**Custom User Model In Django**

A detailed explanation of customizing the user model, admin site, and the inner workings of the authentication and the login flow in Django.

[[Tóth Máté](https://medium.com/@t.mate.laszlo?source=post_page-----98b9a401a6a2--------------------------------)](https://medium.com/@t.mate.laszlo?source=post_page-----98b9a401a6a2--------------------------------)

[Tóth Máté](https://medium.com/@t.mate.laszlo?source=post_page-----98b9a401a6a2--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F10ff50bd9609&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40t.mate.laszlo%2Fcustom-user-model-in-django-98b9a401a6a2&user=T%C3%B3th+M%C3%A1t%C3%A9&userId=10ff50bd9609&source=post_page-10ff50bd9609----98b9a401a6a2---------------------post_header-----------)

16 min read

·

3 days ago

17



AI-generated Django logo

*Note: This tutorial series soon will be available in a short 5-minute and a longer 30-minute version to be able to serve you whether you just want to grab the code or dig deep and understand the concepts in detail. This is the detailed version.*

I’ve spent many days diving into the ins and outs of the user model and the authentication, and to save some time for you and others, here is what I have learned.📚🙇‍♂️

The Django documentation strongly recommends that we use a custom user model from the very start, even before we run the first migration. This practically means that once we create the application using the Django CLI, we don’t run the app, but replace the user model immediately. From my personal experience, it will cause migration conflicts if you want to change the user model halfway through the project, but you will not have problems if you run the initial migrations on a newly created project and then change the model.

That being said, let’s walk through the following:

* [The Built-in User Model](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#f74f)- [User Model](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#7d77)  
  - [ModelManager](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#d57a)
* [Authentication Process](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#451a)  
  - [Creating a Superuser](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#d988)  
  - [Login Page](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#0b38)
* [User Creation Logic on the Admin Page](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#4826)  
  - [ModelAdmin](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#94a0)  
  - [UserCreationForm](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#1780)

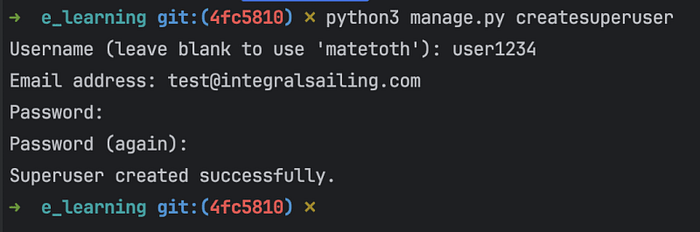
then let’s introduce the two options to tailor the user model:

* [CustomUser(AbstractUser) Implementation](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#8530)  
  - [Create ‘users’ App](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#ad8f)  
  - [Create CustomUser](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#a9a3)  
  - [Create CustomUserManager](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#9cfb)  
  - [Display User Model on the Admin Page](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#fa7d)  
  - [User Creation and Edition Forms](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#6037)  
  - [Customizing the Login Page](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#2a0f)
* [CustomUser(AbstractBaseUser) Implementation](https://medium.com/@t.mate.laszlo/custom-user-model-in-django-98b9a401a6a2#62ce)

*To get the most out of this article, have your IDE ready and open the built-in Django classes mentioned later*

**Built-in User Model**

In your Django app, you will have to create a superuser to be able to log in to the admin page using the python manage.py createsuperuser command. You will be prompted to provide a username, first name, last name, and password(twice).



Command line prompts for superuser details

If you are like me, you might already have a problem with this. Why on Earth would I come up with a unique username and password combination if I can barely remember my girlfriend’s phone number?😅 There are a couple of good reasons actually. It is safer to use a username than an email since if the combination lands in the wrong hands, many people would have sleepless nights due to reused passwords at other websites. It is not possible to recover your password if your application is breached, since Django saves hashed passwords only(random salt is used as well). However, a user can land on a fake copy of your login page and can reveal their email and password there. Another use case is for community-centered websites like **reddit.com** because you might not want to use your email/name for commenting or posting.

**User Model**

Let’s take a look into the DB to understand the default user model. If you were to apply migrations on your newly created app(you don’t have to), then many authentication-related tables would be created including the auth\_user. Here is the creation script for the auth\_user table:

create table auth\_user  
(  
 id integer not null primary key autoincrement,  
 password varchar(128) not null,  
 last\_login datetime,  
 is\_superuser bool not null,  
 username varchar(150) not null unique,  
 last\_name varchar(150) not null,  
 email varchar(254) not null,  
 is\_staff bool not null,  
 is\_active bool not null,  
 date\_joined datetime not null,  
 first\_name varchar(150) not null  
);

A couple of things that we can verify is that the username is indeed a unique column and the user already has a required email address.

*Note: The same user model is used for the superuser, staff, and regular users, however, the model has properties to differentiate between these user types.*

The built-in user model, User(AbstractUser), doesn't implement much on its own, but it extends the  
AbstractUser(AbstractBaseUser, PermissionsMixin), which defines most of the fields that we saw in the DB. The password field and the related saving/verification logic live in the AbstractBaseUser, while the permission-related fields and logic, like verifying whether a user can log into the admin site, are defined in the PermissionsMixin. Since it is not advised by the Django documentation to reinvent password management, we should at least extend the AbstractBaseUser in our custom user model. If we do, then we don't need to implement anything password-related and it will work properly.

**ModelManager**

Every Django model comes with a ModelManager that is assigned to the model's objects property by default. The model manager provides DB capabilities to the specific model, which allows us, for example, to call YourModel.objects.all(). The default manager provides the APIs for the most common use cases, however, sometimes you need custom logic to make the desired change in the DB. This is the case for the AbstractUser as well, and it uses the UserManager implementation to communicate with the DB. The UserManager implements the create user/superuser methods to hash the received password, normalizes(lowercase) the username and email, and sets extra properties for superusers before saving the user instance to the DB. It also provides a method to look up users with certain permissions.

**Authentication Process**

Let’s create a superuser and log in to the admin site while walking through the code.

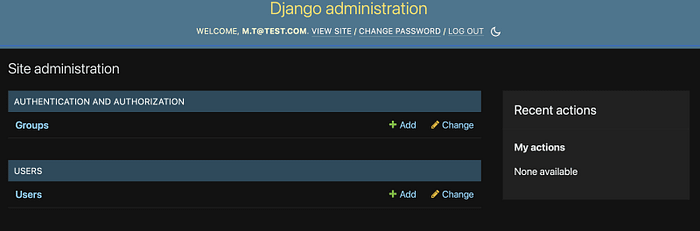
**Creating a Superuser**

When you execute python manage.py createsuperuser, you will be prompted to provide information about the user. What you have to provide depends on the User model, specifically the User.USERNAME\_FIELD, User.PASSWORD\_FIELD, and the fields defined in User.REQUIRED\_FIELDS(the email field is added to the REQUIRED\_FIELDS in the AbstractUser model).  
If you are interested in why the above fields are required and how the superuser creation is implemented, you can dig deeper in this file django/contrib/auth/management/commands/createsuperuser.py.

Once you created a user you can run the  
server(python manage.py runserver) and open the login page(localhost:8000/admin/login).

**Login Page**

What page is rendered at this point is defined in the AdminSite(django/contrib/admin/sites.py). In the login function, the authentication form and the template name are defined to be AdminAuthenticationForm and admin/login.html(django/contrib/admin/templates) respectively. The form and template settings are passed into a LoginView instance which is returned and serves the request at the end. When you fill out the fields and submit the form, the LoginView will take care of the authentication and it will redirect you to the admin landing page after successful login.



Groups and Users links are displayed on the admin page

**User Creation Logic on the Admin Page**

**ModelAdmin**

The models listed after logging in are the models that are registered to the admin site. You can register a model by either calling

admin.site.register(YourModel, YourModelAdmin)

or decorating an admin model with

@admin.register(YourModel)  
class YourModelAdmin(admin.ModelAdmin):  
 pass

If you use the former approach, the YourModelAdmin class is an optional parameter, and if it is omitted, a default will be added by Django. The YourModelAdmin is used to set how you can interact with the model on the admin page, so you can restrict the available fields, override the forms used to create/edit model instances, and set the templates used to display these forms.

When you click on **Add** users, a new page will be loaded that is configured by the UserAdmin(admin.ModelAdmin), which is registered to the default Usermodel. In the UserAdmin, you can find the forms and templates used to render the pages related to the User model.

*Note: The forms used in the UserAdmin model extend the*[*forms.ModelForm*](https://docs.djangoproject.com/en/4.2/topics/forms/modelforms/)*which is a convenience model in Django. It enables automatic form field generation for the model specified in the Meta class of the form class.*

**UserCreationForm**

After filling out the form and submitting it, the UserAdmin calls the save method(defined in ModelAdmin) of the UserCreationForm with the commit=False parameter. This is important because the form will not automatically save the User instance to the DB, but instead, it will hash the raw password and create an object in memory. Once this object is returned to the UserAdmin, the save method of the user object(defined in AbstractBaseUser.save), will be called which stores the user instance to the DB. This flow provides the possibility to customize the user saving mechanism since we can easily override the AbstractBaseUser.save method in the User model.

Now that we understand how users are created in detail, let’s implement our custom user model.👷‍♀️🏗️

**CustomUser(AbstractUser) Implementation**

In this series we will develop an e-learning platform, therefore, I want to make registration as simple as possible so we will use the email/password combination in our user model.

**Create ‘users’ App**

Most of the time you will want to override the AstractUser because many functionality will be provided to you and you will still be able to set your custom fields including replacing the username with an email.

Let’s create a new app called users

python manage.py startapp users

*Note: You could create the folder structure manually, but you will potentially forget to create the migrations folder(or others) and then when we have models, the migration command will not include your app, unless you include the app name in the command. So, just use the command above to make your life easier.*

You can check your structure against this [commit](https://github.com/MateLaszloToth/e_learning/tree/d8fcdcad235e7dc0545e2f0348ec3121f704326f). 📏 ✍️

Then add the users app to the top of the INTALLED\_APPS list.

# e\_learning/settings.py  
...  
INSTALLED\_APPS = [  
 "users", # users app added   
  
 "django.contrib.admin",   
 ...  
]  
...

**Create CustomUser**

Now let’s create our CustomUser model, which uses the email for the USERNAME\_FIELD and it also requires a date of birth.

# users/models.py  
from django.contrib.auth.models import AbstractUser  
from django.db import models  
from django.utils.translation import gettext\_lazy as \_  
  
  
class CustomUser(AbstractUser):  
 """  
 Custom user model which doesn't have a username,   
 but has a unique email and a date\_of\_birth.   
 This model is used for both superusers and   
 regular users as well.  
 """  
 # The inherited field 'username' is nullified, so it will   
 # neither become a DB column nor will it be required.  
 username = None  
 email = models.EmailField(\_("email address"), unique=True)  
 date\_of\_birth = models.DateField(  
 verbose\_name="Birthday",  
 null=True  
 )  
 # Set up the email field as the unique identifier for users.  
 # This has nothing to do with the username  
 # that we nullified above.  
 USERNAME\_FIELD = "email"  
 REQUIRED\_FIELDS = [  
 "first\_name",  
 "last\_name",  
 "date\_of\_birth",  
 ] # The USERNAME\_FIELD aka 'email' cannot be included here  
 objects = CustomUserManager()  
 def \_\_str\_\_(self):  
 return self.email

To better understand the above class, I advise you to open the AbstractUserclass and examine the code. You can see that the username is defined here as a CharField, therefore if we want to remove it from our model, we have to set it to None.

The email field is defined as well, but notice that in the AbstractUser, it can be blank. We want the email to be the unique identifier so we override this property with the unique=True parameter.

The other inherited fields like first\_name, is\_active etc. are quite useful, so we don't touch them in the CustomUser class.

The date of birth of the user is also useful information, so I added that field under the email.

Next up, we have to set the USERNAME\_FIELD="email" because during authentication or form rendering the value of the USERNAME\_FIELD will be used. Django could have called it 'unique identifier' as well because it doesn't have anything to do with the original username field that was assigned to it.

On the following line, we override the REQUIRED\_FIELDS. The fields included here will be required in related forms. Notice that the email and the password fields are not added. This is because the USERNAME\_FIELD and the password are added by default on a deeper layer.

In the next line, we set objects=CustomUserManager() because we will have to override the user creation methods.

**Create CustomUserManager**

Let’s define the CustomUserManager for the CustomUser.

# users/models.py  
from django.contrib.auth.base\_user import BaseUserManager  
from django.contrib.auth.models import AbstractUser  
from django.db import models  
from django.utils.translation import gettext\_lazy as \_  
  
  
class CustomUserManager(BaseUserManager):  
 """  
 Defines how the User(or the model to which attached)  
 will create users and superusers.  
 """  
 def create\_user(  
 self,  
 email,   
 password,  
 date\_of\_birth,  
 \*\*extra\_fields  
 ):  
 """  
 Create and save a user with the given email, password,  
 and date\_of\_birth.  
 """  
 if not email:  
 raise ValueError(\_("The Email must be set"))  
 email = self.normalize\_email(email) # lowercase the domain  
 user = self.model(  
 date\_of\_birth=date\_of\_birth,  
 email=email,  
 \*\*extra\_fields  
 )  
 user.set\_password(password) # hash raw password and set  
 user.save()  
 return user  
 def create\_superuser(  
 self,  
 email,   
 password,  
 date\_of\_birth,  
 \*\*extra\_fields  
 ):  
 """  
 Create and save a superuser with the given email,   
 password, and date\_of\_birth. Extra fields are added  
 to indicate that the user is staff, active, and indeed  
 a superuser.  
 """  
 extra\_fields.setdefault("is\_staff", True)  
 extra\_fields.setdefault("is\_superuser", True)  
 extra\_fields.setdefault("is\_active", True)  
 if extra\_fields.get("is\_staff") is not True:  
 raise ValueError(  
 \_("Superuser must have is\_staff=True.")  
 )  
 if extra\_fields.get("is\_superuser") is not True:  
 raise ValueError(  
 \_("Superuser must have is\_superuser=True.")  
 )  
 return self.create\_user(  
 email,   
 password,   
 date\_of\_birth,   
 \*\*extra\_fields  
 )  
...

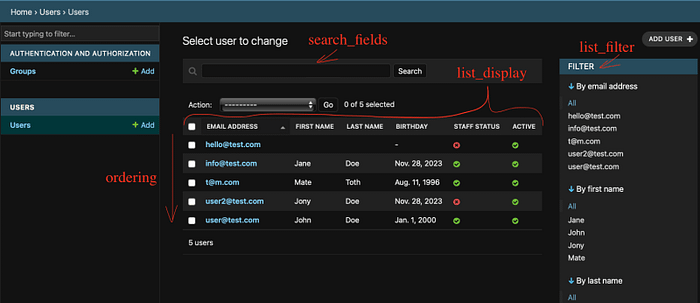
We inherit from the BaseUserManager so that we have the normalize\_email method and other base methods from the Manager class to interact with the DB. We defined additional methods to create users which are used by the createsuperuser command, and we can also use them later in the application. We have to use the CustomUserManager in the CustomUser class because the inherited manager relies on the username field to create a user and we had to replace that with the email field.

**Display User Model on the Admin Page**

Now, let’s define the CustomUserAdmin to be able to customize how the user instances are displayed on the admin site.

# users/admin.py  
from django.contrib import admin  
from django.contrib.auth.admin import UserAdmin  
from users.forms import CustomUserChangeForm, CustomUserCreationForm  
from users.models import CustomUser  
  
  
class CustomUserAdmin(UserAdmin):  
 model = CustomUser  
 list\_display = (  
 "email",  
 "first\_name",  
 "last\_name",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 )  
 list\_filter = (  
 "email",  
 "first\_name",  
 "last\_name",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 )  
 fieldsets = (  
 (None, {"fields": (  
 "first\_name",  
 "last\_name",   
 "email",   
 "password",   
 "date\_of\_birth")}  
 ),  
 ("Permissions", {"fields": (  
 "is\_staff",   
 "is\_active",   
 "groups",   
 "user\_permissions")}  
 ),  
 )  
 add\_fieldsets = (  
 ( None, {"fields": (  
 "first\_name",  
 "last\_name",  
 "email",  
 "password1",  
 "password2",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 "groups",  
 "user\_permissions")}  
 ),  
 )  
 search\_fields = ("email",)  
 ordering = ("email",)

The list\_display, list\_filter, search\_fields, and ordering properties specify what you see on the admin site when you click the 'Users' link.



Users tab on the admin site

The fieldsets and add\_fieldsets properties determine which properties are shown when you open the user details and when you add a user respectively. You can also group the fields like how we did for the 'Permissions'.

You can check the status of the project using this [commit](https://github.com/MateLaszloToth/e_learning/tree/08cecf22f646bd67c1df30c91afc85cd2d539561). 📏 ✍️

**User Creation and Edition Forms**

According to the [documentation](https://docs.djangoproject.com/en/4.2/topics/auth/customizing), we have to override the UserCreationForm and the UserChangeForm if we use a custom user model, however, based on my experience it is not needed if we extend the AbstractUser. In both forms, the Meta class sets the model to be User:

class UserChangeForm(forms.ModelForm):  
 ...  
 class Meta:  
 model = User  
 fields = "\_\_all\_\_" # for UserCreationForm -> fields = ("username",)  
 field\_classes = {"username": UsernameField}

However, when we add a print statement in the form init method,

print("UserChangeForm: ", self.Meta.model)

then the output is UserChangeForm: <class 'users.models.CustomUser'>, so the model is swapped to our custom model in the background.

It is still a good idea to follow the docs, so here is the required implementation.

# users/forms.py  
from django.contrib.auth.forms import UserCreationForm, UserChangeForm  
from users.models import CustomUser  
  
  
class CustomUserCreationForm(UserCreationForm):  
 """  
 Specify the user model created while adding a user  
 on the admin page.  
 """  
 class Meta:  
 model = CustomUser  
 fields = [  
 "first\_name",   
 "last\_name",   
 "email",  
 "password",   
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 "groups",  
 "user\_permissions"  
 ]  
class CustomUserChangeForm(UserChangeForm):  
 """  
 Specify the user model edited while editing a user on the  
 admin page.  
 """  
 class Meta:  
 model = CustomUser  
 fields = [  
 "first\_name",   
 "last\_name",   
 "email",   
 "password",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",   
 "groups",  
 "user\_permissions"  
 ]

Here we override the model and the fields properties in the Meta class. The model is used to render the forms in the UI, while the fields property specifies which model fields are going to be available for the form. You must specify the required fields because otherwise all fields will be rendered automatically. This can cause a security issue when you later add an internal field to the model that you don't want to expose to the user. So, it is highly recommended to explicitly include the allowed fields.

*Note: If you exclude a field in the Meta class, but you include that field in the CustomUserAdmin in one of the properties like fieldsets, then you will receive an error since the admin model expects a field that is not accessible in the model.*

Now let’s tell the admin model to use these forms:

# users/admin.py  
from django.contrib import admin  
from django.contrib.auth.admin import UserAdmin  
from users.forms import CustomUserChangeForm, CustomUserCreationForm  
from users.models import CustomUser  
  
  
class CustomUserAdmin(UserAdmin):  
 add\_form = CustomUserCreationForm  
 form = CustomUserChangeForm  
 ...

Lastly, register your CustomUser model so that it shows up on the admin page, and include the CustomUserAdmin to override the default display.

"""  
Register the 'CustomUser' model to the admin site, so that new  
'CustomUser' model instances can be added to the page.  
The 'CustomUserAdmin' determines which fields and how those fields   
show up in the editor.  
If the 'CustomUserAdmin' is left out, a 'ModelAdmin' is added  
which will include all model fields on the editor page.  
"""  
admin.site.register(CustomUser, CustomUserAdmin)

Now you can make and run migrations

python manage.py makemigrations  
python manage.py migrate

to create the users\_customuser table with the following definition:

create table users\_customuser  
(  
 id integer not null primary key autoincrement,  
 password varchar(128) not null,  
 last\_login datetime,  
 is\_superuser bool not null,  
 first\_name varchar(150) not null,  
 last\_name varchar(150) not null,  
 is\_staff bool not null,  
 is\_active bool not null,  
 date\_joined datetime not null,  
 email varchar(254) not null unique,  
 date\_of\_birth date  
);

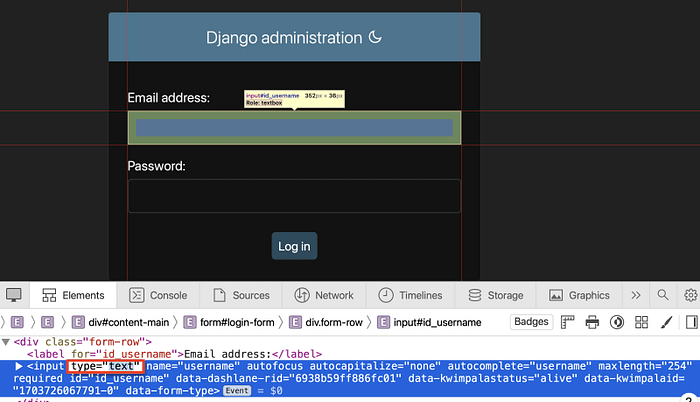
You can check out the project at this stage with this git [commit](https://github.com/MateLaszloToth/e_learning/tree/59916f9e2f9b73e33bd247a239cccf5ae68071cd). 📏 ✍️

**Customizing the Login Page**

Since we already come this far, let’s cover how we can customize the admin login page. The code required for our use case is minimal, however, understanding how it works is a bit more involved.

In short, the AdminSite has a login method that handles the request to the localhost:8000/admin/login path. By default, the built-in AdminAuthenticationForm and the django/contrib/admin/templates/admin/login.html template are used to render the page. The AdminAuthenticationForm specifies what fields are available for the template, while the template specifies how those fields are shown to the user.

The AdminAuthenticationForm inherits the actual field definitions from its parent class, AuthenticationForm. If you open the parent class, you will notice that a username however in the init method that field is overridden by the USERNAME\_FIELD of our custom user model. Thus, the usernameproperty will hold our email field. Therefore, it shouldn't be surprising that if you open the login.html you will see form.username, which will render our email field as a text input.



Email input has text type.

To enable client-side validation and increase the accessibility of the email field, we should render the input field with the email type.

We can create a CustomAdminAuthenticationForm that addresses this issue:

class CustomAdminAuthenticationForm(AuthenticationForm):  
 username = UsernameField(widget=forms.EmailInput(attrs={"autofocus": True}))

Notice that the widget is using EmailInput instead of TextInput which will solve our rendering problem.

Next up let’s create our template, login.html, in the users/template/authenticate folder. We can place here a copy of the original file(django/contrib/admin/templates/admin/login.html). The only thing that we have to update is the path to the CSS file.

<-- users/templates/authentication/login.html -->  
...  
<link rel="stylesheet" href="{% static "css/login.css" %}">  
...

Since we are using the static tag, Django will look into the folder holding static code, which is called 'static' by default. Let's put a copy of the original CSS file(django/contrib/admin/static/admin/css/login.css) into users/static/css/login.html. (You will have to create this folder structure)

Now let’s put the pieces together in the settings.py file and update the URLs.

# settings.py  
...  
urlpatterns = [  
 path('admin/login/', LoginView.as\_view(  
 authentication\_form=CustomAdminAuthenticationForm,  
 template\_name="authentication/login.html"),  
 name='admin\_login'), # this path was added  
 path("admin/", admin.site.urls),  
]

We added a new binding to the path admin/login/, which overrides the Django implementation. The LoginView is a built-in view that will display the login form and handle the login action. At this point, we can specify to use our own CustomAdminAuthenticationForm, and pass in the path to our template.

*Note: The default template path is registration/login.html, therefore, if we renamed authentication to registration we wouldn't need to pass in the template path. However, I want to demonstrate how we can specify a template in a custom location.*

You can view the project at this status using this [commit](https://github.com/MateLaszloToth/e_learning/tree/5abc193674a5cc10dfab079a34678fcb28e27d23). 📏 ✍️

**CustomUser(AbstractBaseUser) Implementation**

If you prefer a bare-bone user model, then you might prefer replacing the AbstractUser and inheriting straight from the AbstractBaseUser. In this case, everything that we discussed in the previous section still holds. The only difference is the implementation of the CustomUserModel and if the user fields are different, then we also have to update the references.

# users/models.py  
from django.contrib.auth.base\_user import BaseUserManager, AbstractBaseUser  
from django.contrib.auth.models import PermissionsMixin  
from django.core.mail import send\_mail  
from django.db import models  
from django.utils import timezone  
from django.utils.translation import gettext\_lazy as \_  
  
  
...  
class CustomUser(AbstractBaseUser, PermissionsMixin):  
 """  
 Custom user model with unique email as the user  
 identifier. This model is used for both superusers and  
 regular users as well.  
 """  
 # The inherited field 'username' is nullified, so it will  
 # neither become a DB column nor will it be required.  
 email = models.EmailField(\_("email address"), unique=True)  
 is\_staff = models.BooleanField(default=False)  
 is\_active = models.BooleanField(default=True)  
 date\_joined = models.DateTimeField(default=timezone.now)  
 date\_of\_birth = models.DateField(  
 verbose\_name="Birthday",  
 null=True  
 )  
 # Set up the email field as the unique identifier for users.  
 # This has nothing to do with the username that we nullified.  
 USERNAME\_FIELD = "email"  
 # The USERNAME\_FIELD aka 'email' cannot be included here  
 REQUIRED\_FIELDS = ["date\_of\_birth"]   
 objects = CustomUserManager()  
 def \_\_str\_\_(self):  
 return self.email  
 class Meta:  
 verbose\_name = \_("user")  
 verbose\_name\_plural = \_("users")  
 abstract = False  
 def clean(self):  
 super().clean()  
 self.email =   
 self.\_\_class\_\_.objects.normalize\_email(self.email)  
 def get\_date\_of\_birth(self): # This method is optional  
 return self.date\_of\_birth  
 def email\_user(  
 self,  
 subject,   
 message,  
 from\_email=None,   
 \*\*kwargs):  
 """Email this user."""  
 send\_mail(  
 subject,  
 message,   
 from\_email,   
 [self.email],   
 \*\*kwargs  
 )

Notice that now we have to inherit from the PermissionsMixin class as well to preserve Django's permissions functionality.

Regarding the fields, we have to keep the is\_staff, and is\_active fields to preserve functionality. I removed the first and last name as well as the username fields since we don't need this information anymore, however, I kept the date of birth and the email fields.

As you can see the USERNAME\_FIELD is untouched, but I had to update the REQUIRED\_FIELDS.

The Meta class, clean, and send\_email methods are copied from the AbstractUser to keep functionality.

Now let’s clean up the references to the first and last names from the CustomUserAdmin:

# users/admin.py  
class CustomUserAdmin(UserAdmin):  
 add\_form = CustomUserCreationForm  
 form = CustomUserChangeForm  
 list\_display = (  
 "email",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 )  
 list\_filter = (  
 "email",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 )  
 fieldsets = (  
 (None, {"fields": (  
 "email",   
 "password",   
 "date\_of\_birth")}  
 ),  
 ("Permissions", {"fields": (  
 "is\_staff",   
 "is\_active",  
 "groups",   
 "user\_permissions")}  
 ),  
 )  
 add\_fieldsets = (  
 (None, {"fields": (  
 "email",  
 "password1",  
 "password2",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 "groups",  
 "user\_permissions")}  
 ),  
 )  
 search\_fields = ("email",)  
 ordering = ("email",)  
...

and from the CustomUserCreationForm and CustomUserChangeForm

# users/forms.py  
...  
class CustomUserCreationForm(UserCreationForm):  
 """  
 Specify the user model/form to be used while  
 adding a user on the admin page.  
 """  
 class Meta:  
 model = CustomUser  
 fields = [  
 "email",  
 "password",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 "groups",  
 "user\_permissions",  
 ]  
  
class CustomUserChangeForm(UserChangeForm):  
 """  
 Specify the user model/form to be used while editing  
 a user on the admin page.  
 """  
 class Meta:  
 model = CustomUser  
 fields = [  
 "email",  
 "password",  
 "date\_of\_birth",  
 "is\_staff",  
 "is\_active",  
 "groups",  
 "user\_permissions",  
 ]

You can view the project at this status using this [commit](https://github.com/MateLaszloToth/e_learning/commit/546f301c3a1896d4f8151b78b1f31fed7c15135d). 📏 ✍️

And that’s it! Now pat yourself on the back! 🎉 🔥

I hope you enjoyed the article and that you now understand the intricacies of the user model and the related topics. If you have any questions or suggestions, leave a comment below or contact me through [LinkedIn](https://www.linkedin.com/in/m%C3%A1t%C3%A9-t%C3%B3th-696834183/). If you liked the content, don’t forget to share it with others. If you would like to fuel my next study sessions, then you are welcome to [sponsor my next bubble tea](https://www.buymeacoffee.com/mate_toth)😁🧋

[Django](https://medium.com/tag/django?source=post_page-----98b9a401a6a2---------------django-----------------)

[Python](https://medium.com/tag/python?source=post_page-----98b9a401a6a2---------------python-----------------)

[User Model](https://medium.com/tag/user-model?source=post_page-----98b9a401a6a2---------------user_model-----------------)

[Tutorial](https://medium.com/tag/tutorial?source=post_page-----98b9a401a6a2---------------tutorial-----------------)

[Beginner](https://medium.com/tag/beginner?source=post_page-----98b9a401a6a2---------------beginner-----------------)