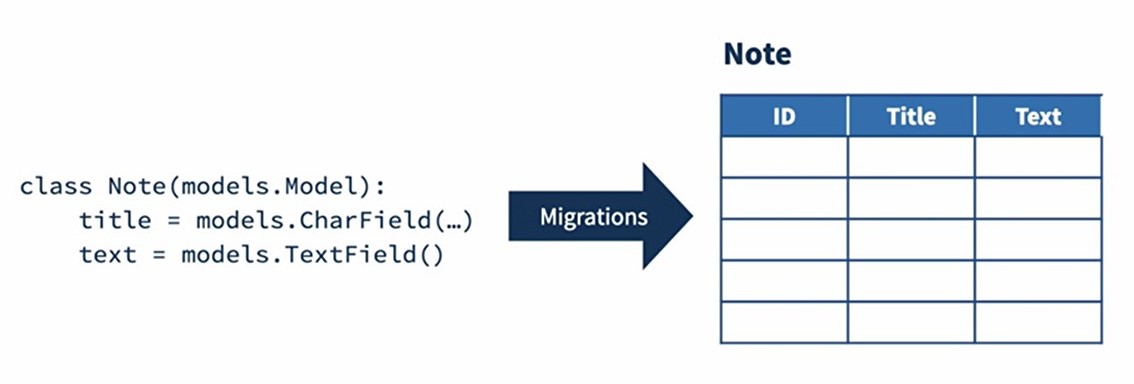
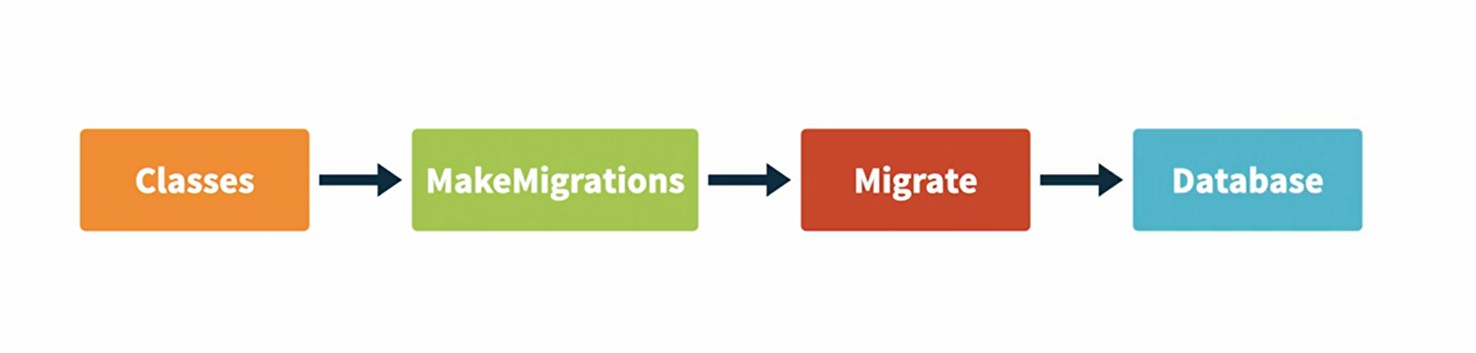
**Django ORMs**

* how to create your own models
* how the structure of creating models work

Django uses an object-relational mapping system or ORM to handle database communication and changes. What you need is to write class models that will then be transformed by migrations into database tables. Each class (known as a model) is a database table and each class attribute is a column.



The way we transform a model into a database table is by the creation of migrations. Migrations will have the step-by-step transformation that a database must do to apply the changes made in the code.

 You've seen that we use the command ‘migrate’ to apply migrations to a database. Similarly, we can use the command ‘make migrations’ to create migration space on the current code.

The process of using a class, defining a model, creating a migration, and applying the immigration and the changes to the database is the ORM's job. And Django's ORM is known for being one of the best ORMs for Python and SQL databases.

Diagram

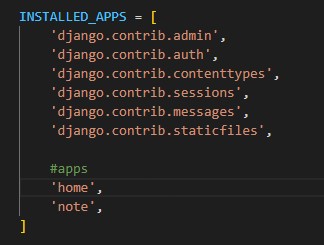
Description automatically generated

**How to create a new model using Django ORM**

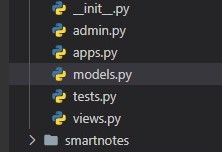
* Create a new app called “notes” by typing in terminal -> django-admin startapp notes.



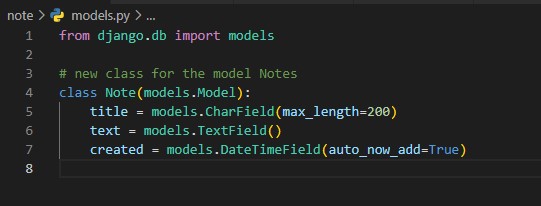
* Go to the settings, and add the new app in the installed app variables.



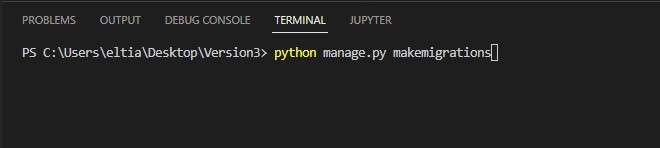
* Now, go back to this new app and open models.py file, where we can create the models that we'll use in this app.

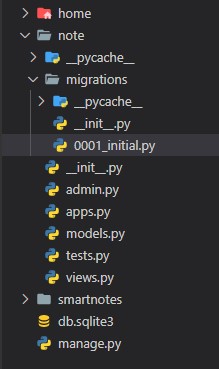


* Create a new class called “Notes” that will inherit from ‘models.Model’ (this way Django knows that it will have an effect on the database).
* Add attributes in our note: title, notes, and created. A ‘title’ is a short text with the type Charfield, which is a limited text view. It also has a parameter called ‘max\_length’, we will set it to 200 in order our title set up to 200 characters -> (max\_length=200). And the ‘notes’ shouldn't have a limit, so instead of using Charfield, we can use the type TextField. ‘created’ is going to be a DateTimeField type, to show when the note was created. So, we need to add a parameter called ‘auto\_now\_add’ equals true, which means that every time a note is created, this field will be correctly populated with the time that this note was created.

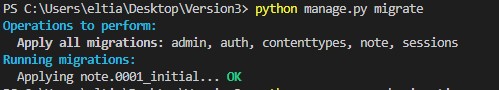


* Create migrations – type ‘python manage.py makemigrations’ in the terminal. After which a folder called ‘migrations’ is created with a file named ‘001\_initial’ inside of it.





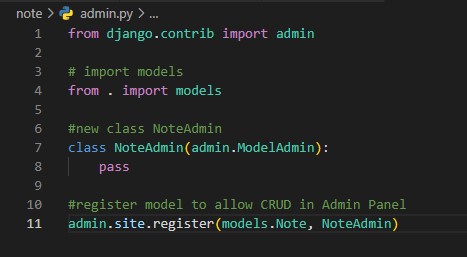
So far, we haven't changed anything in the database, we just created the set of instructions, so everything continues as it is. What we need to do now is to apply the migrations so we can run ‘python manage.py migrate’ in the terminal and we're done.



The changes were applied to the database and we have a new table!

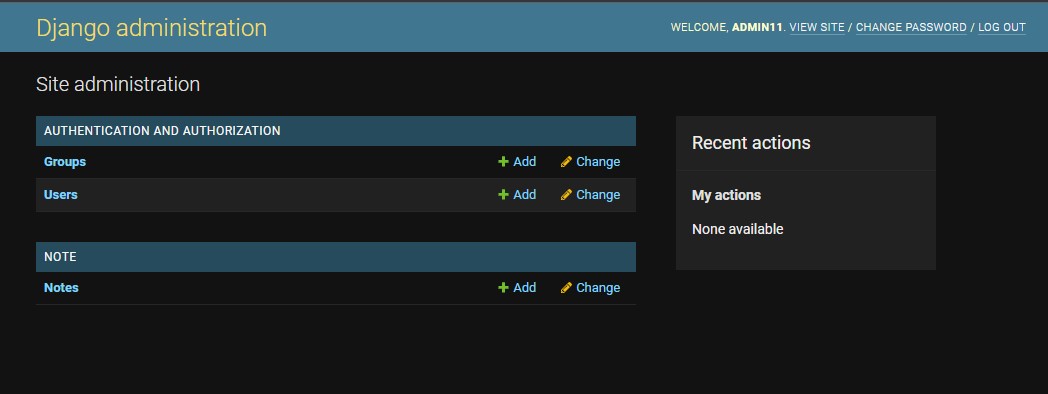
**Using admin for data creation and manipulation**

In order to see table ‘Notes’ in admin interface we need to configure it.



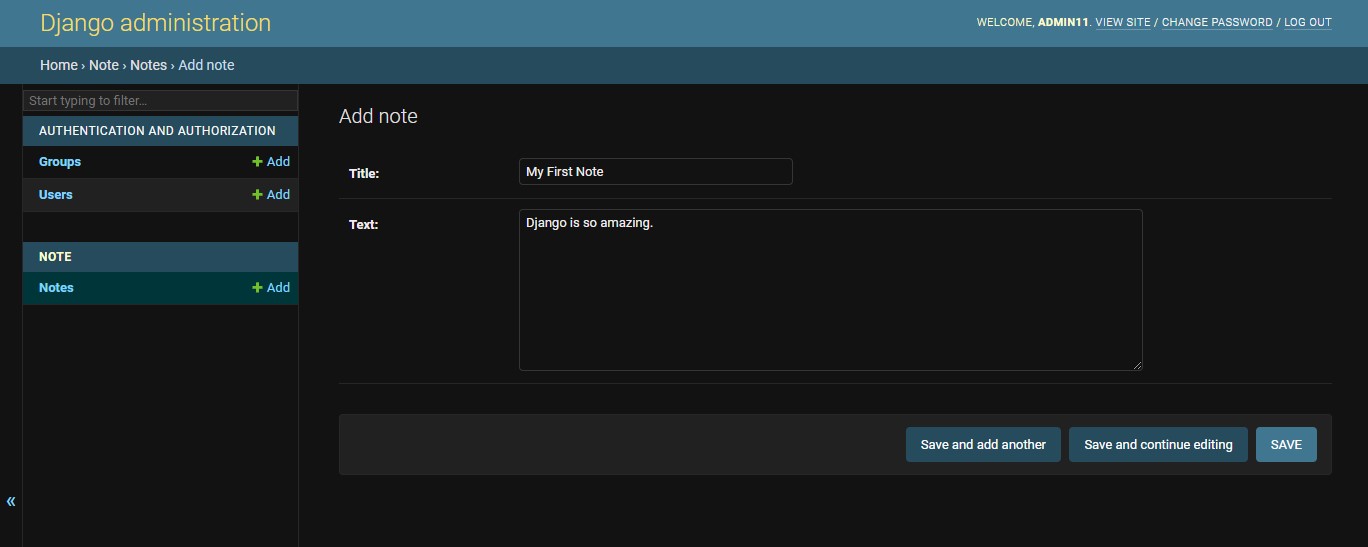
First, go to the ‘note’ app and open a file called admin.py. This is where we go in order to add which models can be displayed. Create a class and call it NoteAdmin. This class should inherit from admin.ModelAdmin. Then import models, and on the bottom of the file, register that the model is attached to this admin model. So, write admin.site.register( models.Note, NoteAdmin).

After, go back to the admin and refresh it.



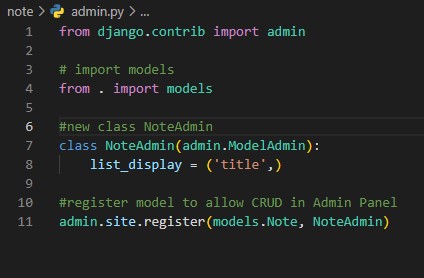
There you go. Now we can see that the ‘Notes’ model is available on the admin interface.

Now use the add button to create a new note. Let's write in Title: ‘My First Note’ and in Text: ‘Django is so amazing’. Save it.

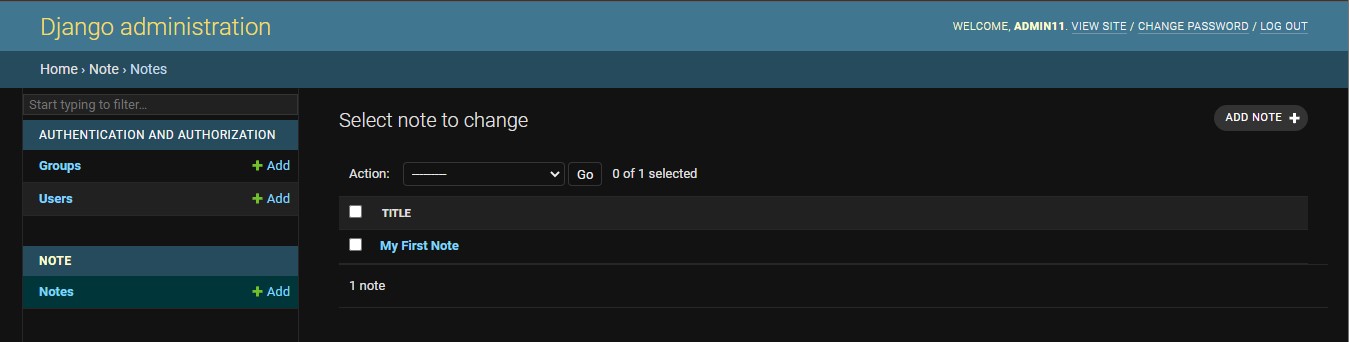


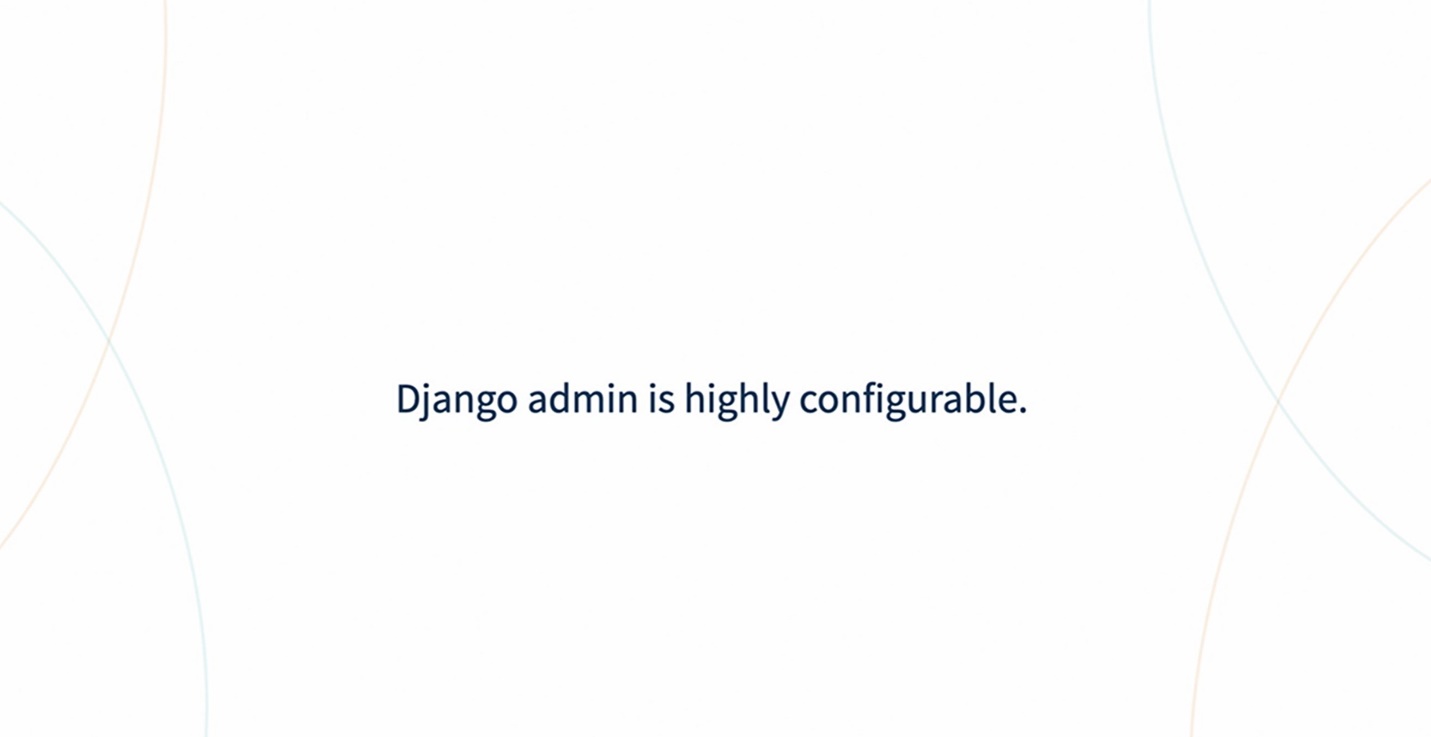
Okay, we have our first note created.

We can see that is displayed as ‘Notes Object(1)’. In case of long list it can be confusing, so go back to the admin class. Instead of pass write ‘list\_display’, which is going to be a tuple, and let's pass ‘title’ as an argument here. Let's save this.



Refresh Admin interface and you can see that  the title of the note being displayed now.

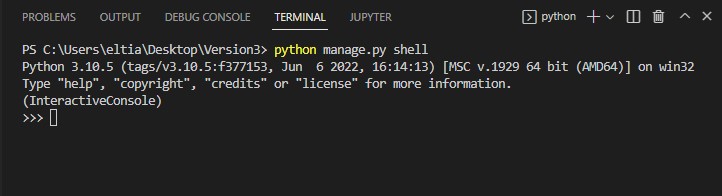
The default configuration of admin also allows that all fields can be changed by all users. However, we can edit the admin model class and start adding some specialized logic. We can remove some fields from being edited. We can allow only staff users to write notes. There's a lot we can do.



**Using Django shell for creating and querying data**

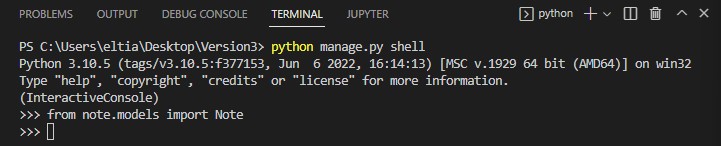
It’s time to learn how to handle models through code. Django has a tool we can use to check the content of a database, which will make our life so much easier, the Django shell.

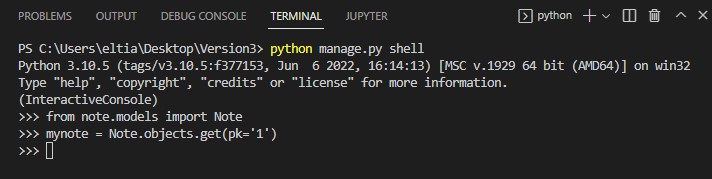
Go to the terminal and type python manage.py shell.



You can see here that we have a Python interpreter, however, this is no ordinary interpreter, it is already tightly coupled with our project.

For instance, we can type from note.models import Note, which is the model we just created, and with this, we can use it to query the objects in the database.

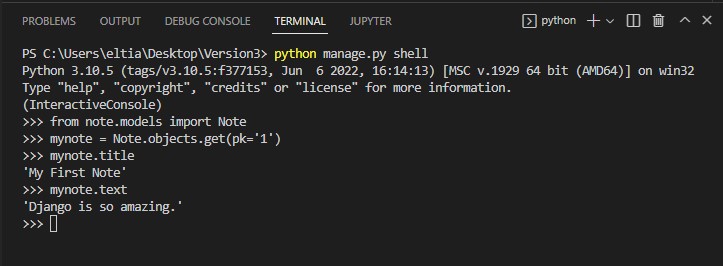


Let's try to get the first note. Type 

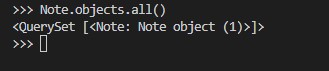
There, we have our note!

 Note.objects is the main way of accessing data from the notes table in the database.

 The .get method will search for one object, where the pk (private key) is equal to one and returns that object. Now we can use it to access attributes of the model by simply typing mynote.title, or mynote.text.



We can also query for all objects in the database by using the method .all instead of the .get, so type -> Note.objets.all()



There you go!

We only have one note so far. The return of this function is a query set, which is a very useful tool, but you can think of it as a list with the superpowers.

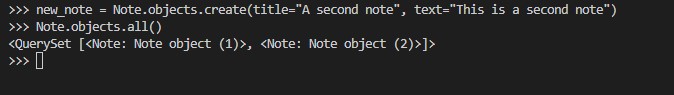
We can also create a new note via the command line,

type ->  new\_note = Note.objects.create(title=”A second note”, text=” This is a second note”)



There! The note was added to the database.

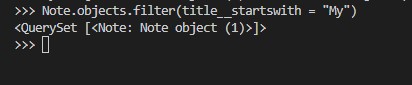
If we query it again, you can see now that we have two objects being returned.



If you prefer, you can open the admin interface and check it out.

We can also filter notes that we want, for instance, we can query for the notes that have titles starting with the word ‘my’.

Type ->  Note.objects.filter(title\_\_startswith = “My”)



The filter also returns a query set, which in this case, returns the first object.

We can also search by something that exists inside the notes, for instance, we can try to find texts that contains the word “Django”.

So we type -> Note.objects.filter(text\_\_icontains = “Django”).



There you go, only the first object has the word Django in the text.

We can also query for the opposite. We can actually filter notes by excluding them, so let's do the opposite.

Type -> Note.objects.exclude(text\_\_contains=”Django”)



You can see now that the filter is returning the second note instead of the first. The fun part is that query sets can also be filtered, meaning that we can add multiple filters at once, for instance, we can filter all the notes containing the word Django, but the title doesn't say anything about Django. Text contains the word Django, but exclude the ones where the title contains the word Django.

>>> Note.objects.filter(text\_\_icontains=“Django”).exclude(title\_\_contains=”Django”)



There you go.

As you can probably imagine, we can go on and on here with thousands of examples on how to query data. Django's ORM has a very neat interface that is very intuitive and yet highly powerful.