

Here's a clear overview of Django's **User model** and how it works:

🔑 The Django User Model

Django provides a built-in authentication system in `django.contrib.auth`, and at its core is the **User model**. It represents users of your app (e.g., customers, admins, staff).

1. Default User Model (`django.contrib.auth.models.User`)

The default model includes common fields for authentication:

- **Username & Password**
 - `username` : unique identifier (by default).
 - `password` : hashed automatically using Django's password hashing system.
 - **Personal Info**
 - `first_name`
 - `last_name`
 - `email`
 - **Permissions**
 - `is_active` : if `False`, the user cannot log in.
 - `is_staff` : gives access to Django admin.
 - `is_superuser` : bypasses all permission checks.
 - `groups` : many-to-many to `Group` (role-based permissions).
 - `user_permissions` : fine-grained control over access.
 - **Timestamps**
 - `last_login`
 - `date_joined`
-

2. Authentication System

- Uses `authenticate()` and `login()` functions.

- Passwords are stored using PBKDF2 (by default).
 - Provides session-based authentication.
-

3. Customizing the User Model

Django allows flexibility when working with users:

a) Extending via One-to-One

- Keep default `User` and attach a `Profile` model with extra fields.
- Quick and simple, but requires two tables.

b) AbstractUser

- Subclass of the default User.
- Lets you add/modify fields without redefining the whole model.
- Example:

```
python

from django.contrib.auth.models import AbstractUser

class CustomUser(AbstractUser):
    age = models.PositiveIntegerField(null=True, blank=True)
```

c) AbstractBaseUser

- Fully custom User model.
- Gives maximum control (e.g., use `email` instead of `username`).
- Requires defining required fields and authentication logic.
- Example:

```
python

from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin
from django.contrib.auth.base_user import BaseUserManager
from django.db import models

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("Email required")
        email = self.normalize_email(email)
```

```

user = self.model(email=email, **extra_fields)
user.set_password(password)
user.save()
return user

def create_superuser(self, email, password=None, **extra_fields):
    extra_fields.setdefault('is_staff', True)
    extra_fields.setdefault('is_superuser', True)
    return self.create_user(email, password, **extra_fields)

class CustomUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    is_staff = models.BooleanField(default=False)
    is_active = models.BooleanField(default=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    objects = CustomUserManager()

```

4. Best Practices

- ✓ Decide early if you need a **custom User model** (changing later is painful).
- ✓ Use `AbstractUser` if you just need extra fields.
- ✓ Use `AbstractBaseUser` if you need to redefine login credentials (e.g., login with email).
- ✓ Always reference users via `settings.AUTH_USER_MODEL`, not directly to `User`.

👉 Do you want me to create a **comparison table** between `AbstractUser` and `AbstractBaseUser` to help decide which one to use?

what is abstraction in python indetailed and then relate this with abstract user and then tell me how do you change the default login from username to email and also how do you link one to one connection between

the user model and a custom model for our convenience.

Great set of questions 🚀 Let's take them step by step — starting from the **concept of abstraction in Python**, then connecting it to Django's **AbstractUser**, then showing how to switch login from **username** → **email**, and finally explaining the **one-to-one User profile extension**.

1. Abstraction in Python

Abstraction is an **OOP principle** where you expose only the **necessary details** of an object and hide the implementation details.

Example with Python's `abc` module:

```
python

from abc import ABC, abstractmethod

class Vehicle(ABC): # Abstract class
    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle): # Concrete class
    def start_engine(self):
        print("Car engine started")

class Bike(Vehicle):
    def start_engine(self):
        print("Bike engine started")

# Usage
v = Car()
v.start_engine() # Car engine started
```

🔑 Key Points

- Abstract class: cannot be instantiated directly.
- Abstract methods: must be implemented by child classes.
- Helps enforce a **contract** between classes while allowing flexibility.

2. How This Relates to `AbstractUser` in Django

- Django's `AbstractUser` is an abstract base class that provides ready-made fields like `username`, `email`, `first_name`, etc.
- You cannot instantiate `AbstractUser` directly. Instead, you extend it to create your own `CustomUser`.
- Django enforces that if you subclass `AbstractUser`, you respect the "contract" (e.g., certain authentication methods remain intact).

So `AbstractUser` = Django's way of saying:

"Here's a skeleton user model with all the basics, you can extend it, but don't use it directly."

3. Changing Default Login from Username → Email

If you want users to log in with email instead of username, you should use `AbstractBaseUser` or tweak `AbstractUser`.

✓ Example: Using `AbstractBaseUser`

python

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin,
BaseUserManager

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("Email must be provided")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save()
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)
```

```
class CustomUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    USERNAME_FIELD = "email"    # 👉 tells Django to use email for login
    REQUIRED_FIELDS = []        # Fields required when creating superuser

    objects = CustomUserManager()
```

Update `settings.py`

```
python

AUTH_USER_MODEL = "myapp.CustomUser"
```

Now, login uses **email + password** instead of username.

4. Linking One-to-One Connection (Profile Model)

If you want to **keep the default User** but add extra fields (like phone, address, etc.), you can use a **One-to-One** relationship.

Example: User Profile

```
python

from django.db import models
from django.conf import settings

class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    phone_number = models.CharField(max_length=15, blank=True, null=True)
    address = models.TextField(blank=True, null=True)

    def __str__(self):
        return f"{self.user.username}'s Profile"
```

Automatically Create Profile When User is Created

python

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.conf import settings
from .models import Profile

@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_or_update_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
    else:
        instance.profile.save()
```

Now, whenever a user is created, a profile is automatically attached.

✓ Summary

- **Abstraction** in Python = hiding implementation details while exposing essential features.
- **AbstractUser** = Django's abstract base class for user models (ready-made fields, extendable).
- **Login with email** = create a `CustomUser` using `AbstractBaseUser` and set `USERNAME_FIELD = "email"`.
- **One-to-one profile model** = extend User with a `Profile` model to store extra info (best when you just need more fields without replacing User).

👉 Would you like me to draw a **decision flowchart** showing **when to use AbstractUser vs AbstractBaseUser vs One-to-One profile**?

please tell me normal way of authentication with file snapshots step by step by reasoning from first principles and then another example with a custom user where authentication happens via email and keep it simple and step by step by placing comments at

required fields and assume i need more explanation and keep comments explicitly every where if it is not too basic

Nice — I'll do this in two parts:

1. a short **first-principles** reasoning of what authentication *is* and what it must do, and then
2. two practical, **file-by-file snapshots** you can copy & paste:
 - **Example A:** the “normal” (default) Django authentication using `username`.
 - **Example B:** a **custom user** that authenticates with **email** (plus a one-to-one `Profile` example).

I'll keep the code simple and put **explicit comments** where the behavior or fields matter. Let's go.

1) Authentication — reasoning from first principles (short & concrete)

- **Goal:** reliably map a *human* (identity) to a *record* (user row) and prove the human is who they claim to be (credentials).
- **Parts required:**
 1. **Identity** — a stable identifier (username or email).
 2. **Credential** — secret material to prove identity (password). Must be **hashed + salted** in storage.
 3. **Verifier** — code that checks the credential (e.g., `User.check_password()` / `authenticate()`).
 4. **Session / token** — a mechanism to remember the authenticated user across requests (Django uses session cookie).
 5. **Authorization** — once authenticated, apply permissions/roles (staff, superuser, groups).
 6. **Safety controls** — rate limiting, account lockouts, password hashing algorithms, secure cookie settings, HTTPS.
- **Django implements** these pieces with:
 - `User` model (identity + metadata),
 - hashed password storage (set via `set_password()`),

- `authenticate()` and `login()` (verifier + session creation),
- middlewares (`SessionMiddleware`, `AuthenticationMiddleware`),
- decorators (`@login_required`) and permission checks (`user.is_staff`, `user.has_perm()`).

2) Example A — Default Django authentication (username + password)

Project: `myproject`, app: `accounts`

Project structure (minimal)

```
arduino

myproject/
  manage.py
  myproject/
    settings.py      # snippet shown below
    urls.py          # snippet shown below
  accounts/
    views.py         # login/logout/protected views
    urls.py
  templates/
    accounts/
      login.html
      dashboard.html
```

`myproject/settings.py` (relevant parts)

```
py

# settings.py (only relevant lines shown)
INSTALLED_APPS = [
    # default apps required for auth + admin + sessions
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions', # session storage (needed for login)
```

```

'django.contrib.messages',
'django.contrib.staticfiles',
'accounts',          # our app
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware', # manages session cookie
    'django.middleware.common.CommonMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware', # populates request.user
    'django.contrib.messages.middleware.MessageMiddleware',
    # ...
]

# Where templates live (so login template is found)
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [], # project-level template dirs (optional)
        'APP_DIRS': True, # allows templates in app/templates/
        ...
    },
]

# After login, where to send user by default
LOGIN_REDIRECT_URL = 'dashboard' # name of the urlpattern to redirect to after login
LOGIN_URL = 'login' # used by @login_required to redirect to login page

```

myproject/urls.py

```

py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path("", include('accounts.urls')), # include our app's URLs
]

```

accounts/urls.py

py

```
from django.urls import path
from . import views

urlpatterns = [
    path('login/', views.login_view, name='login'),    # login page
    path('logout/', views.logout_view, name='logout'), # logout
    path('dashboard/', views.dashboard, name='dashboard'), # protected page
]
```

accounts/views.py (explicit login/logout)

py

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.decorators import login_required

def login_view(request):
    """
    Simple login view using username + password.
    - On POST: read username/password, call authenticate(), then login()
    - On GET: render login form
    """
    if request.method == 'POST':
        # get values from submitted form (names must match the HTML below)
        username = request.POST.get('username') # username string entered by user
        password = request.POST.get('password') # password string entered by user

        # authenticate: checks username & password. Returns User instance or None
        user = authenticate(request, username=username, password=password)

        if user is not None:
            # login: attach user id to the session (session cookie created/updated)
            login(request, user)
            # redirect to a protected page, or use `next` param in real apps
            return redirect('dashboard')
        else:
```

```

        # invalid credentials: show simple error
        return render(request, 'accounts/login.html', {'error': 'Invalid credentials'})

    # GET request: show login form
    return render(request, 'accounts/login.html')

def logout_view(request):
    """
    Log out the current user by clearing the session.
    """
    logout(request)
    return redirect('login')

@login_required # ensures only logged in users can access dashboard
def dashboard(request):
    # request.user is a User instance (populated by AuthenticationMiddleware)
    return render(request, 'accounts/dashboard.html', {'user': request.user})

```

accounts/templates/accounts/login.html

```

html

<!-- simple login template -->
<!doctype html>
<html>
<head><title>Login</title></head>
<body>
    <h1>Login</h1>

    {% if error %}<p style="color:red">{{ error }}</p>{% endif %}

    <form method="post">
        {% csrf_token %}
        <!-- names MUST match what view expects -->
        <label>Username: <input name="username" type="text" required></label><br>
        <label>Password: <input name="password" type="password" required></label><br>
        <button type="submit">Login</button>
    </form>
</body>
</html>

```

```
accounts/templates/accounts/dashboard.html
```

```
html
```

```
<!doctype html>
<html>
<head><title>Dashboard</title></head>
<body>
  <h1>Welcome, {{ user.username }}</h1>
  <p>Your email is: {{ user.email }}</p>
  <p><a href="{% url 'logout' %}">Logout</a></p>
</body>
</html>
```

Shell example: create a user (programmatic)

```
bash
```

```
python manage.py shell
```

```
py
```

```
from django.contrib.auth.models import User

# create_user hashes the password properly (calls set_password under the hood)
u = User.objects.create_user(username='alice', email='alice@example.com',
password='s3cr3t')
# test authenticate
from django.contrib.auth import authenticate
authenticate(username='alice', password='s3cr3t') # returns the User if credentials ok
```

Key Notes for Example A (default):

- `authenticate()` checks credentials; `login()` creates the session cookie.
- Passwords must be set via `User.objects.create_user()` or `user.set_password()` — never store plain text.

- Make sure `SessionMiddleware` and `AuthenticationMiddleware` are enabled (they are by default in new projects).
- Use `@login_required` to protect views.

3) Example B — Custom user model authenticating via email + one-to-one `Profile`

Important: set `AUTH_USER_MODEL` *before* the first `migrate` (or your project will need complex migration surgery). If you already ran migrations with the default `User`, converting later is harder.

Project structure (new app `accounts`):

```
bash

myproject/
  manage.py
  myproject/
    settings.py # add AUTH_USER_MODEL = 'accounts.CustomUser'
  accounts/
    models.py # CustomUser, CustomUserManager, Profile model
    admin.py # admin integration
    forms.py # optional admin forms
    authentication_backends.py # optional email auth backend
    views.py # login/logout/dashboard
    urls.py
    templates/accounts/login_email.html
```

`myproject/settings.py` (relevant lines for custom user)

```
py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'accounts', # our app containing CustomUser
```

```
# ...
]

# Must be set BEFORE running first migrate for the project:
AUTH_USER_MODEL = 'accounts.CustomUser' # points Django to our custom user model

MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    # ...
]

# Optionally: use a custom authentication backend that authenticates by email
AUTHENTICATION_BACKENDS = [
    'accounts.authentication_backends.EmailBackend', # our custom backend (preferred)
    'django.contrib.auth.backends.ModelBackend', # fallback
]
```

accounts/models.py (CustomUser + manager + Profile)

```
py

from django.db import models
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin,
BaseUserManager
from django.utils import timezone
from django.conf import settings

class CustomUserManager(BaseUserManager):
    """
    Manager for CustomUser.
    Always provide create_user and create_superuser.
    """

    def create_user(self, email, password=None, **extra_fields):
        # email is the unique identity for our custom user
        if not email:
            raise ValueError('The Email field must be set')
        email = self.normalize_email(email) # lowercases domain part
        user = self.model(email=email, **extra_fields) # create user instance
        user.set_password(password) # hashes the password
        user.save(using=self._db)
```

```

return user

def create_superuser(self, email, password=None, **extra_fields):
    # ensure superusers have staff/superuser privileges
    extra_fields.setdefault('is_staff', True)
    extra_fields.setdefault('is_superuser', 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, **extra_fields)

class CustomUser(AbstractBaseUser, PermissionsMixin):
    """
    Minimal custom user where `email` is the unique identifier.
    AbstractBaseUser gives password handling and last_login field.
    PermissionsMixin adds groups/permissions & is_superuser.
    """
    email = models.EmailField(unique=True) # unique identity
    first_name = models.CharField(max_length=30, blank=True) # optional
    last_name = models.CharField(max_length=30, blank=True) # optional
    is_staff = models.BooleanField(default=False) # admin site access
    is_active = models.BooleanField(default=True) # can login
    date_joined = models.DateTimeField(default=timezone.now) # when created

    # important: this tells Django which field is used as the identifier
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = [] # e.g., ['first_name'] would prompt when creating superuser

    objects = CustomUserManager() # link manager

    def __str__(self):
        return self.email

# Example: one-to-one Profile model linked to the AUTH_USER_MODEL
class Profile(models.Model):
    # Always refer to the user model via settings.AUTH_USER_MODEL (works for custom
    # models)
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    # extra fields

```



```
phone = models.CharField(max_length=20, blank=True, null=True)
bio = models.TextField(blank=True, null=True)
```

```
def __str__(self):
    return f'Profile of {self.user}'
```

accounts/authentication_backends.py (simple email backend)

This backend searches user by email (case-insensitive) and checks password. You can skip this if you prefer to call `authenticate(username=email, password=...)` because `ModelBackend` uses `USERNAME_FIELD`. But having a clear `EmailBackend` is explicit.

py

```
from django.contrib.auth.backends import ModelBackend
from django.contrib.auth import get_user_model

class EmailBackend(ModelBackend):
    """
    Authenticate using email (case-insensitive).
    ModelBackend expects a username arg; we accept username or email keyword.
    """

    def authenticate(self, request, username=None, password=None, **kwargs):
        UserModel = get_user_model()

        # username may be supplied in the "username" arg or in kwargs['email']
        email = username or kwargs.get('email')
        if email is None or password is None:
            return None

        try:
            user = UserModel.objects.get(email__iexact=email)
        except UserModel.DoesNotExist:
            # No user found with this email
            return None

        # check password and whether the user can authenticate (is_active)
        if user.check_password(password) and self.user_can_authenticate(user):
```

```
return user
return None
```

accounts/forms.py (simple login form for email)

```
py

from django import forms

class EmailAuthenticationForm(forms.Form):
    # explicit form so template names and field names are clear
    email = forms.EmailField(label='Email', max_length=254, widget=forms.EmailInput(attrs=
    {'autofocus': True}))
    password = forms.CharField(label='Password', strip=False, widget=forms.PasswordInput)
```

accounts/views.py (login using email)

```
py

from django.shortcuts import render, redirect
from django.contrib.auth import login, logout, authenticate
from django.contrib.auth.decorators import login_required
from .forms import EmailAuthenticationForm

def email_login_view(request):
    """
    Login view that authenticates using email + password.
    If you use the EmailBackend, `authenticate()` will accept username=email or email=...
    """
    if request.method == 'POST':
        form = EmailAuthenticationForm(request.POST)
        if form.is_valid():
            email = form.cleaned_data['email'] # email provided by user
            password = form.cleaned_data['password'] # password provided by user

            # Option A: If you registered EmailBackend, do:
            user = authenticate(request, email=email, password=password)

            # Option B (if you prefer using ModelBackend and USERNAME_FIELD='email'):
```

```

        # user = authenticate(request, username=email, password=password)

        if user is not None:
            login(request, user) # creates session
            return redirect('dashboard')
        else:
            form.add_error(None, 'Invalid credentials')
    else:
        form = EmailAuthenticationForm()

    return render(request, 'accounts/login_email.html', {'form': form})

def logout_view(request):
    logout(request)
    return redirect('email_login')

@login_required
def dashboard(request):
    return render(request, 'accounts/dashboard.html', {'user': request.user})

```

accounts/admin.py (make the custom user manageable in admin)

```

py

from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as DjangoUserAdmin
from django import forms
from django.contrib.auth.forms import ReadOnlyPasswordHashField
from .models import CustomUser, Profile

# -- Forms used to create/change users in admin --

class CustomUserCreationForm(forms.ModelForm):
    # Form for admin "Add user" page: asks for email + password twice
    password1 = forms.CharField(label='Password', widget=forms.PasswordInput)
    password2 = forms.CharField(label='Password confirmation',
                                widget=forms.PasswordInput)

    class Meta:
        model = CustomUser
        fields = ('email',)

```

```

def clean_password2(self):
    # ensure both passwords match
    p1 = self.cleaned_data.get('password1')
    p2 = self.cleaned_data.get('password2')
    if p1 and p2 and p1 != p2:
        raise forms.ValidationError("Passwords don't match")
    return p2

def save(self, commit=True):
    # set_password will hash the password
    user = super().save(commit=False)
    user.set_password(self.cleaned_data['password1'])
    if commit:
        user.save()
    return user

class CustomUserChangeForm(forms.ModelForm):
    # For admin "change user" page - shows hashed password as read-only
    password = ReadOnlyPasswordHashField()

    class Meta:
        model = CustomUser
        fields = ('email', 'password', 'is_active', 'is_staff')

    def clean_password(self):
        # return the initial value regardless of what the user provides
        return self.initial['password']

# -- Admin registration --
class CustomUserAdmin(DjangoUserAdmin):
    form = CustomUserChangeForm
    add_form = CustomUserCreationForm

    # fields to show in list view
    list_display = ('email', 'is_staff', 'is_superuser')
    list_filter = ('is_staff', 'is_superuser')
    ordering = ('email',)
    search_fields = ('email',)

    # fields shown when editing a user

```

```

fieldsets = (
    (None, {'fields': ('email', 'password')}),
    ('Personal info', {'fields': ('first_name', 'last_name')}),
    ('Permissions', {'fields': ('is_active', 'is_staff', 'is_superuser', 'groups',
'user_permissions')}),
    ('Important dates', {'fields': ('last_login',)}),
)

# fields shown on the "add user" page
add_fieldsets = (
    (None, {
        'classes': ('wide',),
        'fields': ('email', 'password1', 'password2', 'is_staff', 'is_superuser'),
    }),
)

admin.site.register(CustomUser, CustomUserAdmin)
admin.site.register(Profile) # register profile for convenience

```

accounts/templates/accounts/login_email.html

```

html

<!doctype html>
<html>
<head><title>Login with Email</title></head>
<body>
<h1>Login (email)</h1>
<form method="post">{% csrf_token %}
  {{ form.non_field_errors }}
  <p>
    {{ form.email.label_tag }}<br>
    {{ form.email }}
    {{ form.email.errors }}
  </p>
  <p>
    {{ form.password.label_tag }}<br>
    {{ form.password }}
    {{ form.password.errors }}
  </p>
  <button type="submit">Log in</button>

```

```
</form>
</body>
</html>
```

Creating migrations & a superuser (commands)

```
bash

# after adding AUTH_USER_MODEL and creating the models
python manage.py makemigrations accounts
python manage.py migrate

# create a superuser (you will be prompted for email & password)
python manage.py createsuperuser --email admin@example.com
# or simply `python manage.py createsuperuser` and it will prompt for the fields,
# using USERNAME_FIELD (email) when requested.
```

Programmatically creating users with the custom manager

```
py

# from manage.py shell
from accounts.models import CustomUser
u = CustomUser.objects.create_user(email='joe@example.com', password='strongpass')
# check authentication:
from django.contrib.auth import authenticate
authenticate(email='joe@example.com', password='strongpass') # if using EmailBackend
# or
authenticate(username='joe@example.com', password='strongpass') # if relying on
ModelBackend
```

Quick checklist & gotchas (practical tips)

- Set `AUTH_USER_MODEL` before the first migrate. Changing it after migrations is painful.

- **Always use** `settings.AUTH_USER_MODEL` when referencing the user model in `ForeignKey` / `OneToOneField`:

py

```
user = models.OneToOneField(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE)
```

- **When creating users programmatically**, always use `create_user()` or call `set_password()` — *never* assign raw password.
- **Admin integration**: provide custom `UserCreationForm` and `UserChangeForm` or the admin will show wrong fields.
- **Authentication backend**: if you want explicit email auth, register a custom backend and add it to `AUTHENTICATION_BACKENDS`. Alternatively, with `USERNAME_FIELD='email'`, the default `ModelBackend` will work if you pass `username=email` into `authenticate()`.
- **Session & auth middleware** must be enabled (they are by default).
- **Templates & field names**: if your login view expects `email` then your form must post `email` (names must match).

If you'd like I can:

- provide a **single ZIP** of these files ready to paste into a project (I can produce code blocks for every file), or
- show a **minimal complete repo** that you can run locally with `python manage.py runserver`, or
- convert the admin forms + login page to use Django's **class-based** `LoginView` wired to `AuthenticationForm` for the custom user.

Which of those would help you most next?

1/2