high-level architecture

```
[Data Providers/APIs]
  ├─ Sports stats & schedules (NBA/NFL/MLB/UFC)
  ├─ Injury feeds
  ├─ Odds feeds
  └─ News (team/player mentions)
        ↓ (pull via jobs/webhooks)
[Ingestion Workers]
  ├─ Pollers/Schedulers (APS/Celery+Beat)
  └─ Normalizers & Dedupers
        ↓
[FastAPI Backend]
  ├─ REST + WebSocket (live updates)
  ├─ Auth (JWT)
  ├─ Alerts engine (rules for "my teams")
  ├─ Betting odds aggregator
  └─ Admin tools (feature flags, backfill)
        ↓
[PostgreSQL]  [Redis]
  ├─ Core schema    └─ Caching, jobs, rate limits
        ↓
[Clients]
  ├─ React Native (iOS/Android) or Flutter
  ├─ Web (Next.js or simple React)
  └─ Discord bot/webhook feed (notifications)
```

recommended stack (python-first)

- **Backend**: FastAPI + Uvicorn
- **DB**: PostgreSQL (SQLAlchemy + Alembic)
- **Cache/queues**: Redis
- **Scheduling**: APScheduler (simpler) or Celery + Celery Beat (heavy-duty)

- **Realtime**: FastAPI WebSockets (via Socket.IO if you prefer)
- **Auth**: JWT (PyJWT) with OAuth2 password flow
- **Mobile**: React Native (Expo) or Flutter; for “fastest path,” start with a PWA and add RN later
- **Discord**: `discord.py` (bot) or plain webhooks (simpler for push-only)
- **Telemetry**: Prometheus + Grafana (optional but great)
- **Containerization**: Docker + docker-compose

data sources (choose your mix)

you can swap vendors later—normalize everything into your own schema.

- **Multi-sport stats/schedules**: Sportradar, Sportsdata.io, TheSportsDB (community), API-NBA/NFL/MLB community APIs; for UFC: Tapology/typical scrapers are brittle—prefer a paid vendor that covers MMA or use community MMA APIs and expect gaps.
- **Injuries**: Usually bundled with premium sport APIs; some teams/leagues publish reports; community aggregators exist for NBA/NFL.
- **Odds**: The Odds API, OddsJam, Betfair Exchange API, or sportsbook-specific feeds where permitted.
- **News**: NewsAPI, GNews, Bing News Search, or feeds from major outlets; filter by team/player keywords.

legal note: respect each provider’s ToS; scraping often violates ToS and breaks. for betting odds, include disclaimers and geofencing where required.

core database schema (postgres)

```
-- teams & players
CREATE TABLE teams (
  id SERIAL PRIMARY KEY,
  sport TEXT NOT NULL,  -- 'NFL','NBA','MLB','UFC'
  ext_ref TEXT UNIQUE,  -- provider's ID
  name TEXT NOT NULL,
  short_name TEXT,
  market TEXT,          -- city/college/weightclass org for UFC
  created_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE players (
  id SERIAL PRIMARY KEY,
  sport TEXT NOT NULL,
  ext_ref TEXT UNIQUE,
  team_id INT REFERENCES teams(id),
  first_name TEXT,
  last_name TEXT,
  position TEXT,
  status TEXT,          -- active, injured, etc.
  created_at TIMESTAMPTZ DEFAULT now()
);

-- events/games
CREATE TABLE events (
  id SERIAL PRIMARY KEY,
  sport TEXT NOT NULL,
  ext_ref TEXT UNIQUE,
  season TEXT,
  start_time TIMESTAMPTZ,
  venue TEXT,
  home_team_id INT REFERENCES teams(id),
  away_team_id INT REFERENCES teams(id),
  status TEXT,          -- scheduled/live/final
  created_at TIMESTAMPTZ DEFAULT now()
);

-- stats (team & player snapshots)
CREATE TABLE team_stats (
  id BIGSERIAL PRIMARY KEY,
  team_id INT REFERENCES teams(id),
  event_id INT REFERENCES events(id),
  payload JSONB,        -- normalized stats
  collected_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE player_stats (
  id BIGSERIAL PRIMARY KEY,
  player_id INT REFERENCES players(id),
  event_id INT REFERENCES events(id),
  payload JSONB,
  collected_at TIMESTAMPTZ DEFAULT now()
);

-- injuries
CREATE TABLE injuries (
  id BIGSERIAL PRIMARY KEY,
  sport TEXT NOT NULL,
  team_id INT REFERENCES teams(id),
  player_id INT REFERENCES players(id),
  status TEXT,          -- questionable/out/etc.
  note TEXT,
  source TEXT,
  reported_at TIMESTAMPTZ
);

-- odds (multi-book)
CREATE TABLE odds (
  id BIGSERIAL PRIMARY KEY,
  event_id INT REFERENCES events(id),
  bookmaker TEXT NOT NULL,
  market TEXT NOT NULL, -- moneyline/spread/total/props
  selection TEXT,       -- home/away/over/under/fighter
  price NUMERIC,        -- American, or store decimal too
  line NUMERIC,         -- spread points / total
  fetched_at TIMESTAMPTZ DEFAULT now()
);

-- news hits and user alerting
CREATE TABLE news_mentions (
  id BIGSERIAL PRIMARY KEY,
  team_id INT REFERENCES teams(id),
  player_id INT REFERENCES players(id),
  title TEXT,
  url TEXT,
  source TEXT,
  published_at TIMESTAMPTZ
);

CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT UNIQUE NOT NULL,
  password_hash TEXT NOT NULL
);

CREATE TABLE user_subscriptions (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id),
  team_id INT REFERENCES teams(id),
  player_id INT REFERENCES players(id),
  channels TEXT[] NOT NULL DEFAULT ARRAY['mobile'], -- 'mobile','discord','email'
  created_at TIMESTAMPTZ DEFAULT now()
);
```

alerting logic (my-team news & injuries)

- cron every **2–5 minutes**: fetch new articles; fuzzy-match to team/player aliases; dedupe by URL + title hash; insert into `news_mentions`.
- trigger: find users subscribed to the mentioned team or player → push to selected channel(s).
- similar job for **injury updates** and **line moves** (odds change threshold, e.g., ≥ 10 cents or spread shift ≥ 0.5).

project structure

```
sportsapp/
  backend/
    app/
      main.py
      api/            # routers
        teams.py, players.py, events.py, stats.py, odds.py, alerts.py, auth.py
      services/       # business logic
        providers/    # adapters to APIs
          base.py, nba.py, nfl.py, mlb.py, ufc.py, odds.py, news.py
        alerts.py, injuries.py, news_matcher.py
      db/
        models.py, session.py
      jobs/
        scheduler.py, ingest_stats.py, ingest_odds.py, ingest_news.py, ingest_injuries.py
      websocket/
        manager.py
      auth/
        security.py
    tests/
    alembic/
  mobile/ (React Native app)
  discord_bot/
    bot.py
  docker-compose.yml
  .env.example
  README.md
```

sample `.env.example`

```
DATABASE_URL=postgresql+psycopg2://sports:pass@db:5432/sports
REDIS_URL=redis://redis:6379/0
JWT_SECRET=change_me
NEWS_API_KEY=...
ODDS_API_KEY=...
SPORTS_API_KEY=...
DISCORD_BOT_TOKEN=...
```

`FastAPI: minimal endpoints + live socket`

```
# backend/app/main.py
from fastapi import FastAPI, Depends, WebSocket
from fastapi.middleware.cors import CORSMiddleware
from app.api import teams, players, events, odds, stats, alerts
from app.websocket.manager import ws_manager

app = FastAPI(title="Sports Stats API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], allow_credentials=True,
    allow_methods=["*"], allow_headers=["*"]
)

app.include_router(teams.router, prefix="/teams", tags=["teams"])
app.include_router(players.router, prefix="/players",
tags=["players"])
app.include_router(events.router, prefix="/events", tags=["events"])
app.include_router(stats.router, prefix="/stats", tags=["stats"])
app.include_router(odds.router, prefix="/odds", tags=["odds"])
app.include_router(alerts.router, prefix="/alerts", tags=["alerts"])

@app.websocket("/ws/live")
async def live_ws(ws: WebSocket):
    await ws_manager.connect(ws)
    try:
        while True:
            # optional: receive from client (e.g., subscribe to
team_id)
            await ws.receive_text()
    except Exception:
        pass
    finally:
        ws_manager.disconnect(ws)


# backend/app/websocket/manager.py
from typing import Set
from fastapi import WebSocket

class WSManager:
    def __init__(self):
        self.active: Set[WebSocket] = set()

    async def connect(self, ws: WebSocket):
        await ws.accept()
        self.active.add(ws)

    def disconnect(self, ws: WebSocket):
        self.active.discard(ws)

    async def broadcast(self, payload: dict):
        dead = []
        for ws in self.active:
            try:
                await ws.send_json(payload)
            except Exception:
                dead.append(ws)
        for ws in dead:
            self.disconnect(ws)

ws_manager = WSManager()
```

ingestion jobs (scheduler + odds/news pollers)

```
# backend/app/jobs/scheduler.py
from apscheduler.schedulers.asyncio import AsyncIOScheduler
from app.jobs.ingest_news import fetch_and_store_news
from app.jobs.ingest_odds import fetch_and_store_odds
from app.jobs.ingest_stats import fetch_and_store_stats
from app.jobs.ingest_injuries import fetch_and_store_injuries

scheduler = AsyncIOScheduler()

def start():
    scheduler.add_job(fetch_and_store_news, "interval", minutes=3,
id="news")
    scheduler.add_job(fetch_and_store_odds, "interval", minutes=2,
id="odds")
    scheduler.add_job(fetch_and_store_stats, "interval", minutes=1,
id="stats")
    scheduler.add_job(fetch_and_store_injuries, "interval", minutes=5,
id="injuries")
    scheduler.start()


# backend/app/jobs/ingest_news.py
```

```
import httpx, hashlib, datetime as dt
from app.db.session import SessionLocal
from app.db import models

TEAM_KEYWORDS = {
    # 'LAL': ['Lakers','Los Angeles Lakers','LAL'], ...
}

async def fetch_and_store_news():
    # Example with NewsAPI-like interface:
    q = " OR ".join({kw for kws in TEAM_KEYWORDS.values() for kw in
kws})
    url = f"https://newsapi.example.com/search?q={q}&pageSize=50"
    async with httpx.AsyncClient(timeout=10) as client:
        r = await client.get(url, headers={"X-Api-Key": "..."})
        r.raise_for_status()
        data = r.json()["articles"]

    db = SessionLocal()
    try:
        for a in data:
            key = hashlib.sha256((a["title"] +
a["url"]).encode()).hexdigest()
            # resolve to team_id by matching keywords
            team_id = resolve_team(a["title"])
            if team_id:
                db.add(models.NewsMention(
                    team_id=team_id,
                    title=a["title"],
                    url=a["url"],
                    source=a.get("source",{}).get("name"),
                    published_at=a.get("publishedAt",
dt.datetime.utcnow())
                ))
        db.commit()
        # fire alerts async (see alerts service)
    finally:
        db.close()
```

betting odds aggregator (example route)

```
# backend/app/api/odds.py
from fastapi import APIRouter, Query
from sqlalchemy import select
from app.db.session import SessionLocal
from app.db.models import Odds, Events

router = APIRouter()

@router.get("/")
def list_odds(event_id: int | None = None, sport: str | None = None):
    db = SessionLocal()
    try:
        q = select(Odds)
        if event_id:
            q = q.where(Odds.event_id == event_id)
        if sport:
            q = q.join(Events, Events.id ==
Odds.event_id).where(Events.sport == sport)
        return [o.as_dict() for o in db.execute(q).scalars().all()]
    finally:
        db.close()
```

Discord integration (quick push-only)

```
# discord_bot/bot.py
import os, asyncio, json, aiohttp
WEBHOOK_URL = os.getenv("DISCORD_WEBHOOK_URL")

async def send_discord_alert(title: str, url: str):
    async with aiohttp.ClientSession() as s:
        await s.post(WEBHOOK_URL, json={"content": f"📰
{title}\n{url}"})
```

the backend's alert service calls `send_discord_alert(...)` whenever a `news_mentions` row is inserted for a subscribed team/player.

mobile delivery options

1. **React Native (Expo)**: best for iOS/Android with one codebase. Use WebSocket for live ticks, push with **Expo Notifications** or FCM/APNs.
2. **PWA**: quickest MVP (React + WebSocket + Push API). Later, wrap with Capacitor to reach app stores.
3. **Discord-first**: ship an MVP that *notifies immediately*; add mobile later.

MVP scope (what to build first)

- sports: **NFL, NBA, MLB** (add UFC after the core works; MMA data is spikier).
- features:
    - team pages: schedule, standings, **latest game box score**, last 5 games
    - player cards: season & game logs
    - odds: moneyline/spread/total for upcoming games (top 3 books)
    - injuries: consolidated feed per team
    - alerts: **news mentions for my teams** (Discord + in-app)

- realtime: live game state via WebSocket (start with score + quarter/inning/round)

testing & quality

- unit tests for providers (mock API payloads)
- contract tests for DB models (Alembic migrations tested)
- alert engine tests: keyword matching, de-duplication, user targeting
- load test WebSocket broadcast with Locust

deployment

- docker-compose for local:
    - `api` (FastAPI+Uvicorn), `db` (Postgres), `redis`, `jobs` (scheduler)
- cloud:
    - Render/Fly.io/Heroku for API
    - Railway/Neon for Postgres
    - Upstash for Redis
    - Netlify/Vercel for web PWA
- scheduled jobs run inside the same container (APScheduler) or on a worker dyno (Celery Beat)

security & compliance

- rate-limit endpoints (Redis token bucket)
- cache “hot” endpoints (e.g., `/events?today`) for 5–30s
- log PIIs minimally; encrypt at rest where possible
- betting disclaimer + age gate; geofencing if required by your jurisdiction

deliverables checklist (to match typical assignment criteria)

- **Requirements spec**: user stories (“As a fan, I want… so that…”), acceptance criteria
- **Architecture diagram**: included above
- **API contract**: document REST & WebSocket payloads (OpenAPI auto-gen via FastAPI)
- **DB schema & ERD**: provided (convert to a diagram in your report)
- **Prototype code**: shown above; expand into the repo structure
- **Plan for updates**: job schedules & provider adapters
- **Cross-platform plan**: RN/PWA + Discord integrations
- **Testing strategy**: unit + integration + load
- **Risk & mitigation**: API limits, UFC data coverage, odds compliance
- **Timeline**: 2-week MVP: week 1 (teams/events/odds + alerts); week 2 (players/injuries + RN skeleton)