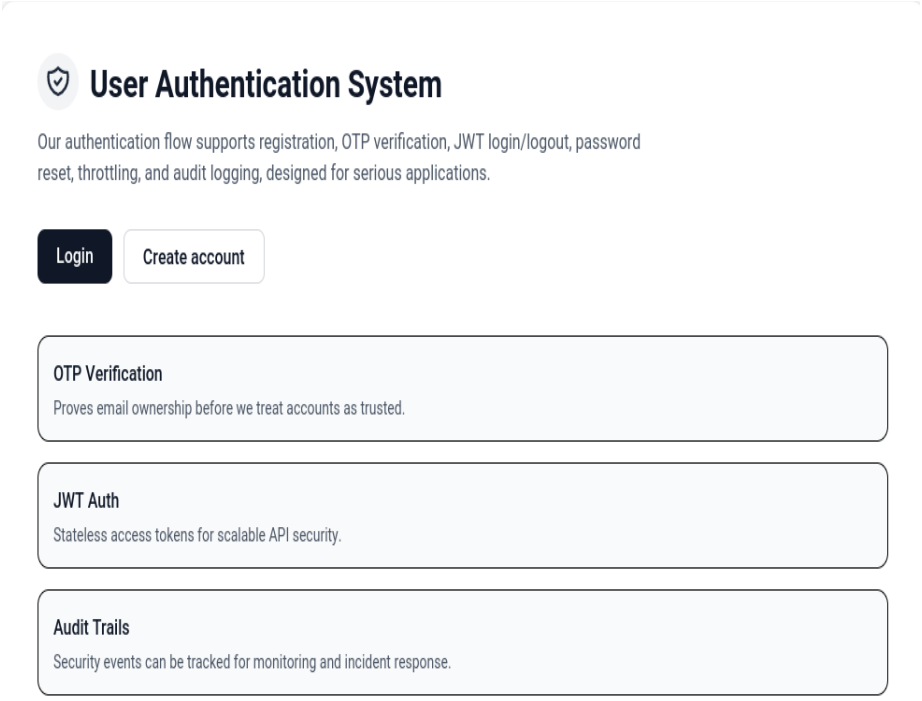
Code by Amin: A Deep Technical Dive into Building a Secure Authentication System with Django, DRF, JWT, and React.

Version 1: A production-style baseline with room for advanced hardening in future editions.



Amin Hydar Ali

Table of Contents

How to Run the Project....9
What needs to be installed....9
1. Clone the project....9
2. Create and activate a virtual environment....9
3. Install backend dependencies....9
4. Create the .env file....10
5. Check settings.py....10
6. Apply database migrations....10
7. Create an admin user....10
8. Start Redis....10
9. Start the Django backend....11
10. Start the Celery worker....11
11. Start the frontend....11
12. Recommended run order....11
How to confirm everything is working....12
Backend checks....12
Email checks....12
Common issues and how to fix them....13
1. OTP or reset email not sending....13
2. Password reset link opens backend instead of frontend....13
3. Login always fails even with correct email and password....13
4. JWT logout seems incomplete....13
5. GeoIP lookup never works....13
6. Frontend cannot reach backend....13
.gitignore....14
What it should include....14
Most important files to never commit....15
Technologies Used....18
A. Django - the foundation and security guardrails....18
B. Django REST Framework - turning Django into a clean auth API....19
C. JWT / SimpleJWT - portable identity tokens for massive scale....19
D. Celery + (typically) Redis - background workers for email and heavy tasks....20
E. GeoIP - adding location context to logins....21
E. zxcvbn - realistic password strength, not just length checks....21
Putting it all together....22
Deep Dive into the Authentication System....22
File Structure....22
Models....24
1. What is a “model” in Django?....24
2. Why do we need models?....24
3. Models in an authentication system (our case)....25
4. How models fit into the bigger picture....25
model/models.py....26
Code Break: Step by Step....29

Model 2: SuspiciousActivity - higher-level alarms, not raw events....35
IP + indexing....35
Reason + indexing....35
Details JSON....36
Timestamp + index....36
Meta indexes....36
String representation....37
models/user_model.py....37
Code Explanation: Step by Step....44
from datetime import timedelta....44
from django.db import models....44
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin, Group, Permission....44
from django.utils import timezone....44
from django.utils.translation import gettext_lazy as _....44
from django.contrib.auth.hashers import make_password, check_password....44
from django.core.exceptions import ValidationError....45
from rest_framework_simplejwt.tokens import RefreshToken....45
from accounts.managers.user_manager import CustomUserManager....45
from accounts.services.email_services import send_otp_email....45
1. class User(AbstractBaseUser, PermissionsMixin):....45
Email....45
Names....45
Username....46
OTP fields....46
Status flags....46
Groups and permissions....46
Timestamps and country....46
Auth config....46
Meta + indexes....47
Methods....47
tokens....47
generate_and_save_otp....47
verify_otp....47
can_resend_otp + resend_otp....47
managers/user_managers.py....48
Code Example: Step by Step....52
Serializers....56
What a serializer is, in plain language....56
Why serializers were created....56
Why we use serializers with Django....57
The “technology behind it” (what DRF is doing under the hood)....58
Where serializers fit into our authentication system....59
serializers/registration_serializers.py....60
Code Explanation: Step by Step....63
Serializer class....63
Fields: why we define these explicitly....64
Meta section....64
validate_email....65

validate_password..... 65
validate (cross-field validation)..... 66
create..... 66
serializers/otp_serializer.py..... 67
Code Explanation: Step by Step..... 72
..... 73
ResendOTPSerializer..... 73
VerifyOTPSerializer..... 75
class VerifyOTPSerializer(serializers.Serializer):..... 75
auth_serializer.py..... 77
UserLoginSerializer..... 79
validate() logic..... 80
UserLogoutSerializer..... 80
password_reset_serializer.py..... 81
Code Explanation: Step by step..... 86
PasswordResetRequestSerializer..... 87
Anti-enumeration design..... 88
SetNewPasswordSerializer..... 88
Confirm password match..... 88
Decode uidb64..... 88
Validate reset token..... 88
Strength check..... 89
ChangePasswordSerializer..... 89
Views..... 90
What a “view” is in Django/DRF..... 90
Why views exist (why we don’t just put everything in models)..... 90
What views do in our auth system (high-level)..... 91
1) Accept and parse requests..... 91
2) Select the correct serializer..... 91
3) Enforce permissions..... 91
4) Throttle abuse (rate limiting)..... 92
5) Log audit events..... 92
6) Return clean, consistent responses..... 92
Why we use DRF GenericAPIView (and what it gives us)..... 92
registration_view.py..... 93
Step-by-step explanation..... 96
Throttle class..... 98
The view class..... 98
The post() method..... 99
Validation block..... 99
User creation + OTP block..... 100
Success response..... 102
otp_view.py..... 103
1. import logging..... 103
Code Explanation: Step by Step..... 107
get_client_ip()..... 107
Throttles: why AnonRateThrottle..... 107
VerifyOTPView success path..... 108
VerifyOTPView failure path..... 108

ResendOTPView success path....108
auth_view.py....108
Code Explanation: Step by Sep....112
IP extraction....113
GeoIP failures should not break login....113
Better audit logging + safer metadata....113
Logout returns a consistent message....113
password_view.py....114
1. import logging....114
Code Explanation: Step by Step....119
IP extraction helper....121
Throttle classes (rate limiting)....121
View 1: PasswordResetRequestView....122
View 2: SetNewPasswordView....124
View 3: ChangePasswordView....126
Big picture: what this file is doing....128
Services....129
Why we create a services/ folder....129
Why services are especially important in authentication systems....129
email_services.py....130
Line-by-line explanation....136
1. _get_from_email()....137
_safe_frontend_url()....138
Celery task: send_email_task....139
send_otp_email(...)....140
build_password_reset_link(user)....141
send_reset_email(...)....141
service/otp_service.py....142
Line-by-line explanation....145
OtpResult....146
send_verification_otp(...)....147
resend_user_otp(...)....149
service/security.py....150
1. from __future__ import annotations....150
Line-by-line Expalanation....154
Defensive imports....155
_get_geoip_reader()....156
get_country_code_from_ip(...)....157
is_unusual_login(...)....157
record_login_country(...)....158
password_strength(...)....158
Utils....160
utils/ip.py....160
The code....161
Security notes for ip.py....163
utils/otp.py....163
1. # accounts/utils/otp.py....163
generate_otp_secret()....167
generate_otp_code()....168

otp_valid() 169
resend_user_otp() 170
.env 171
What each line means 172
1. SECRET_KEY 172
1. DEBUG = True 172
1. EMAIL_HOST_USER 172
1. EMAIL_HOST_PASSWORD 172
celery.py 172
1. import os 173
1. from celery import Celery 173
1. os.environ.setdefault("DJANGO_SETTINGS_MODULE", 'config.settings') 173
1. app = Celery("config") 173
1. app.config_from_object("django.conf:settings", namespace="CELERY") 173
app.autodiscover_tasks() 173
config/urls.py 173
1. from django.contrib import admin 174
1. from django.urls import path, include 174
1. urlpatterns = [...] 174
1. path('admin/', admin.site.urls) 174
1. path('api/v1/auth/', include("accounts.urls")) 174
settings.py 174
1. ALLOWED_HOSTS 175
1. INSTALLED_APPS 175
Good entries 175
MIDDLEWARE 175
CORS 175
Email settings 175
AUTH_USER_MODEL 176
1. REST_FRAMEWORK 176
Authentication 176
Schema class 176
FRONTEND_URL 176
SIMPLE_JWT 176
Needs improvement for stronger security 176
SPECTACULAR_SETTINGS 177
Celery settings 177
GeoIP path 177
Database 177
Password validators 177
accounts/urls.py 178
Good parts 178
admin.py 178
admin_modules/user_admin.py 179
Closing Reflection 180

Authentication is one of the most deceptively simple parts of software engineering. On the surface, it appears to ask a narrow question: *Who is this user, and should they be allowed in?* But beneath that simple question lies a much larger and more difficult problem. Authentication is not merely a login form, nor is it only the act of comparing an email and a password. It is the discipline of establishing trust in an environment where trust cannot be assumed. It is the design of a system that must make decisions under uncertainty, resist abuse under pressure, and remain understandable enough for humans to use correctly.

In modern applications, identity sits at the center of everything. A user account is not just a record in a database; it is a boundary around personal data, private messages, financial actions, permissions, reputation, and ownership. When authentication fails, what breaks is not just access control. What breaks is confidence. A single weak point in registration, password reset, token handling, or verification flow can turn the entire product into a soft target. That is why authentication deserves to be treated not as a minor feature, but as foundational infrastructure.

This document is an attempt to study that infrastructure carefully and honestly. It walks through the design and implementation of a secure authentication system built with Django, Django REST Framework, SimpleJWT, Celery, and React.

At the heart of this project is a simple belief: secure systems should not be mysterious. They should be studied, questioned, documented, and improved in public. Too often, authentication is approached in one of two extremes. Either it is oversimplified into a set of form submissions and API endpoints, or it is spoken about in a way that feels so abstract and inaccessible that beginners are pushed away from understanding it. Neither approach is good enough. Security must be rigorous, but it must also be teachable. A system is stronger when the people building it understand not only the syntax of the code, but the reasoning behind it.

This project therefore does two things at once. It builds a working authentication system, and it explains that system line by line, file by file, and decision by decision. The purpose is not merely to present a final artifact, but to

make the thinking visible. The architecture includes user registration, email verification through OTP, JWT-based login and logout, password reset, password change, activity logging, request throttling, and supporting services for security checks and email delivery. Each piece exists because authentication is not one event; it is a chain of interdependent trust decisions. Registration must establish identity carefully. Verification must prove control over a communication channel. Login must issue credentials safely. Logout must invalidate refresh paths. Password reset must recover access without becoming an attack vector. Logging must preserve evidence without exposing secrets. Security is not one function in one file. It is a posture that emerges from how the pieces are connected.

At the same time, this document does not claim perfection, and it should not. Security is not a finished state. It is not a badge one earns once and keeps forever. It is a moving target shaped by new threats, new attack patterns, operational realities, user behavior, scaling pressures, and the simple fact that every defensive system exists in tension with an intelligent adversary. To claim that any system is perfectly secure would not be a sign of confidence; it would be a sign of misunderstanding. Real security work begins with humility. It begins by recognizing that every design is a baseline, every control has assumptions, and every implementation can be improved.

For that reason, this project should be understood as a strong baseline rather than an absolute endpoint. It is designed to reflect production-minded engineering practices: separation of concerns, explicit validation, secure password handling, token-based API authentication, OTP verification, throttle protection, audit logging, and cautious error design. These are meaningful controls. They raise the cost of attack, reduce common weaknesses, and create a system that is substantially stronger than a naive authentication implementation. But there is still a distinction between a secure baseline and a fully hardened high-assurance system. Additional layers such as device-bound sessions, advanced anomaly scoring, stronger MFA methods, WebAuthn, infrastructure hardening, secret rotation strategy, distributed revocation patterns, and formal threat modeling belong to later stages of maturity. This is

not a weakness of the project; it is an honest recognition of how security evolves.

Github Repository: https://github.com/alaminhydar/secure-authentication-system.git

How to Run the Project

This project is split into two parts: a Django backend and a React frontend. The backend handles authentication, OTP, password reset, logging, throttling, Celery tasks, and email delivery. The frontend consumes those APIs and provides the user interface. The best way to study the system is to get the backend running first, confirm the API works, then start the frontend.

What needs to be installed

Before running the project, install these tools on your machine:

- Python 3.10 or newer
- pip
- virtual environment support
- Redis
- Node.js and npm
- Git

Redis is required because Celery uses it as the broker and result backend. Without Redis, background tasks such as OTP and reset-email delivery will not work properly.

1. Clone the project

```
1. git clone <your-repo-url>
2. cd <your-project-folder>
```

2. Create and activate a virtual environment

On macOS or Linux:

```
1. python -m venv venv
2. source venv/bin/activate
```

On Windows:

```
1. python -m venv venv
2. venv\Scripts\activate
```

This isolates the project dependencies from the rest of your machine.

3. Install backend dependencies

From the backend project root, run:

```
pip install -r requirements.txt
```

If your project uses **django_filters** in **REST_FRAMEWORK** and it is not already in **requirements.txt**, install it too:

```
pip install django-filter
```

4. Create the .env file

In the root of the backend project, create a file named **.env.**

Example:

```
SECRET_KEY=replace-this-with-a-real-secret-key
DEBUG=True
EMAIL_HOST_USER=your_email@example.com
EMAIL_HOST_PASSWORD=your_app_password
```

Important notes:

- Do not commit the real **.env** file to GitHub.
- Use an app password for Gmail, not your regular Gmail password.
- In production, **DEBUG** must be **False**.

5. Check settings.py

Make sure these values are correct for local development:

```
FRONTEND_URL = "http://localhost:5173"
CELERY_BROKER_URL = "redis://localhost:6379/0"
CELERY_RESULT_BACKEND = "redis://localhost:6379/0"
```

6. Apply database migrations

Run:

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

This creates the database tables for the custom user model, logs, blacklist tables, and everything else.

7. Create an admin user

```
1. python manage.py createsuperuser
```

This gives you access to Django admin at:

http://127.0.0.1:8000/admin/

8. Start Redis

On many systems, you can start Redis with:

```
1. redis-server
```

Leave that terminal running.

If Redis is not running, Celery will not be able to queue or process email tasks.

9. Start the Django backend

Open a new terminal, activate the virtual environment again, and run:

```
1. python manage.py runserver
```

The backend will usually start at:

http://127.0.0.1:8000/

Your auth API base will be:

http://127.0.0.1:8000/api/v1/auth/

10. Start the Celery worker

Open another terminal, activate the same virtual environment, and run:

```
1. celery -A config worker --loglevel=info
```

This worker is what actually processes background email tasks.

Without this worker:

- OTP emails may not send
- password reset emails may not send

11. Start the frontend

Go to the frontend folder and install dependencies:

```
1. cd frontend
2. npm install
3. npm run dev
```

The frontend will usually run at:

http://localhost:5173
That should match the FRONTEND_URL in Django settings.

12. Recommended run order

To avoid confusion, start the system in this order:

1. Redis
2. Django backend
3. Celery worker
4. Frontend

That gives you a fully working development environment.

How to confirm everything is working

Backend checks

Open these in the browser or test with Postman:

- Admin:
  http://127.0.0.1:8000/admin/
- Auth API base:
  http://127.0.0.1:8000/api/v1/auth/

You can test flows like:

- register
- login
- verify OTP
- resend OTP
- password reset request

- password change

Email checks

Try registering a user or requesting a password reset.

If everything is correct:

- Django should accept the request
- Celery should pick up the task
- the email should be sent through your SMTP provider

If the request succeeds but no email arrives, check:

- Celery worker is running
- Redis is running
- SMTP credentials are valid
- Gmail app password is correct
- spam folder

Common issues and how to fix them

1. OTP or reset email not sending

Usually caused by one of these:

- Redis not running
- Celery worker not running
- bad email credentials
- Gmail app password not configured

2. Password reset link opens backend instead of frontend

Make sure your reset link builder uses:

FRONTEND_URL = "http://localhost:5173"
and generates links like:

http://localhost:5173/reset-password/<uidb64>/<token>
not backend URLs.

3. Login always fails even with correct email and password

Check your authentication backend. Since the code uses:

```
1. authenticate(request=request, email=email, password=password)
```

your auth backend must support email-based authentication.

4. JWT logout seems incomplete

That is normal if access tokens live too long. Logout blacklists the refresh token, but already-issued access tokens remain valid until they expire. A shorter access token lifetime improves this.

5. GeoIP lookup never works

Check that:

- the GeoLite database file exists
- the filename matches exactly
- the settings path is correct
- the package is installed

6. Frontend cannot reach backend

Check:

- backend is running on port 8000
- frontend is running on port 5173
- CORS_ALLOWED_ORIGINS includes the frontend URL
- the frontend API base URL matches the backend route prefix

.gitignore

A **.gitignore** file tells Git which files and folders should not be tracked or uploaded to GitHub.

This is very important in an authentication project because some files are:

- private
- machine-specific

- temporary
- generated automatically
- dangerous to publish

Think of **.gitignore** as the project's privacy and cleanup filter. It stops us from accidentally pushing secrets, caches, local databases, and environment files into the public repository.

What it should include

For this project, it should at least include:

```
# Python
__pycache__/
*.py[cod]
*.pyo
*.pyd
# Virtual environment
venv/
.env
.env.*
# Django
db.sqlite3
*.log
media/
staticfiles/
# Celery / cache
celerybeat-schedule
# IDE / editor
.vscode/
```

```
.idea/
# OS files
.DS_Store
Thumbs.db
# Node / frontend
node_modules/
dist/
build/
# Test / coverage
.coverage
htmlcov/
.pytest_cache/
# Migrations cache only if needed
*.sqlite3-journal
```

Most important files to never commit

Especially in a security project, never commit:

- .env
- real SMTP credentials
- real SECRET_KEY
- local database files
- venv/
- node_modules/

This system is a complete, user authentication and security layer designed to sit at the heart of any serious application, from everyday social apps to extremely sensitive environments. At its core, it manages the lifecycle of a user's identity: how they register, prove who they are, log in, stay logged in, reset their credentials, and change their password, all while constantly watching for suspicious behaviour and enforcing strong security rules. It does not just **"let people log in"**; it treats authentication as a security-critical workflow that must be monitored, validated, rate-limited, and auditable at every step.

- **The journey begins with registration.** When a new user signs up, they submit basic details such as email, password, and optional profile information. The system does not blindly accept any password: it runs the password through a strength analyzer (using **zxcvbn** under the hood) to evaluate how easy it would be to guess or crack. If the password is weak, the system pushes back with feedback, effectively educating the user to choose a stronger secret. Once a suitable password is provided, the system creates a new user record in the database using a custom user model where email, not username, is the primary identifier. The password is never stored in plain text; it is hashed using Django's secure password hashing mechanisms. After creating the account, the system generates a one-time verification code (OTP), hashes it for storage, sets a strict expiry window, and sends the plain code to the user via email. At this stage, the account exists but is not yet considered "trusted" until email ownership is proven.
- **Email verification sits at the center of trust establishment.** The system requires users to confirm they actually control the email address they registered with. When the user receives the OTP in their email and submits it back to the server, the system checks whether there is a stored OTP hash, verifies that it has not expired, and compares the submitted code against the hash. If everything matches, the user is marked as

verified, and the stored OTP data is cleared, so that the code cannot be reused. If the code is wrong or expired, the system refuses verification and can even indicate whether the user is allowed to request a new code. To prevent abuse and spam, it enforces a cooldown period between OTP resends and applies throttling on OTP verification and resend endpoints. All of this ensures that only legitimate users with access to their email can fully activate their accounts, while attackers attempting to brute-force codes or spam the endpoint are slowed down and logged.

- **Once verified, the user can log in.** The login process uses the email and password combination, validated through Django's authentication framework. If the credentials are invalid, if the account is inactive, or if the email is not yet verified, the system denies access with appropriate error messages. On successful authentication, however, it issues a pair of JSON Web Tokens: a short-lived access token and a longer-lived refresh token, generated via SimpleJWT. These tokens enable stateless, scalable authentication for APIs and frontend clients. Instead of storing heavy session state on the server, the system trusts the cryptographic tokens, which clients then attach to subsequent requests. This is exactly the pattern used by modern large-scale platforms, allowing load-balanced environments to accept user requests without centralized session storage.
- **Logout** is treated as a deliberate security operation, not a mere front-end illusion. When a user chooses to log out, they send their refresh token to the backend. The system validates it and, if it is legitimate, blacklists it. Blacklisting integrates with the JWT library so that even though tokens are stateless, a blacklisted refresh token can no longer be used to obtain new access tokens. Combined with short-lived access tokens, this gives practical revocation: even if someone had captured a token, once it is blacklisted and expired, the attacker cannot silently maintain access. This aligns more closely with robust security environments where session control and revocation need to be explicit and auditable.
- **The system also provides a careful password reset flow**, designed to be both user-friendly and resistant to user enumeration. When a user requests a password reset, they simply provide an email address. The

system always responds as though the request was accepted, regardless of whether the email is actually registered. Internally, if the user exists, it builds a secure reset link containing an encoded user ID and a signed, time-limited token. That link is sent via email using the same asynchronous email infrastructure as OTP messages. When the user clicks the link and submits a new password, the system validates the token, ensures it has not been tampered with or expired, and only then updates the stored password hash. A separate path allows authenticated users to change their password by providing their old password and a new one, again enforcing strong password rules and preventing silent takeover by someone who only has temporary access to a device.

- **Behind all of these user-facing flows, the system is quietly acting as a security operations logger.** It records key events such as login success, login failure, logout, OTP verification attempts, OTP resends, password reset requests, password changes, and completed password resets. Each event is stored with a timestamp, optional user reference, the originating IP address, and additional metadata such as the email used for login or detailed error messages. This persistent audit trail is invaluable for forensics, incident response, anomaly detection, and compliance. Security-sensitive organizations can plug this log into monitoring tools, build dashboards, and alert on patterns like repeated failed logins from the same IP, spikes in password reset requests, or clusters of OTP failures.
- **The system goes further by integrating geographic intelligence into the authentication process.** Using a GeoIP database, it determines the country associated with each login IP address. It stores the last known login country on the user's profile and compares it to the country of each new login attempt. If a user usually logs in from one country but suddenly appears from another, the system flags this as an "unusual login." That event is logged and can trigger alerts or additional verification flows in a more advanced deployment. This is the starting point for behaviour-based anomaly detection: the authentication layer begins to understand what is "normal" for each user and highlights deviations that might represent account compromise, proxy usage, or risky logins.

- **Email delivery itself is architected for scale and reliability.** Rather than sending emails synchronously during registration, OTP, or password reset flows, the system offloads this work to Celery tasks. When an email needs to be sent, the web process creates a job that is placed on a message broker (commonly Redis), and a separate worker process picks it up and performs the actual network I/O. This decoupling has multiple advantages: user-facing API responses remain fast; spikes in email traffic do not directly overload the web server; and the system can be horizontally scaled by increasing worker counts independently of API nodes. For applications serving millions of users, this is essential infrastructure, email sending becomes a background concern, not a bottleneck.
- **Altogether, this authentication system forms a layered defense around user identity.** It ensures that every important step, from registration and verification to login, logout, password management, and anomaly detection is treated as a security event, not a casual form submission. It combines strong password policies, token-based authentication, rate limiting, geographical checks, asynchronous processing, and comprehensive logging into a single coherent architecture. That design makes it suitable not just for simple web apps, but as a foundational building block in systems that demand reliability, observability, and security, including large social platforms and, when augmented with additional enterprise controls, even environments where failure is not an option.

Technologies Used

This system stands on the shoulders of a few powerful technologies that each solve a very specific class of problems in modern, large-scale authentication. At a high level, you can think of the stack as four layers working together:

- a **web framework** layer (Django) to structure the application,
- an **API layer** (Django REST Framework) to expose secure endpoints,
- a **token layer** (JWT / SimpleJWT) to represent identity in a scalable way,

- and a **security + background processing layer** (Celery, Redis, GeoIP, zxcvbn) to harden behaviour and keep things fast under load.

Instead of being random tools thrown together, each one fills a gap that appears when you try to build "**Instagram-level**" auth instead of a simple login form.

A. Django - the foundation and security guardrails

Django is a high-level Python web framework designed to help you build secure, data-driven web applications quickly and with a clean structure. It comes with batteries included: URL routing, ORM, templating, migrations, authentication primitives, and lots of security features like CSRF protection and safe password hashing.

In this system, Django is the foundation. It defines:

- how HTTP requests hit our views,
- how models like **User** and **UserActivityLog** are stored and queried,
- how passwords are hashed (**set_password, check_password**),
- and how configuration, apps and middleware are wired.

The reason it's used here is reliability and security maturity. Django has a very battle-tested auth system and password hashing strategy (PBKDF2 by default), so instead of hand-rolling low-level auth, you build on code that has been reviewed and refined for years. That matters when you are talking about "can serve a million people" and "could be used in very sensitive environments": you don't want to invent your own password hashing or session handling from scratch.

In this project, Django quietly handles the "boring but critical" parts: database access for users and logs, migrations for your custom user model, and the underlying plumbing that makes everything else possible.

B. Django REST Framework - turning Django into a clean auth API

Django REST Framework (DRF) is a toolkit built on top of Django specifically for creating Web APIs. It gives you serializers (to validate and transform input/output), generic views, authentication hooks, and browsable API pages

that make building REST endpoints much easier and safer than manually writing JSON views.

In this system, DRF is the face of the backend, it shapes how clients (like the React frontend or a mobile app) talk to the authentication engine:

- **Serializers** like **UserRegistrationSerializer, UserLoginSerializer, VerifyOTPSerializer, ChangePasswordSerializer** handle:
    - field validation (email format, password strength, matching passwords),
    - converting JSON into Python objects and back,
    - raising consistent errors when something is wrong.
- **GenericAPIView** and DRF's permission + throttle classes make it easy to express:
    - "this endpoint is for anonymous users only" (register, login, reset),
    - "this one requires authentication" (change password, logout),
    - "this one should be rate-limited heavily" (login, OTP, password reset).

The problem DRF solves is that raw Django views don't give you first-class API tools. You'd have to manually parse request bodies, check permissions, construct JSON responses, etc. With DRF, your auth flows become explicit, testable, and consistent: every request is validated by a serializer, every response is a structured API payload, and all the security policies (permissions, throttling) are attached right on the view class.

C. JWT / SimpleJWT - portable identity tokens for massive scale

JSON Web Tokens (JWTs) are a standard way (RFC 7519) to represent claims about a user as a compact, signed JSON object that can be sent between parties, usually as a bearer token. Because the token is signed, any backend that trusts the signing key can verify that the claims (like user ID, expiry time, or roles) haven't been tampered with, without needing to look them up in a session table every time.

In your system, JWTs (via **djangorestframework-simplejwt**) are the language of identity between the client and server:

- After a successful login or OTP verification, the backend gives the client:
    - an **access token** (short-lived) and
    - a **refresh token** (longer-lived).
- The client sends the access token on each request (usually an Authorization: Bearer ... header).
- The server checks the signature and expiry; if valid, it trusts the claims and treats the request as coming from that user.

This solves two big problems:

1. **Scalability -** Instead of storing per-session data in the database or cache and hitting it on every request, you can validate tokens locally in each API instance. That's ideal when traffic is huge and you have many servers behind a load balancer.
2. **Decoupling frontend and backend** - Web, mobile, and third-party services can all use the same token format, without needing to share cookie sessions or be on the same domain.

You also enable **blacklisting** of refresh tokens for logout, which gives you a realistic way to revoke long-lived tokens even though JWTs themselves are stateless.

D. Celery + (typically) Redis - background workers for email and heavy tasks

Celery is a distributed task queue system for Python. In simple terms, it lets you offload slow or expensive work, like sending emails, generating reports, or talking to external services to background workers instead of doing everything in the web request. It's designed to process huge volumes of tasks using a broker like Redis or RabbitMQ.

In this auth system, Celery is the engine room for things that shouldn't block the user:

- When a user registers, the OTP email is sent by a Celery task.
- When a password reset is requested, the reset email is also sent in the background.

The problem this solves is twofold:

- **User experience** - Email sending involves network I/O, DNS lookups, SMTP negotiation... it's slow and unpredictable. If you did that synchronously, your login/registration requests would feel laggy or timeout under load. With Celery, the API can immediately respond "we've sent an email" and let the worker handle the heavy lifting.
- **Scalability and resilience** - If you suddenly have 50,000 users who all request password resets, your Celery workers can scale horizontally (more worker processes, more machines), while your web nodes stay responsive. Email failures or slowdowns don't directly crash your main app.

Redis (or another broker) fits in as the **message bus,** the pipeline over which tasks travel from the web process to the worker processes. That separation of concerns is exactly what lets this system support very large user bases without choking on I/O.

E. GeoIP - adding location context to logins

GeoIP databases like MaxMind's GeoLite map IP addresses to approximate geographic locations such as country. The code uses geoip2 to look up the country for a given IP address and stores that country code on the user's profile.

The high-level problem this helps with is **behaviour-based security**:

- A login from an IP in Nigeria today and another from Japan ten minutes later is suspicious.
- A user who always logs in from one country suddenly appearing from another might have been compromised, or using a VPN/proxy.

In this system, GeoIP is the thing that turns a raw IP string into a meaningful signal. The login view consults the GeoIP reader, compares the country to the last known country, and, if they differ, marks the event as an "unusual login" and

logs it. This doesn't block the user by itself, but it gives us the raw material to build alerts, extra verification steps, or security dashboards that surface risky logins.

E. zxcvbn - realistic password strength, not just length checks

zxcvbn is a password strength estimator originally created by Dropbox. It doesn't just count characters or check for the presence of numbers and symbols. Instead, it uses pattern matching and large dictionaries of common passwords, names, and words to estimate how hard a password would actually be to crack.

The main problem it solves is that most "password strength meters" are either:

- too naive ("P@ssw0rd!" is accepted as 'strong'), or
- annoying in the wrong ways (forcing weird rules without improving real security).

In your system, zxcvbn is used in registration and password change flows to:

- score the password,
- look at how much it overlaps with user inputs (like email),
- and reject passwords that are too easy to guess.

This brings our security model closer to how attackers actually work (using dictionaries, patterns, keyboard sequences), instead of just ticking boxes like "has uppercase letter".

Putting it all together

Taken individually, Django, DRF, JWT, Celery, Redis, GeoIP, and zxcvbn each solve different, very practical problems: structuring the app, exposing APIs, representing identity across services, running heavy work in the background, understanding where a request comes from, and making passwords genuinely strong. In our authentication system, they're woven together so that:

- Django gives you a solid, secure web and data foundation.
- DRF turns that into a clean, validated, permission-aware API surface.

- JWTs (via SimpleJWT) let millions of clients authenticate efficiently with signed tokens instead of fragile server sessions.
- Celery and a broker keep email and other heavy tasks out of the request cycle, so the system stays fast and scalable.
- GeoIP and zxcvbn bring context and realism into your security decisions: where is this login from, and is this password actually safe?

The result isn't just "a login system", but a small ecosystem of technologies working together to make authentication robust, observable, and ready to sit in front of serious applications.

Deep Dive into the Authentication System

File Structure

```
User Authentication App
 accounts
    admin_modules
        user_admin.py
    geoip
        GeoLite2-city.mmdb
    managers
        user_manager.py
    models
        models.py
        user_model.py
    serializers
        auth_serializer.py
        otp_serializer.py
        password_reset_serializer.py
        registration_serializer.py
    services
        email_services.py
        security.py
        otp_service.py
    utils
```

```
        anomaly.py          (empty file, placeholder)
        otp.py
        ip.py
    views
        auth_view.py
        otp_view.py
        password_view.py
        registration_view.py
    tests
        __init__.py
        test_models.py
        test_serializers.py
        test_views.py
    admin.py
    apps.py
    urls.py
config
    .env
    asgi.py
    celery.py
    settings.py
    urls.py
    wsgi.py
frontend
venv
.gitignore
manage.py
requirements.txt
```

Models

When you see a **models** folder or **models.py** in Django, think:

> "This is where we define what exists in our world, what it looks like, and how it's stored."

1. What is a "model" in Django?

In Django, a **model** is a Python class that describes a **type of data** in your application and how it should be stored in the database.

- Each model = one **table** in the database
- Each attribute (field) = one **column** in that table
- Each instance (object) = one **row** in that table

So when you define:

```
1. class UserActivityLog(models.Model):
2.     ip = models.GenericIPAddressField(...)
3.     action = models.CharField(...)
4.     timestamp = models.DateTimeField(...)
```

you're really saying:

> "Create a table called **user_activity_log** with columns for IP address, action, and timestamp, and give me a nice Python interface to read/write rows."

2. Why do we need models?

Without models, you would:

- Manually write SQL (CREATE TABLE, INSERT, SELECT, etc.)
- Manually map raw database rows (dicts, tuples) to Python objects
- Manually keep your DB schema and your Python code in sync

Models solve this by being the single source of truth about your data.

You change the model class → run migrations → Django updates the DB schema for you.

For our auth system, models answer questions like:

- What is a **user**? (fields: email, password hash, OTP, is_verified, etc.)
- How do we **store security logs**? (fields: user, IP, action, metadata, timestamp)
- How do we **record suspicious events**? (fields: IP, reason, details, timestamp)

3. Models in an authentication system (our case)

In your models folder:

1. **user_model.py** defines the **User**
    - This is the *core identity* of our system:
        - Email, names, username
        - Password (hashed)
        - OTP data (hash + expiry)
        - Status flags (is_verified, is_active, is_staff, is_superuser)
        - Timestamps like date_joined, last_login
        - Permission relationships (groups, user_permissions)

    Think of this as the **passport file** for each person who uses our app.
2. models.py defines **UserActivityLog** and **SuspiciousActivity**
    - UserActivityLog:
        - Logs events like logins, logouts, OTP actions, password resets.
        - Has optional user, ip, action, metadata, timestamp.
        - This is your **black box flight recorder**: if something goes wrong, you replay the history.
    - SuspiciousActivity:

- Records events that look dangerous or unusual (e.g. weird login patterns).
- Has ip, reason, details, timestamp.
- This is like a **security incident notebook** the system keeps for you.

Together, these models describe **who the user is**, **what they did**, and **what looked suspicious**, all in a structured way that Django can store in the database and that your code can query.

4. How models fit into the bigger picture

In our overall auth system:

- **Models** = data definitions and storage
- **Serializers** = how that data is validated and transformed for APIs
- **Views** = how requests come in and interact with models/serializers
- **Services/Utils** = reusable logic that uses models (sending emails, checking security, etc.)

So when you "dive into the models", you're really answering:

> "What are the core concepts in this system, and exactly what do we store about them?"

Once that is clear, everything else serializers, views, security logic is just different ways of using those models.

Now Lets deep dive into the code files:

model/models.py

```
1. from django.conf import settings
2. from django.db import models
3. from django.utils.translation import gettext_lazy as _
4.
5. class UserActivityLog(models.Model):
6.   """
```

```
    Immutable-ish audit trail for authentication and security events.

    Security notes:
    - Use SET_NULL so deleting a user does not destroy audit history.
- Avoid storing secrets (passwords, OTP codes, tokens, reset links) in metadata.
    """

        # Prefer constants so codebase doesn't hardcode strings everywhere
        ACTION_LOGIN_SUCCESS = "login-success"
        ACTION_LOGIN_FAILURE = "login-failure"
        ACTION_LOGOUT = "logout"
        ACTION_OTP_REQUEST = "otp-request"
        ACTION_OTP_VERIFY_SUCCESS = "otp-verify-success"
        ACTION_OTP_VERIFY_FAILURE = "otp-verify-failure"
        ACTION_OTP_RESEND_SUCCESS = "otp-resend-success"
        ACTION_OTP_RESEND_FAILURE = "otp-resend-failure"
        ACTION_PASSWORD_RESET_REQUEST = "password-reset-request"
        ACTION_PASSWORD_RESET_COMPLETE = "password-reset-complete"
        ACTION_PASSWORD_CHANGE = "password-change"
        ACTION_UNUSUAL_LOGIN = "unusual-login"

        ACTION_CHOICES = [
          (ACTION_LOGIN_SUCCESS, _("Login Success")),
          (ACTION_LOGIN_FAILURE, _("Login Failure")),
          (ACTION_LOGOUT, _("Logout")),
          (ACTION_OTP_REQUEST, _("OTP Request")),
          (ACTION_OTP_VERIFY_SUCCESS, _("OTP Verify Success")),
          (ACTION_OTP_VERIFY_FAILURE, _("OTP Verify Failure")),
          (ACTION_OTP_RESEND_SUCCESS, _("OTP Resend Success")),
          (ACTION_OTP_RESEND_FAILURE, _("OTP Resend Failure")),
          (ACTION_PASSWORD_RESET_REQUEST, _("Password Reset Request")),
          (ACTION_PASSWORD_RESET_COMPLETE, _("Password Reset Complete")),
          (ACTION_PASSWORD_CHANGE, _("Password Change")),
          (ACTION_UNUSUAL_LOGIN, _("Unusual Login")),
        ]

```

```
43.       user = models.ForeignKey(
44.           settings.AUTH_USER_MODEL,
45.           on_delete=models.SET_NULL,  #keep logs even if user is deleted
46.           null=True,
47.           blank=True,
48.           related_name="activity_logs",
49.           help_text=_("User associated with this event, if available."),
50.           )
51.       ip = models.GenericIPAddressField(
52.         null=True,
53.         blank=True,
54.         help_text=_("IP address from which the action originated."),
55.       )
56.       action = models.CharField(
57.           max_length=50,
58.           choices=ACTION_CHOICES,
59.           db_index=True, # ✅ common filter
60.           help_text=_("Type of activity that occurred."),
61.         )
62.     metadata = models.JSONField(
63.         null=True,
64.         blank=True,
65.         help_text=_("Optional extra context. Never store secrets (OTP, tokens,
passwords)."),
66.       )
67.       timestamp = models.DateTimeField(
68.         auto_now_add=True,
69.         db_index=True, #  common sort/filter
70.         help_text=_("When this activity took place."),
71.       )
72.
73.       class Meta:
74.           verbose_name = _("User Activity Log")
75.           verbose_name_plural = _("User Activity Logs")
76.           ordering = ["-timestamp"]
77.           indexes = [
78.               models.Index(fields=["ip"]),
79.               models.Index(fields=["user", "timestamp"]),
80.            ]
```

```
81.
82.         def __str__(self) -> str:
83.             user_label = getattr(self.user,"email", None) or _("anonymous")
84.             return f"{self.get_action_display()} by {user_label} ({self.ip}) at
{self.timestamp}"
85.
86. class SuspiciousActivity(models.Model):
87. """
88.         Stores higher-level suspicious signals (not every event).
89.
90.         Examples:
91.           - repeated failed logins from one IP
92.           - impossible travel detection
93.           - brute force patterns
94. """
95.
96.          ip = models.GenericIPAddressField(
97.                     null=True,
98.                     blank=True,
99.                     db_index=True,
100.                    help_text=_("IP address associated with this suspicious
activity."),
101.           )
102.          reason = models.CharField(
103.              max_length=200,
104.              db_index=True,
105.              help_text=_("Short reason for flagging this activity."),
106.           )
107.          details = models.JSONField(
108.              null=True,
109.              blank=True,
110.              help_text=_("Structured details about the activity. Avoid secrets."),
111.            )
112.           timestamp = models.DateTimeField(
113.              auto_now_add=True,
114.              db_index=True,
115.              help_text=_("When this suspicious activity was recorded."),
116.          )
117.
```

```
118.        class Meta:
119.            verbose_name = _("Suspicious Activity")
120.            verbose_name_plural = _("Suspicious Activities")
121.            ordering = ["-timestamp"]
122.            indexes = [
123.                models.Index(fields=["ip", "timestamp"]),
124.             ]
125.
126.         def __str__(self) -> str:
127.             return f"{self.reason} from {self.ip or 'unknown IP'} at
{self.timestamp}"
128.
129.
```

Code Break: Step by Step

```
from django.conf import settings
```

This line gives us access to our project's global Django settings. In a big system, you never want to hardcode critical configuration. One example is the user model: this project uses a custom user model, and Django stores the reference to it in **AUTH_USER_MODEL** inside settings. By importing **settings**, we can point our **UserActivityLog.user** field to *whatever user model the project is configured to use*. That means this app stays reusable: if someone plugs it into another project with a different custom user model, our logging still works. **settings** is like the "building blueprint" for the entire application. Instead of saying "the security log always points to Room 12," you say "the security log points to whichever room is configured as 'the User room'."

```
from django.db import models
```

This imports Django's ORM model toolkit. It's the vocabulary used to describe database tables and columns using Python classes.
When we write:

- class UserActivityLog(models.Model): we're saying "this is a database table".
- models.CharField(...) means "this is a text column".
- models.DateTimeField(...) means "this is a timestamp column".
- models.Index(...) means "create a database index for performance".

models is like a kit of LEGO bricks for building database structure. Instead of writing raw SQL, you assemble your schema using these building blocks.

```
from django.utils.translation import gettext_lazy as _
```

This provides _(),a function that marks text as translatable. It's called "lazy" because Django doesn't translate the string immediately; it waits until it actually needs to display it (like in the admin panel or an API response). This is great for global apps.

When the app is running in different language(based on the active locale), Django can automatically show the translated version instead of the English one.
So when you write:

```
_("Login Success")
```

you're saying: "This label might need to be translated later, so keep it compatible with translations."
It's like writing UI labels with a sticky note that says "this can be translated" rather than carving the English text into stone.

→ **UserActivityLog** - the security camera footage of our auth system

```
class UserActivityLog(models.Model):
```

This line creates a Django model, meaning a database table called **UserActivityLog.** Its purpose is to be a permanent record of important security events: logins, failures, OTP usage, password resets, and suspicious behaviour flags.

If our authentication system is a bank vault, this is the security camera footage + visitor logbook. Even if everything is fine today, when something goes wrong later, this is what investigators use.

```
    ACTION_LOGIN_SUCCESS = "login-success"
    ACTION_LOGIN_FAILURE = "login-failure"
    ...
```

Instead of sprinkling strings like "login-success" across your views and serializers, you define them once as constants.

Instead of doing this way:

```
ACTION_CHOICES = [ ('login-success', 'Login Success'), ('login-failure', 'Login Failure'), ('logout', 'Logout'), ('otp-request', 'OTP Request'), ('otp-verify-success', 'OTP Verify Success'), ('otp-verify-failure', 'OTP Verify Failure'), ('otp-resend-success', 'OTP Resend Success'), ('otp-resend-failure', 'OTP Resend Failure'), ('password-reset-request', 'Password Reset Request'), ('password-reset-complete', 'Password Reset Complete'), ('password-change', 'Password Change'), ('unusual-login', 'Unusual Login'), ]
```

This matters for two reasons:

1. **Security correctness:** typos in action names can silently break dashboards/alerting. If one view logs "otp-resend-succes" (missing an “s”), your monitoring rules might never catch it.
2. **Maintainability:** if you ever rename an action, you change it in one place, not in 25 files.

Constants are like standard event codes used by airports. If every airline wrote their own format, you’d never reliably detect “delays” or “cancellations.” Standards let’s us build reliable monitoring.

ACTION_CHOICES: why it’s structured like this

```
1. ACTION_CHOICES = [
2.     (ACTION_LOGIN_SUCCESS, _("Login Success")),
3.     ...
```

```
]
```

This does two things at once:

- It restricts what action can be to a known safe set.
- It provides a human-friendly label for Django admin and debugging.

The first item in each tuple is what is stored in the database ("login-success"). The second is what humans see ("Login Success").

This is a subtle security win: when logs are relied on, you want consistent categories, not random strings.
The **user** field

```
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name="activity_logs",
        help_text=_("User associated with this event, if available."),
    )
```

- **ForeignKey** means each log entry can optionally be linked to a user.
- **settings.AUTH_USER_MODEL** means "link to the configured user model" (plug-and-play).
- **null=True, blank=True** means it's allowed to be empty. That matters because for events like failed logins you may not know the user (maybe the email doesn't exist). You still want the log even if you can't link a user.

Now the optimized part:

- **on_delete=models.SET_NULL** is the "audit integrity" choice.
  - If you used CASCADE, deleting a user would delete all their logs too.
  - That's dangerous: attackers, admins, or cleanup scripts could destroy evidence.
  - With SET_NULL, the logs remain, but the user field becomes empty.

Analogy: If a criminal deletes their ID card, the CCTV footage shouldn't disappear. SET_NULL keeps the footage.

- **related_name="activity_logs"** means you can do:

  user.activity_logs.all()

This is convenient, readable, and makes the model nicer for plugin usage.

- **help_text** helps in Django admin and auto-generated docs. This is not just cosmetic, good help text reduces "developer misuse," which is a real security issue in open-source systems.

The **ip** field

```
ip = models.GenericIPAddressField(
    null=True,
    blank=True,
    help_text=_("IP address from which the action originated."),
)
```

This stores IPv4 or IPv6 addresses.

Why optional? Because sometimes a request might come from internal jobs or proxies where you can't reliably determine the real client IP unless you're correctly handling headers like **X-Forwarded-For**. You'd rather log the event without an IP than fail entirely.

Security note: in production behind a load balancer, you must make sure you are capturing the *real* client IP safely (not trusting headers blindly). That's a view/middleware concern, not a model concern, but the model supports it.

The **action** field + **db_index=True** optimization

```
action = models.CharField(
    max_length=50,
    choices=ACTION_CHOICES,
    db_index=True,
    help_text=_("Type of activity that occurred."),
)
```

This is the main category of the log event.

The optimized part is **db_index=True.** This tells the database: "people will frequently filter by this column." That's true in real systems:

- "Show me all login failures in the last hour"
- "Count OTP failures per IP"
- "Find unusual-login events this week"

Without an index, those queries become slow as the table grows. With an index, the database can jump directly to matching rows rather than scanning millions.

An index is like the table of contents in a big book. Without it, you flip pages forever. With it, you open the exact page.

metadata = models.JSONField(...), powerful, but handle carefully

```
1.    metadata = models.JSONField(
2.        null=True,
3.        blank=True,
4.        help_text=_("Optional extra context. Never store secrets (OTP, tokens, passwords)."),
5.    )
```

This is where you can store extra context like:

- user agent string
- error code
- email used for login attempt (careful with privacy policies)
- throttle reason
- "country_from_geoip": "NG"

But it's also a risk: developers might accidentally store a raw OTP code, password, or JWT. That's why the help_text explicitly warns against storing secrets. You can't fully enforce this at model-level, but writing the warning directly in the schema is a real-world safety measure because it appears in admin and docs.

For high-security deployments, you usually log "enough to investigate" but not "enough to leak secrets."

Timestamp + index

```
timestamp = models.DateTimeField(
    auto_now_add=True,
    db_index=True,
    help_text=_("When this activity took place."),
)
```

auto_now_add=True means Django sets this timestamp once when the row is created and never changes it afterward. That aligns perfectly with audit logs: we don't want timestamps being edited.
db_index=True matters because almost every log query is time-based:

- last 100 events
- events in last 24 hours
- events between dates

Without a timestamp index, the database would slow down badly as log volume grows.

Meta class: where performance and admin clarity come together

```
class Meta:
        verbose_name = _("User Activity Log")
        verbose_name_plural = _("User Activity Logs")
        ordering = ["-timestamp"]
        indexes = [
            models.Index(fields=["ip"]),
            models.Index(fields=["user", "timestamp"]),
        ]
```

verbose_name and **verbose_name_plural** make Django admin readable.

- **ordering = ["-timestamp"]** means when you query logs normally, you get newest first, which is what you want for audit trails.

Now the optimized part: **indexes = [...]**.

- **models.Index(fields=["ip"])**: makes IP-based searches fast, like "show me all failures from this IP".
- **models.Index(fields=["user","timestamp"])**: makes "show me a user's history in time order" fast. This is one of the most common forensic queries.

These indexes are a big part of making the system workable for "million user" scale.

__str__ - readable audit messages

```
    def __str__(self) -> str:
        user_label = getattr(self.user, "email", None) or _("anonymous")
        return f"{self.get_action_display()} by {user_label} ({self.ip}) at
{self.timestamp}"
```

This controls how the object appears in admin and debugging.

- **getattr(self.user,"email", None)** safely tries to access the email.
- If the user is missing (SET_NULL or anonymous event), it prints "anonymous".
- **self.get_action_display()** converts the stored action code into the human label ("Login Success") based on ACTION_CHOICES.

This is like the sentence you'd see in a security logbook:
"Login Failure by anonymous (192.168.1.5) at 2025-12-16 20:01".

Model 2: **SuspiciousActivity** - higher-level alarms, not raw events

```
class SuspiciousActivity(models.Model):
```

This is not meant to record every event. That's what **UserActivityLog** does. This model is for *patterns* or *alerts*, the moment you decide something is suspicious enough to flag.

Examples:

- "20 login failures from same IP in 2 minutes"

- “impossible travel: country changed within 5 minutes”
- “OTP brute force pattern”

It’s like the difference between:

- every footstep recorded on CCTV (activity log)
- versus the security guard saying “this is suspicious, raise an alert” (suspicious activity)

IP + indexing

```
ip = models.GenericIPAddressField(
    null=True,
    blank=True,
    db_index=True,
    help_text=_("IP address associated with this suspicious activity."),
)
```

Indexing IP here makes sense because suspicious patterns are often grouped by IP. You might build alerting like “block this IP temporarily”.

Reason + indexing

```
reason = models.CharField(
    max_length=200,
    db_index=True,
    help_text=_("Short reason for flagging this activity."),
)
```

reason is a short text category like:

- “brute_force_login”
- “otp_spam”
- “impossible_travel”

Indexing reason helps you quickly query “all brute-force events”.

Details JSON

```
details = models.JSONField(
    null=True,
```

```
        blank=True,
        help_text=_("Structured details about the activity. Avoid secrets."),
    )
```

This is where we store structured info:

- count of failures
- time window
- countries involved
- sample action ids

Same warning: don't store tokens, OTP codes, passwords, reset links.

Timestamp + index

```
    timestamp = models.DateTimeField(
        auto_now_add=True,
        db_index=True,
        help_text=_("When this suspicious activity was recorded."),
    )
```

Again, **auto_now_add** makes it immutable and audit-friendly. The index supports time-based filtering ("suspicious activity today").

Meta indexes

```
    class Meta:
        ...
        indexes = [
            models.Index(fields=["ip", "timestamp"]),
        ]
```

This compound index is perfect for queries like:

- "Show suspicious events for this IP in the last 24 hours"
- "List the most recent suspicious events by IP"

Compound indexes are one of the best performance tools when you know the most common query patterns.

String representation

```
    def __str__(self) -> str:
        return f"{self.reason} from {self.ip or 'unknown IP'} at {self.timestamp}"
```

Makes it readable in admin/debugging.

models/user_model.py

```
from datetime import timedelta
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin, Group, Permission
from django.utils import timezone
from django.utils.translation import gettext_lazy as _
from django.contrib.auth.hashers import make_password, check_password
from django.core.exceptions import ValidationError
from rest_framework_simplejwt.tokens import RefreshToken
from accounts.managers.user_manager import CustomUserManager
from accounts.services.email_services import send_otp_email


class User(AbstractBaseUser, PermissionsMixin):
    """
    Custom user model for authentication with secure OTP handling.
        - Email is the primary identifier (USERNAME_FIELD).
        - OTP is stored as a hash + expiry (never store raw codes).
        - Designed to be reusable and safe for high-scale apps.
    """
    email = models.EmailField(
        _("Email Address"),
```

```
        max_length=255,
        unique=True,
        db_index=True,
        help_text=_("Primary email address used for authentication."),
    )
    first_name = models.CharField(
        _("First Name"),
        max_length=100,
        blank=True,
        help_text=_("Optional given name of the user."),
    )
    last_name = models.CharField(
        _("Last Name"),
        max_length=100,
        blank=True,
        help_text=_("Optional family name of the user."),
    )
    username = models.CharField(
        max_length=30,
        unique=True,
        null=True,
        blank=True,
        db_index=True,
        help_text=_("Optional public handle / username."),
    )
```

```
    # OTP: hashed code + expiry time
    otp_hash = models.CharField(
        max_length=120,
        null=True,
        blank=True,
        help_text=_("Hashed one-time code for verification."),
     )
     otp_expiry = models.DateTimeField(
         null=True,
         blank=True,
         db_index=True,
         help_text=_("Expiration time for the current OTP."),
    )

    is_verified = models.BooleanField(
        default=False,
        help_text=_("Designates whether the user's email has been verified."),
     )

     is_active = models.BooleanField(
        default=True,
        help_text=_("Designates whether this user should be treated as active."),
      )
    is_staff = models.BooleanField(
```

```
            default=False,
            help_text=_("Designates whether the user can access the admin site."),
        )
        is_superuser = models.BooleanField(
            default=False,
            help_text=_("Designates that this user has all permissions without
explicitly assigning  them."),
          )

    # Django permissions integration (make blank=True to avoid
admin/migration friction)
     groups = models.ManyToManyField(
            Group,
            related_name="account_user_groups",
            blank=True,
            help_text=_("The groups this user belongs to."),
       )
     user_permissions = models.ManyToManyField(
            Permission,
            related_name="account_user_permissions",
            blank=True,
            help_text=_("Specific permissions for this user."),
      )

    # Timestamps
        date_joined = models.DateTimeField(auto_now_add=True)
```

```
    last_login = models.DateTimeField(auto_now=True)
    last_login_country = models.CharField(
      max_length=2,
      blank=True,
      null=True,
     help_text=_("Last detected login country (ISO 3166-1 alpha-2)."),
   )

   USERNAME_FIELD = "email"
   REQUIRED_FIELDS = ["first_name", "last_name"]
   EMAIL_FIELD = "email"

  objects = CustomUserManager()

 class Meta:
      verbose_name = _("User")
      verbose_name_plural = _("Users")
      ordering = ["-date_joined"]
      indexes = [
         models.Index(fields=["email"]),
         models.Index(fields=["username"]),
         models.Index(fields=["is_verified"]),
      ]

def __str__(self) -> str:
```

```python
        return self.email or _("Unknown user")

    @property
    def full_name(self) -> str:
        return f"{self.first_name} {self.last_name}".strip()

    def get_full_name(self) -> str:
        # Compatibility with Django admin / conventions
        return self.full_name

    def get_short_name(self) -> str:
        return self.first_name or self.email

    @property
    def tokens(self) -> dict:
        """
            Generate JWT refresh and access tokens.
        """
        refresh = RefreshToken.for_user(self)
        return {"refresh": str(refresh), "access": str(refresh.access_token)}

    def generate_and_save_otp(self, *, expiry_minutes: int = 5) -> str:
        """
            Generate OTP, hash it, set expiry, and save to user.
            Returns the raw code (do not log it; send it via email/SMS).
```

```
        """
        from accounts.utils.otp import generate_otp_secret, generate_otp_code
        code = generate_otp_code(digits=6)

        self.otp_hash = make_password(code)  # salted hash
        self.otp_expiry = timezone.now() + timedelta(minutes=expiry_minutes)
        self.save(update_fields=["otp_hash", "otp_expiry"])
        return code

    def verify_otp(self, code: str) -> bool:
        """
        Verify OTP against hashed value and mark user verified if valid.
        Clears OTP fields on success to prevent reuse.
        """
        if (
            self.otp_hash
            and self.otp_expiry
            and timezone.now() <= self.otp_expiry
            and check_password(code, self.otp_hash)
        ):
            self.is_verified = True
            self.otp_hash = None
            self.otp_expiry = None
            self.save(update_fields=["is_verified", "otp_hash", "otp_expiry"])
            return True
```

```
179.         return False
180.
181.     def can_resend_otp(self, *, cooldown_minutes: int = 1) -> bool:
182.         """
183.         Enforce cooldown so users can't spam OTP requests.
184.         """
185.         if not self.otp_expiry:
186.             return True
187.         return timezone.now() >= self.otp_expiry +
timedelta(minutes=cooldown_minutes)
188.
189.     def resend_otp(self, *, expiry_minutes: int = 5, cooldown_minutes: int = 1)
-> str:
190.         """
191.         Resend OTP if allowed, otherwise raise ValidationError with wait time.
192.         """
193.         if not self.can_resend_otp(cooldown_minutes=cooldown_minutes):
194.             wait_seconds = int(
195.                 (self.otp_expiry + timedelta(minutes=cooldown_minutes) -
timezone.now()).total_seconds()
196.             )
197.             raise ValidationError(
198.                 _("Please wait %(seconds)d seconds before requesting a new
code."),
199.                 params={"seconds": wait_seconds},
200.             )
```

```
201.
202.        code = self.generate_and_save_otp(expiry_minutes=expiry_minutes)
203.        send_otp_email(self, code)
204.        return code
```

Code Explanation: Step by Step

```
from datetime import timedelta
```

This is Python's way of representing durations like "5 minutes" or "1 hour." In security flows, time is everything: OTPs must expire. timedelta(minutes=5) is like setting an egg timer. You add it to timezone.now() to compute a strict expiry.

```
from django.db import models
```

This is Django's "database schema language." Every models.XField becomes a database column. When you say **models.EmailField**, you're not just validating an email; you're defining how it's stored, indexed, and constrained in the database.

```
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin, Group, Permission
```

This is Django's authentication foundation.

- **AbstractBaseUser** gives us the hardened password system: hashing, checking passwords, last login updates, etc. It's like inheriting a secure vault door instead of building our own.
- **PermissionsMixin** gives us the permission engine: groups, permissions, is_superuser, etc.
- **Group** and **Permission** are the "roles" and "abilities" objects that connect our user to access control.

```
from django.utils import timezone
```

Timezones matter in distributed systems. **timezone.now()** returns a timezone-aware timestamp that plays nicely with Django settings. It prevents subtle bugs like "OTP expires too early" when servers and DB disagree on time.

```
from django.utils.translation import gettext_lazy as _
```

_() marks strings as translatable. It's mainly used in admin labels and help text. Think of it as saying: "this app can be used globally, so don't lock messages into English."

```
from django.contrib.auth.hashers import make_password, check_password
```

These are cryptographic helpers.

- **make_password("123 456")** produces a salted hash you can safely store.
- **check_password("123 456", stored_hash)** verifies without ever reversing the hash.

This is why our OTP storage is strong: even if someone steals your database, they can't read OTP codes.

```
from django.core.exceptions import ValidationError
```

This is Django's standard "data is invalid" exception. We use it in the model because models are framework-agnostic within Django; they shouldn't depend on DRF.

```
from rest_framework_simplejwt.tokens import RefreshToken
```

This generates JWT refresh/access tokens. JWT is like a signed passport: the server signs it, the client carries it, and any API node can verify it without storing a session.

```
from accounts.managers.user_manager import CustomUserManager
```

This is our "user factory." It controls how users are created (email validation, default flags, hashing password correctly).

```
from accounts.services.email_services import send_otp_email
```

This is our outbound mail pipeline hook. The model generates the code, but the sending mechanism is delegated to a service, which usually uses Celery. This separation improves security (single responsibility) and keeps the model reusable.

The User class: what each line means and why it's designed this way

```
1. class User(AbstractBaseUser, PermissionsMixin):
```

This declares our user table. Inheriting from both means:

- you get secure password storage and checking,
- you get full permission support compatible with Django admin and third-party packages.

If you skipped these and made a plain model, you'd be re-implementing security-critical code yourself (high risk).

Fields: identity + security

Email

```
1. email = models.EmailField(... unique=True, db_index=True)
```

Email is the primary identifier. **unique=True** enforces one account per email at the database level (stronger than only validating in Python). **db_index=True** makes login lookups fast.

Names

```
1. first_name = models.CharField(... blank=True)
2. last_name = models.CharField(... blank=True)
```

Optional user profile fields. **blank=True** means forms/serializers can accept them empty.

Username

```
1. username = models.CharField(unique=True, null=True, blank=True,
db_index=True)
```

Optional public handle. **null=True** + **blank=True** lets it be truly optional. Indexed for quick profile lookup.

OTP fields

```
1. otp_hash = models.CharField(...)
2. otp_expiry = models.DateTimeField(... db_index=True)
```

These turn OTP into a safe, temporary credential:

- Store only the hash (like storing a password hash).
- Store expiry (so it dies automatically).
- Index expiry so queries like "find users with expired OTP" or cleanup tasks are fast.

Status flags

is_verified is our "trust established" indicator. Without it, anyone could register with a fake email and proceed.

is_active is the kill switch. Instead of deleting users (which can destroy audit trails and cause compliance issues), you deactivate them.

is_staff and **is_superuser** integrate admin and permission logic.

Groups and permissions

```
1. groups = models.ManyToManyField(Group, ... blank=True)
2. user_permissions = models.ManyToManyField(Permission, ... blank=True)
```

These are the built-in authorization rails. The optimization here is **blank=True,** without it, admin forms can break and migrations can become annoying.

Timestamps and country

date_joined is when the account was created.

last_login is updated automatically.

last_login_country supports your GeoIP unusual login detection. It's only 2 chars because ISO-3166 country codes are 2 letters (NG, US, GB).

Auth config

```
1. USERNAME_FIELD = "email"
2. REQUIRED_FIELDS = ["first_name", "last_name"]
3. EMAIL_FIELD = "email"
4. objects = CustomUserManager()
```

This tells Django:

- “use email to log in”
- “require these fields for createsuperuser”
- “this is the email field”
- “use my custom manager to create users safely”

Meta + indexes

Indexes are “fast lookup lanes” for our DB. For high-scale apps, they’re what keeps auth fast when you have millions of users.

Methods

tokens

Generates refresh/access tokens. This keeps your views thin, they can just return **user.tokens.**

generate_and_save_otp

Generates OTP, hashes it, sets expiry, saves it. Returns raw code only for sending. Crucially: **don’t log the code**.

verify_otp

Checks expiry + hash match. If correct:

- sets **is_verified=True**
- clears OTP fields (otp_hash AND otp_expiry) to prevent reuse.

Clearing both is important: leaving expiry behind can create confusing state and bugs in resend logic.

can_resend_otp + resend_otp

These are anti-abuse controls. They enforce a cooldown. If someone spams “resend OTP,” they get blocked and told how long to wait.

managers/user_managers.py

```
from __future__ import annotations

from typing import Any, Optional

from django.contrib.auth.models import BaseUserManager
from django.core.exceptions import ValidationError
from django.core.validators import validate_email
from django.db import transaction
from django.utils.translation import gettext_lazy as _


class CustomUserManager(BaseUserManager):
    """
    Manager for the custom User model.

    Responsibilities:
    - Validate and normalize emails.
    - Create regular users safely.
    - Create superusers with strict permission flags.
    """
```

```
21.
22.     def _validate_email(self, email: str) -> str:
23.         """
24.         Validate and normalize email, returning the normalized version.
25.
26.         Raises:
27.             ValidationError: if email is missing or invalid.
28.         """
29.         if not email:
30.             raise ValidationError(_("The email field must be set."))
31.
32.         # normalize_email handles things like case normalization for domain
part
33.         normalized = self.normalize_email(email).strip()
34.
35.         try:
36.             validate_email(normalized)
37.         except ValidationError:
38.             raise ValidationError(_("Please enter a valid email address."))
39.
40.         return normalized
41.
42.     @transaction.atomic
43.     def create_user(
44.         self,
45.         email: str,
```

```
46.         password: Optional[str] = None,
47.         *,
48.         first_name: str = "",
49.         last_name: str = "",
50.         **extra_fields: Any,
51.     ):
52.         """
53.         Create and return a standard user.
54.
55.         Security/correctness notes:
56.         - Email is validated and normalized.
57.         - Password is required in most production systems; if you truly want
58.           passwordless accounts, handle that explicitly.
59.         """
60.         email = self._validate_email(email)
61.
62.         if not password:
63.             raise ValidationError(_("A password must be provided."))
64.
65.         # Prevent privilege escalation via create_user()
66.         extra_fields.pop("is_staff", None)
67.         extra_fields.pop("is_superuser", None)
68.
69.         user = self.model(
70.             email=email,
```

```
            first_name=first_name or "",
            last_name=last_name or "",
            **extra_fields,
        )
        user.set_password(password)  # Django handles hashing + salting
        user.save(using=self._db)
        return user

    @transaction.atomic
    def create_superuser(
        self,
        email: str,
        password: str,
        *,
        first_name: str = "",
        last_name: str = "",
        username: Optional[str] = None,
        **extra_fields: Any,
    ):
        """
        Create and return a superuser (admin).

        Strictly enforces required flags.
        """
        email = self._validate_email(email)
```

```
96.
97.         if not password:
98.             raise ValidationError(_("Superusers must have a password."))
99.
100.        # Provide a default username if your model uses one
101.        if not username:
102.            username = email.split("@", 1)[0]
103.        extra_fields.setdefault("username", username)
104.
105.        extra_fields.setdefault("is_staff", True)
106.        extra_fields.setdefault("is_superuser", True)
107.        extra_fields.setdefault("is_active", True)
108.
109.        if extra_fields.get("is_staff") is not True:
110.            raise ValidationError(_("Superuser must have is_staff=True."))
111.        if extra_fields.get("is_superuser") is not True:
112.            raise ValidationError(_("Superuser must have is_superuser=True."))
113.
114.        # Call create_user to reuse hashing logic
115.        return self.create_user(
116.            email=email,
117.            password=password,
118.            first_name=first_name,
119.            last_name=last_name,
120.            **extra_fields,
```

```
121.         )
```

Code Example: Step by Step

```
from typing import Any, Optional
```

Typing doesn't affect runtime performance much, but it dramatically improves maintainability and correctness. **Any** means "this could be anything," useful for ****extra_fields.** Optional[str] means the value can be a string or None. This prevents subtle bugs like treating None as a string later.

Type hints are like labels on boxes when moving house. You can still open every box without labels, but labels prevent you from putting the TV in the "fragile glass" pile.

```
1. from django.contrib.auth.models import BaseUserManager
```

This is Django's base class for user managers. A "manager" is basically the factory + query interface for a model. By inheriting from BaseUserManager, you get utilities like:

normalize_email()

compatibility with Django's createsuperuser command

conventions Django expects when creating users

if the User model is the "car," the manager is the "factory line" that produces cars correctly.

```
from django.core.exceptions import ValidationError
from django.core.validators import validate_email
```

validate_email checks email formatting according to Django's validation rules. **ValidationError** is the proper exception to raise when data doesn't meet validation requirements.

This is important because a manager is part of the Django domain layer; it shouldn't raise random Python errors that are hard to standardize.

```
from django.db import transaction
```

transaction.atomic ensures user creation is all-or-nothing. If something fails mid-creation (maybe a database constraint, or a signal, or a race condition), the database doesn't end up in a half-written state.

It's like a bank transfer. If you debit one account but fail to credit the other, that's disaster. Transactions prevent "half transfers."

```
from django.utils.translation import gettext_lazy as _
```

Same as before: allows your error messages to be translated and consistent across global deployments.

The manager class

```
class CustomUserManager(BaseUserManager):
```

This declares a custom manager for your custom user model.

Because we're are using **AbstractBaseUser**, Django requires us to define how users are created. If we don't, we'll get inconsistent creation behaviour and broken admin commands.

Email validation helper

```
def _validate_email(self, email: str) -> str:
```

Validation is more useful when it produces the final canonical value you want to store.

Important security/correctness concept: canonicalization

Email case differences can cause confusing identity behaviour. Example:

User signs up with Amin@Example.com

Later logs in using amin@example.com

If your storage and lookups are inconsistent, you get phantom duplicates or login issues. Normalization reduces that.

```
if not email:
    raise ValidationError(_("The email field must be set."))
```

This prevents creating users with blank email. Since your **USERNAME_FIELD** is email, blank email would break authentication.

```
normalized = self.normalize_email(email).strip()
```

normalize_email is built into BaseUserManager. It normalizes domain casing and keeps behaviour consistent with Django. .strip() prevents accidental spaces (" user@example.com ").

```
validate_email(normalized)
```

This ensures the email looks like a real email format.

```
.create_user
```

This is to create user.

So we enforce:

```
if not password:
    raise ValidationError(_("A password must be provided."))
```

If later you want passwordless accounts, you'd support that explicitly (e.g., SSO-only accounts), not by quietly allowing null passwords.

Prevent privilege escalation

This is a big one:

```
extra_fields.pop("is_staff", None)
extra_fields.pop("is_superuser", None)
```

Without this, someone could accidentally call:

```
User.objects.create_user(email=..., password=..., is_superuser=True)
```

Even if this isn't exposed publicly, it's a defense-in-depth measure for open-source reusability. It prevents internal misuse and mistakes.

It's like making sure the "employee onboarding form" can't accidentally include "Make me CEO = True."

```
user = self.model(...)
user.set_password(password)
user.save(using=self._db)
```

self.model(...) builds the user instance.

set_password hashes the password correctly using Django's configured algorithm (PBKDF2/Argon2/etc).

save(using=self._db) supports multi-database setups.

create_superuser

Default username

```
if not username:
    username = email.split("@", 1)[0]
    extra_fields.setdefault("username", username)
```

This is a convenience default. Using split("@", 1) is safer than split("@") because it only splits once.

Enforcing superuser flags

```
extra_fields.setdefault("is_staff", True)
extra_fields.setdefault("is_superuser", True)
extra_fields.setdefault("is_active", True)
```

Then we verify:

```
if extra_fields.get("is_staff") is not True:
        raise ValidationError(...)
```

This stops someone from accidentally creating a "superuser" that isn't staff or isn't superuser, which can lead to broken admin behaviour and security confusion.

Serializers

Serializers are one of those things that feel "extra" when you're new... until you build an API without them and realize you've been doing the same work repeatedly in messy, unsafe ways. In Django REST Framework (DRF), serializers exist because web APIs don't speak "Python objects" or "Django models", they speak **JSON**, and JSON is just raw text and primitive types coming from an untrusted world.

What a serializer is, in plain language

A serializer is a component that sits between:

- **Outside world:** JSON requests/responses (React app, mobile app, Postman, etc.)
- **Inside world:** Python objects and Django models (User objects, database rows)

It does two main jobs:

1. **Deserialization (incoming data)**
    - Take raw JSON and turn it into validated Python data.
    - Example: **"email": "amin@example.com"** becomes a clean, validated Python string you can trust.
2. **Serialization (outgoing data)**
    - Take Python objects (like a User instance) and turn them into JSON output safely.
    - Example: return **"email"** and **"full_name"** but never return **password, otp_hash**, or internal flags unless you explicitly allow it.

If your backend is a restaurant kitchen, JSON is a customer's order written on a napkin. The serializer is the waiter who rewrites it into the kitchen's official ticket format, checks it makes sense ("we don't serve 200 pizzas to one person"), and ensures the kitchen never sees nonsense orders like "cook a password hash."

Why serializers were created

Django originally focused on server-rendered websites: HTML forms, templates, and views. In that world:

- You submit a form
- Django validates it with **forms.Form** / **forms.ModelForm**
- Django saves the data to the database

But modern apps like Instagram/Snapchat-style systems have:

- React/mobile clients

- API endpoints
- JSON request bodies
- authentication tokens
- multiple clients hitting the same backend

So DRF introduced **serializers** as the API equivalent of Django Forms:

- **Django Forms** = validation & cleaning for HTML form submissions
- **DRF Serializers** = validation & cleaning for JSON API submissions

Same spirit, different world.
Django Forms were made for "paper forms at the front desk." DRF Serializers were made for "online forms submitted by millions of devices."

Why we use serializers with Django

Authentication is basically one of the most sensitive data pipeline in our app. Every request is someone trying to:

- create an identity
- prove an identity
- obtain tokens/permissions
- reset credentials

You want a strict, repeatable system that ensures:

- correct types (email is an email, password is a string, otp is 6 digits)
- required fields exist
- values follow rules (strong password, matching passwords)
- bad input never reaches the model layer
- output never leaks secrets

Without serializers, you'd end up scattering validation in:

- views (duplicated logic)
- models (too much responsibility)

- random utility functions (hard to maintain)

Serializers keep it clean: "all input rules live here."
Serializers are like airport security. They don't fly the plane (views do that). They don't build the plane (models do that). They make sure only safe, valid passengers (data) are allowed on board.

The "technology behind it" (what DRF is doing under the hood)

When you write a serializer like:

```
1. class RegisterSerializer(serializers.Serializer):
2.     email = serializers.EmailField()
3.     password = serializers.CharField()
```

DRF uses Python's class system (metaclasses and field descriptors) to:

- collect those fields
- understand expected types
- run validation automatically
- build a structured **validated_data** dictionary
- provide consistent error messages

Then when you call:

```
1. serializer.is_valid(raise_exception=True)
```

DRF runs a pipeline:

1. **Parse request JSON** into Python primitives.
2. **Field validation**:
    - EmailField checks email format
    - CharField checks string type, min length, etc.
3. **Custom validation hooks**:
    - **validate_<field>()**
    - **validate()**

4. If all passes, DRF gives you:
    - **serializer.validated_data**

Then you either:

- call **serializer.save()** (which calls **create()** or **update()**), or
- manually use **validated_data**.

On output, DRF can take a model instance and convert it into JSON again.

So it's a full data pipeline, not just a "validator."

Where serializers fit into our authentication system

In our project structure:

- **models**/: define what data exists + core methods like OTP verification
- **managers**/: define safe user creation rules
- **serializers**/: define safe *input/output rules for each endpoint*
- **views**/: connect HTTP requests to the right serializer + logic
- **services**/ and **utils**/: support functions like email sending, GeoIP, OTP code generation

Serializers are the boundary where "untrusted internet data" becomes "trusted internal data."

A very practical example

Imagine someone sends this payload to our registration endpoint:

```
{
  "email":"amin@example.com",
  "password":"123456",
  "is_superuser": true
}
```

If you just dump **request.data** into **User.objects.create(**request.data)**, you've created a disaster.

But if you use a serializer correctly:

- serializer only accepts allowed fields (email, password, etc.)
- it ignores or rejects **is_superuser**
- it validates password strength
- it ensures passwords match
- it calls **create_user()** which hashes the password safely

That's why serializers are not optional in serious API work.

serializers/registration_serializers.py

```
1. from django.db import transaction
2. from django.utils.translation import gettext_lazy as _
3.
4. from rest_framework import serializers
5. from rest_framework.exceptions import ValidationError as
DRFValidationError
6.
7. from accounts.models.user_model import User
8. from accounts.services.security import password_strength
9.
10.
11. class UserRegistrationSerializer(serializers.ModelSerializer):
```

```
    """
    Handles user registration input validation and safe user creation.

    Key responsibilities:
    - Validate password strength (zxcvbn via password_strength()).
    - Confirm password match.
    - Create the user using the manager (hashing password correctly).
    """

    password = serializers.CharField(write_only=True, min_length=8, trim_whitespace=False)
    confirm_password = serializers.CharField(write_only=True, min_length=8, trim_whitespace=False)

    first_name = serializers.CharField(required=False, allow_blank=True, default="")
    last_name = serializers.CharField(required=False, allow_blank=True, default="")
    username = serializers.CharField(required=False, allow_blank=True, allow_null=True)

    class Meta:
        model = User
        fields = ["email", "username", "first_name", "last_name", "password", "confirm_password"]

    def validate_email(self, email: str) -> str:
```

```
        # Normalize to avoid accidental duplicates like " amin@x.com "
        email = (email or "").strip()
        if not email:
            raise DRFValidationError(_("Email is required."))
        return email

    def validate_password(self, password: str) -> str:
        """
        Enforce password strength. Uses the email as a user_input to penalize
        passwords that contain the email.
        """
        email = (self.initial_data.get("email") or "").strip()
        result = password_strength(password, user_inputs=[email] if email else
[])
        score = result.get("score", 0)

        if score < 3:
            feedback = result.get("feedback") or {}
            # Balanced: helpful but not overly verbose
            warning = feedback.get("warning")
            suggestions = feedback.get("suggestions") or []
            message = warning or (suggestions[0] if suggestions else _("Choose a
stronger password."))

            raise DRFValidationError({"password": _("Password too weak: %
(msg)s") % {"msg": message}})
```

```
56.         return password
57.
58.     def validate(self, attrs):
59.         password = attrs.get("password")
60.         confirm = attrs.pop("confirm_password", None)
61.
62.         if password != confirm:
63.             raise DRFValidationError({"confirm_password": _("Passwords do not match.")})
64.
65.         return attrs
66.
67.     def create(self, validated_data):
68.         """
69.         Create a user safely inside a transaction.
70.
71.         Uses the custom manager's create_user() which hashes the password and
72.         applies creation rules.
73.         """
74.         with transaction.atomic():
75.             user = User.objects.create_user(**validated_data)
76.             return user
77.
```

Code Explanation: Step by Step

```
from django.db import transaction
```

This gives you transactional safety. A transaction is a database "all-or-nothing" wrapper. If registration fails halfway through (say the user is created but a later step fails), a transaction rolls everything back, preventing broken partial accounts.

It's like a bank transfer. Either the whole transfer succeeds, or nothing happens, no "half" state.

```
from django.utils.translation import gettext_lazy as _
```

The _() function marks strings as translatable. It's called "lazy" because the translation happens later when the system knows which language is active.

You're writing labels that can be printed in English today, French tomorrow, without rewriting your code.

```
from rest_framework import serializers
```

This is the DRF serializer system. It is DRF's equivalent of Django Forms, but for APIs.

- Django Forms validate HTML form submissions.
- DRF Serializers validate and convert JSON for API requests.

Django Forms were built for "paper forms at a desk." Serializers are built for "millions of phones and web clients sending JSON."

```
from rest_framework.exceptions import ValidationError as DRFValidationError
```

This is DRF's validation error that produces proper API-friendly error responses (400 with a structured JSON error payload).

We alias it to avoid confusion with Django's ValidationError.

```
from accounts.models.user_model import User
```

We import the User model so the serializer knows which model it is creating and validating against.

```
from accounts.services.security import password_strength
```

This is our strength checker (zxcvbn). It gives a score and feedback, allowing us to reject weak passwords.

Security point: this is one of our strongest moves, weak passwords are one of the biggest reasons accounts get hacked.

Serializer class

```
class UserRegistrationSerializer(serializers.ModelSerializer):
```

A **ModelSerializer** is a serializer that knows how to work with Django models. It can automatically:

- generate fields from our model
- validate uniqueness constraints
- create/update instances when we implement **create()** or **update()**

A ModelSerializer is like a form template already aware of our database structure. We still add custom rules, but we don't rebuild everything from scratch.

Fields: why we define these explicitly

```
password = serializers.CharField(write_only=True, min_length=8, trim_whitespace=False)

confirm_password = serializers.CharField(write_only=True, min_length=8, trim_whitespace=False)
```

- **write_only=True** means the API can accept these fields but will never return them in responses. That prevents accidental password leaks.
- **min_length=8** is a basic baseline.
- **trim_whitespace=False** prevents DRF from silently trimming passwords. Some systems allow spaces in passwords. Trimming could change what the user intended and cause login failures later.

```
first_name = serializers.CharField(required=False, allow_blank=True, default="")

last_name = serializers.CharField(required=False, allow_blank=True, default="")
```

- **required=False** means user can register without these.
- **allow_blank=True** means empty string is allowed.

- **default=""** ensures the field becomes an empty string if missing, preventing None-related issues.

```
username = serializers.CharField(required=False, allow_blank=True, allow_null=True)
```

This makes username optional. If you keep **unique=True** on the model, Django will enforce uniqueness. We just make it flexible for users who don't want a username at signup.

Meta section

```
class Meta:
    model = User
    fields = ["email", "username", "first_name", "last_name", "password", "confirm_password"]
```

This tells DRF:

- which model we're working with
- which fields are allowed through the registration endpoint

Security note: This acts like a whitelist. If someone tries to include dangerous fields like **is_superuser**, the serializer ignores them because they're not in fields.

That is one of the biggest protections serializers provide.

validate_email

```
def validate_email(self, email: str) -> str:
    email = (email or "").strip()
    if not email:
        raise DRFValidationError(_("Email is required."))
    return email
```

This runs automatically for the email field.

- .strip() removes accidental whitespace.

- rejecting empty email prevents invalid accounts.

It's like checking that an address field isn't blank before you ship a package.

validate_password

```
result = password_strength(password, user_inputs=[email] if email else [])
score = result.get("score", 0)
```

This checks strength. The **user_inputs** feature is important: if someone uses their email inside their password, **zxcvbn** penalizes it.

Example:

- email: amin@example.com
- password: amin12 345
  This should be flagged as weak because attackers often guess passwords based on user info.

```
 if score < 3:
     ...
     raise DRFValidationError({"password": _("Password too weak: %(msg)s") %
{"msg": message}})
```

We reject passwords below score 3 (a common practical threshold). We still provide user-friendly feedback, but not so much detail that you help attackers.

This is a security balance:

- too vague = users get frustrated
- too detailed = attackers learn your policy and optimize guessing

validate (cross-field validation)

```
1. password = attrs.get("password")
2. confirm = attrs.pop("confirm_password", None)
3.
4. if password != confirm:
```

```
5.      raise DRFValidationError({"confirm_password": _("Passwords do not match.")})
```

This is “cross-field validation” because we compare two fields.

We use **pop** so **confirm_password** is removed before hitting **create_user(**validated_data)**. That prevents accidentally trying to pass **confirm_password** into model creation (which would error).

We also attach the error to **confirm_password** (the most intuitive field for UI display).

create

```
1. with transaction.atomic():
2.      user = User.objects.create_user(**validated_data)
3.      return user
```

- **transaction.atomic()** ensures consistency.
- **create_user()** ensures password hashing and email validation rules from the manager are used.
- We do **not** call **user.save()** again, because **create_user()** already saves.

serializers/otp_serializer.py

```
1. from datetime import timedelta
2.
3. from django.utils import timezone
4. from django.utils.translation import gettext_lazy as _
5.
```

```python
from rest_framework import serializers
from rest_framework.exceptions import ValidationError as DRFValidationError

from accounts.models.user_model import User
from accounts.utils.otp import resend_user_otp


class ResendOTPSerializer(serializers.Serializer):
    """
    Requests a new OTP for a user who is not yet verified.

    High-security posture:
    - Prevents user enumeration by returning the same success message whether the email exists or not.
    - Does not reveal "email exists" vs "does not exist".
    - Still enforces cooldown if the user exists (but we do not reveal precise timing in the response).
    """

    email = serializers.EmailField()

    def validate_email(self, email: str) -> str:
        # Normalize user input so ' Amin@Example.com ' becomes 'amin@example.com'
        self.normalized_email = (email or "").strip().lower()
```

```
28.         return self.normalized_email
29.
30.     def create(self, validated_data):
31.         """
32.         Always returns a generic response to prevent attackers from discovering
33.         which emails are registered.
34.
35.         If the user exists and is eligible, it sends a new OTP.
36.         Otherwise it does nothing (but still returns success).
37.         """
38.         # Generic response: same message regardless of existence/status.
39.         response = {"message": _("If the email exists, a verification code has
been sent.")}
40.
41.         try:
42.             user = User.objects.get(email=self.normalized_email)
43.         except User.DoesNotExist:
44.             return response    # silent: prevents enumeration
45.
46.         # If the account is disabled or already verified, do nothing.
47.         if (not user.is_active) or user.is_verified:
48.             return response
49.
50.         # Cooldown: only resend if allowed. We do not reveal wait seconds here
(anti-enumeration & anti-abuse).
51.         if user.can_resend_otp(cooldown_minutes=1):
```

```
            resend_user_otp(user)

        return response


class VerifyOTPSerializer(serializers.Serializer):
    """
    Verifies an OTP code for a given email address.

    Security posture:
    - Avoids user enumeration (generic error message).
    - Validates OTP format before hitting DB-heavy operations.
    - Does not reveal whether the email exists.
    """

    email = serializers.EmailField()
    otp = serializers.CharField(
        min_length=6,
        max_length=6,
        write_only=True,
        trim_whitespace=True,
    )

    def validate(self, attrs):
        # Normalize inputs
```

```
77.         email = (attrs.get("email") or "").strip().lower()
78.         otp = (attrs.get("otp") or "").strip()
79.         now = timezone.now()
80.
81.         # Fast format validation (cheap checks before DB lookup)
82.         if len(otp) != 6 or not otp.isdigit():
83.             raise DRFValidationError({"otp": _("OTP must be a 6-digit number.")})
84.
85.         try:
86.             user = User.objects.get(email=email)
87.         except User.DoesNotExist:
88.             # Generic error prevents enumeration
89.             raise DRFValidationError({"detail": _("Invalid email or OTP.")})
90.
91.         # Block verification if account is disabled
92.         if not user.is_active:
93.             raise DRFValidationError({"detail": _("Invalid email or OTP.")})
94.
95.         # Expiry check first for clearer UX (still generic enough)
96.         if user.otp_expiry and user.otp_expiry < now:
97.             raise DRFValidationError({
98.                 "detail": _("Verification code has expired."),
99.                 "can_resend": user.can_resend_otp(cooldown_minutes=1),
100.             })
101.
```

```
        # Verify OTP using the model's secure hash check
        if not user.verify_otp(otp):
            raise DRFValidationError({
                "detail": _("Invalid or expired verification code."),
                "can_resend": user.can_resend_otp(cooldown_minutes=1),
            })

        # Store user for create()
        self.user = user

        # Store normalized email back
        attrs["email"] = email
        return attrs

    def create(self, validated_data):
        """
        Returns tokens upon successful verification.
        """
        tokens = self.user.tokens
        return {
            "message": _("Email verified successfully."),
            "tokens": tokens,
            "user": {
                "email": self.user.email,
                "username": self.user.username,
```

```
            "is_verified": self.user.is_verified,
        },
    }
```

Code Explanation: Step by Step

```
from datetime import timedelta
```

timedelta represents a time duration ("1 minute", "5 minutes"). OTP systems are time-based by nature: codes expire and resend actions have cooldowns. Even if this file only uses cooldown checks indirectly, **timedelta** is the standard tool for expressing "add X minutes to now."

It's like setting the expiry stamp on a parking ticket , "valid for 5 minutes."

```
from django.utils import timezone
```

This gives you **timezone.now()**, which returns a timezone-aware current timestamp. Using **timezone.now()** avoids bugs in environments where servers/database run in different timezones or where daylight saving time exists.

The system uses one official clock on the wall, not everyone's personal watch.

```
from django.utils.translation import gettext_lazy as _
```

This defines **_()** as a translation marker. It lets your messages ("Email verified successfully") be translated into other languages if the app is deployed globally.

```
from rest_framework import serializers
```

DRF serializers are the API equivalent of Django Forms. They:

- validate request input
- clean/normalize data
- shape output responses

Serializers are like the receptionist who checks your form is filled properly before it goes into the company system.

```
from rest_framework.exceptions import ValidationError as DRFValidationError
```

This is DRF's structured error exception. When raised, DRF turns it into a clean JSON response with 400 Bad Request (or other appropriate status when raised in views).

```
from accounts.models.user_model import User
```

We import the user model so we can:

- find users by email
- check status flags like **is_active** and **is_verified**
- call model methods like **verify_otp()**

```
from accounts.utils.otp import resend_user_otp
```

This function encapsulates the OTP generation + sending pipeline (usually via email service). Keeping it in a utility/service prevents duplication and keeps serializer responsibilities focused on validation + orchestration.

Think of it like this, Serializer is the person who approves sending a letter, and **resend_user_otp()** is the mailroom that actually prints and sends it.

ResendOTPSerializer

```
class ResendOTPSerializer(serializers.Serializer):
```

This defines a serializer that isn't tied to creating a database record; it triggers an action ("resend verification OTP").

```
email = serializers.EmailField()
```

DRF ensures the input is a valid email-shaped string before your custom validation runs.

```
def validate_email(self, email: str) -> str:
```

Field-level validation. DRF automatically calls this when validating email.

```
self.normalized_email = (email or "").strip().lower()

return self.normalized_email
```

We normalize email to prevent edge cases:

- users paste emails with spaces
- different casing causes mismatch issues

We store it on **self** so **create()** can use it without re-reading raw input.

```
def create(self, validated_data):
```

In DRF, **.save()** calls **.create()** for serializers that aren't updating an existing instance.

This is where we "perform the resend action."

```
response = {"message": _("If the email exists, a verification code has been sent.")}
```

This is the key anti-enumeration feature.

Even if the email does not exist, the response looks the same. Attackers can't use our resend endpoint to discover which emails are registered.

Think of it like this, If someone calls a hotel asking "Is Mr. X staying there?", a privacy-friendly receptionist replies: "If the guest is staying here, we'll pass along your message." They never confirm.

```
1. try:
2.     user = User.objects.get(email=self.normalized_email)
3. except User.DoesNotExist:
4.     return response
```

If user doesn't exist, we return the same response without revealing anything.

This protects our user base from being harvested.

```
1. if (not user.is_active) or user.is_verified:
2.     return response
```

If the account is disabled, we don't help the attacker by sending codes. If already verified, resending verification OTP is unnecessary.

We still return the generic success message for privacy.

```
1. if user.can_resend_otp(cooldown_minutes=1):
2.     resend_user_otp(user)
```

We respect cooldown rules to prevent abuse and reduce email provider throttling.

Notice what we *don't* do:

- we don't reveal "wait X seconds"
  Because if you reveal precise wait times, attackers can still infer account existence and OTP state.

VerifyOTPSerializer

class VerifyOTPSerializer(serializers.Serializer):

This serializer verifies the OTP and, on success, returns tokens.

```
1. email = serializers.EmailField()
2. otp = serializers.CharField(min_length=6, max_length=6, write_only=True, trim_whitespace=True)
```

- fixed length OTP input
- **write_only** prevents accidental OTP leak in responses
- **trim_whitespace** avoids hidden spaces

```
1. email = (attrs.get("email") or "").strip().lower()
2. otp = (attrs.get("otp") or "").strip()
3. now = timezone.now()
```

Normalize input and cache current time so multiple comparisons use the same timestamp (tiny performance + consistency improvement).

```
1. if len(otp) != 6 or not otp.isdigit():
2.     raise DRFValidationError({"otp": _("OTP must be a 6-digit number.")})
```

This is a fast check before database work. It blocks junk requests early.

Think of it like this, checking if a key is the right shape before trying it in the door lock.

```
1. try:
2.     user = User.objects.get(email=email)
```

```
3. except User.DoesNotExist:
4.     raise DRFValidationError({"detail": _("Invalid email or OTP.")})
```

We avoid enumeration by using a generic error message.

```
1. if not user.is_active:
2.     raise DRFValidationError({"detail": _("Invalid email or OTP.")})
```

Disabled users shouldn't be verified or granted tokens.

```
1. if user.otp_expiry and user.otp_expiry < now:
2.     raise DRFValidationError({...})
```

If expired, we return an error and also **can_resend** which helps the frontend decide whether to show **"Resend OTP"** button.

This is still safe because it only applies once a user was found; if you want extreme privacy, you can also remove **can_resend.**

```
1. if not user.verify_otp(otp):
2.     raise DRFValidationError({...})
```

verify_otp() does the secure check:

- hashes are compared safely via **check_password**
- expiry logic is enforced
- OTP is cleared on success

```
1. self.user = user
2. attrs["email"] = email
3. return attrs
```

Store user so **create()** can generate tokens without another DB lookup.

```
1. def create(self, validated_data):
2.     tokens = self.user.tokens
```

```
3.      return {...}
```

On success, we return JWT tokens and basic user info.

auth_serializer.py

```
1. from django.contrib.auth import authenticate
2. from django.utils.translation import gettext_lazy as _
3.
4. from rest_framework import serializers
5. from rest_framework.exceptions import AuthenticationFailed
6. from rest_framework_simplejwt.tokens import RefreshToken
7. from rest_framework_simplejwt.exceptions import TokenError
8.

9. class UserLoginSerializer(serializers.Serializer):
10.     email = serializers.EmailField()
11.     password = serializers.CharField(write_only=True, trim_whitespace=False)
12.
13.     access = serializers.CharField(read_only=True)
14.     refresh = serializers.CharField(read_only=True)
15.     full_name = serializers.CharField(read_only=True)
16.     user_status = serializers.JSONField(read_only=True)
17.
18.     def validate(self, attrs):
```

```
        request = self.context.get("request")

        # Normalize email (safe). Never normalize passwords.
        email = (attrs.get("email") or "").strip().lower()
        password = attrs.get("password") or ""


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

        # Generic error prevents user enumeration
        if not user or not user.is_active:
            raise AuthenticationFailed(_("Invalid credentials."))


        if not getattr(user, "is_verified", True):
            raise AuthenticationFailed(_("Invalid credentials."))

        token_pair = RefreshToken.for_user(user)

        # Keep user internally for views/audit logging without exposing it in response
        attrs["user"] = user

        return {
            "email": email,
            "access": str(token_pair.access_token),
            "refresh": str(token_pair),
            "full_name": getattr(user, "full_name", "") or user.get_full_name(),
            "user_status": {
                "is_verified": getattr(user, "is_verified", False),
                "is_active": user.is_active,
                "is_staff": user.is_staff,
            },
        }


class UserLogoutSerializer(serializers.Serializer):
    refresh = serializers.CharField(write_only=True)

```

```
    def validate_refresh(self, token: str) -> str:
        try:
            RefreshToken(token)
        except TokenError:
             raise serializers.ValidationError(_("Token is invalid or expired."))
        return token

    def save(self, **kwargs):
         token_str = self.validated_data["refresh"]

         try:
            token = RefreshToken(token_str)
            token.blacklist()
        except AttributeError:
            # Happens if token_blacklist app isn't installed
            raise serializers.ValidationError(
            _("Token blacklisting is not enabled. Add
'rest_framework_simplejwt.token_blacklist' to  INSTALLED_APPS.")
 )
        except TokenError:
                raise serializers.ValidationError(_("Token is invalid or expired."))

        return {"message": _("Successfully logged out.")}

```

This file contains **two DRF serializers** used as "action serializers":

- **Login serializer**: validates credentials and returns **JWT access** + **refresh tokens** (stateless authentication).
- **Logout serializer**: revokes the refresh token by **blacklisting** it (prevents new access tokens).

Think of it like:

- **Login** = "issue a badge (tokens) after checking identity"
- **Logout** = "invalidate the badge re-issuer (refresh token) so the user can't get new badges"

```
1. from django.contrib.auth import authenticate
```

Django's official login entry point.
It runs through the configured authentication backends (database, custom backend, etc.) and returns a **User** if credentials are valid.

```
1. from django.utils.translation import gettext_lazy as _
```

Marks text for translation. _() wraps strings so they can be localized.

```
1. from rest_framework import serializers
```

DRF serializer framework: validates incoming JSON and shapes outgoing response data.

```
1. from rest_framework.exceptions import AuthenticationFailed
```

A DRF exception used for authentication errors (maps cleanly to HTTP 401).
This is more correct than using ValidationError for login.

```
1. from rest_framework_simplejwt.tokens import RefreshToken
```

Used to generate refresh + access tokens.

```
1. from rest_framework_simplejwt.exceptions import TokenError
```

Thrown when a JWT is malformed, invalid, expired, or otherwise unusable.

UserLoginSerializer

```
email = serializers.EmailField()
```

Validates that input looks like an email.

```
password = serializers.CharField(write_only=True, trim_whitespace=False)
```

- **write_only=True** means password will never be returned in API responses.
- **trim_whitespace=False** ensures DRF doesn't silently modify passwords.

```
access = serializers.CharField(read_only=True)
refresh = serializers.CharField(read_only=True)
```

Output-only token fields.

```
full_name = serializers.CharField(read_only=True)
user_status = serializers.JSONField(read_only=True)
```

- full_name is a UI convenience.
- user_status summarizes key flags so the frontend can behave correctly.

validate() logic

```
request = self.context.get("request")
```

We pull the request from serializer context. Some backends use request data (IP, headers).

```
email = (attrs.get("email") or "").strip().lower()
password = attrs.get("password") or ""
```

We normalize the email to avoid cases like " Amin@Example.com " causing confusion.
We do **not** alter password.

```
user = authenticate(request=request, email=email, password=password)
```

Asks Django: "Do these credentials match any user?"

> Important: this assumes your auth backend supports email=.... If not, you'd need a backend that authenticates by email, or use username=email.

```
if not user or not user.is_active:
    raise AuthenticationFailed(_("Invalid credentials."))
```

High-security approach:

- same message whether email exists, password wrong, or account disabled
- prevents attackers learning anything useful

```
if not getattr(user, "is_verified", True):
    raise AuthenticationFailed(_("Invalid credentials."))
```

Also generic (high-security).
If you want user-friendly UX, you could change the message here, but that leaks "account exists".

```
token_pair = RefreshToken.for_user(user)
```

Creates a refresh token for the user; access token comes from it.

```
attrs["user"] = user
```

Keeps the user available internally (views can log events) without exposing the object in output.

```
return {...}
```

Returns the token payload + safe user info.

UserLogoutSerializer

```
refresh = serializers.CharField(write_only=True)
```

Client submits refresh token to revoke it.

```
RefreshToken(token)
```

Checks token is a valid refresh token format.

```
token.blacklist()
```

Revokes it permanently (requires token_blacklist app enabled).

We catch:

- **AttributeError** → blacklist not installed
- **TokenError** → token invalid

password_reset_serializer.py

```
1. from django.contrib.auth.tokens import PasswordResetTokenGenerator
2. from django.db import transaction
3. from django.utils.encoding import force_str
4. from django.utils.http import urlsafe_base64_decode
5. from django.utils.translation import gettext_lazy as _
6.
7. from rest_framework import serializers
8. from rest_framework.exceptions import ValidationError as
DRFValidationError
9.
10. from accounts.models.user_model import User
11. from accounts.services.email_services import build_password_reset_link,
send_reset_email
12. from accounts.services.security import password_strength
13.
14.
15. class PasswordResetRequestSerializer(serializers.Serializer):
16.     """
17.     Request a password reset link.
18.
19.     Security posture:
20.     - Anti-enumeration: Always behave as if the request succeeded.
21.     - If the email exists, send a reset link. If not, do nothing silently.
22.     """
```

```python

    email = serializers.EmailField()

    def validate_email(self, email: str) -> str:
        return (email or "").strip().lower()

    def save(self):
        email = self.validated_data["email"]

        # Generic behaviour: do not reveal whether this email exists
        try:
            user = User.objects.get(email=email)
        except User.DoesNotExist:
            return  # silent by design

        # Optional: if user is inactive, you may decide not to send reset email
        if not user.is_active:
            return

        link = build_password_reset_link(user)
        send_reset_email(to_email=user.email, link=link)


class SetNewPasswordSerializer(serializers.Serializer):
    """
```

```
    Set a new password using a reset link containing uidb64 + token.
    """

    uidb64 = serializers.CharField()
    token = serializers.CharField()

    password = serializers.CharField(write_only=True, min_length=8, trim_whitespace=False)
    confirm_password = serializers.CharField(write_only=True, min_length=8, trim_whitespace=False)

    def validate(self, attrs):
        password = attrs.get("password")
        confirm = attrs.get("confirm_password")

        if password != confirm:
            raise DRFValidationError({"confirm_password": _("Passwords do not match.")})

        # Decode the user ID
        try:
            uid = force_str(urlsafe_base64_decode(attrs["uidb64"]))
            user = User.objects.get(pk=uid)
        except Exception:
            raise DRFValidationError({"detail": _("Invalid reset link.")})

```

```
        # Validate token
        if not PasswordResetTokenGenerator().check_token(user, attrs["token"]):
            raise DRFValidationError({"detail": _("Reset link is invalid or expired.")})

        # Enforce password strength
        # Use user_inputs to penalize passwords that contain the user's email
        result = password_strength(password, user_inputs=[getattr(user, "email", "")])
        if result.get("score", 0) < 3:
            feedback = result.get("feedback") or {}
            warning = feedback.get("warning")
            suggestions = feedback.get("suggestions") or []
            msg = warning or (suggestions[0] if suggestions else _("Choose a stronger password."))
            raise DRFValidationError({"password": _("Password too weak: %(msg)s") % {"msg": msg}})

        self.user = user
        return attrs

    @transaction.atomic
    def save(self):
        self.user.set_password(self.validated_data["password"])
        self.user.save(update_fields=["password"])
        return self.user
```

```
94.
95.
96. class ChangePasswordSerializer(serializers.Serializer):
97.     """
98.     Change password for an authenticated user (requires old password).
99.     """
100.
101.     old_password = serializers.CharField(write_only=True,
trim_whitespace=False)
102.     new_password = serializers.CharField(write_only=True, min_length=8,
trim_whitespace=False)
103.     confirm_password = serializers.CharField(write_only=True,
min_length=8, trim_whitespace=False)
104.
105.     def validate_new_password(self, new_password: str) -> str:
106.         # Check strength
107.         result = password_strength(new_password)
108.         if result.get("score", 0) < 3:
109.             raise DRFValidationError(_("New password is too weak. Choose a
stronger one."))
110.         return new_password
111.
112.     def validate(self, attrs):
113.         new_password = attrs.get("new_password")
114.         confirm_password = attrs.get("confirm_password")
115.         old_password = attrs.get("old_password")
```

```
116.
117.         if new_password != confirm_password:
118.             raise DRFValidationError({"confirm_password": _("Passwords do not match.")})
119.
120.         user = self.context["request"].user
121.
122.         # Block change for inactive users
123.         if not user.is_active:
124.             raise DRFValidationError({"detail": _("Unable to change password.")})
125.
126.         # Verify old password
127.         if not user.check_password(old_password):
128.             raise DRFValidationError({"old_password": _("Old password is incorrect.")})
129.
130.         self.user = user
131.         return attrs
132.
133.     @transaction.atomic
134.     def save(self):
135.         self.user.set_password(self.validated_data["new_password"])
136.         self.user.save(update_fields=["password"])
137.         return self.user
```

Code Explanation: Step by step

```
from django.contrib.auth.tokens import PasswordResetTokenGenerator
```

This is Django's built-in secure token generator for password reset links. It signs tokens based on:

- user id
- password hash
- last login
- timestamps
  so tokens become invalid if the password changes or the token expires.

Think of it like this, it's a tamper-proof wax seal on a letter: if someone changes the letter, the seal breaks.

```
1. from django.db import transaction
```

Ensures password updates are atomic. If anything fails during save, the change is rolled back.

It's like a "commit" button, either everything completes safely or nothing happens.

```
1. from django.utils.encoding import force_str
2. from django.utils.http import urlsafe_base64_decode
```

Reset links contain uidb64 (base64 encoded user ID). These decode it safely into a usable ID string.

It's like decoding a shipping label barcode into the actual address.

```
1. from django.utils.translation import gettext_lazy as _
```

For internationalized error messages.

```
1. from rest_framework import serializers
2. from rest_framework.exceptions import ValidationError as
DRFValidationError
```

DRF serializers validate input and produce clean JSON errors.

```
from accounts.models.user_model import User
```

We need the user to:

- look them up by email or ID
- set/check password

```
from accounts.services.email_services import build_password_reset_link, send_reset_email
```

This constructs the reset URL and sends email (via Celery).

```
from accounts.services.security import password_strength
```

This calls **zxcvbn** to score password strength and provide feedback.

PasswordResetRequestSerializer

```
email = serializers.EmailField()
```

Accepts email.

```
def validate_email(self, email: str) -> str:
    return (email or "").strip().lower()
```

Normalizes email.

Anti-enumeration design

```
try:
    user = User.objects.get(email=email)
except User.DoesNotExist:
    return
```

Even if no user exists, we don't raise an error. This prevents attackers from checking which emails are registered.

Then:

```
link = build_password_reset_link(user, request)
send_reset_email(...)
```

If user exists, we generate a link and send it.

SetNewPasswordSerializer

Confirm password match

```
1. if password != confirm:
2.     raise DRFValidationError({"confirm_password": _("Passwords do not match.")})
```

Decode uidb64

```
1. uid = force_str(urlsafe_base64_decode(attrs["uidb64"]))
2. user = User.objects.get(pk=uid)
```

This finds the user referenced by the reset link.

Validate reset token

```
1. PasswordResetTokenGenerator().check_token(user, attrs["token"])
```

This ensures the link is signed, untampered, and not expired.

Strength check

We added:

```
1. result = password_strength(password, user_inputs=[user.email])
2. if score < 3: raise ...
```

ChangePasswordSerializer

We:

- check password strength
- verify old password
- atomic save

Views

Views are where our authentication system "comes to life" as an API. If models are the data and rules, and serializers are the gatekeepers that validate and clean input, then views are the part that actually handles the conversation with the outside world: they receive a request, decide what workflow it belongs to (register, login, verify OTP, reset password), run the right validations, trigger the right actions, and return a response.

What a "view" is in Django/DRF

A view is a piece of code that answers one question:

> "When someone hits this URL with this HTTP method (POST/GET/PATCH), what should happen?"

In classic Django (template-based), a view often returns an HTML page. In Django REST Framework, a view usually returns JSON because we're building an API that our React frontend (and potentially mobile apps) can call.

So, in our system:

- /auth/register/ should accept user details and create an account.
- /auth/otp/verify/ should accept email + OTP and verify ownership.
- /auth/login/ should accept credentials and issue tokens.
- /auth/password/reset/ should start the reset flow.
- /auth/logout/ should revoke refresh tokens.

Views are the "traffic controllers" for these workflows.

Why views exist (why we don't just put everything in models)

It's tempting to write everything in one place, but that becomes hard to scale and easy to break. Good architecture separates responsibilities:

- **Models:** our database structure + core business rules (OTP verification logic, token helpers, user fields).
- **Serializers:** our validation and translation layer (JSON → safe Python → safe DB).
- **Views:** our orchestration layer (HTTP request → run serializer → call actions → return response).

Think of our system as an airport:

- Models are the airport infrastructure (runways, gates, rules).
- Serializers are security screening (check passports, validate baggage).
- Views are air traffic control + the gate staff (they decide which plane you're boarding, when boarding starts, and what happens if something goes wrong).

The view doesn't "invent" the rules; it just coordinates the right pieces at the right time.

What views do in our auth system (high-level)

In an authentication system, views typically do a few consistent jobs:

1) Accept and parse requests

A user sends JSON, and the view reads it (request.data). It also sees metadata like:

- IP address (request.META)
- headers (user-agent)
- authentication state (request.user)

This metadata matters in security systems because identity isn't just "password correct." It also includes context.

2) Select the correct serializer

Views decide which serializer applies:

- Register uses **UserRegistrationSerializer**
- OTP verify uses **VerifyOTPSerializer**
- Login uses **UserLoginSerializer**
- Reset request uses **PasswordResetRequestSerializer**

If our system is a hospital, views are triage nurses. They look at what you came in for and send you to the right specialist.

3) Enforce permissions

DRF lets views declare who is allowed to call them:

- **AllowAny** for register/login/verify/reset request
- **IsAuthenticated** for logout/change password

This is where we decide: "Do we need a valid token to access this endpoint?"

4) Throttle abuse (rate limiting)

One of the biggest advantages of DRF views is built-in throttling. For security endpoints:

- login attempts must be rate-limited to slow brute-force attacks
- OTP verification must be rate-limited to stop guessing
- OTP resend must be rate-limited to stop email spam
- password reset request must be rate-limited to stop enumeration + spam

Important: serializers validate correctness, but views are where throttling is attached via throttle_classes + scope.
Serializers check if the person is legit. Throttling controls crowd flow so one person can't spam the entrance.

5) Log audit events

Views are the best place to log security-relevant events because they see:

- request context (IP, headers)
- whether the action succeeded or failed
- which endpoint was hit

This is why our **UserActivityLog** model pairs naturally with views. Every auth event becomes an auditable footprint.

6) Return clean, consistent responses

Views package the serializer's output into responses with:

- proper HTTP status codes (200, 201, 400, 401, 429)
- consistent JSON format
- safe messaging (not leaking secrets)

This is a big deal: in high-profile systems, error messages must be carefully designed so they help legitimate users without helping attackers.

Why we use DRF GenericAPIView (and what it gives us)

When we use GenericAPIView, we get:

- **serializer_class** management
- **.get_serializer()** convenience
- easy integration with DRF permissions and throttling
- consistent request/response style

It keeps our views short and readable, which matters for security. Long "do-everything" views hide bugs.
GenericAPIView is like using a proven blueprint to build a house. We still design the interior, but we don't reinvent foundation and plumbing every time.

When we dive into our views folder, our readers should expect each view to follow a familiar pattern:

1. Identify the incoming request type (POST/PATCH).
2. Instantiate the serializer with request data.
3. Validate (is_valid(raise_exception=True)).
4. Execute the business action (save() or a service call).
5. Log the outcome (success/failure).
6. Return a response with the right status code.

Once someone understands that pattern, the views become very predictable and easy to reason about, which is exactly what we want in security-critical code.

registration_view.py

```
1. import logging
2.
3. from django.http import HttpRequest
4. from django.utils.translation import gettext_lazy as _
5.
6. from rest_framework import status, permissions, throttling
```

```
from rest_framework.generics import GenericAPIView
from rest_framework.response import Response
from rest_framework.exceptions import ValidationError as DRFValidationError

from accounts.serializers.registration_serializer import UserRegistrationSerializer
from accounts.services.email_services import send_otp_email
from accounts.utils.ip import get_client_ip

logger = logging.getLogger(__name__)


class RegisterThrottle(throttling.AnonRateThrottle):
    scope = "register"


class RegisterView(GenericAPIView):
    """
    Registration endpoint.

    Flow:
    1) Validate incoming registration data via serializer
    2) Create user
    3) Generate OTP and send email
    4) Return a safe success response
```

```
31.     """
32.
33.     serializer_class = UserRegistrationSerializer
34.     permission_classes = [permissions.AllowAny]
35.     throttle_classes = [RegisterThrottle]
36.
37.     def post(self, request: HttpRequest, *args, **kwargs):
38.         ip = get_client_ip(request)
39.         serializer = self.get_serializer(data=request.data)
40.
41.         try:
42.             serializer.is_valid(raise_exception=True)
43.         except DRFValidationError:
44.             logger.warning(
45.                 "[RegisterView] Invalid registration data. ip=%s errors=%s",
46.                 ip,
47.                 serializer.errors,
48.             )
49.             return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
50.
51.         try:
52.             user = serializer.save()
53.             otp = user.generate_and_save_otp()
54.
55.             send_otp_email(
```

```
56.                 to_email=user.email,
57.                 first_name=user.first_name,
58.                 code=otp,
59.             )
60.
61.         except Exception:
62.             logger.exception(
63.                 "[RegisterView] Unexpected error during registration. ip=%s
email=%s",
64.                 ip,
65.                 (request.data.get("email") or "").strip().lower(),
66.             )
67.             return Response(
68.                 {"detail": _("Registration failed. Please try again later.")},
69.                 status=status.HTTP_500_INTERNAL_SERVER_ERROR,
70.             )
71.
72.         return Response(
73.             {"message": _("User registered successfully. Verification code sent to
email.")},
74.             status=status.HTTP_201_CREATED,
75.         )
```

This view is the **registration entry point** for the authentication system. A user sends their signup data to this endpoint. The view then does four things in sequence: it validates the input, creates the user, generates an OTP, sends that OTP by email, and finally returns a success response. In other words, this file is the **traffic controller** of the registration flow. It does not contain all the

business rules itself. Instead, it coordinates the serializer, the user model, the email service, and the IP helper.

A good analogy is a hospital front desk. The receptionist does not perform surgery, run the lab, or dispense medication. The receptionist checks the patient in, sends them to the right department, records what happened, and gives the right response. That is what this view does for registration.

Step-by-step explanation

```
1. import logging
```

This imports Python's logging system. Logging is how the application records important events for debugging, monitoring, and incident response.

In a security system, logging is like a security camera plus incident notebook. When something goes wrong, logs help answer:

- what happened
- when it happened
- from which IP
- whether it was a validation issue or a server issue

```
1. from django.http import HttpRequest
```

This imports Django's request object type. It is mainly used here as a type hint:

```
1. def post(self, request: HttpRequest, *args, **kwargs):
```

That tells readers and tools that request is a Django HTTP request.

Think of it like labeling a folder clearly so anyone opening it knows what kind of document they are dealing with.

```
1. from django.utils.translation import gettext_lazy as _
```

This makes user-facing messages translatable.

So when we write:

```
1. _("Registration failed. Please try again later.")
```

we are saying: this message can later be shown in Arabic, French, English, or any supported language.

This is not directly a security feature, but it is a good engineering practice.

```
1. from rest_framework import status, permissions, throttling
```

These are three DRF modules:

- status gives readable HTTP codes like HTTP_201_CREATED
- permissions controls who can access the endpoint
- throttling helps rate-limit requests

Security-wise, throttling matters a lot. Registration endpoints are common abuse targets for spam accounts and automated bots.

```
1. from rest_framework.generics import GenericAPIView
```

This gives us a good base class for API views. It already knows how to work with serializers, permissions, and request handling.

Instead of building everything from raw Django views, we use a safer, cleaner DRF abstraction.

```
1. from rest_framework.response import Response
```

This is DRF's response object. It returns JSON cleanly and lets us attach status codes.

```
1. from rest_framework.exceptions import ValidationError as
DRFValidationError
```

This imports DRF's validation exception so we can catch serializer validation failures specifically.

That lets us separate:

- bad user input
  from
- real server-side failures

That separation is important both for clarity and security.

```
1. from accounts.serializers.registration_serializer import
UserRegistrationSerializer
```

This imports the serializer responsible for validating registration input and creating the user.

The view does not itself validate passwords, confirm password, email, or other rules. It delegates that to the serializer.

That separation is healthy architecture.

```
1. from accounts.services.email_services import send_otp_email
```

This imports the email service that sends the OTP email.

The view should not build HTML email itself. That belongs in the service layer.

```
1. from accounts.utils.ip import get_client_ip
```

This imports a helper that extracts the client IP safely.

That is better than repeating:

```
1. request.META.get("REMOTE_ADDR")
```

everywhere, especially if proxies or load balancers are involved.

```
1. logger = logging.getLogger(__name__)
```

This creates a logger specific to this module.

__name__ means the logger is labeled with the current file/module name, which makes logs easier to trace.

Throttle class

```
1. class RegisterThrottle(throttling.AnonRateThrottle):
2.     scope = "register"
```

This creates a rate-limit rule for anonymous registration requests.

It says: registration requests should be throttled according to the "register" rate defined in DRF settings.

For example, in settings.py you might have:

```
1. REST_FRAMEWORK = {
2.     "DEFAULT_THROTTLE_RATES": {
3.         "register": "5/min",
4.     }
5. }
```

This is very important for high-security and high-scale systems because attackers often try to mass-create accounts.

Think of it like a front gate that allows only a certain number of people through per minute.

The view class

```
1. class RegisterView(GenericAPIView):
```

This defines the actual registration endpoint.

```
1.     """
2.     Registration endpoint.
3.     Flow:
4.     1) Validate incoming registration data via serializer
5.     2) Create user
6.     3) Generate OTP and send email
7.     4) Return a safe success response
8.     """
```

This docstring explains the flow clearly. Good documentation matters, especially in authentication systems, because the code has several moving parts.

```
1.     serializer_class = UserRegistrationSerializer
```

This tells DRF which serializer this view uses.

That means when we call:

```
1. self.get_serializer(data=request.data)
```

DRF knows which serializer class to instantiate.

```
1.     permission_classes = [permissions.AllowAny]
```

This means anyone can hit the registration endpoint, even if not authenticated.

That is correct, because new users don't have accounts yet.

```
1.     throttle_classes = [RegisterThrottle]
```

This applies the registration throttle to this endpoint.

So even though the endpoint is public, it is not unprotected.

The post() method

```
1.  def post(self, request: HttpRequest, *args, **kwargs):
```

This defines what happens when a POST request comes to the registration endpoint.

POST is correct here because we are creating a new resource: a user account.

```
1.      ip = get_client_ip(request)
```

This extracts the client's IP address.

Why this matters:

- helps with logging
- helps with abuse analysis
- useful later for suspicious activity detection

In security systems, IP is not identity, but it is useful context.

```
1.      serializer = self.get_serializer(data=request.data)
```

This creates the serializer instance using the JSON body sent by the client.

request.data contains the submitted registration form fields like:

- email
- password
- confirm_password
- first_name
- last_name

Validation block

```
1.      try:
2.          serializer.is_valid(raise_exception=True)
```

This asks the serializer to validate all input.

If validation fails, DRF raises a validation exception automatically because **raise_exception=True.**

```
except DRFValidationError:
```

We specifically catch validation problems here.

This is useful because validation failures are expected client-side issues, not "server crashed" situations.

```
logger.warning(
    "[RegisterView] Invalid registration data. ip=%s errors=%s",
    ip,
    serializer.errors,
)
```

This logs the bad request.

It uses warning because the request is invalid, but this is not a server crash.

The log includes:

- IP address
- structured serializer errors

This helps debugging abuse patterns and client mistakes.

```
return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

This returns a 400 response with field-level validation messages.

This is the right behavior because the client submitted invalid data.

User creation + OTP block

```
try:
    user = serializer.save()
```

If validation passed, we ask the serializer to actually create the user.

The serializer usually handles:

- password hashing

- calling the custom manager
- creating the DB record

```
1.          otp = user.generate_and_save_otp()
```

Once the user is created, we generate a one-time verification code and store it securely.

This method should:

- create the OTP
- hash it before saving
- set expiry
- save it to the user

Important: we save only the hash, not the raw code.

That is a very good security pattern.

```
1.          send_otp_email(
2.              to_email=user.email,
3.              first_name=user.first_name,
4.              code=otp,
5.          )
```

This sends the OTP email using the email service.

This should be asynchronous through Celery, which is excellent for scale.

Why?
Because email delivery is slow network I/O. We do not want the registration request waiting around for mail servers.

Think of it this way: the API receptionist should not walk to the post office personally. It should hand the letter to the mailroom.

```
1.      except Exception:
```

This catches unexpected server-side problems.

This is not for validation problems. This is for things like:

- DB failure
- email service failure
- unexpected coding error

```
1.          logger.exception(
2.              "[RegisterView] Unexpected error during registration. ip=%s email=%s",
3.              ip,
4.              (request.data.get("email") or "").strip().lower(),
5.          )
```

logger.exception() is stronger than **logger.error()** because it includes the full traceback.

That is very useful in production logs for debugging.

We log:

- IP
- normalized email

We do **not** log the password or OTP, which is correct.

```
1.          return Response(
2.              {"detail": _("Registration failed. Please try again later.")},
3.              status=status.HTTP_500_INTERNAL_SERVER_ERROR,
4.          )
```

If the server truly failed, we return a generic 500 response.

This is good security hygiene:

- the user gets a clean generic message
- attackers do not get stack traces or internal details
- developers still get the real details in logs

Success response

```
        return Response(
            {"message": _("User registered successfully. Verification code sent to email.")},
            status=status.HTTP_201_CREATED,
        )
```

If everything succeeds:

- user was created
- OTP was generated
- OTP email was queued

Then we return a 201 Created response.

That is the correct HTTP status because a new user resource was created.

otp_view.py

```
import logging

from django.http import HttpRequest

```

```
from rest_framework import status, permissions, throttling
from rest_framework.generics import GenericAPIView
from rest_framework.response import Response
from rest_framework.exceptions import ValidationError as DRFValidationError

from accounts.serializers.otp_serializer import VerifyOTPSerializer, ResendOTPSerializer
from accounts.models.models import UserActivityLog
from accounts.utils.ip import get_client_ip


logger = logging.getLogger(__name__)


class OTPAttemptThrottle(throttling.AnonRateThrottle):
    # OTP verification is usually unauthenticated, so use AnonRateThrottle
    scope = "otp-attempt"


class OTPResendThrottle(throttling.AnonRateThrottle):
    # OTP resend is also unauthenticated (depends on your design)
    scope = "otp-resend"


class VerifyOTPView(GenericAPIView):
    """
```

```
    Verify a user's OTP code and mark email as verified.

    Throttling slows brute-force attempts.
    Logging writes an audit trail of both success and failure.
    """
    serializer_class = VerifyOTPSerializer
    permission_classes = [permissions.AllowAny]
    throttle_classes = [OTPAttemptThrottle]

    def post(self, request, *args, **kwargs):
        ip = get_client_ip(request)
        serializer = self.get_serializer(data=request.data)

        try:
            serializer.is_valid(raise_exception=True)
            result = serializer.save()

            # Audit log: success
            UserActivityLog.objects.create(
                user=getattr(serializer, "user", None),
                ip=ip,
                action=UserActivityLog.ACTION_OTP_VERIFY_SUCCESS,
            )
            logger.info("[OTP Verify Success] ip=%s email=%s", ip,
getattr(serializer.user, "email", None))
            return Response(result, status=status.HTTP_200_OK)
```

```
64.
65.         except DRFValidationError:
66.             # Audit log: failure (avoid storing secrets)
67.             UserActivityLog.objects.create(
68.                 user=getattr(serializer, "user", None),
69.                 ip=ip,
70.                 action=UserActivityLog.ACTION_OTP_VERIFY_FAILURE,
71.                 metadata={"errors": serializer.errors},
72.             )
73.             logger.warning("[OTP Verify Failure] ip=%s errors=%s", ip,
serializer.errors)
74.             return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
75.
76.
77. class ResendOTPView(GenericAPIView):
78.     """
79.     Resend OTP for verification.
80.
81.     High-security posture:
82.     - Response should be generic (anti-enumeration).
83.     - Throttling prevents email spamming.
84.     """
85.     serializer_class = ResendOTPSerializer
86.     permission_classes = [permissions.AllowAny]
87.     throttle_classes = [OTPResendThrottle]
```

```
88.
89.     def post(self, request, *args, **kwargs):
90.         ip = get_client_ip(request)
91.         serializer = self.get_serializer(data=request.data)
92.
93.         try:
94.             serializer.is_valid(raise_exception=True)
95.             result = serializer.save()
96.
97.             # In anti-enumeration mode, serializer may not set serializer.user.
98.             UserActivityLog.objects.create(
99.                 user=getattr(serializer, "user", None),
100.                 ip=ip,
101.                 action=UserActivityLog.ACTION_OTP_RESEND_SUCCESS,
102.                 metadata={"email": request.data.get("email")},
103.             )
104.             logger.info("[OTP Resend] ip=%s email=%s", ip,
request.data.get("email"))
105.             return Response(result, status=status.HTTP_200_OK)
106.
107.         except DRFValidationError:
108.             # Validation error: 400 (NOT 429). 429 is reserved for throttle
violations.
109.             UserActivityLog.objects.create(
110.                 user=getattr(serializer, "user", None),
111.                 ip=ip,
```

```
112.                action=UserActivityLog.ACTION_OTP_RESEND_FAILURE,
113.                metadata={"errors": serializer.errors, "email":
request.data.get("email")},
114.            )
115.            logger.warning("[OTP Resend Failure] ip=%s errors=%s", ip,
serializer.errors)
116.            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

Code Explanation: Step by Step

```
import logging
```

Logging is our audit trail in real time. Even before the database audit log, logs help us detect attacks and debug issues quickly.

```
from django.http import HttpRequest
```

Used for type hints so it's clearer what kind of object **get_client_ip()** expects.

```
from rest_framework import status, permissions, throttling
```

- status gives readable HTTP status codes.
- permissions decides who can access endpoints.
- throttling applies rate limiting, which is crucial for OTP brute-force protection.

```
from rest_framework.generics import GenericAPIView
```

Gives us DRF's standard view base with serializer handling (**get_serializer()**).

```
from rest_framework.response import Response
```

Creates proper JSON responses.

```
from rest_framework.exceptions import ValidationError as DRFValidationError
```

Caught when serializer validation fails.

```
from accounts.serializers.otp_serializer import VerifyOTPSerializer,
ResendOTPSerializer
```

These serializers implement the actual verification and resend logic.

```
from accounts.models.models import UserActivityLog
```

This model stores audit events in the database, useful for compliance, detection, and forensics.

get_client_ip()

We added this because many apps run behind proxies. Relying only on **REMOTE_ADDR** often logs the load balancer IP, not the real user.

We also note: only trust **X-Forwarded-For** if the proxy is trusted and configured correctly.

Throttles: why AnonRateThrottle

UserRateThrottle throttle keys off authenticated users. But OTP endpoints are often called *before login*, so **UserRateThrottle** may not behave as intended.

So we switched to **AnonRateThrottle** to rate-limit per-IP by default.

If you want stricter control, you can implement custom throttle classes that rate-limit per email + IP.

VerifyOTPView success path

We log:

- event type
- IP
- associated user if available
  We do **not** store the OTP code.

VerifyOTPView failure path

We return **serializer.errors** directly. That gives clean errors consistently.

ResendOTPView success path

We return the serializer's result, which in anti-enumeration mode is generic.

We log metadata with the email provided (not guaranteed to exist) which helps diagnostics without exposing existence to the client.

Actual rates must exist in **settings.py**:

```
REST_FRAMEWORK = {
    "DEFAULT_THROTTLE_RATES": {
        "otp-attempt": "5/min",
        "otp-resend": "2/min",
    }
}
```

auth_view.py

```
from django.http import HttpRequest
from django.utils.translation import gettext_lazy as _
import logging
from rest_framework import status, permissions, throttling
from rest_framework.generics import GenericAPIView
from rest_framework.response import Response
from rest_framework.exceptions import ValidationError as DRFValidationError

from accounts.serializers.auth_serializer import UserLoginSerializer, UserLogoutSerializer
from accounts.models.model import UserActivityLog
```

```
from accounts.services.security import is_unusual_login, record_login_country

from accounts.utils.ip import get_client_ip

logger = logging.getLogger(__name__)


class LoginThrottle(throttling.AnonRateThrottle):
    """
    Login happens before authentication, so throttle should be anonymous (per IP).
    """
    scope = "login"


class LoginUserView(GenericAPIView):
    serializer_class = UserLoginSerializer
    permission_classes = [permissions.AllowAny]
    throttle_classes = [LoginThrottle]

    def post(self, request):
        ip = get_client_ip(request)
        serializer = self.get_serializer(data=request.data, context={"request": request})

```

```
34.         # 1) Validate login credentials
35.         try:
36.             serializer.is_valid(raise_exception=True)
37.         except DRFValidationError:
38.             # Log failed login without leaking too much detail
39.             UserActivityLog.objects.create(
40.                 user=None,
41.                 ip=ip,
42.                 action=UserActivityLog.ACTION_LOGIN_FAILURE,
43.                 metadata={
44.                     "email": (request.data.get("email") or "").strip().lower(),
45.                     "errors": serializer.errors,
46.                 },
47.             )
48.             logger.warning("[Login Failed] ip=%s errors=%s", ip, serializer.errors)
49.             return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
50.
51.         # 2) On success, serializer returns tokens and also attaches user in
validated_data
52.         user = serializer.validated_data.get("user")
53.
54.         # 3) Detect unusual login (GeoIP country change)
55.         try:
56.             if user and is_unusual_login(user, ip):
57.                 UserActivityLog.objects.create(
```

```
58.                 user=user,
59.                 ip=ip,
60.                 action=UserActivityLog.ACTION_UNUSUAL_LOGIN,
61.                 metadata={"signal":"country_change"},
62.             )
63.             logger.warning("[Unusual Login] user=%s ip=%s", getattr(user,
"email", None), ip)
64.         except Exception:
65.             # Never break login due to geoip issues
66.             logger.exception("[GeoIP] Unusual login check failed. ip=%s", ip)
67.
68.         # 4) Record login country (best-effort; should not block login)
69.         try:
70.             if user:
71.                 record_login_country(user, ip)
72.         except Exception:
73.             logger.exception("[GeoIP] Record login country failed. user=%s
ip=%s", getattr(user,"email", None), ip)
74.
75.         # 5) Audit log: success
76.         UserActivityLog.objects.create(
77.             user=user,
78.             ip=ip,
79.             action=UserActivityLog.ACTION_LOGIN_SUCCESS,
80.         )
```

```
81.         logger.info("[Login Success] user=%s ip=%s", getattr(user, "email", None),
ip)
82.
83.         # Return token payload (serializer.data has
access/refresh/full_name/user_status)
84.         return Response(serializer.data, status=status.HTTP_200_OK)
85.
86.
87. class LogoutUserView(GenericAPIView):
88.     serializer_class = UserLogoutSerializer
89.     permission_classes = [permissions.IsAuthenticated]
90.
91.     def post(self, request):
92.         ip = get_client_ip(request)
93.         serializer = self.get_serializer(data=request.data, context={"request":
request})
94.         serializer.is_valid(raise_exception=True)
95.         serializer.save()
96.
97.         UserActivityLog.objects.create(
98.             user=request.user,
99.             ip=ip,
100.             action=UserActivityLog.ACTION_LOGOUT,
101.         )
102.         logger.info("[Logout] user=%s ip=%s", getattr(request.user, "email",
None), ip)
```

```
103.
104.         return Response({"message": _("Successfully logged out.")},
status=status.HTTP_200_OK)
```

Code Explanation: Step by Sep

```
import logging
```

Logging is our "flight recorder." Authentication systems must be observable because attacks and suspicious behavior show up as patterns over time.

```
from django.http import HttpRequest
```

Used for type hints and clarity in get_client_ip().

```
from django.utils.translation import gettext_lazy as _
```

Makes response messages translatable.

from rest_framework import status, permissions, throttling

- status: readable HTTP codes.
- permissions: controls who can call endpoints.
- throttling: rate-limits endpoints to slow brute force.

```
from rest_framework.generics import GenericAPIView
from rest_framework.response import Response
```

- GenericAPIView gives standard serializer + permission handling.
- Response returns structured JSON.

```
from rest_framework.exceptions import ValidationError as DRFValidationError
```

Catches serializer input problems.

```
from accounts.serializers.auth_serializer import UserLoginSerializer,
UserLogoutSerializer
```

Login serializer validates credentials and issues tokens.
Logout serializer validates refresh token and blacklists it.

```
from accounts.models.models import UserActivityLog
```

Stores auditable security events.

```
from accounts.services.security import is_unusual_login, record_login_country
```

GeoIP security signals: detect country changes and store last login country.

IP extraction

```
1. ip = request.META.get('REMOTE_ADDR')
```

This is often wrong behind proxies. We added **get_client_ip()** and optionally use **X-Forwarded-For.**

GeoIP failures should not break login

If GeoIP DB is missing or IP lookup fails, login should still succeed. We wrapped GeoIP calls in try/except.

Better audit logging + safer metadata

We:

- log only what we need (email, errors)
- do not log passwords or tokens
- keep logs consistent with model constants

Logout returns a consistent message

Also logs it in UserActivityLog.

password_view.py

```
1. import logging
2.
3. from django.http import HttpRequest
```

```
from django.utils.translation import gettext_lazy as _

from rest_framework import status, permissions, throttling
from rest_framework.generics import GenericAPIView
from rest_framework.response import Response
from rest_framework.exceptions import ValidationError as DRFValidationError

from accounts.serializers.password_reset_serializer import (
    PasswordResetRequestSerializer,
    SetNewPasswordSerializer,
    ChangePasswordSerializer,
)

from accounts.models.model import UserActivityLog
from accounts.utils.ip import get_client_ip


logger = logging.getLogger(__name__)


class PasswordResetThrottle(throttling.AnonRateThrottle):
    # Rate-limit reset requests to prevent email spam and abuse
    scope = "password-reset"

```

```
29. class PasswordResetConfirmThrottle(throttling.AnonRateThrottle):
30.     # Optional: slows token guessing or repeated attempts
31.     scope = "password-reset-confirm"
32. 
33. 
34. class ChangePasswordThrottle(throttling.UserRateThrottle):
35.     # Authenticated action: rate limit per-user
36.     scope = "password-change"
37. 
38. 
39. class PasswordResetRequestView(GenericAPIView):
40.     """
41.     Starts a password reset flow.
42.     Always responds success message to prevent user enumeration.
43.     """
44.     serializer_class = PasswordResetRequestSerializer
45.     permission_classes = [permissions.AllowAny]
46.     throttle_classes = [PasswordResetThrottle]
47. 
48.     def post(self, request, *args, **kwargs):
49.         ip = get_client_ip(request)
50.         serializer = self.get_serializer(data=request.data, context={"request":
request})
51. 
52.         try:
53.             serializer.is_valid(raise_exception=True)
```

```
54.             serializer.save()
55.         except DRFValidationError:
56.             # Still safe to return validation errors (e.g. invalid email format)
57.             UserActivityLog.objects.create(
58.                 user=None,
59.                 ip=ip,
60.                 action=UserActivityLog.ACTION_PASSWORD_RESET_REQUEST,
61.                 metadata={"errors": serializer.errors},
62.             )
63.             return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
64.
65.         # Audit log (do NOT log reset links/tokens)
66.         UserActivityLog.objects.create(
67.             user=None,
68.             ip=ip,
69.             action=UserActivityLog.ACTION_PASSWORD_RESET_REQUEST,
70.             metadata={"email": (request.data.get("email") or "").strip().lower()},
71.         )
72.         logger.info("[Password Reset Requested] ip=%s email=%s", ip,
request.data.get("email"))
73.
74.         return Response(
75.             {"message": _("If that email exists, a password reset link has been
sent.")},
76.             status=status.HTTP_200_OK,
```

```
        )


class SetNewPasswordView(GenericAPIView):
    """
    Completes password reset using uidb64 + token + new password.
    """
    serializer_class = SetNewPasswordSerializer
    permission_classes = [permissions.AllowAny]
    throttle_classes = [PasswordResetConfirmThrottle]

    def patch(self, request, *args, **kwargs):
        ip = get_client_ip(request)
        data = request.data.copy()
        data['uidb64'] = kwargs.get('uidb64')
        data['token'] = kwargs.get('token')
        serializer = self.get_serializer(data=data)

        try:
            serializer.is_valid(raise_exception=True)
            user = serializer.save()
        except DRFValidationError:
            # Avoid storing secrets; log only generic validation errors
            UserActivityLog.objects.create(
                user=None,
```

```
                ip=ip,
action=UserActivityLog.ACTION_PASSWORD_RESET_COMPLETE,
                metadata={"errors": serializer.errors},
            )
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)

        UserActivityLog.objects.create(
            user=user,
            ip=ip,
            action=UserActivityLog.ACTION_PASSWORD_RESET_COMPLETE,
        )
        logger.info("[Password Reset Complete] user=%s ip=%s", getattr(user,
"email", None), ip)

        return Response({"message": _("Password reset successful.")},
status=status.HTTP_200_OK)


class ChangePasswordView(GenericAPIView):
    """
    Authenticated password change (requires old password).
    """
    serializer_class = ChangePasswordSerializer
    permission_classes = [permissions.IsAuthenticated]
```

```
    throttle_classes = [ChangePasswordThrottle]

    def patch(self, request, *args, **kwargs):
        ip = get_client_ip(request)
        serializer = self.get_serializer(data=request.data, context={"request": request})

        try:
            serializer.is_valid(raise_exception=True)
            user = serializer.save()
        except DRFValidationError:
            # Audit failed attempts too (do NOT log passwords)
            UserActivityLog.objects.create(
                user=request.user,
                ip=ip,
                action=UserActivityLog.ACTION_PASSWORD_CHANGE,
                metadata={"errors": serializer.errors},
            )
            return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

        UserActivityLog.objects.create(
            user=user,
            ip=ip,
            action=UserActivityLog.ACTION_PASSWORD_CHANGE,
        )
```

```
148.         logger.info("[Password Changed] user=%s ip=%s", getattr(user, "email", None), ip)
149.
150.         return Response({"message": _("Password changed successfully.")}, status=status.HTTP_200_OK)
```

Code Explanation: Step by Step

```
import logging
```

This imports Python's built-in logging system. Logging is like the **security camera** + **incident report book** for our app.

- When something succeeds (password reset requested), we log it.
- When something fails (bad request, suspicious activity), we log it.
- When something breaks (unexpected error), we log the stack trace.

In high-security systems, logs are not optional: they're essential for **forensics** and **monitoring**.

```
from django.http import HttpRequest
```

This imports Django's request object type.

We mostly use it here for **type hints**, like:

```
def get_client_ip(request: HttpRequest)
```

This makes the code more readable and helps tools (and learners) understand what **request** really is.

It's like labeling a box "this contains medical supplies" so everyone handling it knows what to expect.

```
from django.utils.translation import gettext_lazy as _
```

This is the translation helper. It marks text as translatable, without translating immediately.

So when we write:

```
_("Password reset successful.")
```

we're saying: "This message can be shown in English now, but later it could automatically show in French, Arabic, etc."

It's like writing a message on a sign but also tagging it with "this sign can be printed in multiple languages later."

```
from rest_framework import status, permissions, throttling
```

This imports three important DRF modules:

- **status:** readable HTTP codes like HTTP_200_OK instead of writing 200.
- **permissions:** decides who can call an endpoint.
- **throttling:** rate-limiting (prevents abuse).

Think of it like this:

- **permissions** = who is allowed into the building.
- **throttling** = how many times someone is allowed to knock on the door per minute.
- **status** = the official response stamp on the reply envelope (200 OK, 400 Bad Request, etc.).

```
from rest_framework.generics import GenericAPIView
```

This is a DRF base class for API views. It provides:

- serializer loading **(get_serializer)**
- consistent request handling
- integration with permissions and throttles

Think of it like this, Instead of building a car from scratch, we're using a reliable chassis and focusing on the features.

```
from rest_framework.response import Response
```

DRF's Response class returns proper API responses (usually JSON) and handles rendering nicely.

It's the standardized "envelope" we use to send responses back to the client.

```
from rest_framework.exceptions import ValidationError as DRFValidationError
```

This is the error DRF raises when input doesn't pass validation rules in serializers.

We import it so we can catch it and also log audit events on failures.

```
from accounts.serializers.password_reset_serializer import (
    PasswordResetRequestSerializer,
    SetNewPasswordSerializer,
    ChangePasswordSerializer,
)
```

These serializers are the "brains" of each password flow:

- **PasswordResetRequestSerializer:** receives email, silently sends link if user exists
- **SetNewPasswordSerializer:** verifies token + sets new password
- **ChangePasswordSerializer:** requires old password + sets new password

Serializers are like **trained clerks** who check forms and verify rules. Views are the **receptionist** who routes forms to the right clerk.

```
from accounts.models.models import UserActivityLog
```

This imports our audit log model.

We use it to store security events:

- reset requested
- reset complete
- password changed
- failures

It's like the official **logbook** the security team keeps.

```
logger = logging.getLogger(__name__)
```

This creates a logger named after the module file (e.g., accounts.views.password_view).

Using **__name__** is standard because it helps you know *which file* generated a log.

It's like stamping each incident report with the department name that filed it.

IP extraction helper

def get_client_ip(request: HttpRequest) -> str:

We're defining a helper function that returns a string IP address for the user.

Throttle classes (rate limiting)

```
1. class PasswordResetThrottle(throttling.AnonRateThrottle):
```

This creates a throttle for **anonymous users**.

Password reset request is usually anonymous because the user might be logged out.

```
1.   scope = "password-reset"
```

This links the throttle to a rate in settings like:

```
1. "password-reset": "3/min"
```

The door guard knows "password reset requests" is a sensitive door, so it sets strict limits.

```
1. class PasswordResetConfirmThrottle(throttling.AnonRateThrottle):
```

This is another throttle for the reset confirmation step (token + new password).

Even though it's not common to brute force tokens, throttling adds extra safety.

```
1. scope = "password-reset-confirm"
```

Rate config in settings:

```
1. "password-reset-confirm": "5/min"
```

```
1. class ChangePasswordThrottle(throttling.UserRateThrottle):
```

Change password requires login, so we throttle per authenticated user.

```
    scope = "password-change"
```

Rate config example:

```
"password-change": "5/min"
```

View 1: PasswordResetRequestView

```
class PasswordResetRequestView(GenericAPIView):
```

This defines the endpoint that starts password reset.

```
    """
    Starts a password reset flow.
    Always responds success message to prevent user enumeration.
    """
```

“User enumeration” means attackers test emails:

- If system says “email exists” → attacker learns which emails are real accounts.
  So we always respond success.

```
    serializer_class = PasswordResetRequestSerializer
```

This tells DRF which serializer to use.

```
    permission_classes = [permissions.AllowAny]
```

Anyone can request a reset link (even logged out).

```
    throttle_classes = [PasswordResetThrottle]
```

Protects endpoint from abuse/spam.

```
    def post(self, request, *args, **kwargs):
```

HTTP POST endpoint.

```
        ip = get_client_ip(request)
```

Extracts IP for logging/auditing.

```
1.        serializer = self.get_serializer(data=request.data, context={"request": request})
```

Creates serializer with incoming JSON (request.data).

We pass **context={"request": request}** because serializer may need request to build absolute reset links.

```
1.        try:
2.            serializer.is_valid(raise_exception=True)
3.            serializer.save()
```

- **is_valid()** runs validations (valid email formatting)
- **save()** sends reset email if user exists (silently)

```
1.        except DRFValidationError:
```

If invalid input (e.g., email not valid format), we catch it.

```
1.            UserActivityLog.objects.create(
2.                user=None,
3.                ip=ip,
4.                action=UserActivityLog.ACTION_PASSWORD_RESET_REQUEST,
5.                metadata={"errors": serializer.errors},
6.            )
```

We write an audit log event.

- **user=None** because we might not know which user this is
- we include the IP
- we store errors (like "invalid email format")

We do **not** store tokens, passwords, or reset links.

```
1.    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

400 means "client gave bad input".

```
1.        UserActivityLog.objects.create(
```

```
            user=None,
            ip=ip,
            action=UserActivityLog.ACTION_PASSWORD_RESET_REQUEST,
            metadata={"email": (request.data.get("email") or "").strip().lower()},
        )
```

If validation succeeded, we log the request as a security event.

We normalize the email for consistent logs.

```
        logger.info("[Password Reset Requested] ip=%s email=%s", ip, request.data.get("email"))
```

This logs to the server logs (separate from database audit log).

- DB log = long-term audit
- server log = operational monitoring

```
        return Response(
            {"message": _("If that email exists, a password reset link has been sent.")},
            status=status.HTTP_200_OK,
        )
```

Always returns success wording to prevent enumeration.

View 2: SetNewPasswordView

```
class SetNewPasswordView(GenericAPIView):
```

This endpoint completes reset.

```
    serializer_class = SetNewPasswordSerializer
    permission_classes = [permissions.AllowAny]
    throttle_classes = [PasswordResetConfirmThrottle]
```

- uses the confirm serializer
- accessible publicly

- throttled to slow abuse

```
1.    def patch(self, request, *args, **kwargs):
```

PATCH means “update existing resource”, here, password update.

```
1.        ip = get_client_ip(request)
2.        serializer = self.get_serializer(data=request.data)
```

Get IP and build serializer using posted token+passwords.

```
1.        try:
2.            serializer.is_valid(raise_exception=True)
3.            user = serializer.save()
```

Validate token + update password.

save() returns the user whose password was reset.

```
1.        except DRFValidationError:
2.            UserActivityLog.objects.create(
3.                user=None,
4.                ip=ip,
5.                action=UserActivityLog.ACTION_PASSWORD_RESET_COMPLETE,
6.                metadata={"errors": serializer.errors},
7.            )
8.            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

On failure:

- log attempt
- return validation errors

Again: never store raw tokens

.

```
1.        UserActivityLog.objects.create(
```

```
            user=user,
            ip=ip,
            action=UserActivityLog.ACTION_PASSWORD_RESET_COMPLETE,
        )
```

On success:

- log completion with actual user

```
        logger.info("[Password Reset Complete] user=%s ip=%s", getattr(user, "email", None), ip)
```

Operational log.

getattr prevents crash if user lacks email.

```
        return Response({"message": _("Password reset successful.")}, status=status.HTTP_200_OK)
```

Success response.

View 3: ChangePasswordView

```
class ChangePasswordView(GenericAPIView):
```

Authenticated password change.

```
    serializer_class = ChangePasswordSerializer
    permission_classes = [permissions.IsAuthenticated]
    throttle_classes = [ChangePasswordThrottle]
```

- only logged-in users
- throttled per user

```
    def patch(self, request, *args, **kwargs):
```

Again using PATCH to update password.

```
        ip = get_client_ip(request)
        serializer = self.get_serializer(data=request.data, context={"request": request})
```

We pass request context because serializer needs **request.user** to check old password.

```
        try:
            serializer.is_valid(raise_exception=True)
            user = serializer.save()
```

Validates:

- old password correct
- new passwords match
- new password strong
  Then saves.

```
        except DRFValidationError:
            UserActivityLog.objects.create(
                user=request.user,
                ip=ip,
                action=UserActivityLog.ACTION_PASSWORD_CHANGE,
                metadata={"errors": serializer.errors},
            )
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

On failure, we still log it because repeated failures can be suspicious.
We store only error messages, never passwords.

```
        UserActivityLog.objects.create(
            user=user,
            ip=ip,
            action=UserActivityLog.ACTION_PASSWORD_CHANGE,
        )
```

On success, log the password change.

```
1.          logger.info("[Password Changed] user=%s ip=%s", getattr(user, "email", None), ip)
```

Operational log.

```
1.          return Response({"message": _("Password changed successfully.")}, status=status.HTTP_200_OK)
```

Success response.

Big picture: what this file is doing

This file builds three "doors" in our authentication building:

1. **Reset Request Door**
   User asks for a reset link → we send it if user exists, but always respond success.
2. **Reset Confirm Door**
   User comes with link token + new password → we verify token + set password.
3. **Change Password Door**
   Logged-in user proves old password → we set new password.

Each door:

- has rate limits
- logs important actions
- returns safe messages
- avoids leaking secrets

Services

Our **services**/ folder is where we keep the parts of the system that do "real work" but don't neatly belong to a **model, serializer, or view.**

If views are the **traffic controllers** (handling requests), serializers are the **gatekeepers** (validating and cleaning data), and models are the **database blueprints** (storing and enforcing core rules), then services are the **specialist teams** we call when we need an operation performed, sending emails, generating security signals, checking password strength, geo-locating logins, and so on.

Why we create a services/ folder

In a beginner project, it's common to dump logic into views or serializers because it "works." But in serious systems, that becomes a problem:

- views become huge and hard to read
- logic gets duplicated in multiple places
- testing becomes painful
- security fixes become risky because you have to patch the same behavior in many files

A services layer solves that by centralizing reusable operations into clean, focused modules.
Think of our authentication system like a company building.

- **Views** are the reception desk: they accept requests and route them.
- **Serializers** are the document verification desk: they check forms for correctness.
- **Models** are the company records department: they store official data and rules.

- **Services** are the internal departments that do specialized work: **security**, **email**, **fraud detection**, **notifications**.

The receptionist shouldn't be the one sending emails or running fraud checks. They should call the right department. That's what our service layer does.

Why services are especially important in authentication systems

Authentication is not just "check password." It includes:

- password strength scoring
- OTP generation and delivery
- token workflows
- geo-risk checks
- audit logging signals
- anomaly detection hooks

These are *security-critical* tasks that should be:

- consistent (one source of truth)
- testable (unit tests can target them directly)
- easy to upgrade (swap email providers, add MFA, add device fingerprinting)
- reusable (used by multiple endpoints)

Putting this logic into a services layer is one of the biggest steps from "toy auth" to "production-grade auth."

email_services.py

```
1. from __future__ import annotations
2.
3. from typing import Optional
4.
5. from celery import shared_task
```

```python
from django.conf import settings
from django.contrib.auth.tokens import PasswordResetTokenGenerator
from django.core.mail import EmailMultiAlternatives
from django.utils.encoding import force_bytes
from django.utils.http import urlsafe_base64_encode


def _get_from_email() -> str:
    """
    Central place to fetch DEFAULT_FROM_EMAIL.
    Fail fast if misconfigured (better than silently dropping mail).
    """
    from_email = getattr(settings, "DEFAULT_FROM_EMAIL", "")
    if not from_email:
        raise RuntimeError("DEFAULT_FROM_EMAIL is not configured")
    return from_email


def _safe_frontend_url() -> str:
    """
    Returns FRONTEND_URL with basic sanity checks.
    Prevents mistakes like missing scheme (http/https) in production.
    """
    url = getattr(settings, "FRONTEND_URL", "").rstrip("/")
    if not url:
```

```
        raise RuntimeError("FRONTEND_URL is not configured")
    if not (url.startswith("http://") or url.startswith("https://")):
        raise RuntimeError("FRONTEND_URL must start with http:// or https://")
    return url


@shared_task(
    bind=True,
    autoretry_for=(Exception,),
    retry_backoff=True,
    retry_jitter=True,
    retry_kwargs={"max_retries": 5},
)
def send_email_task(
    self,
    *,
    subject: str,
    to_email: str,
    text_content: str,
    html_content: Optional[str] = None,
) -> None:
    """
    Send email in background via Celery.

    Security:
```

```
    - We never log OTPs/tokens here.
    - We fail loudly if the mail config is broken (so we notice).

    Performance/scale:
    - Async task keeps API fast.
    - Automatic retries handle transient SMTP/provider issues.
    """
    from_email = _get_from_email()

    # Always send a text version (deliverability + accessibility)
    msg = EmailMultiAlternatives(subject, text_content, from_email, [to_email])

    # Optional HTML for nicer emails
    if html_content:
        msg.attach_alternative(html_content, "text/html")

    # fail_silently=False so celery can retry on failure
    msg.send(fail_silently=False)


def send_otp_email(*, to_email: str, first_name: str = "", code: str, expires_minutes: int = 5) -> None:
    """
    Queue an OTP verification email.

```

```
    Security:
    - We don't include any extra sensitive metadata in the task call.
    - OTP expires quickly (ex: 5 minutes).

    Scale:
    - We pass only primitive values (strings/ints) to Celery (safer serialization).
    """
    subject = "Verify Your Email Address"

    safe_name = (first_name or "there").strip()[:60]  # guard against huge names
    text_content = f"Your verification code is {code}. It expires in {expires_minutes} minutes."

    html_content = f"""
    <div style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto; padding: 20px;">
        <h2 style="color: #333;">Verify Your Email Address</h2>
        <p>Hi {safe_name},</p>
        <p>Your verification code is:</p>
        <div style="background-color: #f4f4f4; padding: 10px; margin: 15px 0;
                    text-align: center; font-size: 24px; letter-spacing: 5px;">
            <strong>{code}</strong>
        </div>
        <p>This code will expire in {expires_minutes} minutes.</p>
        <p>If you didn’t request this code, please ignore this email.</p>
```

```
    </div>
    """

    send_email_task.delay(
        subject=subject,
        to_email=to_email,
        text_content=text_content,
        html_content=html_content,
    )


def build_password_reset_link(user) -> str:
    """
    Build a password reset link for the FRONTEND, containing uidb64 + token.

    Security:
    - Token is generated with Django's PasswordResetTokenGenerator.
    - Token automatically becomes invalid if password changes, etc.
    """
    frontend = _safe_frontend_url()

    uidb64 = urlsafe_base64_encode(force_bytes(user.pk))
    token = PasswordResetTokenGenerator().make_token(user)

    # This should match our React route:
```

```
    # /reset-password/:uidb64/:token
    return f"{frontend}/reset-password/{uidb64}/{token}"


def send_reset_email(*, to_email: str, link: str) -> None:
    """
    Queue a password reset email (link-based).

    Security:
    - We do NOT log the link anywhere (it contains a reset token).
    - Email content should be generic and safe.
    """
    subject = "Reset Your Password"
    text_content = f"Reset your password using this link: {link}"

    html_content = f"""
    <div style="font-family: Arial, sans-serif; max-width: 600px; margin: auto; padding: 20px;">
        <h2>Password Reset Request</h2>
        <p>We received a request to reset your password. Click the link below:</p>
        <p>
            <a href="{link}" style="display:inline-block; padding:10px 20px;
                background-color:#4CAF50; color:white; text-decoration:none; border-radius:5px;">
                Reset Password
```

```
            </a>
        </p>
        <p>If you didn't request this, you can ignore this email.</p>
    </div>
    """

    send_email_task.delay(
        subject=subject,
        to_email=to_email,
        text_content=text_content,
        html_content=html_content,
    )
```

Think of this file as **our "mailroom + courier dispatch"**:

- **Mailroom**: prepares the email content (OTP email, reset email)
- **Courier dispatch**: hands the email to Celery (background worker) so our API doesn't "stand in line at the post office"
- **Security rules**: avoids logging secrets (OTP/token links), prevents misconfiguration mistakes
- **Scale**: uses async tasks + retry logic so email sending stays reliable under heavy traffic

Line-by-line explanation

```
1. from __future__ import annotations
```

- Allows using type hints like User | None (modern typing) *without importing the class at runtime.*
- Helps avoid circular import issues and keeps imports lighter.

```
from typing import Optional
```

- Optional[str] means: "this can be a string or None."
- Used for html_content since some emails may not include HTML.

```
from celery import shared_task
```

- Celery is our "background worker system".
- **shared_task** turns a function into an async task we can queue with **.delay()**.

```
from django.conf import settings
```

- Gives access to our settings like **DEFAULT_FROM_EMAIL, FRONTEND_URL.**

```
from django.contrib.auth.tokens import PasswordResetTokenGenerator
```

- Django's built-in secure token generator for password resets.
- Tokens expire and automatically become invalid if password changes (and other security checks).

```
from django.core.mail import EmailMultiAlternatives
```

- Django email utility that supports **plain text** + **HTML** in the same email.

```
from django.utils.encoding import force_bytes
from django.utils.http import urlsafe_base64_encode
```

- Helps build a safe version of the user's ID (uidb64) to put into URLs.
- **urlsafe_base64_encode()** ensures the value won't break a URL.

```
_get_from_email()
def _get_from_email() -> str:
```

- Private helper function (underscore = internal use).
- Returns a string (the "from" email address).

```
    """
    Central place to fetch DEFAULT_FROM_EMAIL.
```

```
3.    Fail fast if misconfigured (better than silently dropping mail).
4.    """
```

- This explains why we do it:
    - If email config is broken, it's safer to crash loudly than silently "pretend email was sent."

```
1.    from_email = getattr(settings, "DEFAULT_FROM_EMAIL", "")
```

- Reads **DEFAULT_FROM_EMAIL** from Django settings.
- If missing, returns empty string instead of crashing immediately.

```
1.    if not from_email:
2.        raise RuntimeError("DEFAULT_FROM_EMAIL is not configured")
```

- If empty, we raise an error immediately.
- This protects us from "emails not sending but nobody knows."

```
1.    return from_email
```

- Returns the email address.

This is like checking we actually have a **return address label** before sending mail.

_safe_frontend_url()

```
1. def _safe_frontend_url() -> str:
```

- Another private helper: gets the frontend base URL.

```
1.    """
2.    Returns FRONTEND_URL with basic sanity checks.
3.    Prevents mistakes like missing scheme (http/https) in production.
4.    """
```

- Ensures we don't generate broken reset links.

```
1.    url = getattr(settings, "FRONTEND_URL", "").rstrip("/")
```

- Reads **FRONTEND_URL**, removes trailing / so links don't become //reset-password/....

```
if not url:
    raise RuntimeError("FRONTEND_URL is not configured")
```

- Must exist, otherwise our reset link cannot work.

```
if not (url.startswith("http://") or url.startswith("https://")):
    raise RuntimeError("FRONTEND_URL must start with http:// or https://")
```

- Prevents **localhost:5173** (missing scheme) from producing invalid links.
- In production, we should use https://....

```
return url
```

- Returns the validated base URL.

This is like verifying our "delivery address format" before we print shipping labels.

Celery task: **send_email_task**

@shared_task(

- Decorator that registers this function as a Celery task.

bind=True,

- Gives the task access to self (Celery task instance).
- Helpful when we want retries, task metadata, etc.

autoretry_for=(Exception,),

retry_backoff=True,

retry_jitter=True,

retry_kwargs={"max_retries": 5},

)

- **autoretry_for**: retry if any exception occurs.
- **retry_backoff:** waits longer each retry (exponential backoff).

- **retry_jitter**: adds randomness so many retries don't happen at the same second (avoids "retry stampede").
- **max_retries=5**: prevents infinite retry loops.

If the courier can't deliver now (email provider down), we automatically try again later instead of giving up instantly.

```
def send_email_task(
    self,
    *,
    subject: str,
    to_email: str,
    text_content: str,
    html_content: Optional[str] = None,
) -> None:
```

- This is the async task itself.
- The * forces all parameters to be **keyword-only**. That prevents mistakes like swapping to_email and subject.
- Returns nothing (None).

```
    """
    Send email in background via Celery.
    ...
    """
```

- Explains security and scale reasons.

```
    from_email = _get_from_email()
```

- Gets validated sender email.

```
    msg = EmailMultiAlternatives(subject, text_content, from_email, [to_email])
```

- Creates the email object.
- Always includes plain text (text_content).

```
1.  if html_content:
2.      msg.attach_alternative(html_content, "text/html")
```

- If HTML exists, attach it as an alternative.
- Email clients choose the best version.

```
1.  msg.send(fail_silently=False)
```

- Sends email.
- **fail_silently=False** means: if it fails, Celery sees the error → retry happens.

send_otp_email(...)

```
1. def send_otp_email(*, to_email: str, first_name: str = "", code: str, expires_minutes: int = 5) -> None:
```

- Sends an OTP email.
- Again * forces keyword args for safety.
- Takes only **primitive values** (strings/ints) → best for Celery serialization.

```
1.  subject = "Verify Your Email Address"
```

- Email subject.

```
1.  safe_name = (first_name or "there").strip()[:60]
```

- Uses "there" if we don't have a name.
- Strips whitespace.
- Limits to 60 chars (prevents huge/untrusted strings from breaking our email layout).

```
1.  text_content = f"Your verification code is {code}. It expires in {expires_minutes} minutes."
```

- Plain text version (best for deliverability and accessibility).

```
1.  html_content = f""" ... """
```

- HTML version for nicer formatting.

```
1.  send_email_task.delay(...)
```

- Queues the email to Celery.
- Our API returns instantly instead of waiting for SMTP.

build_password_reset_link(user)

```
1. def build_password_reset_link(user) -> str:
```

- Builds a link pointing to **our frontend reset page**, not backend.

```
1.  frontend = _safe_frontend_url()
```

- Gets validated frontend base URL.

```
1.  uidb64 = urlsafe_base64_encode(force_bytes(user.pk))
```

- Converts user ID into a URL-safe string.
- **force_bytes** ensures it's bytes so base64 encoding works.

```
1.  token = PasswordResetTokenGenerator().make_token(user)
```

- Generates secure reset token tied to the user.
- Automatically invalidates after password change and time rules.

```
1.  return f"{frontend}/reset-password/{uidb64}/{token}"
```

- Produces a link matching our React route:
    - /reset-password/:uidb64/:token

This is like generating a **one-time entry pass** plus the **ID badge number**, then printing both into a link.

send_reset_email(...)

```
1. def send_reset_email(*, to_email: str, link: str) -> None:
```

- Queues password reset email.

```
1.  subject = "Reset Your Password"
```

- Subject.

```
1.    text_content = f"Reset your password using this link: {link}"
```

- Plain text fallback.

```
1.    html_content = f""" ... """
```

- HTML button email.

```
1.    send_email_task.delay(...)
```

- Sends asynchronously via Celery.

service/otp_service.py

```
 1. from __future__ import annotations
 2.
 3. from dataclasses import dataclass
 4. from typing import Optional
 5.
 6. from django.core.exceptions import ValidationError
 7. from django.utils.translation import gettext_lazy as _
 8.
 9. from accounts.services.email_services import send_otp_email
10.
11.
12. @dataclass(frozen=True)
13. class OtpResult:
14.     sent: bool
15.     email: str
16.     otp_code: Optional[str] = None  # only for tests/dev when explicitly
requested
```

```
17.
18.
19. def send_verification_otp(user, *, expiry_minutes: int = 5, return_code: bool =
False) -> OtpResult:
20.     """
21.     Generate + save OTP on user model, then email it.
22.
23.     This uses the User model's generate_and_save_otp(), which stores:
24.     - otp_hash (hashed)
25.     - otp_expiry (time-bound)
26.
27.     Args:
28.         expiry_minutes: how long OTP remains valid in DB
29.         return_code: if True returns OTP (ONLY for tests/dev)
30.
31.     Returns:
32.         OtpResult
33.     """
34.     code = user.generate_and_save_otp(expiry_minutes=expiry_minutes)
35.
36.     send_otp_email(
37.         to_email=user.email,
38.         first_name=user.first_name,
39.         code=code,
40.         expires_minutes=expiry_minutes,
41.     )
```

```
42.
43.     return OtpResult(
44.         sent=True,
45.         email=user.email,
46.         otp_code=code if return_code else None,
47.     )
48.
49.
50. def resend_user_otp(
51.     user,
52.     *,
53.     cooldown_minutes: int = 1,
54.     expiry_minutes: int = 5,
55.     return_code: bool = False,
56. ) -> OtpResult:
57.     """
58.     Resend OTP with cooldown enforcement.
59.
60.     Raises:
61.         ValidationError if cooldown not satisfied.
62.     """
63.     if not user.can_resend_otp(cooldown_minutes=cooldown_minutes):
64.         raise ValidationError(_("You must wait before requesting a new OTP."))
65.
66.     code = user.generate_and_save_otp(expiry_minutes=expiry_minutes)
```

```
67.
68.     send_otp_email(
69.         to_email=user.email,
70.         first_name=user.first_name,
71.         code=code,
72.         expires_minutes=expiry_minutes,
73.     )
74.
75.     return OtpResult(
76.         sent=True,
77.         email=user.email,
78.         otp_code=code if return_code else None,
79.     )
```

This file is the **OTP coordination layer**.

The OTP itself is not fully handled here. The user model already knows how to:

- generate an OTP
- hash it
- set its expiry
- save it into the database

The email service already knows how to:

- format the OTP email
- queue it to Celery
- send it asynchronously

So what does **otp_service.py** do?

It acts like a dispatcher between those two systems.

Think of it like this:

- the **user model** is the vault that creates and stores the one-time code securely
- the **email service** is the courier that delivers the message
- the **OTP service** is the coordinator that says: "create the code, store it correctly, and send it to the user"

That is why this file belongs in **services**/. It is not just a tiny helper. It expresses a real business workflow.

Line-by-line explanation

```
1. from __future__ import annotations
```

This enables postponed evaluation of type hints.

It helps modern Python typing behave more flexibly and avoids some circular import issues.

```
1. from dataclasses import dataclass
```

This imports dataclass, which is a Python feature used to create lightweight structured objects.

Instead of returning a plain dictionary like:

```
1. {"sent": True, "email": "...", "otp_code": "..."}
```

we return a structured object called **OtpResult.**

That makes the return value clearer and more self-documenting.

```
1. from typing import Optional
```

This allows us to say a field may either have a value or be None.

For example:

```
1. otp_code: Optional[str] = None
```

means the OTP code might be a string, or it might be absent.

```
1. from django.core.exceptions import ValidationError
```

This is Django's validation error class.

We use it when the user tries to resend an OTP too early.

```
1. from django.utils.translation import gettext_lazy as _
```
This marks text as translatable.

So the message:

```
1. _("You must wait before requesting a new OTP.")
```
can later be translated into another language if needed.

```
1. from accounts.services.email_services import send_otp_email
```
This imports the email service responsible for actually sending the OTP email.

The OTP service does not build the email itself. It delegates delivery to the dedicated email service.

That separation is good architecture.

OtpResult

```
1. @dataclass(frozen=True)
2. class OtpResult:
```
This creates a data class named **OtpResult**.

A data class is a simple container for structured data.

frozen=True means the object becomes immutable after creation. That is good because the result should be a final record of what happened, not something we accidentally mutate later.

```
1.     sent: bool
```
This tells us whether the OTP was successfully sent.

In this file it is always returned as True when successful.

```
1.     email: str
```
This stores the email address the OTP was sent to.

```
1.     otp_code: Optional[str] = None
```
This optionally stores the raw OTP code.

That is useful only for:

- tests
- local development

- very controlled debugging

In real production behavior, we should almost never return raw OTP codes to callers.

That is why the comment says:

> only for tests/dev when explicitly requested

send_verification_otp(...)

```
1. def send_verification_otp(user, *, expiry_minutes: int = 5, return_code: bool = False) -> OtpResult:
```

This defines a service function that sends a verification OTP.

The parameters mean:

- **user**: the user object we are generating the OTP for
- **expiry_minutes=5:** OTP is valid for 5 minutes by default
- **return_code=False**: by default, do not expose the raw OTP
- returns **OtpResult**

The * means everything after it must be passed by keyword. That reduces mistakes and makes calls more explicit.

```
1.     code = user.generate_and_save_otp(expiry_minutes=expiry_minutes)
```

This line calls the user model's OTP helper.

That helper is expected to:

- create a random OTP
- hash it
- save the hash to otp_hash
- save the expiry to otp_expiry

This is excellent architecture because the user model is responsible for its own OTP storage logic.

The service does not manually touch **otp_hash** or **otp_expiry.** It delegates to the model.

```
send_otp_email(
    to_email=user.email,
    first_name=user.first_name,
    code=code,
    expires_minutes=expiry_minutes,
)
```

This sends the OTP to the user via the email service.

The service layer is acting as the coordinator:

- first store the OTP safely
- then deliver it

This order matters. We store first, then send. That way, the code in the email already corresponds to what is in the database.

```
return OtpResult(
    sent=True,
    email=user.email,
    otp_code=code if return_code else None,
)
```

This returns a structured result.

If **return_code=False**, otp_code becomes None.

If **return_code=True**, the raw OTP is included. That is useful for tests, but should be avoided in production flows.

resend_user_otp(...)

```
def resend_user_otp(
    user,
    *,
    cooldown_minutes: int = 1,
```

```
    expiry_minutes: int = 5,
    return_code: bool = False,
) -> OtpResult:
```

This function resends an OTP, but with a cooldown.

It is similar to **send_verification_otp**, but adds one security rule: do not let users spam OTP requests continuously.

```
    if not user.can_resend_otp(cooldown_minutes=cooldown_minutes):
```

This checks whether the user is allowed to request a new OTP yet.

The cooldown is a protection against:

- abuse
- email spam
- excessive OTP generation
- brute-force style resend flooding

```
        raise ValidationError(_("You must wait before requesting a new OTP."))
```

If the cooldown has not expired yet, we stop the process and raise a validation error.

That is a good design because the request is not a server crash. It is a user action that violates a business rule.

```
    code = user.generate_and_save_otp(expiry_minutes=expiry_minutes)
```

If resend is allowed, generate a fresh OTP and store it.

Notice that we do not reuse the old OTP. A fresh OTP is safer.

```
    send_otp_email(
        to_email=user.email,
        first_name=user.first_name,
        code=code,
        expires_minutes=expiry_minutes,
    )
```

Email the newly generated OTP.

```
    return OtpResult(
        sent=True,
        email=user.email,
        otp_code=code if return_code else None,
    )
```

Returns the final structured result just like the first function.

service/security.py

```
from __future__ import annotations

from typing import Optional

from django.conf import settings

try:
    import geoip2.database
except Exception:  # geoip may not be installed in some environments/tests
    geoip2 = None  # type: ignore

try:
    from zxcvbn import zxcvbn
except Exception:
    zxcvbn = None  # type: ignore

```

```
17.
18. _GEOIP_READER = None  # cached reader instance
19.
20.
21. def _get_geoip_reader():
22.     """
23.     Lazy-load the GeoIP reader so the app doesn't crash at import time
24.     if the GeoIP file path is missing/misconfigured.
25.     """
26.     global _GEOIP_READER
27.
28.     if _GEOIP_READER is not None:
29.         return _GEOIP_READER
30.
31.     if geoip2 is None:
32.         return None
33.
34.     path = getattr(settings, "GEOIP_DATABASE_PATH", None)
35.     if not path:
36.         return None
37.
38.     try:
39.         _GEOIP_READER = geoip2.database.Reader(path)
40.     except Exception:
41.         _GEOIP_READER = None
```

```
    return _GEOIP_READER


def get_country_code_from_ip(ip_address: str) -> Optional[str]:
    """
    Returns ISO country code (e.g. "NG", "US") for a given IP address.
    Returns None if lookup fails.
    """
    reader = _get_geoip_reader()
    if reader is None:
        return None

    try:
        return reader.country(ip_address).country.iso_code
    except Exception:
        return None


def is_unusual_login(user, ip_address: str) -> bool:
    """
    True if the country of this login differs from user's last_login_country.
    """
    new_country = get_country_code_from_ip(ip_address)
    if not new_country:
```

```
        return False

    old_country = getattr(user, "last_login_country", None)
    return bool(old_country and new_country != old_country)


def record_login_country(user, ip_address: str) -> None:
    """
    Saves the user's current login country to their profile (best-effort).
    """
    country = get_country_code_from_ip(ip_address)
    if not country:
        return

    user.last_login_country = country
    user.save(update_fields=["last_login_country"])


def password_strength(password: str, user_inputs: Optional[list] = None) -> dict:
    """
    Evaluate password strength using zxcvbn if available.

    Returns:
        dict with score/feedback-like fields.
        If zxcvbn isn't installed, returns a minimal safe response.
```

```
    """
    user_inputs = user_inputs or []

    if zxcvbn is None:
        # Minimal fallback: we can't score, but we can avoid crashing.
        # Strongly recommended: keep zxcvbn installed in production.
        return {
            "score": 0,
            "feedback": {
                "warning":"Password strength checker unavailable (zxcvbn not installed).",
                "suggestions": ["Install zxcvbn to enable password strength checks."],
            },
        }

    return zxcvbn(password, user_inputs=user_inputs)
```

This file is the security support layer for two different concerns:

1. **Geo-location awareness**
    - get country from IP
    - detect unusual login country changes
    - store login country
2. **Password quality evaluation**
    - score passwords with **zxcvbn**
    - fail safely if the package is missing

That means this file is not one single security feature. It is a set of **shared security services** that other parts of the authentication system can call.

Think of it as a risk analysis desk inside the app.

When a login happens, this file helps answer:

- “Does this location look unusual?”
- “Should we remember this location?”

When a password is being set, it helps answer:

- “Is this password weak or strong?”

Line-by-line Expalanation

```
1. from __future__ import annotations
```

Modern typing support, same as before.

```
1. from typing import Optional
```

Used for type hints like Optional[str], meaning a value may be a string or None.

```
1. from django.conf import settings
```

Imports Django settings so we can read things like:

- GEOIP_DATABASE_PATH

Defensive imports

```
1. try:
2.     import geoip2.database
3. except Exception:
4.     geoip2 = None  # type: ignore
```

This tries to import the GeoIP library.

If the package is missing, instead of crashing the entire app, the code sets **geoip2 = None.**

This is a very thoughtful design choice.

It means:

- the app still runs
- login still works
- only GeoIP-based features become unavailable

That is strong reliability engineering.

```
try:
    from zxcvbn import zxcvbn
except Exception:
    zxcvbn = None  # type: ignore
```

This does the same thing for the password strength package.

If **zxcvbn** is not installed, the app does not crash during import.

Instead, the **password_strength()** function later handles that case gracefully.

```
_GEOIP_READER = None
```

This is a module-level cache.

The GeoIP database reader can be expensive to open repeatedly, so we store it once and reuse it.

Think of it like opening a heavy reference book once and leaving it on the desk instead of reopening it every time someone asks a question.

That improves performance.

_get_geoip_reader()

```
def _get_geoip_reader():
```

Private helper function that lazily loads the GeoIP reader.

```
    global _GEOIP_READER
```

This tells Python we want to use and possibly update the module-level **_GEOIP_READER** variable, not create a new local variable.

```
    if _GEOIP_READER is not None:
        return _GEOIP_READER
```

If the reader is already loaded, just return it.

That avoids reopening the database file repeatedly.

Good for performance.

```
1.  if geoip2 is None:
2.      return None
```

If the geoip2 package is not installed, GeoIP features cannot work, so return None.

Again, this is safe failure instead of crashing.

```
1.  path = getattr(settings,"GEOIP_DATABASE_PATH", None)
```

Reads the GeoIP database path from settings.

```
1.  if not path:
2.      return None
```

If there is no path configured, return None.

This is another safe guardrail.

```
1.  try:
2.      _GEOIP_READER = geoip2.database.Reader(path)
3.  except Exception:
4.      _GEOIP_READER = None
```

Try to open the GeoIP database.

If it fails, store None instead.

That means:

- bad path
- missing file
- broken DB

will not crash the whole app.

```
1.  return _GEOIP_READER
```

Return the reader if it loaded successfully, otherwise None.

get_country_code_from_ip(...)

```
1. def get_country_code_from_ip(ip_address: str) -> Optional[str]:
```

This accepts an IP address string and returns either:

- a country code like "NG" or "US"
- or None if lookup fails

```
1.     reader = _get_geoip_reader()
2.     if reader is None:
3.         return None
```

If the GeoIP reader is unavailable, we give up safely and return None.

```
1.     try:
2.         return reader.country(ip_address).country.iso_code
3.     except Exception:
4.         return None
```

If the lookup works, we extract the ISO country code.

If anything goes wrong, we return **None.**

This keeps the rest of the authentication system stable even if IP lookup fails.

is_unusual_login(...)

```
1. def is_unusual_login(user, ip_address: str) -> bool:
```

This checks whether the current login country is different from the user's previous recorded login country.

```
1.     new_country = get_country_code_from_ip(ip_address)
2.     if not new_country:
3.         return False
```

If we cannot determine the country, we do not flag the login as unusual.

That is important: it avoids false alarms caused by missing GeoIP data.

```
1.     old_country = getattr(user, "last_login_country", None)
```

Reads the previously recorded country from the user object.

getattr avoids crashing if the attribute is missing.

```
    return bool(old_country and new_country != old_country)
```

This returns True only if:

- we have an old country
- and the new country is different

If there is no old country yet, the login is not considered unusual because there is no baseline for comparison.

This is a simple but effective anomaly signal.

record_login_country(...)

```
def record_login_country(user, ip_address: str) -> None:
```

This stores the user's current login country for future comparisons.

```
    country = get_country_code_from_ip(ip_address)
    if not country:
        return
```

If the country cannot be determined, do nothing.

Again, fail safely.

```
    user.last_login_country = country
    user.save(update_fields=["last_login_country"])
```

This updates only the **last_login_country** field in the database.

Using **update_fields** is slightly more efficient than saving the entire model and is a good habit when updating a single field.

password_strength(...)

```
def password_strength(password: str, user_inputs: Optional[list] = None) -> dict:
```

This function evaluates password strength.

It returns a dictionary because **zxcvbn** returns structured information like:

- score
- warning

- suggestions

```
1.    user_inputs = user_inputs or []
```

If no user-specific inputs are passed, default to an empty list.

These inputs are useful because password strength tools should penalize passwords that contain:

- email
- first name
- username

```
1.    if zxcvbn is None:
```

If the library is not available, we take the fallback path.

```
1.        return {
2.            "score": 0,
3.            "feedback": {
4.                "warning": "Password strength checker unavailable (zxcvbn not installed).",
5.                "suggestions": ["Install zxcvbn to enable password strength checks."],
6.            },
7.        }
```

This is a fail-closed behavior.

It returns a score of 0, meaning effectively “weak”.

That is a good security choice, because if the strength checker is unavailable, we should not silently approve passwords as if nothing happened.

This can surprise developers in local setup, but it is safer than pretending everything is fine.

```
1.    return zxcvbn(password, user_inputs=user_inputs)
```

If **zxcvbn** is installed, return its real evaluation result.

This gives you a proper strength analysis.

is_unusual_login() is a simple heuristic, not a full anomaly detection engine. A country change is only one signal. It does not automatically mean compromise.

Utils

The utils/ folder is where we place small supporting tools that other parts of the system can reuse.

Think of it like a toolbox in a workshop.

- The **services** folder is where we keep the specialists: security checks, email delivery, and other higher-level jobs.
- The **utils** folder is where we keep the simple tools: an IP extractor, OTP generator, small reusable helpers.

A good rule is:

- if a file mostly contains **small helper functions**, it belongs in utils
- if a file coordinates workflows, touches multiple parts of the system, or expresses business/security policy, it belongs in services

utils/ip.py

```
1. from django.http import HttpRequest
2.
3.
4. def get_client_ip(request: HttpRequest) -> str:
5.     """
6.         Best-effort client IP extraction.
7.         Only trust X-Forwarded-For if your proxy/load balancer is trusted &
configured.
```

```
    """
    xff = request.META.get("HTTP_X_FORWARDED_FOR")
    if xff:
        return xff.split(",")[0].strip()
    return request.META.get("REMOTE_ADDR", "unknown")
```

This file gives us a reusable way to get the client's IP address from a Django request.

That matters because many security decisions rely on IP information:

- audit logging
- throttling
- suspicious activity detection
- geo-location checks
- incident investigation

Instead of repeating this logic in every view, we place it once in utils/ip.py and import it wherever needed.

Think of it like a single trusted measuring tape. Instead of every person in the workshop guessing the length differently, everybody uses the same tool.

The code

```
from django.http import HttpRequest
```

This imports Django's request type.

It is mainly used for clarity and type hinting. By writing:

```
def get_client_ip(request: HttpRequest) -> str:
```

we are telling readers and tools that this function expects a Django HTTP request object and returns a string.

That makes the code easier to understand and easier to check with editors and linters.

This defines the helper function.

Its job is simple: inspect the request and return the best guess of the user's IP address.

The return type **-> str** means the function always tries to return a string, even if it falls back to "unknown".

```
1.
```

```
2.    """
3.      Best-effort client IP extraction.
4.      Only trust X-Forwarded-For if your proxy/load balancer is trusted & configured.
5.    """
```

This docstring is very important.

It tells us that IP extraction is not always perfect. In real deployments, a request often goes through:

- Nginx
- a load balancer
- Cloudflare
- a reverse proxy

So the IP seen by Django may not always be the actual user's IP unless the infrastructure is configured correctly.

This is why the phrase **"best-effort"** is used.

Think of it like looking at the return address on a forwarded package. Sometimes the label shows the original sender. Sometimes it shows only the forwarding center.

```
1.    xff = request.META.get("HTTP_X_FORWARDED_FOR")
```

This checks whether the request has the X-Forwarded-For header.

When traffic passes through proxies or load balancers, they often add this header to preserve the original client IP.

Django stores request metadata inside **request.META,** so this line is saying:

“Let me see if a trusted upstream component has already told us who the original client was.”

```
1.    if xff:
2.        return xff.split(",")[0].strip()
```

If **X-Forwarded-For** exists, this line takes the **first IP** from the list.

Why the first one?

Because X-Forwarded-For can look like this:

203.0.113.10, 10.0.0.2, 10.0.0.3
The first value is usually the original client, while the later ones are intermediate proxies.

.split(",")[0] grabs the first part.
.strip() removes any spaces around it.

This is useful, but only safe if your proxy setup is trusted and configured properly. Otherwise, a malicious client could fake this header.

```
1.    return request.META.get("REMOTE_ADDR", "unknown")
```

If there is no forwarded header, this falls back to REMOTE_ADDR.

That is the IP address of the machine that connected directly to Django.

In local development, this is often enough. In production, it may be the proxy instead of the real user unless the proxy is set up properly.

If even that is unavailable, the function returns "unknown" instead of crashing.

That is good defensive coding.

Security notes for ip.py

This helper is good and should stay simple.

For higher-security deployments, the real improvement is not usually inside this function. It is in the infrastructure:

- trust only known proxies
- configure Nginx or the load balancer correctly

- ensure Django is not directly exposed if it relies on forwarded headers

So the code is fine, but the environment matters.

utils/otp.py

```python
# accounts/utils/otp.py
from __future__ import annotations

import secrets
from django.core.exceptions import ValidationError

from accounts.models.user_model import User


def generate_otp_secret() -> str:
    """
    Backwards-compatible helper.

    Your current system does NOT store a secret on the user model, so we do not
    implement true TOTP verification. Keeping this function prevents breaking
    older documentation and imports.

    Returns:
        str: A random string (not used by verification in this architecture).
    """
    return secrets.token_urlsafe(16)
```

```
22.
23.
24. def generate_otp_code(*, digits: int = 6, secret: str | None = None) -> str:
25.     """
26.     Generate a cryptographically secure numeric OTP code.
27.
28.     Why this design:
29.     - Your system stores OTP as a hash in the DB (otp_hash), not as a TOTP
secret.
30.     - Verification happens via check_password(code, otp_hash).
31.     - Therefore, we only need a strong random one-time code.
32.
33.     Args:
34.         digits: length of OTP (default 6)
35.         secret: accepted only for backwards compatibility (ignored)
36.
37.     Returns:
38.         str: zero-padded numeric OTP code (e.g. "042913")
39.     """
40.     if digits < 4 or digits > 12:
41.         # guardrails: too short = brute-forceable, too long = annoying UX
42.         raise ValueError("digits must be between 4 and 12")
43.
44.     max_value = 10**digits
45.     code_int = secrets.randbelow(max_value)
46.     return str(code_int).zfill(digits)
```

```
47.
48.
49. def otp_valid(secret: str, code: str) -> bool:
50.     """
51.     Deprecated in this architecture.
52.
53.     True TOTP validation requires storing a per-user secret (or deriving it
54.     deterministically) and verifying against time windows.
55.     Your current system verifies OTP using otp_hash + check_password, so
this
56.     function is not used.
57.
58.     Returns:
59.         bool: Always False (kept only to avoid breaking old imports/docs).
60.     """
61.     return False
62.
63.
64. def resend_user_otp(user: User) -> str:
65.     """
66.     Resend OTP using the User model's OTP helpers.
67.     Returns the raw code (do not log it; only send to the user).
68.     """
69.     if not user.can_resend_otp():
70.         raise ValidationError("You must wait before requesting a new OTP.")
71.
```

```
72.     code = user.generate_and_save_otp()
73.     from accounts.services.email_services import send_otp_email
74.
75.     send_otp_email(to_email=user.email, first_name=user.first_name,
code=code)
76.     return code
```

This file provides helper functions related to OTP generation and resend behavior.

The overall purpose is:

- generate a secure one-time code
- preserve backward compatibility with older imports/docs
- support OTP resend through the user model

Think of this file as the **OTP toolkit**.

It is not the full OTP workflow by itself. The real workflow is spread across:

- the user model, which stores OTP hash and expiry
- the email service, which sends the OTP
- serializers/views, which validate and coordinate the flow

This file focuses on the supporting tools.

```
1. from __future__ import annotations
```

This is a modern Python feature that makes type hints more flexible and avoids some forward-reference issues.

It is not directly about security, but it is clean modern Python style.

```
1. import secrets
```

This is very important.

The secrets module is Python's cryptographically secure random generator.

For security-sensitive codes like OTPs, we should use secrets, not random.

Why?

Because OTPs are part of the authentication system. If the generator is predictable, attackers may guess codes more easily.

So this is the correct choice.

```
1. from django.core.exceptions import ValidationError
```

This imports Django's validation error class.

It is used when resend attempts are not allowed yet.

```
1. from accounts.models.user_model import User
```

This imports our custom user model for typing and helper logic.

This is acceptable, but it means this file is not a "pure utility" anymore, because it now depends on our application's model layer.

generate_otp_secret()

```
1. def generate_otp_secret() -> str:
```

This function returns a string and exists mainly for backward compatibility.

```
1.     """
2.     Backwards-compatible helper.
3.     Your current system does NOT store a secret on the user model, so we do not
4.     implement true TOTP verification. Keeping this function prevents breaking
5.     older documentation and imports.
6.     """
```

This explanation is accurate and very important.

Our current architecture is not true TOTP.

A true TOTP system usually works like authenticator apps:

- generate/store a secret
- derive the code from that secret and time
- verify by recalculating from the same secret

But our system instead does this:

- generate a random OTP
- hash the OTP
- store the hash and expiry in the database
- verify using **check_password**

That is completely valid and often better for email verification, but it means the "secret" is not actually used in verification.

So this helper is here only so old code and imports do not break.

```
1.     return secrets.token_urlsafe(16)
```

This generates a random URL-safe token string.

Since the function is only for compatibility, the exact value does not matter much in our architecture.

It is essentially a placeholder that keeps old interfaces alive.

generate_otp_code()

```
1. def generate_otp_code(*, digits: int = 6, secret: str | None = None) -> str:
```

This is the real OTP generator.

- **digits=6** means six-digit OTP by default
- **secret** exists only for backward compatibility and is ignored

The * means arguments after it must be passed by keyword. That prevents mistakes like accidentally mixing up arguments.

```
1.     """
2.     Generate a cryptographically secure numeric OTP code.
3.     """
```

This is the core of the function.

The goal is to produce a strong random numeric code suitable for email/SMS verification.

```
1.  """  Why this design:
2.     - Your system stores OTP as a hash in the DB (otp_hash), not as a TOTP
secret.
```

```
3.      - Verification happens via check_password(code, otp_hash).
4.      - Therefore, we only need a strong random one-time code."""
```

This is exactly right.

This design is simpler than full TOTP and fits our current architecture better.

We are not building Google Authenticator style OTP.
We are building server-generated, database-hashed, short-lived verification codes.

That is a valid and secure architecture.

```
1.  if digits < 4 or digits > 12:
2.      raise ValueError("digits must be between 4 and 12")
```

This adds guardrails.

If the OTP is too short, it becomes brute-forceable more easily.

If it is too long, it becomes annoying for users and may harm usability.

So this enforces a reasonable range.

```
1.  max_value = 10**digits
```

For 6 digits, this gives 1 000 000.

This is the maximum space of possible numeric OTP values.

```
1.  code_int = secrets.randbelow(max_value)
```

This generates a secure random integer between 0 and max_value - 1.

This is the correct way to generate a numeric OTP securely.

```
1.  return str(code_int).zfill(digits)
```

This converts the number to a string and zero-pads it.

That matters because if the random number is 421, a six-digit OTP should be:

000 421
not just 421.

otp_valid()

```
1. def otp_valid(secret: str, code: str) -> bool:
```

This function exists only for compatibility.

```
    """
    Deprecated in this architecture."""
```

That is correct.

It is not part of the real verification path anymore.

```
    return False
```

This always returns False.

That may look strange, but it is deliberate.

It prevents old imports from failing, while making it clear that this function is not actually valid in your current design.

This is safe, but it is also a signal that you may eventually want to remove it in a future version once backward compatibility is no longer important.

resend_user_otp()

```
def resend_user_otp(user: User) -> str:
```

This function coordinates an OTP resend for a user.

This is the part of **otp.py** that starts to behave more like a service than a pure utility, because it is no longer just generating a code. It is now:

- checking resend policy
- calling model methods
- sending email

That is business workflow logic.

It still works fine here, but architecturally you may later move it to a service file like **otp_service.py.**

```
    if not user.can_resend_otp():
        raise ValidationError("You must wait before requesting a new OTP.")
```

This prevents OTP spam.

That is important both for security and cost control:

- prevents abuse of the email system

- prevents brute-force resend loops
- reduces spam complaints and email reputation damage

```
code = user.generate_and_save_otp()
```

This asks the user model to generate the OTP, hash it, set expiry, and save it.

That is a good separation of responsibility:

- otp.py does not manage database fields directly
- the user model owns its own OTP storage behavior

```
from accounts.services.email_services import send_otp_email
```

This is a local import.

Local imports are often used to avoid circular import issues.

Since otp.py already depends on User, and email services may depend on other parts of the account system, importing inside the function reduces import-time problems.

```
send_otp_email(user, code)
```

This line is conceptually correct but, with your newer email service signature, it should be updated.

It should become:

```
send_otp_email(
    to_email=user.email,
    first_name=user.first_name,
    code=code,
)
```

That keeps it aligned with your new email service API and avoids runtime errors.

```
return code
```

This returns the raw OTP code.

That is okay internally, but it is important never to log or expose this code outside trusted internal flow.

The raw OTP should only be sent to the user over the delivery channel, never stored in logs or client-visible debug output.

.env

The **.env** file is the project's private configuration vault. It keeps values out of the codebase so we do not hardcode secrets into **settings.py.**

A good analogy is that the source code is the building design, while the .env file is the locked room containing the live keys.

Your file contains:

```
1. SECRET_KEY ='django-insecure-g$%^(!m$i6p@395!
yiacf(8fkox7%z3g4_e918r@mrav6^c=w4dghtrhjuy'
2. DEBUG = True
3. EMAIL_HOST_USER ='your email'
4. EMAIL_HOST_PASSWORD ='your password'
```

What each line means

```
1. SECRET_KEY
```

This is Django's core cryptographic secret. It is used in multiple security-sensitive operations such as token signing and session-related cryptography.

This must be private. In a real system, if this leaks, it is serious.

```
1. DEBUG = True
```

This means the app is running in development mode.

That is okay for local development, but for production it must be False.

```
1. DEBUG = config("DEBUG", cast=bool, default=False)
```

```
1. EMAIL_HOST_USER
```

This is the SMTP sender account.

```
1. EMAIL_HOST_PASSWORD
```

This is the SMTP password or app password.

For Gmail, we usually need an App Password, not our main account password.

celery.py

```
1. import os
2. from celery import Celery
3. os.environ.setdefault("DJANGO_SETTINGS_MODULE", 'config.settings')
4. app = Celery("config")
5. app.config_from_object("django.conf:settings", namespace="CELERY")
6. app.autodiscover_tasks()
```

This file initializes Celery for the Django project.

If Django is the main office, Celery is the background operations team. Django handles incoming requests immediately. Celery handles tasks that should happen in the background, like sending emails.

```
1. import os
```

Imports Python's OS module so we can work with environment variables.

```
1. from celery import Celery
```

Imports the Celery application class.

```
1. os.environ.setdefault("DJANGO_SETTINGS_MODULE", 'config.settings')
```

This tells Celery which Django settings module to use if it is not already set.

Without this, Celery would not know how to load our Django configuration.

```
1. app = Celery("config")
```

Creates a Celery app instance named "config".

```
1. app.config_from_object("django.conf:settings", namespace="CELERY")
```

Loads Celery settings from Django settings, but only keys prefixed with CELERY_.

So in **settings.py**, values like **CELERY_BROKER_URL** will be picked up automatically.

app.autodiscover_tasks()

Tells Celery to search installed apps for task definitions automatically.

That means functions decorated with **@shared_task** can be found without manually registering every file.

config/urls.py

```
1. from django.contrib import admin
2. from django.urls import path, include
3. from django.conf.urls.static import static
4. from django.conf import settings
5. urlpatterns = [
6.     path('admin/', admin.site.urls),
7.     path('api/v1/auth/', include("accounts.urls"))
8. ]
```

This is the project-level URL router. It decides which app gets which incoming request.

Think of it like the main reception desk in a building. It does not solve the visitor's problem itself. It directs the visitor to the correct department.

```
1. from django.contrib import admin
```

Imports Django admin site support.

```
1. from django.urls import path, include
```

- **path** defines a route
- **include** hands routing over to another URL file

```
1. urlpatterns = [...]
```

This is the main routing table.

```
1. path('admin/', admin.site.urls)
```

Sends /admin/ requests to Django admin.

```
1. path('api/v1/auth/', include("accounts.urls"))
```

All authentication-related API routes are delegated to accounts.urls under the prefix **/api/v1/auth/.**

That versioned prefix is good practice because it gives you room to evolve the API later.

settings.py

This is the central configuration file of the Django project. It controls installed apps, database, authentication, email, Celery, REST framework behavior, JWT behavior, and more.

Think of it as the control room of the application.

```
1. from pathlib import Path
2. import os
3. from decouple import config
```

- **Path** helps build filesystem paths safely
- **os** is used for environment and path operations
- **config** from **python-decouple** reads values from .env

```
1. BASE_DIR = Path(__file__).resolve().parent.parent
```

This calculates the root folder of the project.

```
1. SECRET_KEY = config('SECRET_KEY')
2. DEBUG = config('DEBUG', cast=bool, default=False)
```

```
1. ALLOWED_HOSTS
```

```
2. ALLOWED_HOSTS = []
```

This is okay for early development, but not for deployment.

In production, you must list the real domains.

```
1. INSTALLED_APPS
```

This controls which Django and third-party apps are active.

Good entries

- 1. rest_framework
- 2. rest_framework_simplejwt.token_blacklist
- 3. corsheaders
- 4. drf_spectacular
- 5. accounts

MIDDLEWARE

This is the ordered chain each request passes through.

CORS

```
1. CORS_ALLOWED_ORIGINS = [
2.     "http://localhost:3000",
3.     "http://localhost:5173"
4. ]
5. CORS_ALLOW_CREDENTIALS = True
```

This allows our frontend dev servers to call the backend.

This is correct for development.

For production, these should be locked down to real frontend domains only.

Email settings

These configure SMTP email sending.

They are mostly fine.

But in real systems:

- do not use raw account password
- use app passwords or provider credentials
- never commit real credentials

AUTH_USER_MODEL

```
AUTH_USER_MODEL = 'accounts.User'
```

This tells Django to use our custom User model instead of the default built-in one.

```
REST_FRAMEWORK
```

This section defines DRF's authentication, throttling, filters, and schema generation.

Authentication

```
'DEFAULT_AUTHENTICATION_CLASSES': [
    'rest_framework_simplejwt.authentication.JWTAuthentication',
],
```

This means protected endpoints expect Bearer JWT tokens.

Schema class

drf_spectacular is used for API schema generation and documentation.

FRONTEND_URL

```
FRONTEND_URL = "http://localhost:5173"
```

This is important because password reset links should send the user to the frontend reset page, not directly to the backend API.

SIMPLE_JWT

This is one of the most important security sections.

```
'ACCESS_TOKEN_LIFETIME': timedelta(days=2),
'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
```

```
3. 'ROTATE_REFRESH_TOKENS': True,
4. 'BLACKLIST_AFTER_ROTATION': True,
```

Needs improvement for stronger security

Two-day access tokens are long.

For stronger security, shorten access token lifetime. A more common pattern is minutes, not days.

Example:

```
1. 'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),
2. 'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
```

SPECTACULAR_SETTINGS

This configures API docs.

Celery settings

These tell Celery where Redis is and how task data should be serialized.

These are fine for local development.

GeoIP path

```
1. GEOIP_DATABASE_PATH = os.path.join(BASE_DIR, 'accounts', 'geoip', 'GeoLite2-City.mmdb')
```

Database

SQLite is fine for learning and development.

For real scale or serious production, you would move to PostgreSQL or another production-grade DB.

Password validators

```
1. AUTH_PASSWORD_VALIDATORS = [
2.     {
```

```
3.         'NAME':
'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
4.     },
5.     {
6.         'NAME':
'django.contrib.auth.password_validation.MinimumLengthValidator',
7.     },
8.     {
9.         'NAME':
'django.contrib.auth.password_validation.CommonPasswordValidator',
10.     },
11.     {
12.         'NAME':
'django.contrib.auth.password_validation.NumericPasswordValidator',
13.     },
14. ]
```

These are Django's built-in password validators.

Combined with our **zxcvbn** check, that gives you stronger password quality enforcement.

accounts/urls.py

This file defines all account-related API routes.

It is the app-level routing table for auth functionality.

Good parts

- registration
- login/logout
- token refresh

- password reset confirm with uid/token in URL
- change password
- verify/resend OTP
- test auth endpoint

admin.py

```
1. from django.contrib import admin
2. from accounts.admin_modules.user_admin import register_user_admin
3. register_user_admin()
```

This file is the entry point for registering models with Django admin.

It imports a helper function and executes it.

This is a slightly unusual but valid pattern. Instead of decorating or registering models directly inside **admin.py**, you delegated that work to **admin_modules/user_admin.py.**

Think of **admin.py** as the front desk that says: "Go call the registration function."

admin_modules/user_admin.py

```
1. from django.contrib import admin
2. from accounts.models.model import (
3.     UserActivityLog,
4.     SuspiciousActivity
5. )
6. from accounts.models.user_model import User
7. def register_user_admin():
8.     admin.site.register(UserActivityLog)
```

```
9.  admin.site.register(SuspiciousActivity)
10.  admin.site.register(User)
```

This file registers your models so they appear in Django admin.

That allows administrators to inspect:

- users
- activity logs
- suspicious activities

Closing Reflection

Authentication begins with a simple question: who is this user, and how certain should the system be before granting trust? But as this project has shown, the answer is never contained in a single password check, a single OTP code, or a single token response. Real authentication is layered. It is a sequence of defensive decisions, each one designed to reduce uncertainty, resist abuse, and preserve trust under pressure. It is not only about access. It is about evidence, verification, revocation, recovery, and accountability.

This project was built with that philosophy in mind. It does not treat authentication as a narrow login feature, but as foundational infrastructure that sits at the heart of a serious application. Through custom user modeling, OTP verification, JWT-based authentication, password recovery flows, activity

logging, throttling, password strength analysis, GeoIP-based risk signals, and asynchronous email delivery, this system establishes a meaningful and security-conscious baseline for identity management. It is already far stronger than a naive authentication implementation, not because it is perfect, but because it was designed with structure, separation of concerns, and defensive thinking from the beginning.

At the same time, this document should be read with honesty. This is Version 1. It is a strong baseline, but it is not a finished endpoint. It still needs major improvement, deeper hardening, broader testing, and more advanced controls before it could be considered a mature high-assurance authentication system. Future versions should push further into areas such as shorter-lived access tokens, stronger multi-factor authentication, WebAuthn, session/device management, more advanced anomaly detection, infrastructure hardening, secrets management, and formal threat modeling. These are not signs that the project is weak. They are signs that security is a discipline of continuous refinement.

That is perhaps the most important lesson of this work: security is never complete. No responsible engineer should claim a system is perfectly secure, because security is not a final badge of success. It is an ongoing practice of questioning assumptions, reducing exposure, tightening weak points, and improving architecture over time. Every serious system begins somewhere, and the goal of Version 1 is not to pretend to be the final answer. Its goal is to establish a thoughtful, extensible, and well-reasoned foundation that can support stronger iterations in the future.

Version 1 is where the system begins. The real work, as always in security, is in what comes next.