

ClubManager — Model & Domain Architecture

Baseline reference for implementing the domain models. This describes the **intended shape** of the data model: what exists today, what is planned, and the conventions every app should follow. It is a living document — update it when the model changes.

Tenancy: multi-tenant (row-based / shared-schema). ClubManager is designed as a **multi-tenant platform** — one deployment serves many clubs, with `Club` as the **tenant root**. Isolation is **row-based**: a shared database and schema where every club-owned row carries a `club` FK (via `ClubScopedModel`), and *all* access is scoped to the current tenant. The mechanics — tenant resolution, scoping manager, per-club uniqueness — are specified in §2.4.

⚠ **This supersedes** `CLAUDE.md` , which currently states the app is "deliberately *not* multi-tenant ... there is no `club_id` tenancy." That guidance and the project memory are now **out of date** and must be updated to match this document — see the **banner at the foot of this file**. Where the two disagree, this architecture is the intended direction.

1. App decomposition

The `isort` known-first-party roadmap lists nine apps. The build split the original `accounts` app into `authentication` + `club` . Per the decisions in §7, the people models (`Member` , `Family`) **move out of** `authentication` **into a dedicated** `members` **app**, and two apps (`formbuilder` , `shop`) are added **beyond the original roadmap** — add all new labels to `known-first-party` in `pyproject.toml` when they land. The target decomposition:

App	Status	Responsibility	Models
authentication	built	Login identity + tenancy/role services (global, cross-club)	User
members	planned	People: person records, families (extract from <code>authentication</code>)	Member , Family , FamilyMembership
club	built	Tenant root, season , season-scoped affiliation, club roles	Club , Season (<i>planned</i>), ClubMembership , ClubRole (<i>planned</i>)
teams	planned	Teams and season rosters	Team , TeamMembership , StaffAssignment
events	planned	Training / matches / social events + attendance	Event , Attendance

App	Status	Responsibility	Models
news	planned	Editorial news for the public site	Article , Category
pages	planned	Flat CMS pages for the public site	Page
home	planned	Homepage composition / featured content	HomeConfig (per-club) or config-only
formbuilder	planned	Admin-defined dynamic forms + submissions + reporting	Form , Field , Submission , Answer
shop	planned	Cart-like shop, orders, payments, PDF invoices	Product , Cart , CartItem , Order , OrderLine , Payment , Invoice
search	planned	Site search (likely no models; index/config only)	—

User **stays global** (one login identity across the whole platform); everything else that belongs to a club is tenant-scoped (§2.4). This is why `Member` — a *person within a club* — lives in its own app and carries a `club` FK, while `User` does not.

Open decision (§7): whether to migrate `Member` / `Family` out of `authentication` into a dedicated `members` app. Recommendation: keep them in `authentication` for now; revisit only if the app grows unwieldy. The roadmap `members` name is reserved either way.

2. Shared conventions

These are already established in code — every new model follows them.

- **UUID primary keys.** Inherit `clubmanager.base.UUIDModel` (`id = UUIDField(default=uuid4)`). Never expose sequential integer PKs.
- `ClubScopedModel` (`clubmanager.base`) adds the tenant `club` FK (`related_name="%sclub"`) and, under multi-tenancy, a tenant-aware manager + auto club-stamping `save()` (§2.4). **Every aggregate-root model inherits it**; leaf rows reachable only via a scoped parent (e.g. `Attendance` via `Event`) may inherit scope from the parent, though denormalising `club` onto them is recommended (§2.4).
- **i18n everywhere.** Every field gets a `gettext_lazy` verbose name; every model sets `verbose_name` / `verbose_name_plural` and a sensible `Meta.ordering` . `TextChoices` labels are translated too.
- `__str__` **on every model**, human-readable.
- **Enumerations** use nested `models.TextChoices` (see `FamilyMembership.FamilyRole`).
- **Phone numbers** use `PhoneNumberField` (from `django-phonenumbers-field`), nullable.
- **Through models** for many-to-many relationships that carry data (role, jersey number, attendance status) — never a bare `ManyToManyField` when the link has attributes.
- **Business logic in** `services/` , not fat models or views (see `authentication/services/member_csv_importer.py`). Management commands are thin wrappers over

services (see `import_members_csv`).

- **Migrations are generated, not hand-edited** (ruff-excluded).

Naming & relations

- `related_name` is explicit and plural on the "many" side, chosen to read naturally from the parent (`club.members` , `family.memberships` , `member.family_memberships`).
- Deletion policy is deliberate per FK: `CASCADE` for owned children, `SET_NULL` (+ `null=True`) where the child should survive its parent (e.g. `Member.user`), `PROTECT` for references that must not silently disappear (planned: `Season` on rosters/events — see §5).

2.4 Multi-tenancy (row-based, shared schema)

`Club` is the **tenant root**. One deployment, one database, one schema; tenants are separated by a `club` FK on every owned row and by disciplined scoping of every query. This is the lightest multi-tenancy model and matches the existing `ClubScopedModel` scaffolding — no Postgres schemas, no per-tenant databases, no `django-tenants` .

What gets a `club` FK. Every *aggregate root* inherits `ClubScopedModel` and so carries `club` (`Member` , `Family` , `Season` , `Team` , `Event` , `ClubRole` , `Article` , `Category` , `Page` , `Form` , `Product` , `Cart` , `Order` , `Invoice` , ...). Leaf rows reachable only through a scoped parent (`Answer` → `Submission` , `OrderLine` / `Payment` → `Order` , `CartItem` → `Cart` , `Attendance` → `Event` , `TeamMembership` / `StaffAssignment` → `Team`) may inherit scope via the parent — but **denormalising `club` onto them too is recommended** for leak-proof filtering and DB-level constraints. `User` is the **only** global identity model; it has no `club` .

People vs. logins under tenancy. A `User` is one platform-wide login that may belong to several clubs; a `Member` is that person *within one club*. So:

- `Member.user` becomes a `ForeignKey` (not `OneToOneField`) — one user → many members (at most one per club): `unique_together (club, user)` .
- `User.get_full_name()` can no longer assume a single member; resolve the member **for the current club** (via the tenant context below), falling back to email.

Tenant resolution → `request.club` . A `ClubTenantMiddleware` resolves the active club per request (recommended: **subdomain**, `ajax-united.clubmanager.app` ; path-prefix `/c/<slug>/` is the alternative) and stores it on `request.club` and in a context variable so non-request code (services, management commands) can read it:

```
# clubmanager/tenancy.py
from contextvars import ContextVar
_current_club: ContextVar = ContextVar("current_club", default=None)

def set_current_club(club):
    _current_club.set(club)
def get_current_club():
    return _current_club.get()
def require_current_club():
    # raises if unset – use in write paths
    club = _current_club.get()
    if club is None:
        raise RuntimeError("No active club in context")
    return club

class ClubTenantMiddleware:
    # resolves subdomain -> Club, sets both
```

```
... # request.club = club; set_current_club(club)
```

Scoping manager. `ClubScopedModel` gets a tenant-aware manager so day-to-day queries can't accidentally cross tenants:

```
class TenantQuerySet(models.QuerySet):
    def for_club(self, club): return self.filter(club=club)
    def current(self):       return self.filter(club=require_current_club())

class ClubScopedModel(UUIDModel):
    club = models.ForeignKey("club.Club", on_delete=models.CASCADE, related_name="%s" % (class_name))
    objects = TenantQuerySet.as_manager()
    def save(self, *args, **kwargs):
        # auto-stamp club from context if unset
        if self.club_id is None: self.club = require_current_club()
        super().save(*args, **kwargs)
    class Meta: abstract = True
```

Prefer **explicit** `.for_club(club)` / `.current()` in views and services over a fully automatic global filter — auto-filtering via context state is convenient but hides tenant boundaries and bites hard in shells, tasks, and tests. Keep scoping visible.

Per-club uniqueness. Every constraint that was globally unique becomes **unique per club**. Concretely:

`Season.name`, all public `slug s` (`Article`, `Page`, `Form`, `Product`), `Team (season, name)`, `jersey numbers (team, jersey_number)`, and the human-readable counters `Order.number` / `Invoice.number` are scoped by / allocated per club. A bare `unique=True` on a tenant model is almost always a bug — use `UniqueConstraint(fields=["club", ...])`.

Cross-cutting consequences (checklist for every feature):

- Admin: register a `club` list-filter and scope `get_queryset` for non-superusers.
- Roles are **per-club** — see §3 (global Django Groups don't fit; use `ClubRole`).
- Sequential numbers (invoices) allocate per club inside a transaction — never `count()+1`.
- Tests must set a current club (a `with_club(club)` context-manager helper).
- Config: wildcard host + CSRF for subdomains; see §8.

3. Roles & access control (RBAC)

Authorization is **service-layer, and per-club** (§2.4). Because roles differ per tenant — a user can be a treasurer at club A and merely a member at club B — global Django `Group s` (which are platform-wide) **do not fit**. Roles are stored as tenant-scoped `ClubRole` rows and *all* permission decisions go through a single service module; no `django-guardian`, no per-club Group hacks. Django's own permission framework is retained **only** for the platform-operator layer (`is_staff` / `is_superuser` in Django admin).

3.1 Two layers

1. **Platform operators** — `User.is_superuser` / `is_staff`. Global, cross-club; run the Django admin, manage the tenant list. Not a club role.

2. **Club roles** — a member's standing *within one club*, stored per tenant. Two kinds:
- **Club-wide roles** → `ClubRole` rows (below).
 - **Object-scoped roles** → derived from the domain graph, no extra rows:
 - Coach / manager **of a specific team** → `StaffAssignment(team, member, role)` .
 - Parent / guardian **of a specific member** → `FamilyMembership` + the family graph.
 - Purchaser vs. beneficiary → `Order / OrderLine` + `ClubMembership` .

3.2 `ClubRole` — club-wide role assignments (*app: club*)

```
ClubRole(ClubScopedModel)           # ClubScopedModel -> carries `club` (§2.4)
  member FK Member (CASCADE, related_name="roles")
  role   CharField (TextChoices: MEMBER | EDITOR | TREASURER | BOARD)
  Meta: unique_together (club, member, role)
```

Role	Grants (representative)
<i>Public</i>	Anonymous — no row; read-only public site of that club.
MEMBER	View own + family data, own rosters/attendance, own orders/invoices, submit member-only forms.
EDITOR	Manage that club's news , pages , formbuilder content.
TREASURER	Manage that club's shop : products, orders, payments, issue/void invoices.
BOARD	Full management of that club: members, roles, all of the above.

COACH / TEAM_MANAGER are deliberately **not** `ClubRole`s — being a coach is always *of a team*, so it lives on `StaffAssignment` (§5.3). "Is this user a coach at this club?" = "do they have any `StaffAssignment` on a team in this club?".

3.3 The permission service

One module — `club/services/access.py` (or `authentication/services/access.py`) — answers every authorization question, always taking the club/object as an argument:

```
has_club_role(user, club, role)      -> bool          # ClubRole lookup
roles_in_club(user, club)            -> set[str]      # incl. derived COACH/MANAGER
teams_managed_by(user, club)         -> QuerySet[Team]
members_visible_to(user, club)       -> QuerySet[Member] # self + family + managed teams
can_edit_event(user, event)          -> bool
can_manage_shop(user, club)          -> bool          # TREASURER or BOARD
```

Views, admin, and templates call these — never re-derive access inline. Each helper scopes to the given club (§2.4), so a user's powers in club A never leak into club B.

3.4 Keeping roles in sync with domain state

ClubRole membership is **reconciled from domain facts**, not hand-assigned:

- An **active** ClubMembership for the club's current season → grant the MEMBER role; a lapsed one → revoke. (Season-scoped membership: §5.1.)
- A StaffAssignment needs no ClubRole — coach status is derived (§3.2).

Implement as `club/services/access.py::reconcile_roles(user, club)`, invoked on the state changes that matter (membership activation/lapse), so authorization never drifts from data.

4. Built models (as-is, + planned tenancy changes)

authentication

User — custom auth model, email is the login (`USERNAME_FIELD = "email"` , no username). UUID PK. `objects = UserManager()` (email-based `create_user` / `create_superuser`). **Stays global — the one model with no club FK** (§2.4). Because a user may belong to several clubs, `get_full_name` / `get_short_name` resolve the Member **for the current club** (via tenant context), falling back to email — they can no longer assume a single member.

members (*planned — extract from authentication*)

Member , Family , FamilyMembership **move here** and become tenant-scoped (`ClubScopedModel`).

Member — a *person within one club*. `user` becomes a `ForeignKey` (was `OneToOneField`), still nullable (`on_delete=SET_NULL`), so one login maps to one member *per club* (`unique_together (club, user)`) and members can exist without logins (children, imports). Holds name, DOB, contact email / phone / emergency_phone . `contact_email` prefers the member's own email, else the login email. `guardians` returns parent/guardian members reachable through shared families (all within the same club).

Family + FamilyMembership — households, tenant-scoped. FamilyMembership carries a `FamilyRole` (`parent` / `child` / `guardian` / `other`), `unique_together (family, member)` ; `Family.guardians` / `Family.children` are role-derived querysets. Powers the "parents see their children's data" object-scope (§3.1).

club

Club — **the tenant root** (§2.4). Currently just `name` ; extend with `slug` (unique, drives subdomain/path resolution), `contact`, and `branding`. Provide `Club.objects.current()` and resolve it in middleware — never hardcode a PK.

ClubMembership — links a Member to the Club with a `license` string. **Being made season-scoped** (§5.1): it gains a `season` FK and `sign-up` / `fee-status` fields, so each row is one member's affiliation for one season (`unique_together (club, member, season)`). This is the record the MEMBER role and shop fulfilment key off of (§3.4, §5.7).

5. Planned models (design)

Field lists below are **sketches** to implement against, not final migrations.

All sketches below are tenant-scoped: aggregate roots inherit `ClubScopedModel` (the `club` FK is shown implicitly and every listed `unique_together` is *within a club*, §2.4).

5.1 Season — the central organizing concept (*app: club*)

Everything time-bound hangs off a season. Rosters, events, and attendance are **season-scoped via FK** — never global state. Seasons are **per club** — each club runs its own.

```
Season(ClubScopedModel)          # -> carries `club`
  name          CharField         # e.g. "2025-2026"
  start_date    DateField
  end_date      DateField
  is_current    BooleanField      # exactly one true PER CLUB; enforce in save()/service
  Meta: unique_together (club, name); ordering = ["-start_date"]; get_latest_by = "start_date"
```

- Provide `Season.objects.current()` (scoped to the current club, §2.4) rather than scattering `is_current=True` filters.
- Referenced by `Team`, `Event`, and `ClubMembership`. Use `on_delete=PROTECT` on those FKs — deleting a season with data should be blocked.

Season-scoped `ClubMembership` (evolution of the built model, §4):

```
ClubMembership(ClubScopedModel)  # -> carries `club`
  member          FK Member (CASCADE, related_name="club_memberships")
  season          FK Season (PROTECT, related_name="memberships")
  license          CharField (blank)      # federation license for that season
  status          CharField (TextChoices: pending | active | lapsed | cancelled)
  fee_status       CharField (TextChoices: unpaid | partial | paid | waived)
  signed_up_at    DateTimeField (null)    # when the member registered for the season
  activated_at    DateTimeField (null)    # when membership became active (usually on payment)
  Meta: unique_together (club, member, season); ordering = ["-season__start_date", ...]
```

- One row per member **per season** — sign-up and fee payment are tracked independently each season. `unique_together` moves from `(club, member)` → `(club, member, season)` (a data migration must backfill existing rows with the current season).
- `fee_status` is the **source of truth for whether dues are paid**; it is driven by the shop (§5.7) — a paid membership `Order` flips `fee_status` → `paid` and `status` → `active`. Keep it denormalized here (fast to query "who hasn't paid") but only mutate it through a service that reconciles against `Payment`s, never by hand.
- `status = active` (for the club's current season) is the fact that grants the member's user the `MEMBER ClubRole` (§3.4).

5.2 teams

```

Team(ClubScopedModel)          # -> carries `club`
  season      FK Season (PROTECT, related_name="teams")
  name        CharField          # "U12 A"
  age_group   CharField (choices, optional)
  Meta: unique_together (season, name); ordering = ["season", "name"]

TeamMembership(UUIDModel)      # roster entry — through model, club/season implied by team
  team        FK Team (CASCADE, related_name="roster")
  member       FK Member (CASCADE, related_name="team_memberships")
  position     CharField (TextChoices, optional)
  jersey_number PositiveSmallIntegerField (null=True)
  Meta: unique_together (team, member);
      UniqueConstraint(team, jersey_number) WHERE jersey_number IS NOT NULL

StaffAssignment(UUIDModel)     # coach / manager on a team, per season
  team        FK Team (CASCADE, related_name="staff")
  member       FK Member (CASCADE, related_name="staff_assignments")
  role         CharField (TextChoices: coach | assistant | manager)
  Meta: unique_together (team, member, role)

```

A `Member` plays on one *or more* `Team` s per season, each with its own position + jersey number — modeled by `TeamMembership` , exactly matching the domain note.

- **Jersey numbers are unique within a team** (decision §7 #4): a partial `UniqueConstraint(fields=["team", "jersey_number"], condition=~Q(jersey_number=None))` . Nullable so a roster spot can exist before a number is assigned; `NULL` s are exempted so several unnumbered entries don't collide. `team` already implies club + season, so no extra tenancy field is needed on the constraint.
- `StaffAssignment` drives the coach/manager object-scope (§3.1–3.2) — it *is* the "is a coach of this team" fact; no `ClubRole` mirrors it.

5.3 events

```

Event(ClubScopedModel)        # -> carries `club`
  season      FK Season (PROTECT, related_name="events")
  team        FK Team (SET_NULL, null=True, related_name="events") # null = club-wide
  kind        CharField (TextChoices: training | match | tournament | social | meeting)
  title       CharField
  location     CharField (blank)
  starts_at   DateTimeField
  ends_at     DateTimeField (null=True)
  opponent    CharField (blank) # for matches
  Meta: ordering = ["starts_at"]

Attendance(UUIDModel)         # through model Event <-> Member
  event       FK Event (CASCADE, related_name="attendances")
  member      FK Member (CASCADE, related_name="attendances")
  status      CharField (TextChoices: present | absent | excused | maybe)
  note        CharField (blank)
  Meta: unique_together (event, member)

```

`Event.season` is redundant with `team.season` when a team is set, but events can be club-wide (`team=None`), so `season` stays a first-class FK. Keep it consistent in a service/clean().

5.4 news , pages , home (public site / editorial)

```
news.Article(ClubScopedModel)      # -> carries `club`
  title, slug (SlugField), body (TextField)
  excerpt (blank), cover_image (ImageField, null)
  author      FK members.Member (SET_NULL, null, related_name="articles")
  category    FK news.Category (SET_NULL, null)
  is_published BooleanField; published_at DateTimeField (null)
  Meta: unique_together (club, slug); ordering = ["-published_at"]

news.Category(ClubScopedModel): name, slug  # Meta: unique_together (club, slug)

pages.Page(ClubScopedModel)        # flat CMS pages: "About", "Contact", ...
  title, slug, body (TextField)
  is_published BooleanField; menu_order (int)
  Meta: unique_together (club, slug)
  # If nested navigation is needed, add: parent = FK self (SET_NULL, null)

home.HomeConfig(ClubScopedModel)    # one row PER CLUB: featured articles/teams, hero content
                                     # (unique_together (club,) - one per tenant). May be config-only.
```

- `Article.author` **links to** `members.Member` (decision §7 #5) — attribution is to a club person, not a raw login; `SET_NULL` so deleting a member doesn't erase their posts.
- `slug` s back clean public URLs and feed search ; they are **unique per club** (§2.4), so two clubs can both have `/news/season-kickoff` . Resolve within the request's club.
- `cover_image` / hero images use `ImageField` → **media storage must be configured** (§8). If page/news trees grow, consider a tree library later — start flat.

5.5 search

Likely **no models** — a search view over `Article` , `Page` , `Team` , `Event` . If moving to Postgres full-text or an external index, add config here, not domain tables.

5.6 formbuilder — dynamic forms (*new app*)

Admins/editors define forms with a **variable number of fields** at runtime; submissions are stored so they can be **reported on** later. This uses the classic EAV (entity- attribute-value) shape with **normalized** `Answer` **rows as the single source of truth** (decision §7 #9) — one row per answered field, with a `JSON` `value` to stay flexible across field types. No parallel `JSON` blob on the submission — reporting reads `Answer` s directly.

```
Form(ClubScopedModel)              # -> carries `club`
  title, slug, description (blank)
  is_active      BooleanField
  login_required BooleanField      # members-only vs public submission
  opens_at / closes_at DateTimeField (null)  # optional submission window
  max_submissions_per_user PositiveInteger (null)  # null = unlimited
  Meta: unique_together (club, slug); ordering = ["title"]

Field(UUIDModel)                   # a form's field definition (club implied by form)
  form      FK Form (CASCADE, related_name="fields")
  key       SlugField              # stable machine name, unique per form (for reporting)
```

```

label          CharField
field_type CharField (TextChoices: text | textarea | number | email | date |
                                choice | multichoice | checkbox | file)

required       BooleanField
help_text      CharField (blank)
order          PositiveSmallIntegerField
is_active      BooleanField (default=True)  # soft-retire instead of deleting (see below)
options        JSONField (default=list) # choices for choice/multichoice: [{value,label}]
Meta: unique_together (form, key); ordering = ["form", "order"]

Submission(UUIDModel)                # container only – no answer data on it
form          FK Form (CASCADE, related_name="submissions")
member        FK Member (SET_NULL, null)  # set when submitter is logged in
submitted_at  DateTimeField
Meta: ordering = ["-submitted_at"]

Answer(UUIDModel)                    # CANONICAL store – one per answered field
submission FK Submission (CASCADE, related_name="answers")
field        FK Field (PROTECT, related_name="answers")
value        JSONField                # scalar, list (multichoice), or file ref
Meta: unique_together (submission, field)

```

Design notes:

- Answer **is canonical; there is no denormalized JSON snapshot**. A submission's values are always read/aggregated from its Answer rows. The submit service (`formbuilder/services/submit.py`) validates the dynamic form and writes the Submission
 - its Answer s in one transaction. (If a flat per-submission view ever becomes a hotspot, add a *derived, rebuildable* cache later — but the model stays the source.)
- Field.key **is immutable once submissions exist** — reporting joins on it. Deleting a field with answers is blocked (PROTECT); **retire via** is_active=False instead.
- **Reporting** = a service/view producing per-field aggregates (counts per choice, numeric averages, response rate) plus a wide CSV/Excel export (one column per field, one row per submission). No extra model needed; add a saved-report model later only if users need to persist report definitions.
- Rendering a Form to a Django form (and validating a submission) is a service concern — build the form class dynamically from Field rows; don't hand-write form classes.

5.7 shop — cart, orders, payments & PDF invoices (*new app*)

A cart-like shop where a member (or parent) "buys" products — chiefly a **season membership** — with payment-status tracking and generated invoices. Fulfilment of a membership product writes back to the season-scoped ClubMembership (§5.1).

```

Product(ClubScopedModel)                # -> carries `club`
name, slug, description (blank)
kind          CharField (TextChoices: membership | event_fee | merchandise | donation)
price         DecimalField(max_digits=8, decimal_places=2)
season        FK Season (PROTECT, null)  # set for membership/event products
is_active      BooleanField
Meta: unique_together (club, slug)
# membership products fulfil into a ClubMembership for the chosen season + beneficiary

Cart(ClubScopedModel)                   # -> carries `club`; one open cart per (club, user)

```

```

user          FK User (CASCADE, related_name="carts")
status        CharField (TextChoices: open | checked_out | abandoned)
Meta: UniqueConstraint(club, user) WHERE status = open

CartItem(UUIDModel)                                # club implied by cart
cart          FK Cart (CASCADE, related_name="items")
product       FK Product (PROTECT)
beneficiary   FK Member (PROTECT, null) # who this membership is FOR (parent buys for child)
quantity      PositiveSmallIntegerField (default=1)
unit_price    DecimalField                # snapshot of price at add-to-cart time
Meta: unique_together (cart, product, beneficiary)

Order(ClubScopedModel)                             # -> carries `club`; immutable, created at checkout
number        CharField                    # human ref, allocated PER CLUB, e.g. "ORD-2026-00042"
purchaser     FK Member (PROTECT, related_name="orders")
status        CharField (TextChoices: pending | paid | partially_paid | cancelled | refunded)
total         DecimalField
created_at    DateTimeField
Meta: unique_together (club, number); ordering = ["-created_at"]

OrderLine(UUIDModel)                              # club implied by order
order         FK Order (CASCADE, related_name="lines")
product       FK Product (PROTECT)
beneficiary   FK Member (PROTECT, null)
quantity      PositiveSmallIntegerField
unit_price    DecimalField                # snapshot
line_total    DecimalField
fulfilled_at  DateTimeField (null)        # when this line's ClubMembership was activated

Payment(UUIDModel)                                # club implied by order; an order may have several (partial)
order         FK Order (CASCADE, related_name="payments")
amount        DecimalField
method        CharField (TextChoices: bank_transfer | cash | card | online)
status        CharField (TextChoices: pending | confirmed | failed | refunded)
reference     CharField (blank)           # bank/gateway reference
paid_at       DateTimeField (null)

Invoice(ClubScopedModel)                         # -> carries `club`
number        CharField                    # sequential PER CLUB per year, e.g. "INV-2026-00042"
order         OneToOneField Order (PROTECT, related_name="invoice")
issued_at     DateTimeField
due_date      DateField (null)
billing_snapshot JSONField                # name/address frozen at issue time
pdf           FileField (null)            # rendered HTML->PDF, cached in PRIVATE storage (§8)
Meta: unique_together (club, number)

```

Flow & design notes:

- **Cart → checkout → order.** Checkout converts the open `Cart` into an immutable `Order`
 - `OrderLine` s, snapshotting prices (products may reprice later). The cart is marked `checked_out` . All in one transactional service (`shop/services/checkout.py`).
- **Payment status is derived, not typed by hand.** A service sums `confirmed` `Payment` s and sets `Order.status` (`pending` → `partially_paid` → `paid`). Online payments arrive via a gateway webhook that creates/confirmes a `Payment` ; manual methods (bank transfer/cash) are confirmed by a `TREASURER` (§3.2).
- **Fulfilment writes back to membership.** When an order (or a membership line) reaches `paid` , a service creates/activates the `ClubMembership`(`member=beneficiary`, `season=...`) , flips its `fee_status` → `paid` /

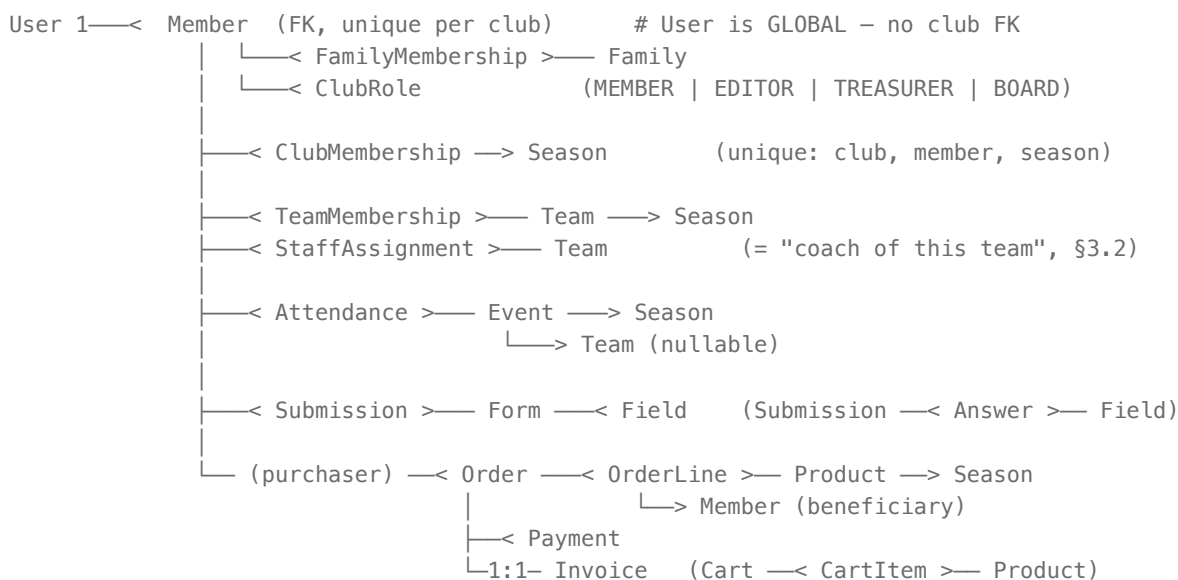
status → active , stamps OrderLine.fulfilled_at , and triggers role reconciliation (§3.4). This is the seam that ties the shop to the domain.

- **Beneficiary vs. purchaser** is first-class: a parent (purchaser) buys memberships for several children (beneficiary) in one order. Both are Member s of the same club.
- **Invoice = HTML → PDF**. Render a Django template to HTML, convert with **WeasyPrint** (see §8 for the exact dependency + native-library setup). Generate on order confirmation, store the file on Invoice.pdf in **private** storage, and serve it only through a permission-checked view (never a public media URL — invoices are tenant-private, §8). Invoice.number uses a gap-free counter **per club per year** — allocate it in a transaction/service (e.g. a select_for_update sequence row), not from count() .
- **Money =** DecimalField , never float. Snapshot prices onto cart items / order lines / invoices so historical records stay correct when Product.price changes.

6. Entity-relationship overview

Club (TENANT ROOT — every model below except User carries `club`; §2.4)

└─ Season, Member, Family, Team, Event, ClubRole, Article, Page, Form, Product, Order, Invoice, ...



Season —< Team, Event, ClubMembership, (membership/event) Product # all within one club

news.Article —> news.Category, (author) members.Member

pages.Page (self-parent, optional)

Legend: —< one-to-many, >—< many-to-many via a through model. Everything under Club is one tenant's data; joins never cross clubs (§2.4).

7. Decisions (resolved) & remaining questions

Resolved (this revision):

1. ☒ **members app split** — Member/Family/FamilyMembership **move to a dedicated** members **app** (§1, §4). Migration reshuffles app labels + tables.
2. ☒ **Season-scoped** ClubMembership — yes (§5.1). Data migration backfills existing rows with the current season and re-scopes `unique_together` .
3. ☒ **Full multi-tenancy** — adopt **row-based multi-tenancy**, `Club` as tenant root (§2.4). Requires: `ClubScopedModel` on every aggregate root, `Member.user → ForeignKey (+ unique(club, user) )`, tenant middleware + `clubmanager/tenancy.py` context, tenant-aware manager, per-club uniqueness, per-club roles (§3). **Supersedes** `CLAUDE.md` .
4. ☒ **Jersey uniqueness** — unique **within a team** via a partial `UniqueConstraint ( NULL s exempt)` (§5.2).
5. ☒ `Article.author` — links to `members.Member` (§5.4).
6. ☒ **RBAC mechanism** — **service layer**, no `django-guardian` ; per-club `ClubRole` rows + a single access service (§3). Django's own perms only for platform admin.
7. ☒ `formbuilder storage` — **normalized** `Answer` **is canonical**; no denormalized JSON snapshot (§5.6).

Infrastructure/config for the above (media storage, dependencies + exact setup) is specified in **§8**.

Still open:

- **Tenant resolution mechanism** — subdomain (recommended) vs. path-prefix `/c/<slug>/` . Affects DNS/TLS, `ALLOWED_HOSTS` , cookies, and local dev (§8). Pick before building `ClubTenantMiddleware` .
- **Auto-scoping vs. explicit scoping** — should the tenant manager filter *automatically* from context, or stay explicit (`.for_club()` / `.current()`)? Doc currently recommends **explicit** (§2.4).
- **Cross-club users** — can one person be a `BOARD` member of several clubs, switching context in one session? The model allows it; confirm the UX (club switcher) is in scope.
- **Payment gateway** — which provider (Mollie / Stripe / none-yet)? Only needed when online payments go live (§8).

8. Infrastructure & configuration notes

Config required by the models above — settings + dependencies to add as each app lands.

8.1 Media & file storage (needed for `news` , `home` , `shop` , **form file fields**)

Two classes of files with **different exposure**:

- **Public media** — `Article.cover_image` , home hero images. Served from the normal media URL is fine.
- **Private, tenant-scoped files** — `Invoice.pdf` and `formbuilder` file uploads. These **must not** be publicly reachable. Serve them only through a permission-checked Django view (`X-Accel-Redirect` / `X-Sendfile` in prod), never a guessable public URL, and scope access to the file's club (§2.4).

Setup:

- **Dev:** `MEDIA_ROOT` / `MEDIA_URL` on local disk; private files under a non-served path.
- **Prod:** object storage (S3-compatible) via `django-storages` (add to deps) with **separate public and private buckets/backends** (Django 5.1+ `STORAGES` setting). Keep the private backend non-public and generate signed/short-lived URLs or stream via the view.
- Organise keys by club (e.g. `club/<club_id>/invoices/...`) so tenant data is easy to isolate, audit, and delete.

8.2 HTML → PDF invoices — WeasyPrint

- Add the dependency: `uv add weasyprint` .
- **Native libraries required** (WeasyPrint wraps Pango/Cairo) — install at the OS/image level, not via pip:
 - macOS (dev): `brew install pango gdk-pixbuf libffi` (Cairo/GLib come along).
 - Debian/Ubuntu (CI + prod image): `apt-get install libpango-1.0-0 libpangocairo-1.0-0 libcairo2 libgdk-pixbuf-2.0-0 libffi-dev` (exact names per distro/WeasyPrint version).
 - Document these in the Dockerfile/CI so PDF rendering isn't a "works on my machine" trap.
- Render a Django template → HTML string → `weasyprint.HTML(string=...).write_pdf()` ; store onto `Invoice.pdf` (private storage, §8.1). Generation is a service, ideally async/queued if volume grows.

8.3 Tenancy runtime config (needed once `ClubTenantMiddleware` lands)

- **Hosts:** wildcard `ALLOWED_HOSTS` for the chosen base domain (e.g. `.clubmanager.app`) if using subdomain resolution; add `DJANGO_ALLOWED_HOSTS` accordingly.
- **CSRF:** `CSRF_TRUSTED_ORIGINS` must cover the wildcard scheme+host set (`https://*.clubmanager.app`).
- **Cookies:** to share login across club subdomains, set `SESSION_COOKIE_DOMAIN` / `CSRF_COOKIE_DOMAIN` to the base domain; otherwise keep per-subdomain sessions (decide with the "cross-club users" question in §7).
- **Local dev:** map a wildcard to localhost (e.g. `*.localhost` resolves on most systems, or use `dnsmasq`) so subdomain resolution works without editing `/etc/hosts` per club.
- **Middleware order:** place `ClubTenantMiddleware` after `AuthenticationMiddleware` (needs `request.user` to fall back to a user's default club when no subdomain is present).

8.4 Optional dependencies

- **Payment gateway** (only if online payments): provider SDK (e.g. `mollie-api-python` or `stripe`) + webhook endpoint that creates/confirm `Payment` s (§5.7).
- **Excel export** for form reporting beyond CSV: `openpyxl` .

Conventions cross-reference: `clubmanager/base.py` (`UUIDModel` , `ClubScopedModel`), `clubmanager/tenancy.py` (to add — tenant context/middleware, §2.4), `authentication/managers.py` (`UserManager`), `authentication/services/` (service-layer pattern).

Banner: supersedes `CLAUDE.md`

This architecture adopts **full multi-tenancy** (§2.4), which **directly contradicts** the current `CLAUDE.md` ("ClubManager is a **single-club** app ... deliberately *not* multi-tenant — there is no `club_id` tenancy") and the project memory (`project_overview` — "Single-club (NOT multi-tenant)").

Action required before/alongside implementation: update `CLAUDE.md` and the memory to describe ClubManager as a **multi-tenant platform (row-based, `Club` = tenant root)**. Until that is done, where the two disagree **this document is authoritative**.