

**Introduction**:

In this section we will learn about Django’s authentication system for identifying users. So we will learn,

* How the authentication system works
* User model and customizing it
* Extending the user model
* Creating profiles
* Managing groups and permissions

**Django Authentication System**:

Every Django application comes with a full featured and flexible authentication system. Let us dive in and take a closer look.

Go to list of *installed apps*,

INSTALLED\_APPS = [

    "django.contrib.admin",

    "django.contrib.contenttypes",

    "django.contrib.sessions",

    "django.contrib.auth",------------->Authentication system app

    "django.contrib.messages",

    "django.contrib.staticfiles",

    "django\_filters",

    "corsheaders",

    "rest\_framework",

    "playground",

    "debug\_toolbar",

    "store",

    "store\_custom",

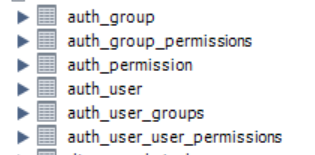
    "tags",

    "likes",

]

Here, *django.contrib.auth* is the authentication system and *using this system we can identify users, allow them to log in, log out, change their password and so on*.

In this app we have a bunch of models like *user group* and *permission* (*obviously we have tables for persisting these models*).

🡨These are tables for the *auth* app.

Let us look at the auth\_user table,



In this table we have

🡪 user id for each user.

🡪 We have password which is stored in encrypted format.

🡪 last login date and time.

🡪 *is\_superuser* tells *if this user has all the privileges. If it is set to 1 then this user can do everything in this application*.

🡪 Then we have username, first\_name, last\_name and email.

🡪 *is\_staff* tells if the user can log into the admin area. *So we can create additional users and give them access to the admin area or we can control the level of access they have to this area*.

🡪 Last we have is\_active and date\_joined, pretty straightforward.

Now we can easily customize this table and add additional columns, but before we do that let’s go back to settings module.

In this module we have another section called *middleware*. *A middleware is a function that takes a request and either passes that request to another middleware or returns a response*.

MIDDLEWARE = [

    "corsheaders.middleware.CorsMiddleware",

    "debug\_toolbar.middleware.DebugToolbarMiddleware",

    "django.middleware.security.SecurityMiddleware",

    "django.contrib.sessions.middleware.SessionMiddleware",

    "django.middleware.common.CommonMiddleware",

    "django.middleware.csrf.CsrfViewMiddleware",

    "django.contrib.auth.middleware.AuthenticationMiddleware",

    "django.contrib.messages.middleware.MessageMiddleware",

    "django.middleware.clickjacking.XFrameOptionsMiddleware",

]

In Django when we receive a request, at some point that request is going to be passed to a *view* and during this time *Django is going to run that request through these middleware functions in order*.

Now each function can take the request and add something to it or it can return a response. *If it returns a response the next middleware function is not executed*.

"django.contrib.auth.middleware.AuthenticationMiddleware",

This *AuthenticationMiddleware*’s job is *to read the user information from the request and set the user attribute on the request object*.

To understand it, let us go to one of our views,

class CollectionViewSet(ModelViewSet):

    queryset = Collection.objects.annotate(products\_count=Count("product")).all()

    serializer\_class = CollectionSerializer

    def destroy(self, request, \*args, \*\*kwargs):

        if Product.objects.filter(collection\_id=kwargs["pk"]).count() > 0:

            return Response(

                {

                    "error": "This collection cannot be deleted because it is associated with a product"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        return super().destroy(request, \*args, \*\*kwargs)

In the *CollectionViewSet* and its destroy method, we have a *request* object.

    def destroy(self, request, \*args, \*\*kwargs):

        request.user 🡪 there is a user attribute from request object.

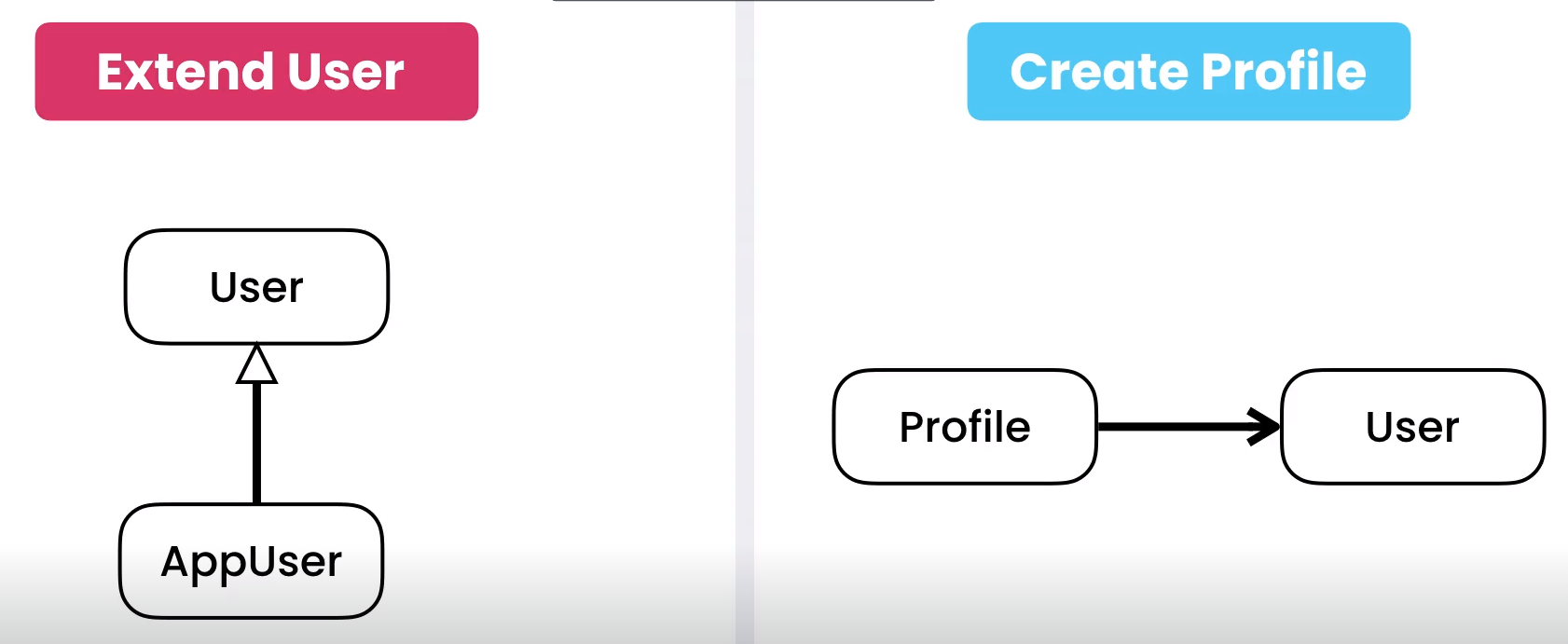
        if Product.objects.filter(collection\_id=kwargs["pk"]).count() > 0:

This request object at runtime is going to have an attribute called *user*, which is either set to an instance of the anonymous user class or an actual user object.

This is the *job of authentication middleware, which is to read the user information from the request and set this attribute*.

**Customizing the User Model**:

The user table in Django authentication system is pretty good but sometimes we need to store additional data about the user.



Here have two options,

🡪 *Use inheritance to extend the user model*.

We can create another model called *AppUser* and this model will extend the User model in Django.

🡪 Or we can *create a profile*.

We can create a profile model and in this model we add a one to one link with the user model. In this scenario we won’t use inheritance but *composition* instead (*Profile model is composed of User model*).

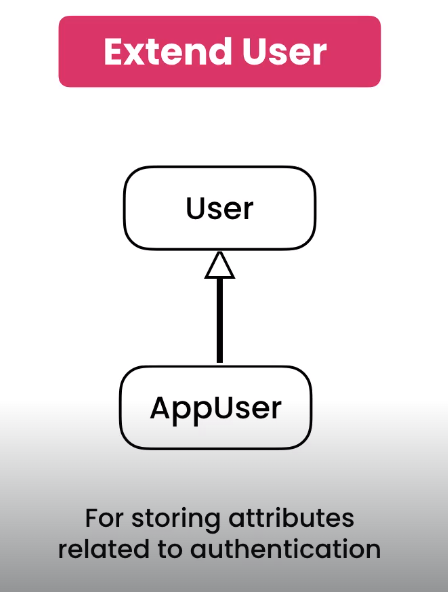
What are the practical differences and when should we use which?

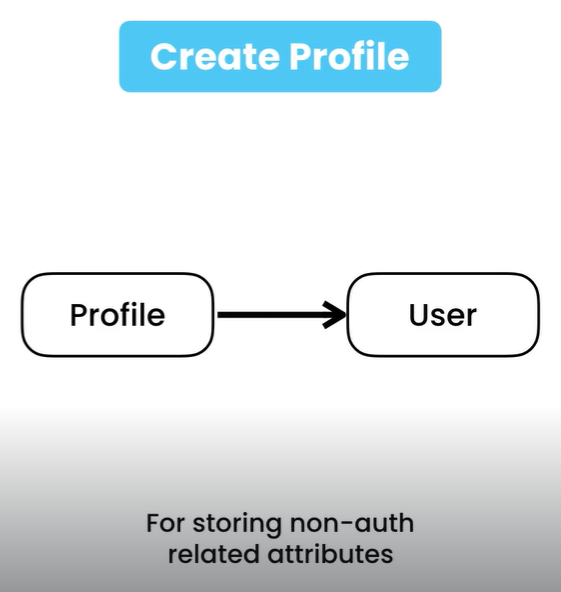
🡪 In terms of database with the first approach we will end up extending the user table.



So any extra attributes that we add in our custom user model will end up in this table.

🡪 With second approach we are not going to extend this table, instead we will have a separate table and in that table we will have a foreign key to the User table.

🡨First approach should be used only for storing attributes related to authentication (*anything that impacts the authentication process*).

🡨 Anything extra that is not related to authentication should really go to in a profile table (*user’s birth date, address etc.*).

With this approach, we allow each app to have a different concept of a user’s profile.



For example in the Sales app, Customer model represent user’s profile, where in a HR app, Employee model represents user.

Note: We cannot use extend user approach in the middle of a project (*it’s a little tricky*).

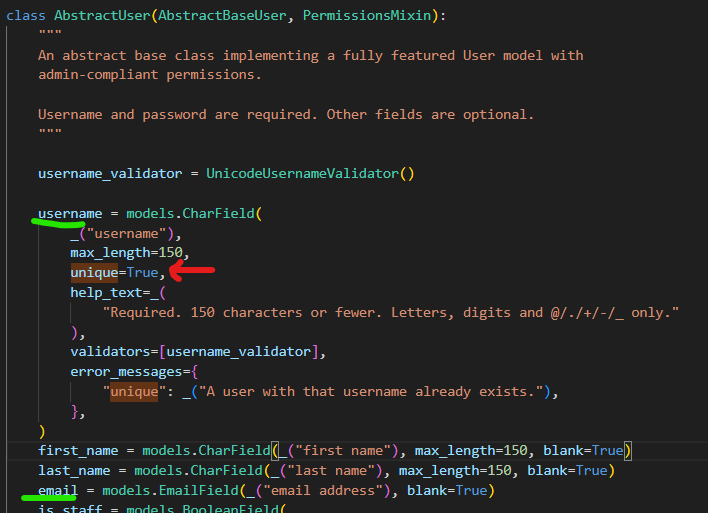
Most of the time we use second approach (*Create Profile*) to customize user model.

**Extending the User Model**:

View *AbstractUser* class from auth.models module,

from django.contrib.auth.models import AbstractUser

Our username column has a unique constraint, so we cannot have multiple users with the same username.



But *email* has no such unique constraint, so multiple users can have same email.

*In the future we might need to add a unique constraint on email column as well which is a valid use case for extending the User model because we are changing something related to authentication*.

So we will create a new User model, but where should we add it? Because we don’t want to add it in our store app because this app is only for building an online store (*nothing to do with authentication*).

What we are trying to solve here is something very specific in this project. The other app is ***store\_custom***, *we created this in first part of the course to combine features from different apps*.

So the code that we write here is very specific to this project. Let us rename this app to ***core*** (*since this is essentially the core of this project*).

After Renaming an app in installed apps we need to do these things,

🡪 Rename the class name in app.py from StoreCustomConfig to CoreConfig and name to *core*.

from django.apps import AppConfig

class CoreConfig(AppConfig):

    default\_auto\_field = "django.db.models.BigAutoField"

    name = "core"

🡪 Change the name in list of installed apps.

    "tags",

    "likes",

    "core",

]

It is suggested to add this app in the last after all the reusable apps (better readability).

Back to the core app, in the models module,

from django.contrib.auth.models import AbstractUser

from django.db import models

class User(AbstractUser):

    email = models.EmailField(unique=True)

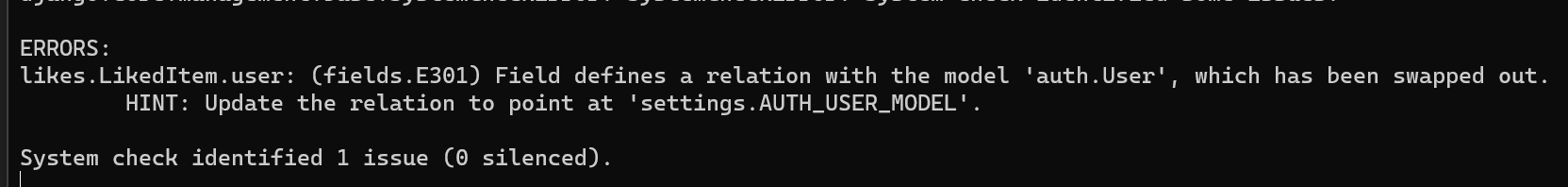
Here we create a new class called User which will extend the *AbstractUser* class in the authentication system.

Now we need to tell Django that we will use this *User* class instead of user class in the authentication system. To do this..

🡪 Go to settings module and here we will define a new setting,

AUTH\_USER\_MODEL = "core.User"

Now our server is throwing an error,



likes.LikedItem.user: (fields.E301) Field defines a relation with the model 'auth.User', which has been swapped out.

HINT: Update the relation to point at 'settings.AUTH\_USER\_MODEL'.

*We are seeing this error because we swapped out the* ***built in User model in the auth app*** *with our* ***custom User model***.

So let us jump to LikedItem class,

class LikedItem(models.Model):

    user = models.ForeignKey(User, on\_delete=models.CASCADE)

    content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

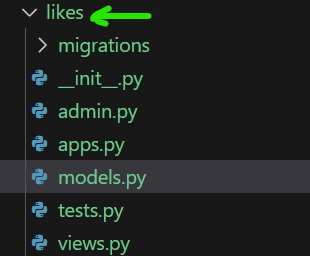
Here we have a foreign key to the *User* model define in, *auth.models*.

from django.contrib.auth.models import User //Built in User model

One way to solve this is by using *custom User* model.

from core.models import User //Custom User model

But there is a problem with this approach which is now our LIKES app is dependent on the *User* model in core app.



So *it is no longer an independent reusable app, we can no longer distribute it without distributing Core app*.

Therefore we do not want to explicitly import this User class here. Instead we will import settings module.

from django.conf import settings

And from the settings module, we can read the *AUTH\_USER\_MODEL*.

class LikedItem(models.Model):

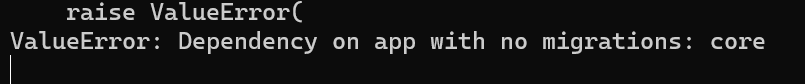
    user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

    content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

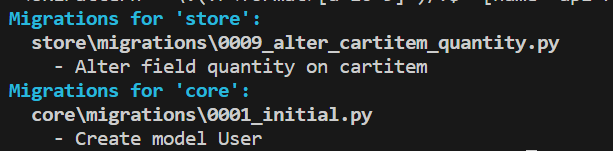
Now our error is solved, but we see a new error,



*We are seeing this error because our LIKES app is indirectly dependent on the core app because here we have a new custom user model but currently there is no migration to create a table for this model*.

So let us run,

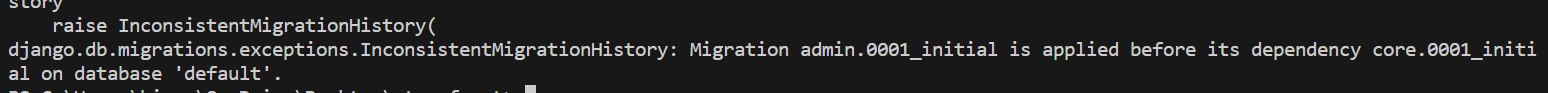
🡪 python manage.py makemigrations



And then migrate command,

🡪 python manage.py migrate

We get an error,



raise InconsistentMigrationHistory(

django.db.migrations.exceptions.InconsistentMigrationHistory: Migration admin.0001\_initial is applied before its dependency core.0001\_initial on database 'default'.

*The reason this is happening is because we decided to swap out the User model in the middle of the project*.

So first migration in the admin module (*admin.0001\_initial*) is dependent on the User model and later on we decided to change that User model to a different User model defined in *core.0001\_initial*.

This is why we cannot easily extend and swap out the User model in the middle of the project. So *as a best practice we should always create a custom User model at the beginning of our project even if there is no requirement to change the authentication flow in the future*.

class User(AbstractUser):

    pass

Just create an empty class using *pass* keyword and this will ensure that later on, if you want to replace this class, you are not going to have any problems.

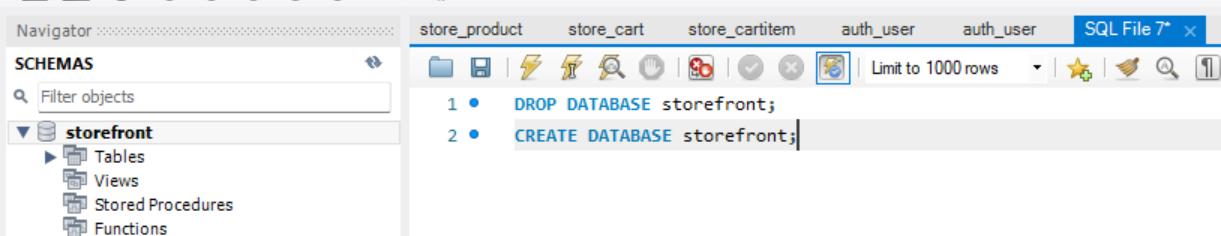
*The only way we can solve this problem is by* ***dropping and recreating*** *our database* (*nuclear bomb solution*).

To destroy and create Database,

🡪 Open MySQL query console,

DROP DATABASE storefront;

CREATE DATABASE storefront;



Run these queries,



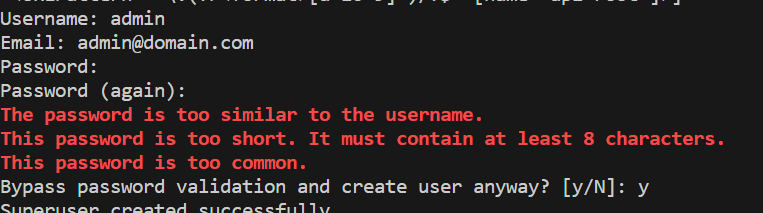
🡪 Now migrate our database one more time,

python .\manage.py migrate

This time all migrations were applied successful and our server is running successfully.

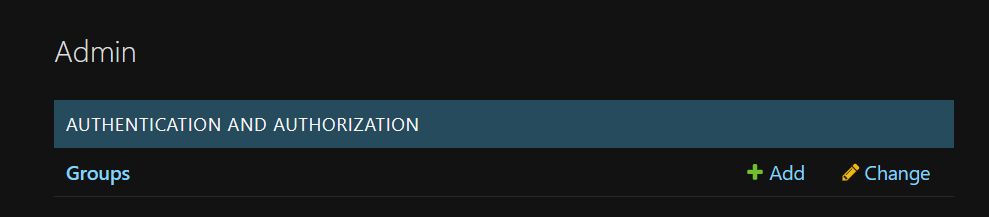
But there are a couple of issues we need to fix in the admin panel. We cannot login into admin panel now, because we created our database from scratch, so we need to create a new *superuser*.

python .\manage.py createsuperuser



Password: admin

We can now login to admin page,



But notice we no longer have User page under Authentication app.

So in the CORE app, we need to go to *admin* module and *register an admin model for registering our users*.

We should define a new class called *UserAdmin* which will extend *UserAdmin* in the auth app. So that we get all the functionality already implemented for us.

class UserAdmin()

Since we have a name conflict, we use alias as *BaseUserAdmin*,

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

Now our class will extend BaseUserAdmin,

class UserAdmin(BaseUserAdmin):

    pass

Finally register it with our *User* model, that we created.

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

from .models import User

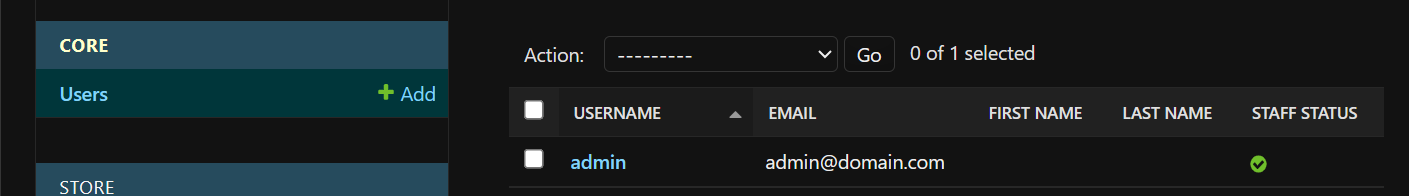
from django.contrib import admin, messages

@admin.register(User)

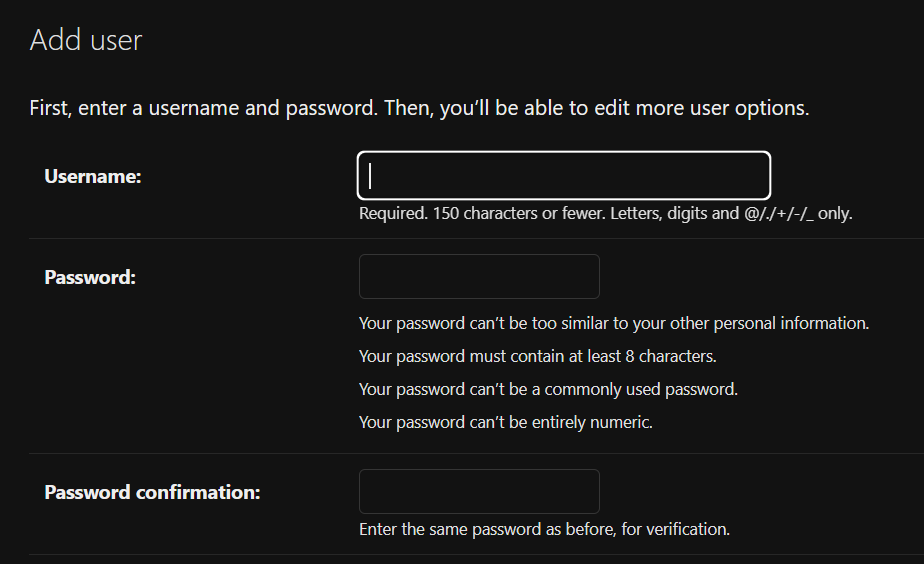
class UserAdmin(BaseUserAdmin):

    pass

In the admin panel, we can see CORE app where we can access all our users.



Let us add a new User,



Currently we can see only three fields Username, password and password confirmation, but we also need to capture email because we applied a unique constraint on the email field.

*If we do not supply a value for email, we will end up with an empty string in the database. So next time we create another User, that user will also have a blank email and we will get a duplicate record error*.

Let us look at our BaseUserAdmin,

@admin.register(User)

class UserAdmin(admin.ModelAdmin):

    add\_form\_template = "admin/auth/user/add\_form.html"

    change\_user\_password\_template = None

    fieldsets = (

        (None, {"fields": ("username", "password")}),

        (\_("Personal info"), {"fields": ("first\_name", "last\_name", "email")}),

        (

            \_("Permissions"),

            {

                "fields": (

                    "is\_active",

                    "is\_staff",

                    "is\_superuser",

                    "groups",

                    "user\_permissions",

                ),

            },

        ),

        (\_("Important dates"), {"fields": ("last\_login", "date\_joined")}),

    )

    add\_fieldsets = (

        (

            None,

            {

                "classes": ("wide",),

                "fields": ("username", "password1", "password2"),

            },

        ),

    )

....................-->*its much more than this…*

In this class we have an attribute called *add\_fieldsets*.

Let us add this in part of code into our UserAdmin along with *email*, *first\_name* and *last\_name* fields (*even though first and last name are not compulsory, but by capturing them here we won’t end up with blank values in these columns*).

@admin.register(User)

class UserAdmin(BaseUserAdmin):

    add\_fieldsets = (

        (

            None,

            {

                "classes": ("wide",),

                "fields": (

                    "username",

                    "password1",

                    "password2",

                    "email",

                    "first\_name",

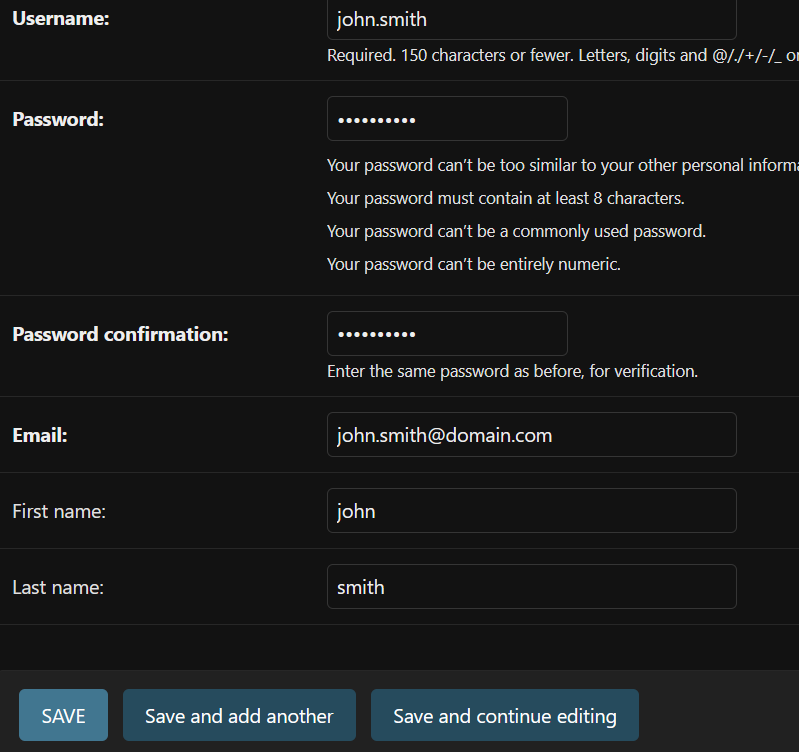
                    "last\_name",

                ),

            },

        ),

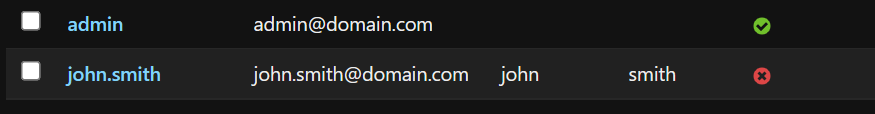
    )



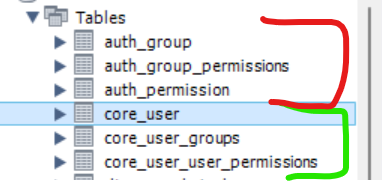
We successfully registered a new user called John Smith(*password:* ***QAQwi4.2hs.NF:p***)

Note: Fields in Bold are compulsory.

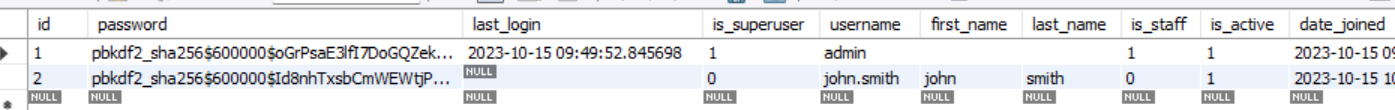
We get our new user,



In the database, observe that User table is no longer part of the AUTH app, it is now part of the CORE app.



And our user table has a new user,



*Recap*:

🡪 To extend the User model, first we create a new model that extends AbstractUser.

🡪 Then in settings module, we set AUTH\_USER\_MODEL to our custom user model.

🡪 From this point forward we never reference the user model directly, instead we use settings.AUTH\_USER\_MODEL.

**Creating user profiles**:

This customer model we have defined here is essentially the profile of a user in the sales app.

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    first\_name = models.CharField(max\_length=255)

    last\_name = models.CharField(max\_length=255)

    email = models.EmailField(unique=True)

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    def \_\_str\_\_(self) -> str:

        return f"{self.first\_name} {self.last\_name}"

    class Meta:

        ordering = ["first\_name", "last\_name"]

Only thing missing is the link to the *User* model. So let us define a new field called *user* here. This is going to be a *oneToOneField*.

*Here we do not want to refer User model in Django or in the core app because the sales app is going to be a reusable app. So we should be able to reuse it in any project, no matter what model is used for representing users*.

In other words we do not want to link this model to the built in User model in Django or a custom user model, instead *we want to link this to AUTH\_USER\_MODEL of the containing project*.

So we want to import the settings module and reference auth user model.

from django.conf import settings

Import settings module from django.conf.

Then we reference *AUTH\_USER\_MODEL* from here,

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    first\_name = models.CharField(max\_length=255)

    last\_name = models.CharField(max\_length=255)

    email = models.EmailField(unique=True)

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE) ---> user field here...

    def \_\_str\_\_(self) -> str:

        return f"{self.first\_name} {self.last\_name}"

    class Meta:

        ordering = ["first\_name", "last\_name"]

Note: By default this setting is set to User model in Django. But in this project we changed it to the User model in the core app.

AUTH\_USER\_MODEL = "core.User"

For *on\_delete* we set it *cascading*, so if we delete a user then the associated customer record is also deleted automatically.

In this Customer model, we have some redundant fields like first\_name, last\_name and email. All these fields exist in the User model so we need to delete them from here.

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

    def \_\_str\_\_(self) -> str:

        return f"{self.first\_name} {self.last\_name}"

    class Meta:

        ordering = ["first\_name", "last\_name"]

Then we need to make a few more changes. Since we removed the first\_name and last\_name fields, here in the *\_\_str\_\_* method we need to reference the user object. As well as for *ordering* use \_\_ (*double underscore for referencing*)

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

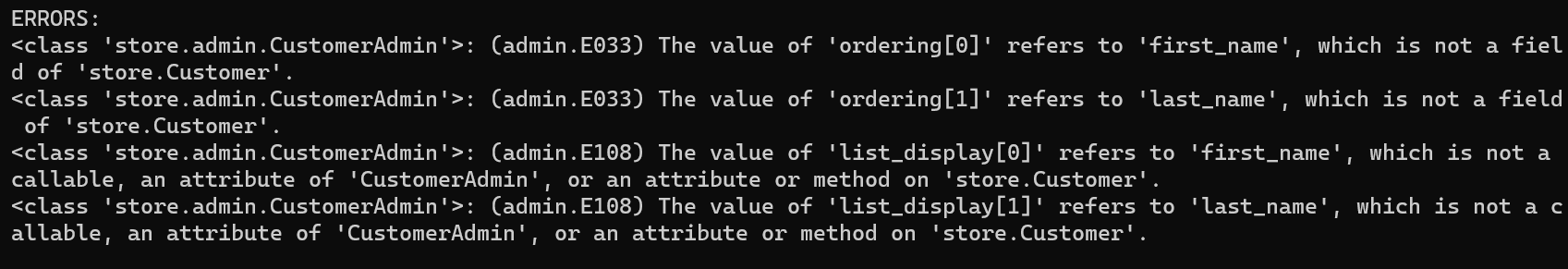
    def \_\_str\_\_(self) -> str:

        return f"{self.user.first\_name} {self.user.last\_name}"

    class Meta:

        ordering = ["user\_\_first\_name", "user\_\_last\_name"]

In the terminal we see 4 errors,



Let us see these one by one,

<class 'store.admin.CustomerAdmin'>: (admin.E033) The value of 'ordering[0]' refers to 'first\_name', which is not a field of 'store.Customer'.

So let us go to *CustomerAdmin* model,

@admin.register(models.Customer)

class CustomerAdmin(admin.ModelAdmin):

    list\_display = ["first\_name", "last\_name", "membership", "orders\_count"]

    list\_editable = ["membership"]

    list\_per\_page = 10

    ordering = ["first\_name", "last\_name"]

    search\_fields = ["first\_name\_\_istartswith", "last\_name\_\_istartswith"]

    @admin.display(ordering="orders\_count")

    def orders\_count(self, customer):

        url = (

            reverse("admin:store\_order\_changelist")

            + "?"

            + urlencode({"customer\_\_id": str(customer.id)})

        )

        return format\_html('<a href="{}">{}</a>', url, customer.orders\_count)

    def get\_queryset(self, request):

        return super().get\_queryset(request).annotate(orders\_count=Count("order"))

Look at the ordering attribute, it means we need to reference the user field here.

ordering = ["user\_\_first\_name", "user\_\_last\_name"]

First two errors are gone now.

Note: When loading customers, we want to eager load them with our users otherwise for each customer a separate query will be sent to database.

list\_select\_related = ["user"]

Next error…

<class 'store.admin.CustomerAdmin'>: (admin.E108) The value of 'list\_display[0]' refers to 'first\_name', which is not a callable, an attribute of 'CustomerAdmin', or an attribute or method on 'store.Customer'.

This one is complaining about *list\_display* attribute.

@admin.register(models.Customer)

class CustomerAdmin(admin.ModelAdmin):

    list\_display = ["first\_name", "last\_name", "membership", "orders\_count"]

Here we cannot use user field here like this, “user\_\_first\_name”. This syntax is not supported here. So the workaround is to define a method called first\_name in the customer model and return user.first\_name from there.

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

    def first\_name(self):

        return self.user.first\_name ------> return first\_name

    def last\_name(self):

        return self.user.last\_name ------> return last\_name

    def \_\_str\_\_(self) -> str:

        return f"{self.user.first\_name} {self.user.last\_name}"

    class Meta:

        ordering = ["user\_\_first\_name", "user\_\_last\_name"]

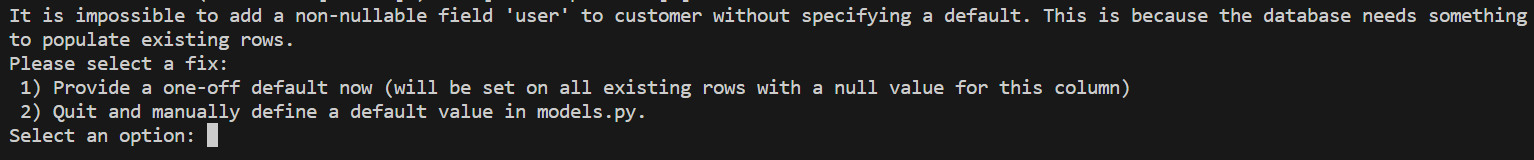
Now all our errors are gone.

So let us create a migration to update the database.

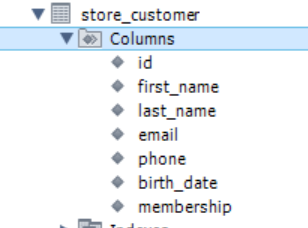
python manage.py makemigrations

We get a *warning* saying,

*It is impossible to add a non-nullable field 'user' to customer without specifying a default. This is because the database needs something to populate existing rows*.



In the database look at the customer table,



We are trying to add a new column to this table and this column does not accept null values. So what is going to happen to existing customers?

That is why we need to supply a default value.

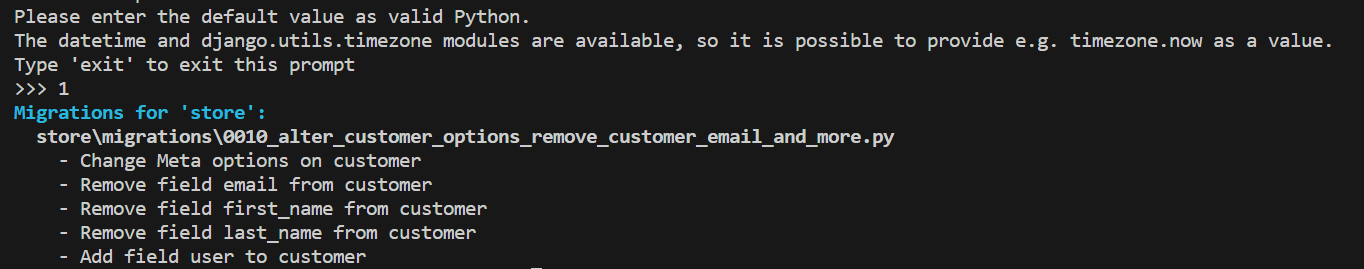
Here we have two choices,

1) Provide a one-off default now (will be set on all existing rows with a null value for this column)

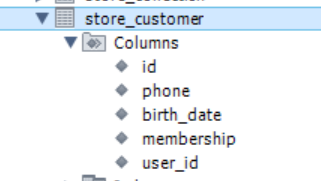
2) Quit and manually define a default value in models.py.

*We do not want to hardcode a default value in our model because otherwise every time we create a new customer, that customer will be associated with a specific user*.

So we will select the first option and provide a one-off default value one (*id of the admin user*).

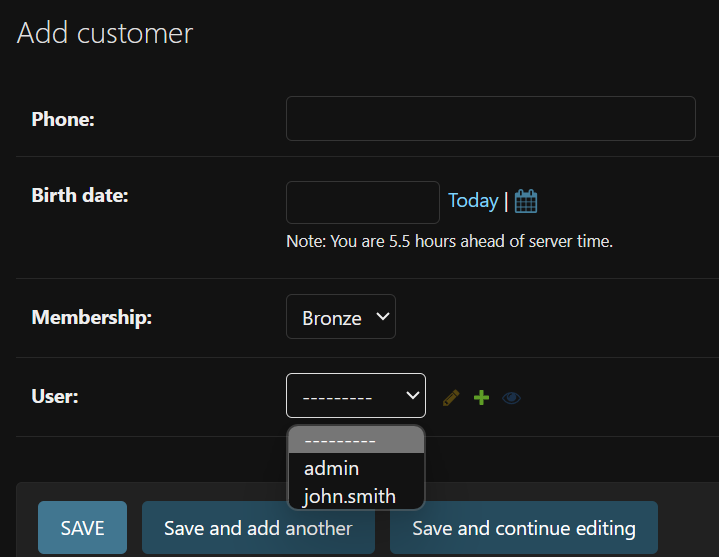


Now python manage.py migrate.



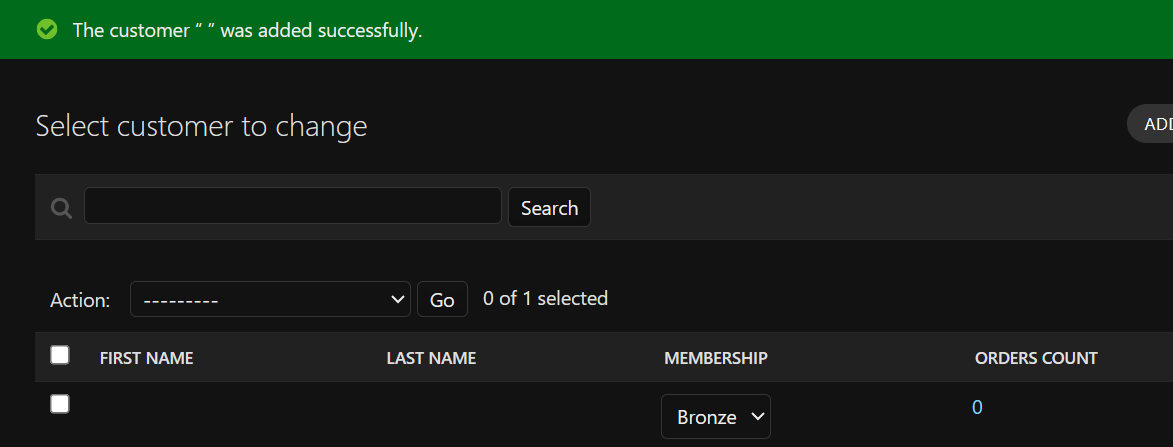
Now in the database we have *user\_id* which is foreign key to the User table and also first\_name, last\_name and email are removed from this table.

Now let us add a new customer to see if everything is working fine.

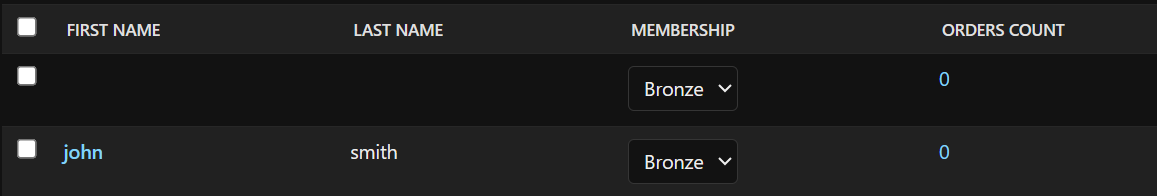


Notice that we have a dropdown list for selecting a user. So we can associate a customer with an existing user (*in the real world scenario we might need to replace this dropdown with an auto complete field 🡪* ***additional exercise***).

For now we will associate it with admin user.



Currently we do not have first name and last name for the admin, so let us add another customer and associate it with john smith (*have a first and last name*).



Now we can see first and last name of this customer is coming from the associated user model.

But there is a problem here, we cannot sort this table by first or last name, we can only sort them by membership or orders.

To fix this problem, back to our Customers model, we need to apply *@admin.display* decorator.

So first import admin module,

from django.contrib import admin

Then we use our decorator and specify the field that we want to use for sorting,

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

    @admin.display(ordering="user\_\_first\_name") 🡪 ordering for first\_name

    def first\_name(self):

        return self.user.first\_name

    @admin.display(ordering="user\_\_last\_name")🡪 ordering for last\_name

    def last\_name(self):

        return self.user.last\_name

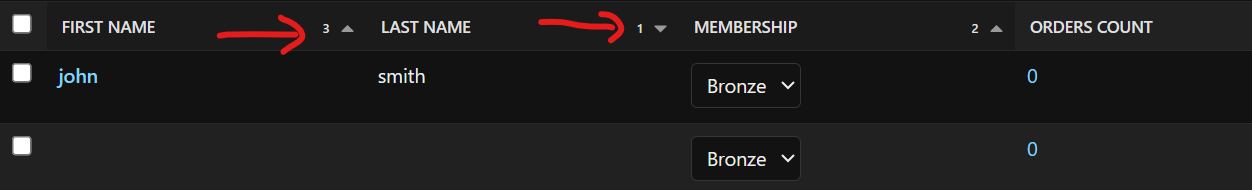
    def \_\_str\_\_(self) -> str:

        return f"{self.user.first\_name} {self.user.last\_name}"

    class Meta:

        ordering = ["user\_\_first\_name", "user\_\_last\_name"]

Now we can sort by first and last name in our admin panel.



*Recap*:

🡪 To define a user profile, we simply create a profile model (*in this case customer*) and in this profile model we add a oneToOneField with the User model.

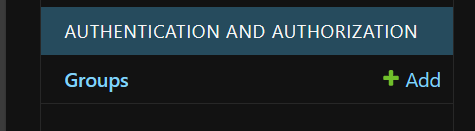
user = models.OneToOneField(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

or more accurately with the AUTH\_USER\_MODEL of the containing project.

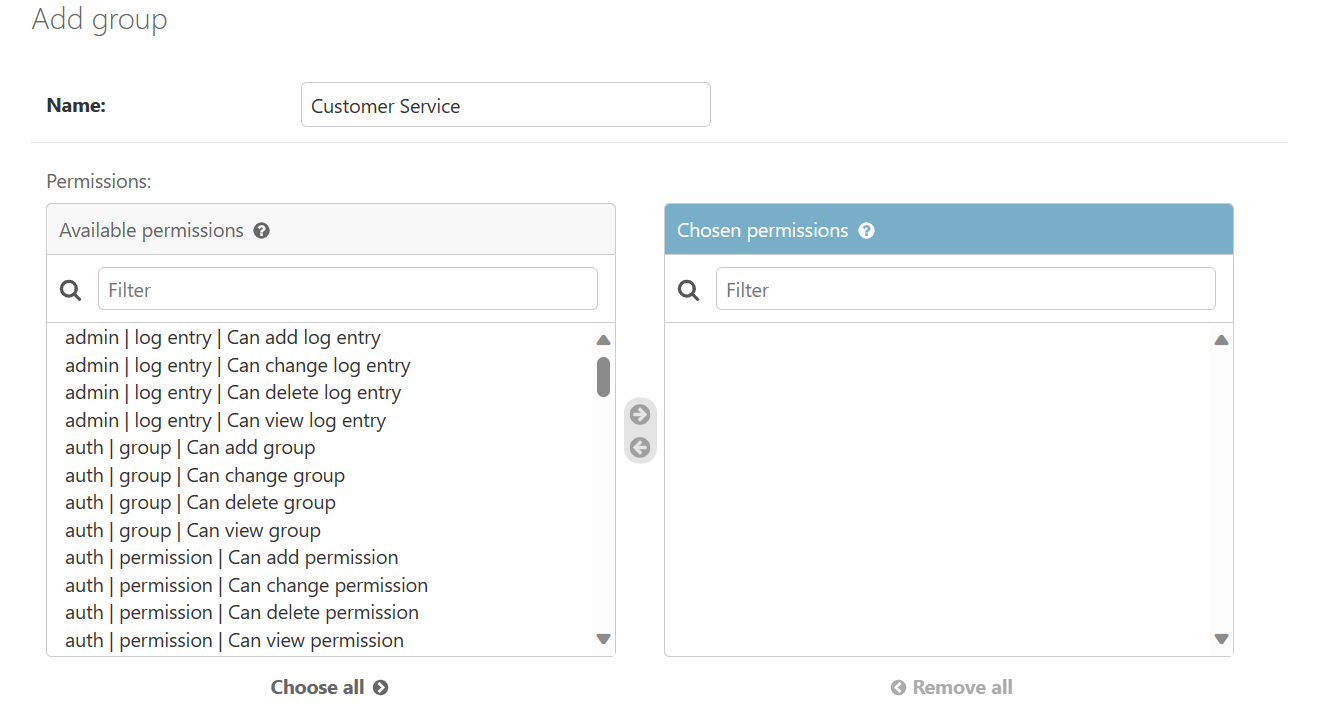
This makes our apps more reusable so they are not coupled to a specific user model implementation.

**Groups and Permissions**:

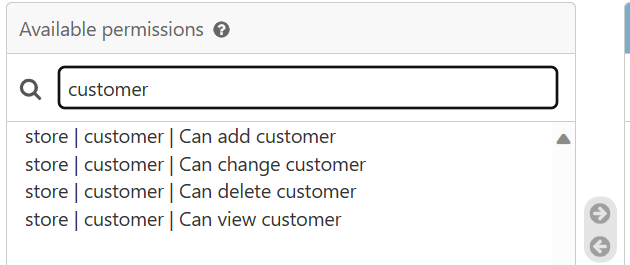
**“***A group is a collection of permissions. So instead of assigning a bunch of permissions on an ad hoc basis, we can add them to one or more groups and each group can contain a bunch of permissions***”**.



In this Group table, currently we do not have any groups. So let us add a new group called ‘*Customer Service*’.



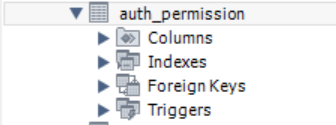
In the box on left side, we can see all the permissions available in our application. So let us filter by customer.

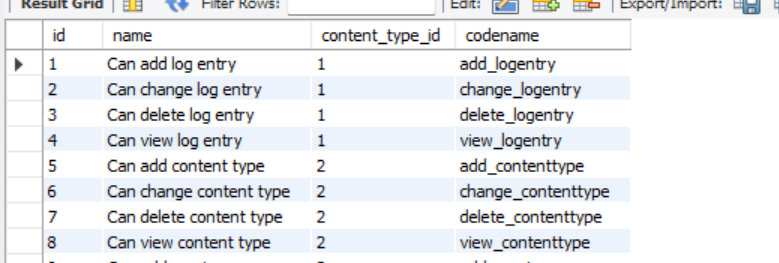


In the store app, for the customer model, we have these permissions. We can add a customer, change it, view it or delete it.

*So every time we create a model and migrate our database, Django automatically creates these permissions for us.*

Let us see where they are stored,

 Go to auth app where we have a permission table.



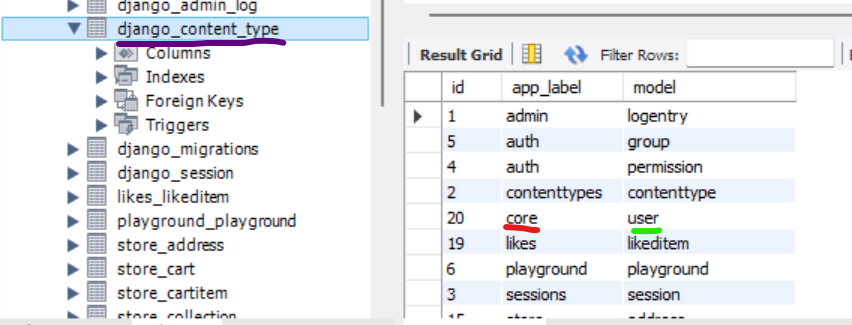
Look at the last record, *can view user*.



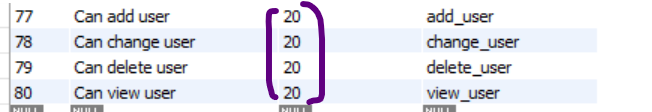
So because we created a custom user model, Django created this permission for us.

This permission has unique identifier, a descriptive name, it also has a codename (*also a unique identifier, but it’s a string that we can reference in our code*).

Notice the *content\_type\_id* column. In this database we have a table called *django\_content\_type* which specifies all models in our application.

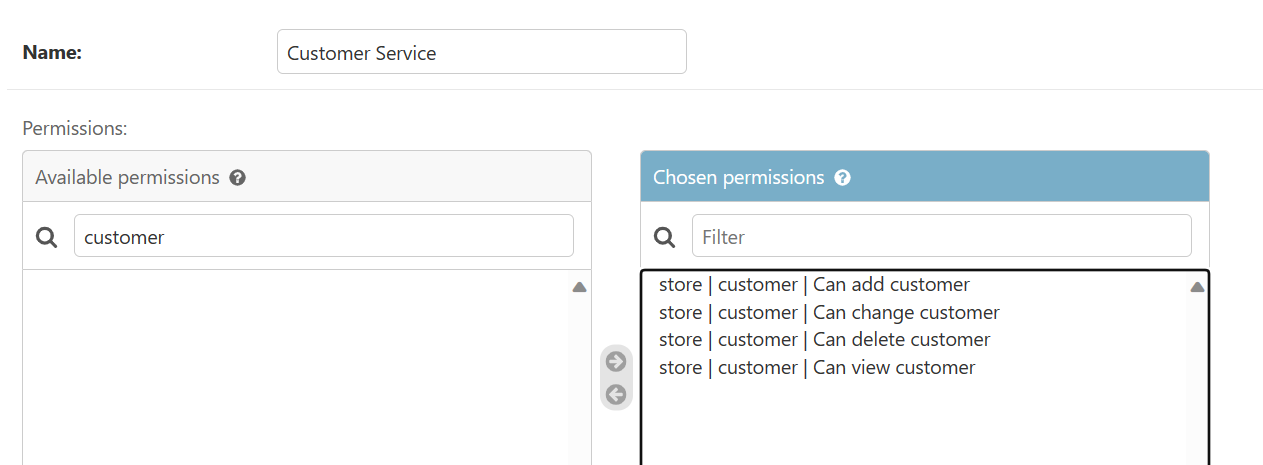


For example, in the *core* app we have a model called *user* and *20 is the id of this model*.

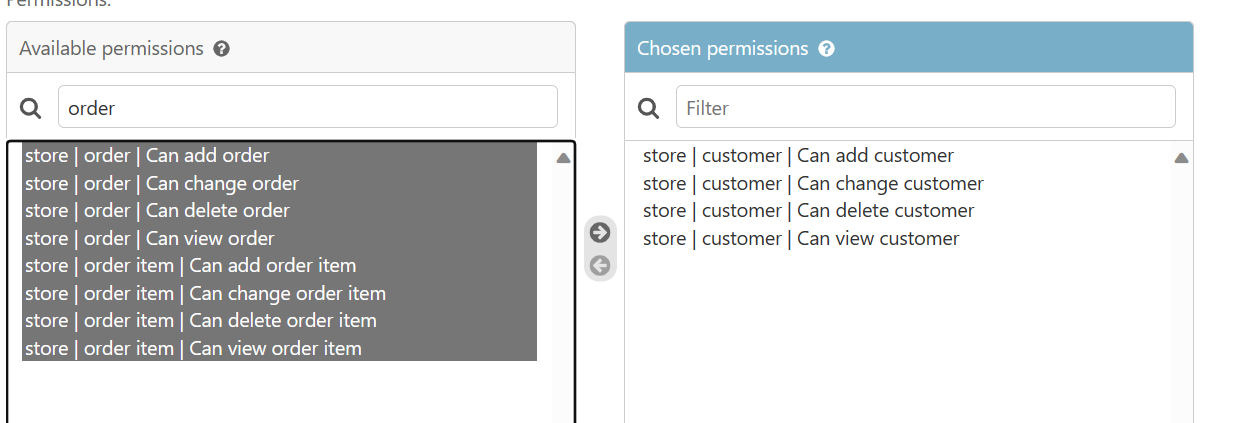


So in the *auth\_permission* table, each permission is associated with this id which is content\_type\_id.

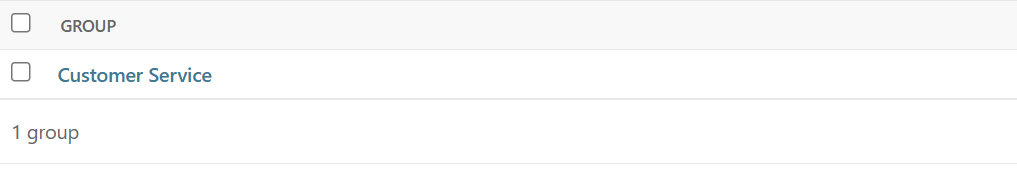
So in the group we will add all the customer related permissions.



For customer service we will also assign permissions to manage orders.

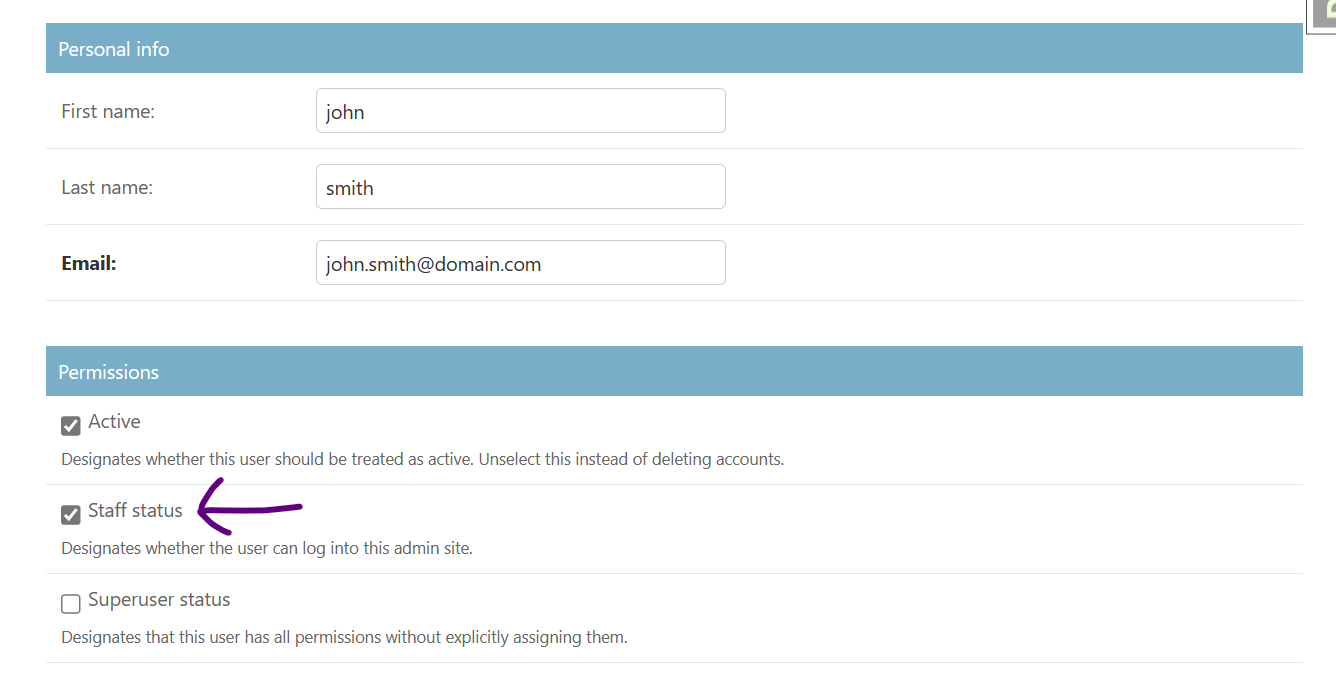


After saving after Customer Service group is created.

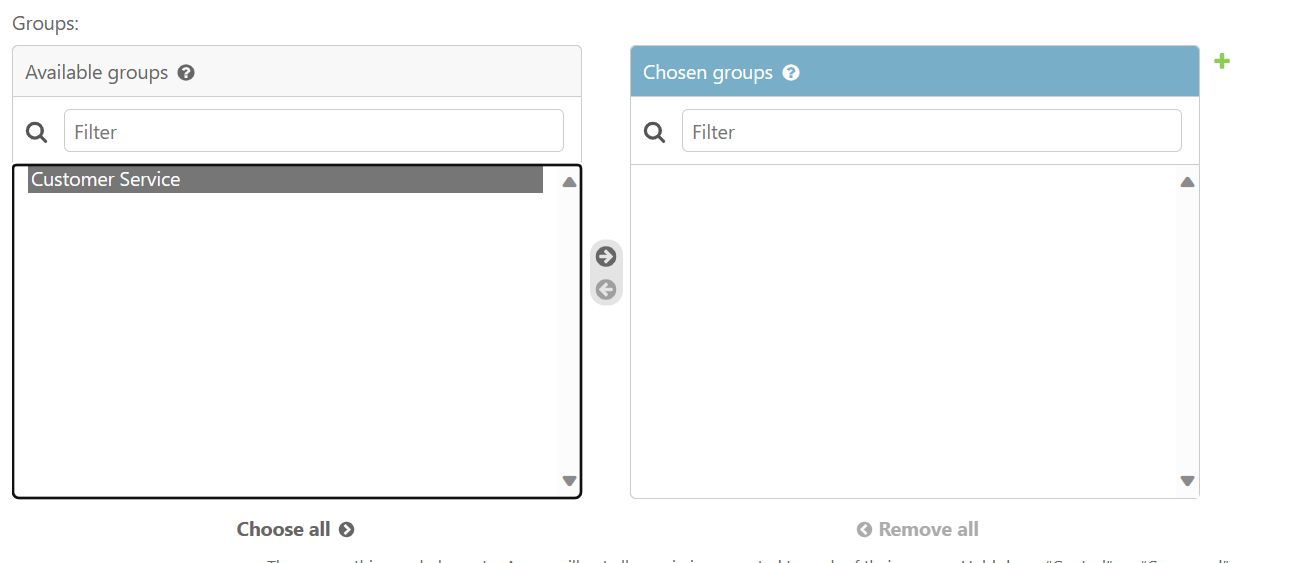


Now let us add our user *John Smith* to the customer service group.

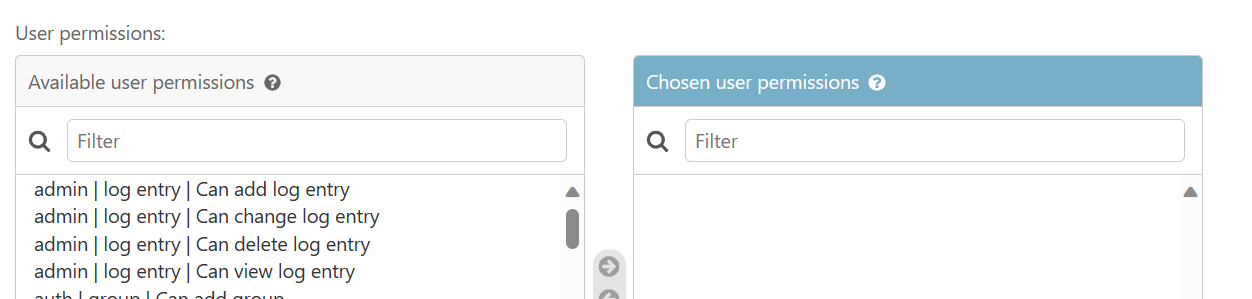
Go to Users 🡪 John Smith 🡪 Select Staff Status (so that this user can login inside admin area)



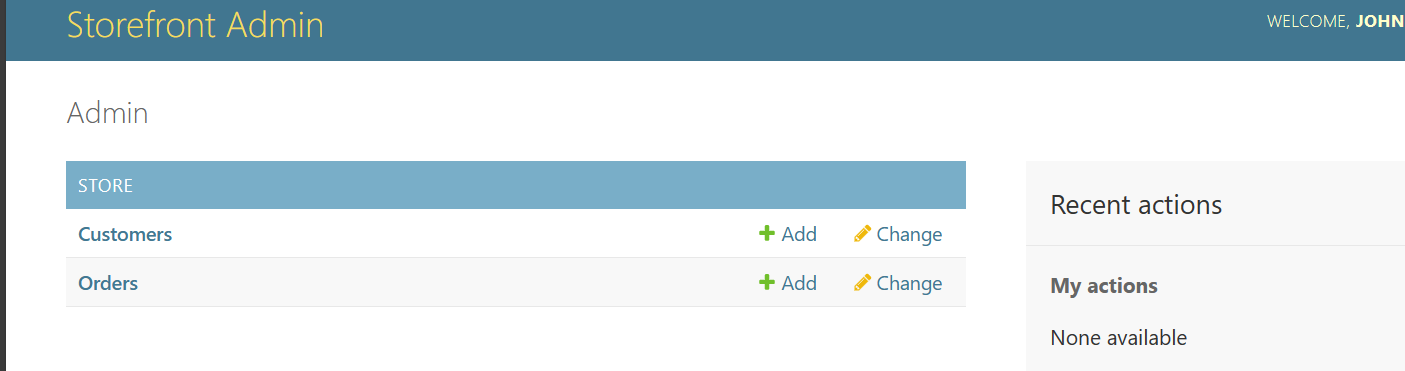
Further down we can see all available groups,



Optionally we can assign john a bunch of permissions explicitly(*not recommended*).



Now save the changes after giving permission then logout and login as john smith.



Now john smith has access to admin panel but he can only manage customers and orders. This is how groups and permissions work.

**Creating Custom Permissions**:

Sometimes we have operations that are not quite about creating, updating or deleting data.

For example think of cancelling an order, it is a special kind of update. So by cancelling an order, we do not want to delete it, we just want to change the status to cancelled.

*We can give some customers the ability to cancel an order but not update it, this is where we need custom permission*.

So let us go to Order class and to create a custom permission we create a *Meta* class inside it.

class Order(models.Model):

    PAYMENT\_STATUS\_PENDING = "P"

    PAYMENT\_STATUS\_COMPLETE = "C"

    PAYMENT\_STATUS\_FAILED = "F"

    PAYMENT\_STATUS\_CHOICES = [

        (PAYMENT\_STATUS\_PENDING, "Pending"),

        (PAYMENT\_STATUS\_COMPLETE, "Complete"),

        (PAYMENT\_STATUS\_FAILED, "Failed"),

    ]

    placed\_at = models.DateTimeField(auto\_now\_add=True)

    payment\_status = models.CharField(

        max\_length=1, choices=PAYMENT\_STATUS\_CHOICES, default=PAYMENT\_STATUS\_PENDING

    )

    customer = models.ForeignKey(Customer, on\_delete=models.PROTECT)

    class Meta:

        permissions = [ --->List of permissions

            ('cancel\_order', 'Can cancel order')

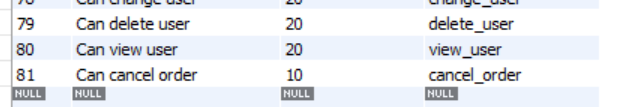
            ]

In the Meta class, we set *permissions* to a list of tuples, where each tuple represents a permission.

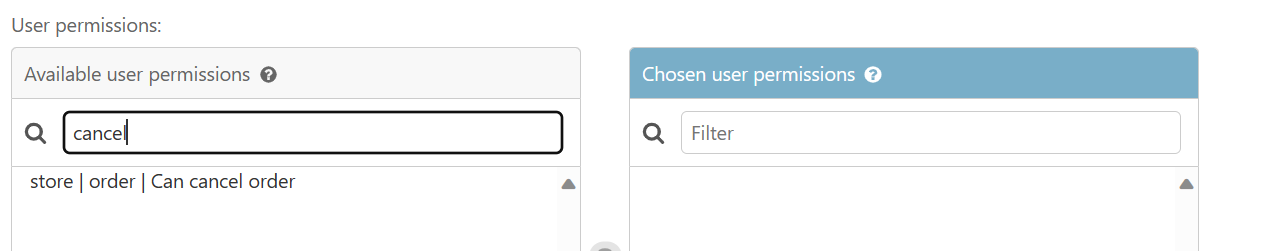
First value in the tuple is *code\_name* (*a unique identifier that we will use in our code*) and second value is just the description.

Now that we have modified our model, we need to create a migration and run it.

After refreshing the database, we can see new permission in our auth\_permission table.



Now we can login to admin and give someone this particular permission.



We can assign this particular permission to John smith.

But what about act of cancelling an order?

That’s the topic for next section. So *in the next section we are going to apply these permissions to our API end points to make them secure*.