code::core

THE DJANGO PRODUCTION FIELD MANUAL

From runserver to Enterprise-Grade Deployment — for Ugandan Production Systems

---

Prepared for David Emiru Egwell · codeandcore

PART I — DEV TO PRODUCTION · PART II — CONTAINERISATION · PART III — USER MANAGEMENT & LOGIN

Contents

The Django Production Field Manual

How to use this manual
Why this stack, and not something else
A short glossary, since the tutorials assume you already know this

PART I — FROM DEV SERVER TO PRODUCTION SERVER

1.1 The architecture you're actually building
1.2 Server provisioning
1.3 Per-application isolation
1.4 Django settings for production
1.5 Static and media files
1.6 Database
1.7 Gunicorn + systemd
1.8 Nginx
1.9 Cloudflare: DNS, proxying, and certificates
1.10 Deploy process and zero(ish)-downtime
1.11 Monitoring, logging, and backups

PART II — CONTAINERISING DJANGO: DEV → PROD → HANDOVER

2.1 Why containerise at all, given Part I already works
2.2 The shape of a containerised Django app
2.3 The Dockerfile
2.4 Environment variables across dev → staging → prod
2.5 Local development in containers
2.6 Production docker-compose
2.7 Building, tagging, and shipping the image
2.8 Handover to the client

PART III — USER MANAGEMENT & LOGIN

3.1 Start every project with a custom user model
3.2 Session-based login, done properly
3.3 Password policy
3.4 Brute-force and login-abuse protection
3.5 Admin site hardening
3.6 Multi-factor authentication
3.7 Permissions and groups
3.8 Auth checklist summary

APPENDIX — COPY-PASTE REFERENCE CONFIGS

A.1 systemd unit — `/etc/systemd/system/appname.service`
A.2 Nginx server block — `/etc/nginx/sites-available/appname.conf`
A.3 `.env.example`
A.4 Production `Dockerfile` (multi-stage)
A.5 Production `docker-compose.yml`

A.6 `docker-compose.override.yml` (local dev — auto-loaded)
A.7 Minimal GitHub Actions deploy pipeline — `.github/workflows/deploy.yml`
A.8 Security quick-reference (both deployment styles)

The Django Production Field Manual

From `runserver` to Enterprise-Grade Deployment — for Ugandan Production Systems

---

How to use this manual

This is not a checklist you tick blind. Every part explains **what you're building and why**, walks you through it **step by step with real commands**, and only then gives you the compressed checklist — as a recap you use on your second, third, and fifth deployment, once you already understand what each line is doing. Read the explanations the first time through a part. After that, the checklists are enough.

Three parts, one appendix:

1. **Part I — From Dev Server to Production Server.** The classic Nginx + Gunicorn + PostgreSQL + systemd stack, plus Cloudflare DNS and certificates. This is the foundation — understand this before you containerise anything, because Part II just moves these same pieces into boxes.

2. **Part II — Containerising Django.** Docker from your laptop to a client's VM to handover, built on the same concepts as Part I.

3. **Part III — User Management & Login.** Custom user model, sessions, password policy, brute-force protection, 2FA — the parts of Django auth that the tutorials skip.

4. **Appendix.** Copy-paste configs: systemd unit, Nginx block, Dockerfile, docker-compose, `.env` template, CI pipeline, security quick-reference.

Assumptions throughout: Ubuntu Server 22.04/24.04 LTS, PostgreSQL, Cloudflare for DNS/proxy/certs, a custom user model with Django's session-based auth. Five client systems, one VM, real Ugandan hosting conditions (variable power, ISP routing, RAM-constrained VPS tiers) — this isn't written for an AWS reference architecture, it's written for what you'll actually be plugging into.

Why this stack, and not something else

You could deploy Django a dozen different ways — Heroku-style PaaS, a managed platform, serverless. For SME clients in Uganda who need you to own the whole stack, self-managed Ubuntu + Nginx + Gunicorn + PostgreSQL wins for three reasons: you control cost (a client paying for one VM instead of per-request PaaS billing in USD), you control data residency (the database sits where you put it, which matters if a client cares where their data physically lives), and you control the failure modes (when something breaks, you're not waiting on a platform's support queue — you SSH in and look at a log file).

The trade-off is that you own operations. Nobody auto-patches your OS, auto-scales your workers, or auto-renews your certificates for you unless you set that up. That's

what most of this manual actually is: the operational scaffolding that a PaaS would otherwise hide from you.

A short glossary, since the tutorials assume you already know this

WSGI — Web Server Gateway Interface. The standard Python defines for how a web server talks to a Python web application. Django *is* a WSGI application; it doesn't include a production-grade server to run it — that's Gunicorn's job.

Gunicorn — "Green Unicorn." A WSGI HTTP server. It loads your Django app once, then spawns multiple worker processes that each handle requests. This is what actually executes your Python code in production. `runserver` does this too, badly, single-threaded, with auto-reload and debug features that are actively dangerous in production.

Reverse proxy — A server (Nginx, here) that sits in front of your application and forwards requests to it, while also doing things the application shouldn't have to: serving static files directly, terminating TLS, buffering slow clients, load balancing across multiple app processes.

systemd — Ubuntu's process supervisor. It starts your Gunicorn process, restarts it if it crashes, starts it again on reboot, and gives you `journalctl` for logs. Without it, your app dies the moment your SSH session ends or the VM reboots.

Reverse DNS vs proxying — Cloudflare, when "proxied" (orange cloud), sits between the public internet and your VM: visitors hit Cloudflare's IP, not yours, and Cloudflare forwards the request to your origin server. This is what gives you their TLS certificates, DDoS protection, and caching for free.

Origin certificate — A TLS certificate Cloudflare issues specifically for the connection between Cloudflare and *your* server (the "origin"). It's not trusted by browsers directly — it doesn't need to be, because browsers only ever talk to Cloudflare, which presents its own publicly-trusted certificate.

PART I — FROM DEV SERVER TO PRODUCTION SERVER

1.1 The architecture you're actually building

`python manage.py runserver` is a development convenience: single-threaded, auto-reloading, serves static files inefficiently through Django itself, and prints stack traces to whoever hits an error if `DEBUG=True`. None of that belongs anywhere near a client's real traffic. The production stack that replaces it:

```
Internet → Cloudflare (DNS, proxy, TLS) → VM public IP:443
   → Nginx (reverse proxy, static/media files, TLS termination to origin)
      → Gunicorn (WSGI application server, running your Django app)
         → PostgreSQL (database)
         → Redis (cache / sessions / celery broker, if used)
   → systemd (keeps Gunicorn, Celery, and everything else alive and restarted on crash/reboot)
```

Walk the request path once so the pieces make sense: a browser requests `https://client.co.ug/dashboard/`. Cloudflare receives it, terminates the visitor's TLS connection, and forwards the request to your VM over its own encrypted connection. Nginx receives it on port 443, decides "this isn't a static file," and passes it to Gunicorn over a Unix socket. Gunicorn hands it to a free worker process, which runs your Django view, queries PostgreSQL, and returns HTML. Nginx passes that response back through Cloudflare to the browser. If the request had instead been for `/static/app.css`, Nginx would have served that file directly off disk — Gunicorn, and therefore your Python code, is never involved.

That last point is why the split exists at all: static files are the majority of requests on most pages (CSS, JS, images), and Nginx serving them directly from disk is 10-100x faster than routing them through a Python process. Keep dynamic and static traffic separated and each layer stays simple and fast at its one job.

For 5 client systems on one VM, you have two reasonable topologies. **One VM per client system** gives you the cleanest isolation and the easiest independent handover, at the cost of more RAM overhead running 5 separate OS instances. **One VM, multiple systems**, each with its own PostgreSQL database, Linux user, and Nginx server block, is cheaper and fine up to moderate load, but demands more careful firewalling so one client's app can't see another's files or database.

Default to **one system = one dedicated Linux user + one dedicated PostgreSQL database + one Gunicorn socket + one Nginx server block**, even when several systems share a VM. This single decision is what makes the isolation checklist in 1.3 actually enforceable — without it, "5 systems on one VM" quietly becomes "5 systems that can all read each other's files."

1.2 Server provisioning

Why this matters before you touch Django at all

A Django app is only as secure and stable as the server underneath it. Most production incidents on small VPS deployments aren't Django bugs — they're an open SSH port with password auth, a firewall that was never enabled, or a disk that filled up because nobody set up log rotation. This section is entirely about the OS layer, before any Python is involved.

Walkthrough

Provision the VM first — for in-country hosting, Raxio, Africa Data Centres Kampala, Roke Telkom, or Simba/Onix are worth checking for Ugandan data residency and lower-latency routing to local users; otherwise any KVM-based VPS provider works. Once you have root access and an IP:

```
# Update the base system first, always
sudo apt update && sudo apt upgrade -y

# Create your working user — never operate as root day-to-day
sudo adduser deploy
sudo usermod -aG sudo deploy
```

Log out and back in as `deploy` from here on. Set up SSH key auth (generate a deploy-specific key on your own machine, don't reuse your personal key across every client — if one client's key is ever compromised, you don't want it to be the same key sitting on four other servers):

```
# on your local machine
ssh-keygen -t ed25519 -C "deploy-clientname" -f ~/.ssh/clientname_deploy
ssh-copy-id -i ~/.ssh/clientname_deploy.pub deploy@server-ip
```

Then lock the server down — disable root login and password auth entirely, so a leaked/guessed password is no longer even a valid attack:

```
sudo nano /etc/ssh/sshd_config
# set: PermitRootLogin no
# set: PasswordAuthentication no
sudo systemctl restart sshd
```

Set the hostname and timezone (Africa/Kampala matters more than it seems — every log timestamp, cron job, and certificate renewal window will be easier to reason about in local time):

```
sudo hostnamectl set-hostname clientname-prod01
sudo timedatectl set-timezone Africa/Kampala
```

Enable the firewall, default-deny, and only open what you need:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow OpenSSH
sudo ufw allow 80,443/tcp
sudo ufw enable
```

Install `fail2ban`, which watches auth logs and temporarily bans IPs after repeated failed login attempts — cheap insurance against automated SSH scanning, which every public IP receives constantly:

```
sudo apt install fail2ban -y
```

```
sudo systemctl enable --now fail2ban
```

If the VM has less than 4GB RAM — common on entry-tier Ugandan VPS plans — add swap, because Django + PostgreSQL + Nginx running together on a 1-2GB box will otherwise hit the OOM killer under any real load spike:

```
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
```

Finally, turn on unattended security upgrades so the OS patches itself against known vulnerabilities without you having to remember to `apt upgrade` on 5 servers every week:

```
sudo apt install unattended-upgrades -y
sudo dpkg-reconfigure unattended-upgrades
```

Checklist — 1.2 Server provisioning

- ☐ VM provisioned, Ubuntu 22.04/24.04 LTS confirmed (`lsb_release -a`)
- ☐ Deploy user created, SSH key-based login working, deploy-specific key (not your personal key)
- ☐ Root login and password auth disabled in `sshd_config`, `sshd` restarted
- ☐ Hostname and timezone (`Africa/Kampala`) set
- ☐ `ufw` enabled, default-deny, only SSH/80/443 open
- ☐ `fail2ban` installed and running
- ☐ Swap added if RAM < 4GB
- ☐ Unattended security upgrades enabled
- ☐ Full reboot tested (`sudo reboot`) — confirm SSH access survives it before going further

1.3 Per-application isolation

Why this matters

If 5 client systems share a VM under one Linux user with one shared database role, a bug or compromise in one app can read or corrupt another client's data. This is the difference between "we had an incident in Client A's app" and "we had an incident in Client A's app that also exposed Client B's database." Isolation is cheap to set up and expensive to retrofit after a client's already in production, so do it from day one on every system.

Walkthrough

For each of the 5 systems, repeat this pattern with the app's own name substituted in:

```
# A system user with no interactive shell — it only exists to own files and run the process
sudo adduser --system --group appname

# Its own directory, not under /home or /var/www
sudo mkdir -p /srv/appname
sudo chown appname:appname /srv/appname
```

Create a dedicated PostgreSQL role and database scoped to nothing else:

```
sudo -u postgres psql
```

```
CREATE USER appname_user WITH PASSWORD 'a-long-random-password';
CREATE DATABASE appname_db OWNER appname_user;
REVOKE ALL ON DATABASE appname_db FROM PUBLIC;
\q
```

That `REVOKE ALL ... FROM PUBLIC` matters — by default PostgreSQL grants some connect privileges broadly, and this closes that back down so only `appname_user` can touch `appname_db`.

Inside `/srv/appname`, create the Python virtual environment as the app user (or skip this entirely if you're using Part II's container approach for this particular system):

```
sudo -u appname python3 -m venv /srv/appname/venv
```

And the `.env` file, locked to owner-only read/write:

```
sudo -u appname touch /srv/appname/.env
sudo chmod 600 /srv/appname/.env
```

Checklist — 1.3 Per-application isolation

- ☐ Dedicated system user created per app, no login shell
- ☐ Dedicated `/srv/appname/` directory, owned by that user
- ☐ Dedicated PostgreSQL role + database, `REVOKE ALL ... FROM PUBLIC` applied
- ☐ Dedicated virtualenv inside the app directory (non-containerised systems)
- ☐ Dedicated `.env`, permissions `600`, owned by the app user

1.4 Django settings for production

Why a settings split, not one file with `if DEBUG`

One settings.py with if DEBUG: ... else: ... scattered through it is how a dev-only setting quietly ships to production, or a production security setting gets accidentally disabled while debugging locally. A settings *package* makes environments explicit files, not runtime branches:

```
config/
  settings/
    __init__.py
    base.py       # shared: INSTALLED_APPS, MIDDLEWARE, templates, etc.
    dev.py        # DEBUG=True, local DB, django-debug-toolbar
    prod.py       # DEBUG=False, security headers, prod DB from env
```

base.py never hardcodes a secret — every value that differs between your laptop and the client's server comes from os.environ, typically via django-environ for convenient parsing:

```
# config/settings/base.py
import environ

env = environ.Env()
environ.Env.read_env()   # reads .env in dev; in prod, real env vars are already set

SECRET_KEY = env('SECRET_KEY')
DATABASES = {'default': env.db('DATABASE_URL')}
```

prod.py then imports everything from base.py and layers production-only settings on top:

```
# config/settings/prod.py
from .base import *

DEBUG = False
ALLOWED_HOSTS = env.list('ALLOWED_HOSTS')
CSRF_TRUSTED_ORIGINS = [f'https://{h}' for h in ALLOWED_HOSTS]

SECURE_SSL_REDIRECT = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
SECURE_HSTS_SECONDS = 3600  # raise to 31536000 once TLS is confirmed stable
X_FRAME_OPTIONS = 'DENY'
```

SECURE_PROXY_SSL_HEADER is the one every guide forgets and every deployment breaks on. Nginx terminates TLS and talks to Gunicorn over plain HTTP on the loopback/socket — from Django's point of view, every request looks like plain HTTP unless you tell it to trust the X-Forwarded-Proto header Nginx sets. Without this line,

`SECURE_SSL_REDIRECT` sends every request into an infinite redirect loop, because Django thinks the "secure" request it just received over HTTPS was actually insecure.

Checklist — 1.4 Production settings

- ☐ Settings split into `base.py` / `dev.py` / `prod.py`
- ☐ `DEBUG = False`, `SECRET_KEY` from environment, `ALLOWED_HOSTS` exact (never `['*']`)
- ☐ `CSRF_TRUSTED_ORIGINS` set for Django 4+
- ☐ `SECURE_SSL_REDIRECT`, `SESSION_COOKIE_SECURE`, `CSRF_COOKIE_SECURE` all `True`
- ☐ `SECURE_PROXY_SSL_HEADER` set — the redirect-loop fix
- ☐ `SECURE_HSTS_SECONDS` set, raised once TLS is confirmed stable over a few days
- ☐ `STATIC_ROOT`/`MEDIA_ROOT` set to absolute paths outside the git repo
- ☐ Logging configured to a file or stdout that systemd/journald captures
- ☐ Real SMTP email backend configured, not the console backend
- ☐ `DJANGO_SETTINGS_MODULE=config.settings.prod` set explicitly via `.env`/systemd, never left to guesswork

1.5 Static and media files

Why Nginx, not Django, serves these

Django *can* serve static files (via `whitenoise`, for instance), and for a very small internal tool that's a fine simplification. But Nginx serving them directly off disk is dramatically faster and doesn't tie up a Gunicorn worker — a worker busy streaming a 2MB image is a worker not available to handle the next dynamic request. At any real scale this matters; at small scale it costs nothing to do it right from the start.

Walkthrough

Every deploy runs:

```
python manage.py collectstatic --noinput
```

This copies every app's static files into one `STATIC_ROOT` directory that Nginx points at directly (see the Nginx config in 1.8 and the Appendix). For media (user uploads), if you need multi-server access or CDN delivery later, `django-storages` pointed at an S3-compatible bucket is the standard move — Cloudflare R2 is a natural pairing here since you're already using Cloudflare for DNS, and its egress pricing is friendlier than AWS S3 for a bandwidth-conscious Ugandan client.

Set `ManifestStaticFilesStorage` so filenames get content hashes (`app.a1b2c3.css`), which lets you set aggressive browser caching without worrying about serving a stale file after a deploy — the filename itself changes when the content does.

Match your upload limits in both places, or you'll get the classic bug where a large upload silently fails with no clear error: Nginx's `client_max_body_size` and Django's `DATA_UPLOAD_MAX_MEMORY_SIZE`/`FILE_UPLOAD_MAX_MEMORY_SIZE` need to agree.

Checklist — 1.5 Static and media

- ☐ `collectstatic` run every deploy
- ☐ Nginx serves `/static/` and `/media/` directly, Gunicorn never touches these requests
- ☐ `ManifestStaticFilesStorage` (or equivalent) for cache-safe static filenames
- ☐ `django-storages` + S3-compatible bucket considered for media if multi-server/CDN is needed
- ☐ Upload size limits matched between Nginx and Django

1.6 Database

Why PostgreSQL, and what "production-ready" actually means here

SQLite is a file, not a database server — it can't handle concurrent writes from multiple Gunicorn workers safely at any real scale, and it has no user/permission model at all. PostgreSQL is the default for a reason: mature, well-documented, and every Ugandan hosting provider's Ubuntu image has it in the standard repos.

"Production-ready" mostly means: connections are restricted, backups exist and are tested, and you won't run out of connections the moment 5 apps are live at once.

Walkthrough

Install from the official Ubuntu repo (or `apt.postgresql.org` if you need a specific newer version):

```
sudo apt install postgresql postgresql-contrib -y
```

Confirm it only listens locally — check `pg_hba.conf` and `postgresql.conf` — unless a specific system genuinely needs remote DB access, in which case restrict that to a specific IP, never `0.0.0.0`.

Set up automated daily backups, written *off* the VM's own disk — a backup that lives next to the database it's backing up isn't a backup, it's a false sense of security:

```
#!/bin/bash
# /srv/scripts/backup-appname.sh
DATE=$(date +%F)
pg_dump -U appname_user appname_db | gzip > /tmp/appname_$DATE.sql.gz
# then push /tmp/appname_$DATE.sql.gz somewhere off-VM: object storage, another server, etc.
rm /tmp/appname_$DATE.sql.gz
```

Cron it daily, then — this is the step almost everyone skips — actually restore that backup once, onto a scratch database, before you trust it:

```
createdb appname_test_restore
gunzip -c appname_2026-08-09.sql.gz | psql appname_test_restore
```

If 5 apps are sharing one PostgreSQL instance, watch `max_connections` (default 100) — each Gunicorn worker can open its own connection, and across 5 apps × 3 workers × connection pooling overhead, you can exhaust that faster than expected. `pgbouncer` in front of Postgres pools connections and is worth adding once you're running more than 2-3 apps against one instance.

Checklist — 1.6 Database

- ☐ PostgreSQL installed, connections restricted to localhost/Unix socket
- ☐ Dedicated role/database per app (from 1.3)
- ☐ Automated daily backups, stored off the VM
- ☐ Backup restore actually tested at least once
- ☐ `pgbouncer` considered once running several apps against one Postgres instance
- ☐ `pg_stat_statements` enabled for later query performance visibility

1.7 Gunicorn + systemd

Why you need a process supervisor at all

If you start Gunicorn by hand in an SSH session (`gunicorn config.wsgi:application`), it dies the moment that SSH session ends. systemd is what keeps it running permanently, restarts it if it crashes, and starts it again automatically after a server reboot — the difference between "the app went down at 2am and stayed down until you noticed" and "the app restarted itself in the two seconds it took to crash."

Walkthrough

Install Gunicorn inside the app's own virtualenv (never system-wide — that couples every app on the VM to one Gunicorn version):

```
sudo -u appname /srv/appname/venv/bin/pip install gunicorn
```

Bind Gunicorn to a Unix socket rather than a TCP port. Sockets avoid picking arbitrary port numbers across 5 apps and are slightly faster since they skip the TCP/IP stack entirely for a connection that never leaves the machine:

```
mkdir -p /srv/appname/run
```

The systemd unit (full version in the Appendix) is the piece that ties it together — walk through what each line does:

```
[Unit]
Description=Gunicorn daemon for appname
After=network.target postgresql.service
# Don't start until networking and Postgres are up
```

```
[Service]
Type=notify
User=appname
Group=appname
WorkingDirectory=/srv/appname
EnvironmentFile=/srv/appname/.env
# Loads DATABASE_URL, SECRET_KEY, etc. into the process environment
ExecStart=/srv/appname/venv/bin/gunicorn --workers 3 --bind unix:/srv/appname/run/gunicorn.sock
--timeout 30 config.wsgi:application
ExecReload=/bin/kill -s HUP $MAINPID
# HUP reloads workers gracefully without dropping in-flight connections
Restart=always
RestartSec=5
# If Gunicorn crashes, systemd restarts it 5 seconds later, indefinitely

[Install]
WantedBy=multi-user.target
```

Worker count follows the `(2 × CPU cores) + 1` rule of thumb, adjusted down if RAM is the binding constraint rather than CPU — each worker is a full Python process holding its own memory. On a 1-2 vCPU / 2GB Ugandan VPS running 2 apps, 2-3 workers per app is often more realistic than the textbook formula.

```
sudo systemctl daemon-reload
sudo systemctl enable --now appname
sudo systemctl status appname
journalctl -u appname -f    # live logs
```

Then actually reboot the VM and confirm the service comes back on its own — this is the test people skip and then discover the hard way during a power outage.

Checklist — 1.7 Gunicorn + systemd

- ☐ Gunicorn installed in the app's own virtualenv
- ☐ Bound to a Unix socket, not a TCP port
- ☐ Worker count set deliberately for the VM's actual CPU/RAM, not just copied from a tutorial
- ☐ systemd unit created, `Restart=always`
- ☐ `daemon-reload` + `enable --now` done, `status` clean
- ☐ Reboot tested — service comes back without manual intervention

1.8 Nginx

What Nginx is doing in this stack, precisely

Nginx is the only thing in this stack that talks directly to the internet on ports 80/443 (well — Cloudflare does, but Nginx is what the VM itself exposes). It terminates TLS from Cloudflare, decides "static file or app request," and either serves the file itself or

forwards to Gunicorn. It's also your first line of defense: a malformed or oversized request often never reaches your Python code at all.

Walkthrough

One server block per app, `/etc/nginx/sites-available/appname.conf`, symlinked into `sites-enabled`:

```
sudo ln -s /etc/nginx/sites-available/appname.conf /etc/nginx/sites-enabled/
```

The essential shape (full version in the Appendix): TLS config, static/media locations served with `alias`, and everything else proxied to the Gunicorn socket with headers that tell Django what the original request actually looked like:

```
location / {
    proxy_pass http://unix:/srv/appname/run/gunicorn.sock;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto https;
}
```

That last header, `X-Forwarded-Proto https`, is what `SECURE_PROXY_SSL_HEADER` in 1.4 reads — without it Django can't tell the request arrived over HTTPS.

Disable the default Nginx server block so random IP-address scans (which every public IP receives) don't get routed to your app by accident:

```
sudo rm /etc/nginx/sites-enabled/default
```

Always test config before reloading — a syntax error in a live `nginx.conf` takes down every app on the VM, not just the one you were editing:

```
sudo nginx -t && sudo systemctl reload nginx
```

Checklist — 1.8 Nginx

- ☐ One server block per app, symlinked into `sites-enabled`
- ☐ `proxy_pass` to the Gunicorn Unix socket, with `X-Forwarded-Proto` set
- ☐ `/static/` and `/media/` served via `alias`, with cache headers
- ☐ `client_max_body_size` matched to Django's upload limits
- ☐ Default Nginx server block disabled
- ☐ `nginx -t` passes before every reload
- ☐ Access/error logs per app, rotated via `logrotate`

1.9 Cloudflare: DNS, proxying, and certificates

Why this is the part everyone gets wrong

The "too many redirects" loop almost everyone hits with Cloudflare + Django comes from one specific mistake: setting Cloudflare's SSL mode to "Flexible" while also forcing `SECURE_SSL_REDIRECT` in Django. Flexible means Cloudflare talks to your origin over plain HTTP — so Django, correctly, keeps trying to redirect that "insecure" request to HTTPS, Cloudflare receives the redirect, and loops forever. Understanding *why* this happens means you'll never make this mistake, instead of just following steps that happen to avoid it.

Walkthrough

DNS. Point the domain's nameservers at Cloudflare (done once at your registrar — for `.co.ug`/`.ug` domains that's whichever CoCCA-based registrar you registered through; for `.com` it's your usual registrar). Create an `A` record pointing at the VM's public IP, and set it to **Proxied (orange cloud)** — this is what routes traffic through Cloudflare rather than straight to your IP.

Certificates — Pattern A, the one to use by default. In the Cloudflare dashboard, go to SSL/TLS → Origin Server and generate an Origin Certificate (free, valid up to 15 years). This certificate is for the Cloudflare-to-your-VM leg only — browsers never see it. Copy it onto the VM:

```
sudo mkdir -p /etc/nginx/ssl
sudo nano /etc/nginx/ssl/appname.pem   # paste the certificate
sudo nano /etc/nginx/ssl/appname.key   # paste the private key
sudo chmod 600 /etc/nginx/ssl/appname.key
```

Point Nginx's `listen 443 ssl` at this cert/key pair. Then, critically, set Cloudflare's SSL/TLS encryption mode to **Full (strict)** — not "Full," which accepts *any* certificate at the origin including a self-signed one, and not "Flexible," which is the redirect-loop trap above. Full (strict) actually validates the origin certificate, which is exactly what you want since you just installed a real one.

The resulting picture: browser ↔ Cloudflare is public-CA-trusted HTTPS (visitors see a valid padlock); Cloudflare ↔ your VM is also encrypted, and Cloudflare validates that it's really talking to your server. Both hops are secure, correctly.

Pattern B — Let's Encrypt on the origin, only if you need the origin reachable directly (bypassing Cloudflare), e.g. for a health-check endpoint some monitoring service hits directly:

```
sudo apt install certbot python3-certbot-nginx -y
sudo certbot --nginx -d yourdomain.co.ug
```

If the domain stays orange-cloud proxied, standard HTTP-01 validation won't work — use `certbot-dns-cloudflare` for DNS-01 validation instead, which works regardless of proxy status.

Finally, in Cloudflare: turn on "Always Use HTTPS" and "Automatic HTTPS Rewrites," and test from outside your own network — a phone on mobile data is a genuinely useful test in Uganda, given how differently local ISPs route traffic compared to your office wifi:

```
curl -I https://yourdomain.co.ug
# expect HTTP/2 200, no redirect loop
```

Checklist — 1.9 Cloudflare

- ☐ Nameservers pointed at Cloudflare, `A` record proxied (orange cloud)
- ☐ Origin Certificate generated and installed on the VM, Nginx configured to use it
- ☐ Cloudflare SSL/TLS mode set to **Full (strict)** — never Flexible
- ☐ "Always Use HTTPS" and "Automatic HTTPS Rewrites" enabled
- ☐ Tested from outside the VM's own network, confirmed no redirect loop

1.10 Deploy process and zero(ish)-downtime

Why "just git pull and restart" isn't quite enough

A deploy touches code, dependencies, the database schema, and static files, roughly simultaneously, while the previous version might still be handling in-flight requests. Doing this by memory means eventually skipping a step under time pressure — usually `collectstatic`, which fails silently until someone notices the CSS looks wrong.

Walkthrough

Script the deploy, don't perform it from memory:

```
#!/bin/bash
set -e   # stop on first error, don't push a half-broken deploy forward
cd /srv/appname
git pull
source venv/bin/activate
pip install -r requirements.txt
python manage.py migrate
python manage.py collectstatic --noinput
sudo systemctl reload appname   # HUP reload — graceful, no dropped connections
```

Write migrations to stay backward-compatible with the *currently running* code when you can — add a new column in one deploy, backfill it, and only remove the old column in a later deploy. Otherwise a request that lands mid-deploy, hitting old code against a new schema (or vice versa), can 500.

Know your rollback command before you need it, not after: keep the previous release reachable (a git tag, a previous directory) and know the one-line revert.

Checklist — 1.10 Deploy process

- ☐ Deploy is a script, not a sequence of remembered commands
- ☐ Migrations are backward-compatible where practical
- ☐ `collectstatic` runs every deploy, unconditionally
- ☐ Reload is graceful (`HUP`/`reload`), not a hard `restart` that drops connections
- ☐ Smoke test after every deploy: homepage, login, `/admin/`
- ☐ Rollback command known before you deploy, not improvised after a failure

1.11 Monitoring, logging, and backups

Why this is the part that gets skipped

Nothing about monitoring is visible until the day it's the only thing standing between you and a client finding out about an outage before you do. It's genuinely easy to ship five client systems without it and get away with it for months — until you don't.

Walkthrough

Wire up application error tracking — Sentry's free tier is enough for 5 systems to start, and the value is immediate: you get a stack trace and the request that caused it, emailed to you, often before the client even notices.

```
pip install sentry-sdk
```

```
# settings/prod.py
import sentry_sdk
sentry_sdk.init(dsn=env('SENTRY_DSN'), traces_sample_rate=0.1)
```

Add uptime monitoring per domain — UptimeRobot or Better Stack's free tiers, or a simple cron job hitting the homepage and alerting via a webhook if it fails, are all fine starting points. Given how much more variable power and connectivity can be for Ugandan hosting compared to a tier-1 cloud region, this matters more here than the tutorials assume.

Confirm `logrotate` is actually rotating your Nginx and Postgres logs — an unrotated access log on a small VPS disk will eventually fill it, and a full disk takes down everything on that VM, not just the log-writing process:

```
cat /etc/logrotate.d/nginx   # should already exist from the nginx package; confirm it's enabled
```

And repeat the backup restore test from 1.6 — it belongs in more than one checklist, because it's the single most commonly skipped step under deadline pressure, and the only one that turns a disaster into an afternoon.

Checklist — 1.11 Monitoring, logging, backups

- ☐ Application error tracking wired up (Sentry or equivalent)
- ☐ Uptime monitoring on each domain
- ☐ Disk space alerting in place
- ☐ `logrotate` confirmed actually rotating Nginx/Postgres logs
- ☐ Backup restore tested (again)
- ☐ A one-page runbook written per client: how to restart, where the logs are, who to call

---

PART II — CONTAINERISING DJANGO: DEV → PROD → HANDOVER

2.1 Why containerise at all, given Part I already works

Part I is a completely legitimate, permanent way to run production Django — plenty of serious systems never move past it. Containerise when any of these are actually true for a given client system, not by default: you want your laptop, staging, and the client's VM running the exact identical environment, which kills "it worked on my machine"; you want a client's own IT person to be able to restart the app with `docker compose restart` without knowing what a virtualenv or systemd unit is; you're running several of these 5 systems on one VM and want real isolation between their dependencies without maintaining 5 separate venvs by hand; or you want a CI pipeline that builds one artifact and deploys that exact artifact everywhere, provably.

The concepts underneath containers are the same ones from Part I — you still have a WSGI app server, a reverse proxy, a database, and a process supervisor. Docker just packages each of those into its own isolated, portable box, and Docker Compose replaces systemd as the thing that keeps them running.

A few Docker concepts, briefly, since the rest of this part assumes them

An **image** is a read-only template — your app's code plus everything it needs to run, built once. A **container** is a running instance of an image — the same way a class and an object relate in code. A **Dockerfile** is the recipe that builds an image. A **volume** is how a container gets persistent storage that survives the container being destroyed and recreated (your database's actual data, for instance — the *container* is disposable, the *volume* is not). A **network** is how containers talk to each other by name instead of IP address.

2.2 The shape of a containerised Django app

```
project/
  Dockerfile
```

```
docker-compose.yml          # production
docker-compose.override.yml # local dev overrides (auto-loaded by compose)
.env.example                # committed, no real secrets
.env                        # gitignored, real values
requirements.txt
config/settings/...
manage.py
```

Three containers minimum: `web` (Django via Gunicorn), `db` (PostgreSQL), `nginx` (reverse proxy + static files) — plus `redis` and a `worker`/`beat` pair if you're running Celery. Each container maps to exactly one process from the Part I architecture diagram; nothing conceptually new is happening, it's the same request path, just each hop now lives in its own box.

2.3 The Dockerfile

Why multi-stage, why non-root, why layer order matters

A naive single-stage Dockerfile ships your build toolchain (compilers, headers) inside the production image, making it needlessly large and giving an attacker more tools if they ever get a shell inside the container. A multi-stage build compiles dependencies in a throwaway `builder` stage, then copies only the finished result into a slim final image.

Running as root inside a container is a much bigger deal than it sounds — a container escape vulnerability (rare, but they happen) hands an attacker root on the host if the container process was root. Running as a non-root user inside the container costs you nothing and closes that off.

Layer order matters because Docker caches each instruction as a layer, and reuses cached layers if nothing above them changed. If you `COPY . .` (your whole app) before `RUN pip install`, then *every single code change* invalidates the pip install cache and forces a full dependency reinstall on every build. Copying `requirements.txt` and installing dependencies *first*, then copying the rest of your code, means a pure code change only invalidates the fast final layer.

Walkthrough (full file in the Appendix)

```
# ---- builder: has the compiler toolchain, nothing else does ----
FROM python:3.12-slim-bookworm AS builder
WORKDIR /app
RUN apt-get update && apt-get install -y --no-install-recommends build-essential libpq-dev
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# ---- final: slim, no compiler, non-root ----
FROM python:3.12-slim-bookworm
WORKDIR /app
RUN apt-get update && apt-get install -y --no-install-recommends libpq5 \
    && adduser --system --group appuser
COPY --from=builder /root/.local /home/appuser/.local
COPY --chown=appuser:appuser . .
```

```
USER appuser
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "config.wsgi:application"]
```

Note `libpq-dev` (the Postgres client headers/compiler support) only exists in the builder stage; the final image only has the much smaller `libpq5` runtime library, since that's all the running app actually needs.

A `.dockerignore` file (mirroring `.gitignore`, roughly) keeps `.git`, `venv/`, `__pycache__/`, and your real `.env` out of the build context entirely — without it, Docker copies all of that into the image, bloating it and, worse, potentially baking secrets into an image layer that then sits in your registry forever.

Checklist — 2.3 Dockerfile

- ☐ Multi-stage build: builder stage has compilers, final stage doesn't
- ☐ Base image pinned to a specific tag, never `:latest`
- ☐ Non-root user created and used via `USER appuser`
- ☐ Dependencies installed before app code is copied, to preserve layer caching
- ☐ `.dockerignore` present and excludes `.git`, `venv/`, `.env`, `__pycache__/`
- ☐ No secrets in `ENV`/`ARG` — build args persist in image history even if unused later
- ☐ Image tagged with both a version (git SHA) and `latest`

2.4 Environment variables across dev → staging → prod

Why this is where secrets most often leak

The single most common way a client's production secret ends up in a public GitHub repo is a developer committing a working `.env` "just to save it somewhere," then forgetting it's there. The discipline here is simple but has to be absolute: `.env.example` (no real values) is committed; `.env` (real values) is gitignored *before* it ever contains a real secret, not after.

Walkthrough

```
# the moment you create the project, before any real secret exists
echo ".env" >> .gitignore
cp .env.example .env
# now fill in real values in .env — it's already excluded from git
```

Dev `.env` holds weak/dummy values on purpose (`DEBUG=True`, a throwaway `SECRET_KEY`, a local Postgres container). Prod `.env` holds real values and lives *only* on the target host — transferred by `scp` over SSH, never by email, chat, or a shared doc. `docker-compose.yml` reads it via `env_file: .env`; Django reads every setting from `os.environ` inside the container exactly as it would outside one — the settings split from Part I 1.4 carries over unchanged.

The dev/prod compose files should differ only where they need to: dev mounts your code as a live volume so edits reload instantly and runs `manage.py runserver` inside the container (fine in dev, since only your machine can reach it); prod copies code into the image at build time and always runs Gunicorn — `runserver` inside an internet-facing container is exactly as unsafe as it is outside one.

Checklist — 2.4 Environment variables

- ☐ `.env.example` committed with every variable name and a placeholder value
- ☐ `.env` gitignored from the first commit, never after the fact
- ☐ Prod `.env` transferred only via SSH/secrets manager, never chat or email
- ☐ Dev compose runs `runserver` + live-mounted code; prod compose runs Gunicorn + baked-in code
- ☐ Django reads every environment-specific value from `os.environ`, nothing hardcoded per-environment in code

2.5 Local development in containers

The point of this step

A new developer (or you, six months later, on a different laptop) should be able to clone the repo and run one command to get a working local environment — no "first install Python 3.12, then Postgres, then..." checklist of their own.

Walkthrough

```
docker compose up
```

This should bring up `web` + `db` (+ `redis` if used) together, with `docker-compose.override.yml` automatically layered over `docker-compose.yml` since Compose loads both by default. Give the database a named volume so `docker compose down` doesn't wipe your local data every time you stop the stack:

```
volumes:
  postgres_data:/var/lib/postgresql/data
```

From here, the normal Django workflow runs through `exec`:

```
docker compose exec web python manage.py migrate
docker compose exec web python manage.py createsuperuser
docker compose exec web python manage.py shell
```

Port mapping (`8000:8000`) only belongs in the dev override file — production exposes nothing from `web` directly to the host; only the `nginx` container publishes ports,

matching the same principle from Part I where Gunicorn was never reachable except through Nginx.

Checklist — 2.5 Local development

- ☐ `docker compose up` brings up the full stack with one command
- ☐ Named volume for Postgres data, survives `docker compose down`
- ☐ Code bind-mounted in dev for hot reload
- ☐ `docker compose exec web ...` is the standard workflow for management commands
- ☐ Host ports exposed only in the dev override, never in the prod compose file

2.6 Production docker-compose

Walkthrough

The production compose file (full version in the Appendix) encodes the same reliability guarantees systemd gave you in Part I, in Compose's vocabulary:

```
services:
  web:
    image: ghcr.io/yourorg/appname:latest
    restart: unless-stopped        # systemd's Restart=always, Compose-flavoured
    depends_on:
      db:
        condition: service_healthy # don't start web until Postgres is actually ready, not just started
```

`depends_on` alone only waits for the container to *start*, not for Postgres to actually be accepting connections — pair it with a real `healthcheck:` on the `db` service, or you'll intermittently see `web` crash-loop on boot because it tried to connect a few seconds too early.

Resource limits matter more here than on a single-app VM, because one runaway container can now starve four sibling apps sharing the same host:

```
    deploy:
      resources:
        limits:
          memory: 512M
```

And logging needs an explicit rotation policy — container logs otherwise accumulate on disk exactly like the unrotated Nginx logs in 1.11, just less visibly:

```
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
```

Checklist — 2.6 Production docker-compose

- ☐ `restart: unless-stopped` (or `always`) on every service
- ☐ Only `nginx` publishes host ports; `web`/`db`/`redis` stay on the internal network
- ☐ `depends_on` paired with a real `healthcheck` on `db`
- ☐ Memory/CPU limits set per service
- ☐ Log rotation configured explicitly
- ☐ Named volumes for anything that must survive a redeploy: Postgres data, media, certs

2.7 Building, tagging, and shipping the image

Why the VM should never run `docker build`

If the production VM builds its own image, you can never be fully certain what's actually running there matches what was tested — a `pip install` during that build could silently pull a newer dependency version than the one your CI tested against. Building in CI and only ever pulling a pinned, tested image on the VM closes that gap: what's running in production is provably the exact artifact that passed your pipeline.

Walkthrough

Pick a registry — GitHub Container Registry (`ghcr.io`) is the simplest choice if your code already lives on GitHub, since it needs no separate billing setup and integrates directly with GitHub Actions. A basic pipeline (full version in the Appendix): on push to `main`, build the image, tag it with the git SHA and `latest`, push both tags, then SSH into the VM and run:

```
docker compose pull
docker compose up -d
docker compose exec -T web python manage.py migrate --noinput
```

That sequence mirrors the deploy script from 1.10 almost exactly — pull new code (here, a new image instead of a `git pull`), bring the stack up with it, run migrations. The concepts didn't change moving from Part I to Part II; only the packaging did.

Checklist — 2.7 Building and shipping

- ☐ Images built and pushed from CI, never built by hand on the production VM
- ☐ Registry chosen (`ghcr.io` is the simplest default), pull-scoped credentials on the VM
- ☐ CI pipeline: build → tag (SHA + `latest`) → push → SSH deploy → `docker compose pull && up -d` → `migrate`

2.8 Handover to the client

Why this is part of the technical work, not an afterthought

A client who can't restart their own app without calling you isn't fully operational — and a support relationship that was never made explicit in writing is the single most common source of an awkward, unpaid, indefinite support obligation. This section is as much about protecting the relationship as it is about documentation.

What to actually hand over

A one-page runbook: how to check if the app is up (`docker compose ps`), how to restart it (`docker compose restart`), how to view logs (`docker compose logs -f web`), and who to call if that's not enough. Credentials through a proper channel — a shared password manager entry, not a WhatsApp message — covering VM SSH access, Cloudflare account access (if the client owns the domain), database admin credentials, and the Django admin superuser. Explicit, written clarity on domain/DNS ownership: does the client hold the Cloudflare account, or do you manage it under an ongoing support agreement — settle this before go-live, not after a dispute. A demonstrated (not just documented) backup restore. And, if you're not staying on as the operator, a handover complete enough that a different competent developer could pick it up cold from the repo alone — the architecture diagram and `.env.example` left in the repo are what make that possible a year later, including for you.

Checklist — 2.8 Handover

- ☐ One-page runbook: status check, restart, logs, escalation contact
- ☐ Credentials handed over via a proper channel, never plaintext chat/email
- ☐ Domain/Cloudflare ownership clarified in writing before go-live
- ☐ Backup restore demonstrated to the client or their IT contact
- ☐ Support/maintenance agreement in place if you're staying on as operator
- ☐ `.env.example` and an architecture diagram left in the repo for future maintainers

---

PART III — USER MANAGEMENT & LOGIN

3.1 Start every project with a custom user model

Why this is the one decision you cannot cheaply undo

Django's default `User` model works fine — until the day you need to add a `phone_number` field, or switch to email-based login, or add an `organisation` foreign key, and discover that swapping `AUTH_USER_MODEL` after the first migration has already run means hand-editing migration history across every app that has a `ForeignKey` to `User`. It's not impossible, but it's genuinely painful, and entirely avoidable by spending five minutes on this before `migrate` ever runs the first time.

Walkthrough

Before your first migration, create an `accounts` app:

```
python manage.py startapp accounts
```

```
# accounts/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models

class User(AbstractUser):
    phone_number = models.CharField(max_length=20, blank=True)
    organisation = models.CharField(max_length=120, blank=True)
```

```
# settings/base.py — before the first migrate, not after
AUTH_USER_MODEL = 'accounts.User'
```

If you're changing how users are identified (email instead of username, for instance), you also need a custom manager:

```
# accounts/models.py
from django.contrib.auth.models import BaseUserManager

class UserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra):
        user = self.model(email=self.normalize_email(email), **extra)
        user.set_password(password)
        user.save()
        return user
```

Everywhere else in the codebase, reference the user model through settings, never by importing it directly — this is what keeps the model swappable and avoids circular imports:

```
# correct, everywhere in your codebase
from django.conf import settings
author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

# wrong — couples every app to one concrete model
from accounts.models import User
```

Checklist — 3.1 Custom user model

- ☐ `accounts` app with a custom `User(AbstractUser)` created before the first migration
- ☐ `AUTH_USER_MODEL = 'accounts.User'` set before `migrate` ever runs
- ☐ Custom `UserManager` written if changing the identifying field (e.g. email login)

- ☐ Every `ForeignKey`/`OneToOneField` to the user uses `settings.AUTH_USER_MODEL`, never a direct import

3.2 Session-based login, done properly

What's actually happening when a user logs in

Django's session framework stores a signed cookie in the browser containing a session ID; the actual session data (including "which user is this") lives server-side, in the database by default. `login()` writes that association; `logout()` clears it. This is simpler and more secure by default than rolling your own token scheme, and is the right default for anything that's primarily a browser-based application rather than an API consumed by a separate mobile app or SPA.

Walkthrough

Use Django's built-in views rather than hand-rolling authentication — `authenticate()` and `login()` already handle password hashing comparison, timing-safe checks, and session creation correctly:

```
# accounts/views.py
from django.contrib.auth import authenticate, login
from django.contrib.auth.views import LoginView

class MyLoginView(LoginView):
    template_name = 'accounts/login.html'
```

Tune session behaviour deliberately rather than leaving Django's defaults unexamined:

```
# settings/prod.py
SESSION_COOKIE_AGE = 1800          # 30 minutes — tighter for financial/sensitive systems
SESSION_EXPIRE_AT_BROWSER_CLOSE = True   # for genuinely sensitive systems
```

Logout should be a POST, not a clickable GET link — recent Django versions require this by default with `LogoutView`, and it's correct: a GET-based logout can be triggered by a malicious `<img src="/logout/">` on another page, logging a user out without their intent (a minor CSRF-adjacent annoyance, but a real one).

Checklist — 3.2 Session-based login

- ☐ Login implemented via Django's built-in `LoginView`/`authenticate()`, not hand-rolled password checking
- ☐ `SESSION_COOKIE_AGE` set deliberately for the system's sensitivity level
- ☐ `SESSION_EXPIRE_AT_BROWSER_CLOSE` considered for sensitive systems
- ☐ Logout implemented via POST, using Django's `LogoutView`

3.3 Password policy

Why the defaults are a floor, not a ceiling

Django's `AUTH_PASSWORD_VALIDATORS` ship with reasonable defaults (minimum length, common-password rejection, no all-numeric passwords, no obvious similarity to the user's own name/email). They're a genuine floor worth keeping — but "genuine floor" means raising the minimum length for anything client-facing, not just accepting the default 8 characters.

Walkthrough

```
# settings/base.py
AUTH_PASSWORD_VALIDATORS = [
    {'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator'},
    {'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
     'OPTIONS': {'min_length': 10}},
    {'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator'},
    {'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator'},
]
```

Wire password reset through Django's built-in view chain rather than building a custom flow — it already handles the security-sensitive parts correctly (single-use tokens, expiry, not leaking whether an email exists):

```
# urls.py
from django.contrib.auth import views as auth_views

urlpatterns += [
    path('password-reset/', auth_views.PasswordResetView.as_view(), name='password_reset'),
    path('password-reset/done/', auth_views.PasswordResetDoneView.as_view(),
name='password_reset_done'),
    path('reset/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(),
name='password_reset_confirm'),
    path('reset/done/', auth_views.PasswordResetCompleteView.as_view(),
name='password_reset_complete'),
]
```

This requires real SMTP (see Part I, 1.4) — never email a plaintext password to a user, ever, for any reason; the built-in flow never does this, it emails a single-use link.

Checklist — 3.3 Password policy

- ☐ `AUTH_PASSWORD_VALIDATORS` kept enabled, `min_length` raised to 10-12 for client-facing systems
- ☐ Password reset uses Django's built-in view chain with real SMTP
- ☐ No custom "email me my password" flow, ever
- ☐ `update_session_auth_hash` behaviour confirmed after password changes (built-in views already handle this)

3.4 Brute-force and login-abuse protection

The gap Django's built-in auth leaves open

Django's auth system has no rate limiting at all by default — nothing stops an attacker from trying thousands of password guesses against a login form in an automated script. This is the single most commonly missing piece in a from-scratch Django deployment, because the tutorials never mention it — auth "works" without it, right up until it doesn't.

Walkthrough

```
pip install django-axes
```

```
# settings/base.py
INSTALLED_APPS += ['axes']
AUTHENTICATION_BACKENDS = [
    'axes.backends.AxesStandaloneBackend',
    'django.contrib.auth.backends.ModelBackend',
]
MIDDLEWARE += ['axes.middleware.AxesMiddleware']

AXES_FAILURE_LIMIT = 5
AXES_COOLOFF_TIME = 1   # hours
```

Tune the threshold to the client's risk profile — a public-facing client portal warrants a stricter limit than an internal staff tool only reachable from an office network. Log failed logins (never the attempted password itself) so a credential-stuffing pattern is visible in your monitoring before it succeeds.

Checklist — 3.4 Brute-force protection

- ☐ `django-axes` (or equivalent) installed and configured
- ☐ Lockout threshold tuned to the system's actual risk profile
- ☐ Failed logins logged, without ever logging the attempted password
- ☐ CAPTCHA added only in response to observed abuse, not by default

3.5 Admin site hardening

Why the default `/admin/` URL is a liability

Automated scanners probe `/admin/` on every public IP constantly, looking for exactly the login form Django's admin ships by default. None of the following steps are about being unbreakable — they're about not being the easiest target on the internet.

Walkthrough

```
# urls.py — change the URL to something non-obvious
urlpatterns = [
    path('mgmt-portal-x7k/', admin.site.urls),
]
```

Layer an IP restriction at Nginx for client staff with predictable IPs (office network, VPN range) — genuine defense in depth, since it means a stolen admin password alone isn't enough:

```
location /mgmt-portal-x7k/ {
    allow 41.x.x.x;   # office IP range
    deny all;
    proxy_pass http://unix:/srv/appname/run/gunicorn.sock;
}
```

Never hand a client a shared `admin/admin`-style login — every staff member gets their own named account, both for accountability (you can tell *who* changed a record) and so revoking one person's access doesn't require rotating a password everyone shares.

Checklist — 3.5 Admin hardening

- ☐ Default `/admin/` URL changed to something non-obvious
- ☐ Admin access IP-restricted at Nginx where the client's staff have predictable IPs
- ☐ Every admin user has their own named account, never a shared login

3.6 Multi-factor authentication

Why this is the highest-leverage addition on this list

A leaked or guessed password is common; a leaked password *and* a live 2FA code from the same person's phone is rare. For any system touching client financial or personal data, this single addition does more to prevent account takeover than everything else in Part III combined.

Walkthrough

```
pip install django-otp qrcode
```

```
# settings/base.py
INSTALLED_APPS += ['django_otp', 'django_otp.plugins.otp_totp']
MIDDLEWARE += ['django_otp.middleware.OTPMiddleware']
```

TOTP (Google Authenticator/Authy-compatible) is the right default over SMS-based 2FA here — it works entirely offline once enrolled, which matters given how variable mobile network reliability can be, and it avoids the cost and delivery-reliability problems of SMS gateways entirely.

Enforce it for staff/admin roles at minimum; offer it optionally to regular users on sensitive systems. Generate backup/recovery codes at enrolment and show them exactly once — a lost phone without a recovery code means a permanently locked-out user.

Checklist — 3.6 Multi-factor authentication

- ☐ `django-otp` (TOTP) integrated
- ☐ 2FA enforced for staff/admin accounts on any system touching financial or personal data
- ☐ Backup/recovery codes generated and shown once at enrolment

3.7 Permissions and groups

Why to reach for Django's built-in system first

Most SME applications need 2-4 role tiers (staff, manager, admin) — Django's `Group` + `Permission` system covers this without adding a dependency. Reach for something heavier (`django-guardian` for per-object permissions) only when you have a concrete requirement it actually solves, not speculatively.

Walkthrough

```
# a one-time setup script or a data migration
from django.contrib.auth.models import Group, Permission

managers = Group.objects.create(name='Managers')
managers.permissions.add(Permission.objects.get(codename='change_invoice'))
```

```
# checking in a view
if request.user.groups.filter(name='Managers').exists():
    ...
```

Keep `is_staff` reserved strictly for admin-site access, and `is_superuser` for actual superusers — model your application's own roles separately (a `role` field, or group membership), rather than overloading Django's admin-access flags to also mean "has permission X in my app." Conflating the two eventually produces a user who needs app-level manager permissions but shouldn't have Django admin access, and there's no clean way to express that if you've been using `is_staff` for both jobs.

Checklist — 3.7 Permissions and groups

- ☐ Role tiers modelled with Django's `Group`/`Permission` system before reaching for a third-party package
- ☐ `django-guardian` (object-level permissions) added only for a concrete, confirmed need
- ☐ `is_staff`/`is_superuser` reserved for admin-site access only, application roles modelled separately

3.8 Auth checklist summary

Paste this into your project's kickoff doc — the compressed version of everything above, once you already understand why each line is there:

- ☐ Custom user model created before first migration
- ☐ Session-based login via Django's built-in views, cookie age tuned
- ☐ Password validators tuned, reset flow uses real SMTP
- ☐ Brute-force protection (`django-axes`) installed
- ☐ Admin URL obscured + IP-restricted for client-facing systems
- ☐ 2FA on all staff/admin accounts for any system touching client financial/personal data
- ☐ Roles modelled via groups, not by overloading `is_staff`

APPENDIX — COPY-PASTE REFERENCE CONFIGS

A.1 systemd unit — `/etc/systemd/system/appname.service`

```
[Unit]
Description=Gunicorn daemon for appname
After=network.target postgresql.service

[Service]
Type=notify
User=appname
Group=appname
RuntimeDirectory=appname
WorkingDirectory=/srv/appname
EnvironmentFile=/srv/appname/.env
ExecStart=/srv/appname/venv/bin/gunicorn \
    --workers 3 \
    --bind unix:/srv/appname/run/gunicorn.sock \
    --timeout 30 \
    config.wsgi:application
ExecReload=/bin/kill -s HUP $MAINPID
```

```
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

A.2 Nginx server block — `/etc/nginx/sites-available/appname.conf`

```
server {
    listen 443 ssl http2;
    server_name yourdomain.co.ug;

    ssl_certificate     /etc/nginx/ssl/appname.pem;
    ssl_certificate_key /etc/nginx/ssl/appname.key;

    client_max_body_size 20M;

    location /static/ {
        alias /srv/appname/static/;
        expires 30d;
        access_log off;
    }

    location /media/ {
        alias /srv/appname/media/;
        expires 7d;
    }

    location / {
        proxy_pass http://unix:/srv/appname/run/gunicorn.sock;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
    }

    add_header X-Frame-Options DENY;
    add_header X-Content-Type-Options nosniff;
}

server {
    listen 80;
    server_name yourdomain.co.ug;
    return 301 https://$host$request_uri;
}
```

A.3 `.env.example`

```
DJANGO_SETTINGS_MODULE=config.settings.prod
SECRET_KEY=change-me
```

```
DEBUG=False
ALLOWED_HOSTS=yourdomain.co.ug

DATABASE_URL=postgres://appname_user:password@localhost:5432/appname_db

EMAIL_HOST=smtp.mailgun.org
EMAIL_PORT=587
EMAIL_HOST_USER=postmaster@yourdomain.co.ug
EMAIL_HOST_PASSWORD=change-me
EMAIL_USE_TLS=True

REDIS_URL=redis://localhost:6379/0
```

A.4 Production `Dockerfile` (multi-stage)

```
# ---- builder ----
FROM python:3.12-slim-bookworm AS builder
WORKDIR /app
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
RUN apt-get update && apt-get install -y --no-install-recommends build-essential libpq-dev && rm -rf
/var/lib/apt/lists/*
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# ---- final ----
FROM python:3.12-slim-bookworm
WORKDIR /app
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1 PATH=/home/appuser/.local/bin:
$PATH
RUN apt-get update && apt-get install -y --no-install-recommends libpq5 && rm -rf
/var/lib/apt/lists/* \
    && adduser --system --group appuser
COPY --from=builder /root/.local /home/appuser/.local
COPY --chown=appuser:appuser . .
USER appuser
RUN python manage.py collectstatic --noinput --settings=config.settings.prod || true
EXPOSE 8000
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "--workers", "3", "config.wsgi:application"]
```

A.5 Production `docker-compose.yml`

```
services:
  web:
    image: ghcr.io/yourorg/appname:latest
    restart: unless-stopped
    env_file: .env
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - static_volume:/app/staticfiles
      - media_volume:/app/media
```

```
    deploy:
      resources:
        limits:
          memory: 512M

  db:
    image: postgres:16-alpine
    restart: unless-stopped
    env_file: .env
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER}"]
      interval: 5s
      timeout: 5s
      retries: 5

  nginx:
    image: nginx:1.27-alpine
    restart: unless-stopped
    ports:
      - "443:443"
      - "80:80"
    volumes:
      - ./nginx/appname.conf:/etc/nginx/conf.d/default.conf:ro
      - ./nginx/ssl:/etc/nginx/ssl:ro
      - static_volume:/app/staticfiles:ro
      - media_volume:/app/media:ro
    depends_on:
      - web

volumes:
  postgres_data:
  static_volume:
  media_volume:
```

A.6 `docker-compose.override.yml` (local dev — auto-loaded)

```
services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    env_file: .env
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db

  db:
    ports:
      - "5432:5432"
```

A.7 Minimal GitHub Actions deploy pipeline — `.github/workflows/deploy.yml`

```
name: Build and Deploy
on:
  push:
    branches: [main]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Log in to GHCR
        run: echo "${{ secrets.GHCR_TOKEN }}" | docker login ghcr.io -u ${{ github.actor }} --
password-stdin

      - name: Build and push
        run: |
          docker build -t ghcr.io/yourorg/appname:${{ github.sha }} -t
ghcr.io/yourorg/appname:latest .
          docker push ghcr.io/yourorg/appname:${{ github.sha }}
          docker push ghcr.io/yourorg/appname:latest

      - name: Deploy over SSH
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ secrets.PROD_HOST }}
          username: ${{ secrets.PROD_USER }}
          key: ${{ secrets.PROD_SSH_KEY }}
          script: |
            cd /srv/appname
            docker compose pull
            docker compose up -d
            docker compose exec -T web python manage.py migrate --noinput
```

A.8 Security quick-reference (both deployment styles)

- ☐ `DEBUG=False` in every production environment, verified after every deploy
- ☐ `SECRET_KEY` unique per environment, never committed
- ☐ Database never exposed to the public internet
- ☐ TLS via Cloudflare Full (strict) + Origin CA cert
- ☐ Firewall default-deny, only 22/80/443 open
- ☐ Non-root process/container users everywhere
- ☐ Automated, tested backups
- ☐ 2FA on all admin/staff accounts for client-facing systems
- ☐ Dependency updates checked periodically (`pip list --outdated`, Dependabot/Renovate on the repo)

End of manual. Treat every checklist above as a template — copy it into your project's own `DEPLOYMENT.md` per client, and tick items off for real rather than trusting memory across 5 concurrent systems.