**Summary — how many models now?**

**8 core models** (plus Django User which you already have). Reason: these cover templates, runtime tenant instances, per-tenant services, deployments and step-by-step logs, scheduled job metadata, and encrypted integration secrets.

Models:

1. ProjectTemplate
2. ServiceTemplate
3. Tenant
4. TenantService
5. Deployment
6. DeploymentStep (structured step/log entries)
7. JobRecord
8. IntegrationSecret

(Optionally add AuditEntry later for user actions — not required for core orchestration.)

**Detailed models & fields (design + purpose)**

**1) ProjectTemplate — blueprint for a product (DMS, HRMS, ...)**

Purpose: define default behavior and base-domain for all tenants of this project.

Important fields:

* id, name, slug, description
* base\_domain (example: delivery.pk OR hrms.pk) — used to create tenant subdomains like client1.delivery.pk
* db\_required (bool) — the simplified UX question: yes/no
* db\_type (choice) — default 'postgres' (choices: postgres, mysql, mongodb, others)
* default\_env (JSON, optional) — environment variables templates
* notify\_emails (JSON list or Text) — email(s) to notify on success/failure
* created\_by, created\_at, updated\_at, active (bool)

Why: base\_domain + db\_required let the UI present the two DB questions you requested. ServiceTemplates for frontend/backend are still supported if user wants to define extra services.

**2) ServiceTemplate**

Purpose: describes a service type inside a ProjectTemplate (backend, frontend, worker, etc.). All services are optional; a project may have none or many.

Important fields:

* id, project (FK ProjectTemplate)
* name (e.g., backend or frontend or search-worker)
* service\_type (choice: backend, frontend, db, worker, cron, etc.)
* repo\_url (nullable) — repo for this service (empty if none)
* repo\_branch (nullable)
* build\_config (JSON) — build options (dockerfile path, publish dir, static SPA flag...)
* expose\_domain (bool) — whether to create a public domain for this service (frontend typically true, backend often true)
* internal\_provision\_endpoint (nullable) — relative path to call (e.g. auth/create-superuser-view/)
* internal\_provision\_token\_secret (FK to IntegrationSecret or null) — token stored securely
* order (int) — orchestration order (lower runs earlier)
* active (bool)
* created\_at, updated\_at

Why: one model per declared service. The UI can create service templates; in simple flows you create 2 templates (backend, frontend) or only one. For the DB simplified UI you can create a DB ServiceTemplate automatically when db\_required is true.

**3) Tenant**

Purpose: a deployed instance of a ProjectTemplate — each tenant has own subdomain(s) and will receive deployments.

Important fields:

* id, project (FK), name (client name), client\_ref (optional external id)
* subdomain (string, unique, sanitized, lowercase) — e.g., client1
* status (choice: pending, provisioning, waiting\_for\_internal\_provision, running, failed, completed)
* created\_by, created\_at, updated\_at, completed\_at
* detail (short text/capped)
* active (bool)

Why: this is the tenant identifier used across UI and scheduling. subdomain uniqueness prevents collisions.

**4) TenantService**

Purpose: per-tenant runtime artifact for each service (maps to a ServiceTemplate). Holds Dokploy ids, domain, db credentials and boolean flags for resume idempotency.

Important fields:

* id, tenant (FK), service\_template (FK), name, service\_type
* **repo fields**: repo\_url, repo\_branch (copied from template or overridden)
* runtime identifiers: app\_id (dokploy application id), db\_id, domain, domain\_id
* flags for idempotency: created (bool), git\_attached (bool), build\_configured (bool), env\_configured (bool), deploy\_triggered (bool), deployed (bool)
* health info: health\_status (ok/unhealthy/unknown), health\_tries (int), next\_wait\_seconds (int)
* last\_deployed\_at, created\_at, updated\_at
* detail (short field for summary)

Why: keeps per-tenant-per-service runtime state and booleans so your orchestration can be resume-aware exactly like current flags but scoped to the service (not to a big ProvisionRequest).

**5) Deployment**

Purpose: a single deployment attempt for a Tenant (or redeploy/resume). Central record that UI will read to show timeline / logs.

Important fields:

* id, tenant (FK), triggered\_by (FK user), trigger\_reason (enum: initial, redeploy, resume, manual\_fix), status (pending,running,failed,succeeded)
* started\_at, ended\_at, duration\_seconds (computed)
* meta (JSON) — optional: overrides, progress markers
* summary (text; small)
* created\_at, updated\_at

Why: every deploy gets a record to group step logs and to resume the same deployment.

**6) DeploymentStep (structured logs + step state)**

Purpose: this model represents a **single ordered step** in a Deployment. It gives the exact step names you specified (project created, backend created, git provider set, build config done, DB created, DB deployed, backend deployed, frontend created, domain created, domain propagation, health-check, retries, internal-provision, email). Having explicit step records makes UI trivial to show completed/remaining steps and resume behavior.

Important fields:

* id, deployment (FK Deployment), tenant\_service (FK TenantService, optional) — link step to a service when relevant
* step\_key (string, e.g. project.create, service.backend.create, service.backend.git\_attach, service.backend.build\_config, db.create, db.deploy, service.backend.deploy, domain.create, domain.propagation\_wait, health.check, health.retry, internal.provision, email.notify)
* order (int) — ordering for display / enforcement
* status (enum: pending, running, success, failed, skipped)
* attempts (int)
* started\_at, ended\_at
* message (text short) — last human-friendly log line
* meta (JSON) — free-form (API responses, error codes, job ids)
* created\_at, updated\_at

How UI uses it:

* Show ordered list of DeploymentSteps for a given Deployment with their status and message.
* To resume, orchestrator queries the DeploymentSteps: run the next step with status in (pending, failed) as per your resume semantics (you can choose to resume only failed steps or continue from last pending).
* Each step can be retried and attempts incremented; step meta store Dokploy job ids, response ids, and scheduler job ids for health/email.

Why: this replaces the huge free-form detail string — it's structured, searchable, joinable, and limited in size.

**7) JobRecord**

Purpose: metadata for scheduled APScheduler jobs (so UI and API can cancel/resume jobs; tracks attempts and next run).

Important fields:

* id, job\_id (string APScheduler id), deployment (FK), step (FK DeploymentStep), job\_type (enum: health, email, retry), attempt (int), next\_run\_at (datetime), status (scheduled/running/completed/cancelled), result\_meta (JSON), created\_at, updated\_at

Why: although APScheduler has its own persistence, storing a DB record lets your UI list jobs and control them easily.

**8) IntegrationSecret**

Purpose: centrally store encrypted secrets for projects or services (internal provision tokens, Dokploy API key if per-project).

Important fields:

* id, project (FK optional) or service\_template (FK optional), name, secret\_type (enum), encrypted\_value (encrypted field), created\_by, created\_at, updated\_at, last\_rotated\_at

Why: don't store tokens in plain text in ServiceTemplate. Use encrypted field or a vault integration. Link token to template so tenant inherits it.

**How logs / steps reflect the precise flow you described**

We will model the deployment as an ordered list of standardized step\_keys. Example canonical sequence (example for a project with backend+db+frontend):

1. project.create — project record creation (if applicable)
2. service.{backend}.create — backend app creation (sets app\_id)
3. service.{backend}.git\_attach
4. service.{backend}.build\_config
5. service.{backend}.env\_set (includes DB envs)
6. db.create (if db\_required)
7. db.deploy (trigger DB deploy)
8. service.{backend}.deploy (trigger backend deploy)
9. service.{backend}.wait\_deploy (optional wait)
10. service.{frontend}.create
11. service.{frontend}.git\_attach
12. service.{frontend}.build\_config
13. service.{frontend}.deploy
14. domains.create
15. domains.wait\_propagation
16. health.check (per service as separate steps)
17. health.retry (if check failed; repeated step instances)
18. internal.provision — call internal endpoint
19. email.notify\_success / email.notify\_failure

Each of these is a DeploymentStep row. The step model stores attempts, message, meta (API responses), and status. When a job runs (APScheduler job), it should update the corresponding DeploymentStep to running and then to success or failed, increment attempts, and append meta. The UI will therefore show exactly which steps succeeded and which are pending/failed and how many tries happened.

Resuming:

* On resume you load the Deployment; find the first step with status in pending or failed. The orchestrator runs that step. Because TenantService stores idempotency flags (created/git\_attached/build\_configured/...), the step should be implemented idempotently (skips if already done). The DeploymentStep status is the single source-of-truth for visible logs and progress.

Example UI display per step:

* Step name: service.backend.create
* Status: success
* Attempts: 1
* Started: 2025-09-08T12:00:00Z
* Ended: 2025-09-08T12:00:03Z
* Message: applicationId=abc123
* Button: View details (shows meta JSON or raw response)

**Why this is resilient & foolproof**

* **Structured steps + idempotent flags:** the orchestrator can always determine what to run next and skip already-done work (no guesswork from free-form detail).
* **JobRecord + job ids persisted:** scheduler jobs can be cancelled or re-scheduled from UI.
* **Separation of template vs runtime state:** templates are immutable blueprints; TenantService holds runtime identifiers, avoiding mixing template and runtime.
* **Attempt counters & backoff:** DeploymentStep attempts + JobRecord support exponential backoff and gives full visibility.
* **IntegrationSecret encryption:** tokens are not exposed in DB nor in UI (except masked).
* **Logs searchable & bounded:** each DeploymentStep message/meta is limited and structured, preventing unbounded growth.

**Example counts and storage concerns**

* Each deployment will produce ~10–30 DeploymentStep rows depending on project complexity — small and fast to query.
* DeploymentStep.meta can hold response snippets; cap size (text limit) or store large payloads in separate log storage (S3) and reference via meta.
* Keep retention policy for Deployment and DeploymentStep (e.g., delete older than X days, archive).