Goal

Deliver a production-ready relational database for PsychSync with: (1) a complete ERD, (2) schemas & migrations wired to FastAPI, (3) isolated dev/staging/prod environments, and (4) a repeatable workflow. Target DB: **PostgreSQL 16**.

---

0) Quick Decisions (stick to these unless there's a strong reason to change)

- **Engine**: PostgreSQL 16
- **ORM**: SQLAlchemy 2.x (async) + Alembic
- **Driver**: `asyncpg`
- **IDs**: UUID v4 (database-generated default `gen_random_uuid()`)
- **Timestamps**: `timestamptz` with `created_at` / `updated_at` triggers
- **Soft delete**: `deleted_at` nullable
- **Tenancy**: Org-scoped multi-tenant via foreign keys (no shared row data across orgs)
- **Schema**: single `public` schema (keep it simple for v1)
- **Naming**: snake_case for tables/columns; singular table names for entities with junction tables pluralized (e.g., `team_members`)
- **Constraints**: NOT NULL + CHECKs + UNIQUE where relevant; `ON DELETE CASCADE` only on pure junctions
- **Indexes**: compound indexes for hot queries noted below

---

1) Core Entities (MVP)

Identity & Access - `users` – platform accounts - `organizations` – customer accounts - `org_members` – user↔org membership (role at org level) - `teams` – teams belong to an organization - `team_members` – user↔team membership (role at team level) - `roles` (seeded: `owner`, `admin`, `member`, `viewer`) - `permissions` (coarse-grained, map to roles)

Assessment Framework - `frameworks` – e.g., MBTI, Big Five, DISC - `questions` – belongs to a framework - `question_options` – options for multiple choice - `assessments` – an assigned/taken assessment instance (user, framework, team, status) - `responses` – per question answer for an assessment - `scores` – normalized result per dimension (e.g., Big Five OCEAN)

Analytics & Optimization - `compatibility_rules` – rules used by the optimization engine - `team_metrics` – aggregated metrics per team & period

Operations - `invitations` – email invites to join org/team - `audit_logs` – security & data change logs - `notifications` – queued in-app/email events - `files` – uploaded artifacts (optional for v1) - `subscriptions` – (optional: Stripe integration in later phase)

Nice-to-have later: `segments`, `experiments`, `webhooks_outbox` (for reliable integrations), `idempotency_keys` for API writes.

---

2) ERD (Mermaid)

Copy/paste into docs/wiki to render. (Reflects MVP above.)

```
erDiagram
    organizations ||--o{ users : "owner (via org_members)"
    organizations ||--o{ org_members : has
    users ||--o{ org_members : has

    organizations ||--o{ teams : has
    teams ||--o{ team_members : has
    users ||--o{ team_members : has

    roles ||--o{ org_members : grants
    roles ||--o{ team_members : grants

    frameworks ||--o{ questions : has
    questions ||--o{ question_options : has

    frameworks ||--o{ assessments : assigned_as
    users ||--o{ assessments : takes
    teams ||--o{ assessments : context

    assessments ||--o{ responses : has
    assessments ||--o{ scores : produces

    organizations ||--o{ invitations : issues
    organizations ||--o{ audit_logs : records
    users ||--o{ audit_logs : actor
    teams ||--o{ team_metrics : aggregates
```

---

3) SQL DDL (PostgreSQL) – Seed Migrations

Create **initial migration** with these representative tables. You can expand with more columns as needed.

```
-- Enable extensions
CREATE EXTENSION IF NOT EXISTS pgcrypto;          -- for gen_random_uuid
CREATE EXTENSION IF NOT EXISTS citext;            -- case-insensitive email/
username
```

```sql
-- Timestamp trigger function
CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
  NEW.updated_at = NOW();
  RETURN NEW;
END;$$ LANGUAGE plpgsql;

-- organizations
CREATE TABLE organizations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  slug TEXT UNIQUE NOT NULL,
  created_at timestamptz NOT NULL DEFAULT NOW(),
  updated_at timestamptz NOT NULL DEFAULT NOW(),
  deleted_at timestamptz
);
CREATE TRIGGER organizations_set_updated
BEFORE UPDATE ON organizations FOR EACH ROW EXECUTE PROCEDURE set_updated_at();

-- users
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email CITEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  full_name TEXT,
  avatar_url TEXT,
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  created_at timestamptz NOT NULL DEFAULT NOW(),
  updated_at timestamptz NOT NULL DEFAULT NOW(),
  deleted_at timestamptz
);
CREATE INDEX users_email_idx ON users (email);
CREATE TRIGGER users_set_updated
BEFORE UPDATE ON users FOR EACH ROW EXECUTE PROCEDURE set_updated_at();

-- roles (seed data in a later migration)
CREATE TABLE roles (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  scope TEXT NOT NULL CHECK (scope IN ('org','team')),
  code TEXT NOT NULL, -- e.g., 'owner','admin','member','viewer'
  UNIQUE(scope, code)
);

-- org_members
CREATE TABLE org_members (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  role_id UUID NOT NULL REFERENCES roles(id),
  created_at timestamptz NOT NULL DEFAULT NOW(),
  UNIQUE (org_id, user_id)
);
CREATE INDEX org_members_org_user_idx ON org_members (org_id, user_id);

-- teams
CREATE TABLE teams (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  description TEXT,
  created_at timestamptz NOT NULL DEFAULT NOW(),
  updated_at timestamptz NOT NULL DEFAULT NOW(),
  deleted_at timestamptz,
  UNIQUE (org_id, name)
);
CREATE TRIGGER teams_set_updated
BEFORE UPDATE ON teams FOR EACH ROW EXECUTE PROCEDURE set_updated_at();
CREATE INDEX teams_org_idx ON teams (org_id);

-- team_members
CREATE TABLE team_members (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  team_id UUID NOT NULL REFERENCES teams(id) ON DELETE CASCADE,
  user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  role_id UUID NOT NULL REFERENCES roles(id),
  created_at timestamptz NOT NULL DEFAULT NOW(),
  UNIQUE (team_id, user_id)
);
CREATE INDEX team_members_team_user_idx ON team_members (team_id, user_id);

-- frameworks
CREATE TABLE frameworks (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  code TEXT NOT NULL UNIQUE, -- 'MBTI','BIG5','DISC'
  name TEXT NOT NULL,
  created_at timestamptz NOT NULL DEFAULT NOW(),
  updated_at timestamptz NOT NULL DEFAULT NOW()
);
CREATE TRIGGER frameworks_set_updated
BEFORE UPDATE ON frameworks FOR EACH ROW EXECUTE PROCEDURE set_updated_at();

-- questions
CREATE TABLE questions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
  framework_id UUID NOT NULL REFERENCES frameworks(id) ON DELETE CASCADE,
  code TEXT,         -- optional external code
  body TEXT NOT NULL,
  kind TEXT NOT NULL CHECK (kind IN ('single','multi','likert','text')),
  position INT NOT NULL,
  created_at timestamptz NOT NULL DEFAULT NOW()
);
CREATE INDEX questions_framework_pos_idx ON questions (framework_id, position);

-- question_options
CREATE TABLE question_options (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  question_id UUID NOT NULL REFERENCES questions(id) ON DELETE CASCADE,
  label TEXT NOT NULL,
  value TEXT NOT NULL,  -- store raw value
  weight NUMERIC,       -- optional scoring weight
  position INT NOT NULL
);
CREATE INDEX question_options_q_pos_idx ON question_options (question_id,
position);

-- assessments (an instance of a user taking a framework)
CREATE TABLE assessments (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  team_id UUID REFERENCES teams(id) ON DELETE SET NULL,
  user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  framework_id UUID NOT NULL REFERENCES frameworks(id) ON DELETE RESTRICT,
  status TEXT NOT NULL CHECK (status IN
('assigned','in_progress','completed','expired')),
  started_at timestamptz,
  completed_at timestamptz,
  created_at timestamptz NOT NULL DEFAULT NOW()
);
CREATE INDEX assessments_lookup_idx ON assessments (org_id, team_id, user_id,
framework_id);

-- responses
CREATE TABLE responses (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  assessment_id UUID NOT NULL REFERENCES assessments(id) ON DELETE CASCADE,
  question_id UUID NOT NULL REFERENCES questions(id) ON DELETE CASCADE,
  answer_json JSONB NOT NULL, -- supports multi/likert/text
  created_at timestamptz NOT NULL DEFAULT NOW(),
  UNIQUE (assessment_id, question_id)
);
CREATE INDEX responses_assessment_idx ON responses (assessment_id);
```

```sql
-- scores (normalized dimension scores)
CREATE TABLE scores (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  assessment_id UUID NOT NULL REFERENCES assessments(id) ON DELETE CASCADE,
  dimension TEXT NOT NULL,     -- e.g., 'O','C','E','A','N' or 'I/E'
  value NUMERIC NOT NULL,      -- 0..100 or normalized 0..1
  created_at timestamptz NOT NULL DEFAULT NOW(),
  UNIQUE (assessment_id, dimension)
);
CREATE INDEX scores_assessment_dim_idx ON scores (assessment_id, dimension);

-- invitations
CREATE TABLE invitations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  email CITEXT NOT NULL,
  token TEXT NOT NULL UNIQUE,
  expires_at timestamptz NOT NULL,
  accepted_at timestamptz
);
CREATE INDEX invitations_org_email_idx ON invitations (org_id, email);

-- audit_logs
CREATE TABLE audit_logs (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  actor_user_id UUID REFERENCES users(id) ON DELETE SET NULL,
  action TEXT NOT NULL,        -- e.g., 'user.invited','assessment.completed'
  entity TEXT NOT NULL,        -- table name
  entity_id UUID,
  meta JSONB,
  created_at timestamptz NOT NULL DEFAULT NOW()
);
CREATE INDEX audit_logs_org_time_idx ON audit_logs (org_id, created_at DESC);
```

Seed roles (separate migration):

```sql
INSERT INTO roles (scope, code) VALUES
 ('org','owner'),('org','admin'),('org','member'),('org','viewer'),
 ('team','admin'),('team','member'),('team','viewer')
ON CONFLICT DO NOTHING;
```

---

4) Wire Up SQLAlchemy (Async) + Alembic (FastAPI)

Install deps

```
pip install "sqlalchemy[asyncio]" asyncpg alembic psycopg2-binary python-dotenv
```

Project structure (backend)

```
backend/
  app/
    db/
      __init__.py
      models.py
      session.py
    api/
    core/
    ...
  alembic/
  alembic.ini
```

`app/db/session.py`

```
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker
from sqlalchemy.orm import DeclarativeBase
import os

DATABASE_URL = os.getenv("DATABASE_URL")  # e.g., postgresql+asyncpg://
user:pass@localhost:5432/psychsync

class Base(DeclarativeBase):
    pass

engine = create_async_engine(DATABASE_URL, pool_pre_ping=True)
AsyncSessionLocal = async_sessionmaker(engine, expire_on_commit=False)
```

`app/db/models.py` (sample subset)

```
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import ForeignKey, text
from .session import Base
```

```
class Organization(Base):
    __tablename__ = "organizations"
    id: Mapped[str] = mapped_column(primary_key=True,
server_default=text("gen_random_uuid()"))
    name: Mapped[str]
    slug: Mapped[str]

class User(Base):
    __tablename__ = "users"
    id: Mapped[str] = mapped_column(primary_key=True,
server_default=text("gen_random_uuid()"))
    email: Mapped[str]
    password_hash: Mapped[str]
```

Initialize Alembic

```
alembic init -t async alembic
```

Edit `alembic.ini` - Set `script_location = alembic`

Edit `alembic/env.py` – load URL from env & use async engine

```
from logging.config import fileConfig
from sqlalchemy import pool
from sqlalchemy.engine import Connection
from alembic import context
import os

# Import models for autogenerate
from app.db.session import Base
from app.db import models  # noqa

config = context.config
fileConfig(config.config_file_name)

target_metadata = Base.metadata

def get_url():
    return os.getenv("DATABASE_URL")

def run_migrations_offline():
    url = get_url()
    context.configure(url=url, target_metadata=target_metadata,
literal_binds=True, dialect_opts={"paramstyle": "named"})
```

```
    with context.begin_transaction():
        context.run_migrations()

def run_migrations_online():
    from sqlalchemy.ext.asyncio import async_engine_from_config
    connectable = async_engine_from_config(
        {"sqlalchemy.url": get_url()}, prefix="sqlalchemy.",
poolclass=pool.NullPool
    )
    async def do_run_migrations():
        async with connectable.connect() as connection:
            await connection.run_sync(lambda conn:
context.configure(connection=conn, target_metadata=target_metadata))
            await connection.run_sync(lambda conn: context.begin_transaction())
            await connection.run_sync(lambda conn: context.run_migrations())
    import asyncio
    asyncio.run(do_run_migrations())

if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

Create migrations

```
# 1) Revision for extensions & triggers (manual SQL)
alembic revision -m "extensions & triggers"
# paste SQL into the revision's upgrade() using op.execute(...)

# 2) Autogenerate tables from models
alembic revision --autogenerate -m "core tables"

# 3) Seed roles
alembic revision -m "seed roles"
# add op.execute(INSERT ...) in upgrade()

# Apply
alembic upgrade head
```

---

5) Environments

Env Vars (in each env)

```
DATABASE_URL=postgresql+asyncpg://psychsync:psychsync@localhost:5432/psychsync
DATABASE_URL_TEST=postgresql+asyncpg://psychsync:psychsync@localhost:5432/
psychsync_test
```

Docker Compose (dev)

```yaml
version: '3.8'
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_USER: psychsync
      POSTGRES_PASSWORD: psychsync
      POSTGRES_DB: psychsync
    ports: ["5432:5432"]
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U $$POSTGRES_USER"]
      interval: 5s
      timeout: 5s
      retries: 10
volumes:
  pgdata: {}
```

Staging/Prod - Use managed Postgres (AWS RDS/Azure Flexible Server) - Private networking + TLS required - Automated daily snapshots; PITR enabled - Separate DB per env (no shared data), distinct credentials

Migrations in CI/CD - Run `alembic upgrade head` on deploy (staging, then prod) - Block deploy if migrations fail - Keep a `down_revision` chain linear; squash old migrations periodically

---

6) Data & Access Patterns (index plan)

- `assessments_lookup_idx (org_id, team_id, user_id, framework_id)` – dashboard filters
- `responses(assessment_id)` – loading answer sets
- `scores(assessment_id, dimension)` – reading per-dimension quickly
- `teams(org_id, name)` – unique per org; search by name within org
- `audit_logs(org_id, created_at desc)` – recent activity feed

---

7) Security & Compliance Basics (v1)

- Store only necessary PII; use `citext` for emails
- Hash passwords with **Argon2id** (app layer) & long random salts
- Restrict by org: **every query** that reads/writes domain data must filter by `org_id`
- Do not expose internal IDs in URLs without authorization checks
- Enable **pg_stat_statements** in staging/prod for performance analysis
- Periodic anonymized snapshots for staging (no raw PII)

---

8) Test Data & Seeding

Create a Python seed script to populate a dev org, users, teams, and a minimal Big Five framework with 5–10 questions to unblock UI work.

```
# scripts/seed.py (run with: uvicorn not needed; use Python)
import asyncio
from app.db.session import AsyncSessionLocal, engine
from app.db.models import Organization, User

async def main():
    async with AsyncSessionLocal() as s:
        acme = Organization(name="Acme Corp", slug="acme")
        s.add(acme)
        await s.flush()
        u = User(email="founder@acme.com", password_hash="dev_only",
                 full_name="Dev Founder")
        s.add(u)
        await s.commit()

asyncio.run(main())
```

---

9) Definition of Done (for this task)

- ✅ ERD checked into repo (`/docs/erd.mmd` + exported PNG)
- ✅ Alembic migrations: extensions/triggers, core tables, seed roles
- ✅ Local dev DB up via Docker; `alembic upgrade head` succeeds
- ✅ FastAPI connects with async SQLAlchemy; healthcheck endpoint passes
- ✅ Basic seed script creates org, user, team, framework, questions
- ✅ Staging DB provisioned with credentials stored in secrets manager
- ✅ CI job runs migrations automatically on staging

---

10) 16-Hour Work Plan (Backend Lead)

1. **Design** (3h): finalize entities & ERD, review with Tech Lead
2. **Scaffold** (2h): SQLAlchemy base, session, Alembic init
3. **Migrations** (4h): extensions/triggers + core tables + seeds
4. **Env Setup** (2h): Docker Compose, .envs, Makefile targets
5. **Wiring** (2h): FastAPI startup checks, connection lifecycle
6. **Seeding & Smoke** (2h): seed script + CRUD sanity
7. **Docs & Handover** (1h): README, ERD image, usage notes

---

11) Makefile (quality of life)

```
.PHONY: db-up db-down migrate upgrade seed

DB_URL=postgresql+asyncpg://psychsync:psychsync@localhost:5432/psychsync

db-up:
    docker compose up -d db
    sleep 3

migrate:
    alembic revision --autogenerate -m "$(m)"

upgrade:
    DATABASE_URL=$(DB_URL) alembic upgrade head

seed:
    python scripts/seed.py
```

---

12) Next Actions (execute in order)

1) `docker compose up -d db` 2) Set `DATABASE_URL` in `.env` and shell 3) `alembic upgrade head` 4) `make seed` (or `python scripts/seed.py`) 5) Run FastAPI; hit `/health` to verify DB connectivity 6) Export ERD PNG, add to repo, share in sprint review