عالی رفیق — بزنیم به کار. 🌱  
من همین الان یادداشت می‌کنم که این طراحی و نقشه‌کشی **همین امروز** انجام شد تا تاریخ و زمانش ثبت باشه:

**تاریخ و زمان:** ۱۰ اوت ۲۰۲۵ / ساعت 15:39 (Europe/London)

در ادامه یه **نقشهٔ معماری ماژولارِ مونولیت** برای «فرح» می‌ذارم — طوری که برای شروع سریع مناسب باشه، سبک بمونه، و در آینده بتونیم مرحله‌ای به میکروسرویس مهاجرت کنیم.

**۱) اصول کلی تصمیم**

* از **مونولیت ماژولار** شروع کنیم: سرعت توسعه بالا، پیچیدگی کمتر، نگهداری آسان‌تر برای تیم کوچک (تو و احتمالاً یه نفر دیگه).
* کد و لایه‌ها را **ماژولار** بنویسیم تا استخراج ماژول‌ها به سرویس‌های مجزا در آینده آسان باشه.
* اولویت: **سادگی، قابل‌درک بودن، تست‌پذیری** و حداقل هزینهٔ زیرساخت.
* آماده‌سازی برای آینده: قرار بدیم که هر ماژول API‌محور باشه و ایونت‌ها (events) لاگ بشن — بعداً می‌شه این‌ها رو برش زد و بیرون کشید.

**۲) لیست ماژول‌ها (اپ‌ها)**

هر ماژول = یک app در جنگو (یا ماژولی مشابه اگر فریم‌ورک دیگه انتخاب بشه).

۱. core / common — توابع و مدل‌های پایه، هندلر خطا، middlewareهای عمومی  
۲. accounts — احراز هویت، پروفایل کاربر، نقش‌ها و مجوزها (RBAC)  
۳. organizations — شرکت/فروشگاه/کسب‌وکار (multi-tenant logical)  
۴. stores (اگر لازم جدا) — داده‌های مربوط به هر فروشگاه (در صورت چند فروشگاهی)  
۵. chart\_of\_accounts — ساختار حساب‌ها، سرفصل‌ها، ترازها  
۶. transactions — ثبت، ویرایش، حذف تراکنش‌ها، اسناد پیوست‌شده  
۷. invoices & billing — فاکتورها، رسیدها، پرداخت‌ها (در صورت نیاز)  
۸. reporting — گزارش‌های مالی، ترازنامه، گردش حساب، اکسپورت CSV/PDF  
۹. imports\_exports — ایمپورت از اکسل/CSV و خروجی گرفتن  
۱۰. api — لایهٔ API (REST/GraphQL) و نسخه‌بندیِ API  
۱۱. web\_frontend — صفحات سمت کاربر (در صورت SSR با جنگو) یا صرفاً proxy به SPA  
۱۲. integrations — درگاه پرداخت، پیامک، ایمیل، POS یا سایر سیستم‌ها  
۱۳. tasks — پردازش‌های پس‌زمینه، scheduled jobs  
۱۴. admin — پنل مدیریتی (مبتنی بر Django admin یا پنل شخصی‌سازی شده)  
۱۵. tests — تست‌های اتوماتیک و fixtures

**۳) رابطه بین ماژول‌ها (خلاصه)**

* accounts ⇄ organizations: یک کاربر می‌تواند متعلق به چند سازمان باشد (در صورت نیاز)؛ هر سازمان چندین کاربر و چندین فروشگاه می‌تواند داشته باشد.
* organizations ⇄ chart\_of\_accounts: هر سازمان یک/چند شِمای سرفصل دارد (قابلیت کپی یا تمپلیت برای ساختن سریع).
* transactions ⇄ chart\_of\_accounts: هر تراکنش به حساب‌های مبدا/مقصد (debit/credit) ارجاع می‌دهد.
* transactions ⇄ invoices/billing: تراکنش‌ها می‌توانند مرتبط با فاکتورها باشند.
* reporting از داده‌های transactions و chart\_of\_accounts و organizations گزارش می‌گیرد.
* api لایه‌ای است که ماژول‌ها را به بیرون (پنل وب یا اپ موبایل) عرضه می‌کند.
* tasks پردازش‌های سنگین (مثلاً محاسبهٔ تراز دوره‌ای، تولید PDF دسته‌ای، ایمیل یادآورها) را اجرا می‌کند.

**۴) پیشنهاد دیتابیس و چندتِنَنتی (multi-tenancy)**

برای شروع **ساده ولی مقیاس‌پذیر** پیشنهاد می‌کنم:

* **یک دیتابیس مشترک (PostgreSQL)**
* مدل Organization (یا Company) که همه‌ی رکوردهای مربوط به آن سازمان به آن ارجاع داشته باشند (FK یا tenant\_id) — یعنی **row-level tenancy**.  
  مزایا: ساده، بکاپ و مدیریت آسان‌تر، هزینهٔ کمتر.  
  معایب: با رشد فوق‌العاده زیاد ممکنه به محدودیت برسیم (قابل‌حل با شاردینگ یا مهاجرت به schema-per-tenant بعداً).
* در آینده، در صورت نیاز می‌تونیم به schema-per-tenant یا حتی DB-per-tenant مهاجرت کنیم.

**۵) پیشنهاد tech stack (شروع سریع و قابل نگهداری)**

* Backend: **Django** (چون قبلاً پایه‌اش بودی) — یا Django + Django REST Framework
* DB: **PostgreSQL**
* Queue / Broker: **Redis** (برای ساده‌سازی) + **Celery** یا **Dramatiq** (برای tasks) — پیشنهاد: Celery چون اکوسیستم بزرگتر داره.
* Caching: Redis
* Frontend:
  + ساده: قالب‌های Django (برای MVP خیلی سریع)
  + مدرن: **React** یا **Vue** (SPA) که از API مصرف کنه — این مسیر جدا کردن UI از backend را ساده می‌کند.
* Containerization: **Docker** (شروع با docker-compose)
* CI/CD: GitHub Actions یا GitLab CI (پای‌هزینه کم و ساده)
* Observability: Sentry (برای خطا) + Prometheus/Grafana در آینده یا سرویس‌های مدیریت‌شده
* Auth: JWT یا session-based (برای API بهتره JWT/OAuth2) — ولی از روز اول از Django's auth و DRF token support استفاده کن تا ساده باشه.

**۶) ساختار فولدر پیشنهادی (Django)**

bash

CopyEdit

farah\_project/

├─ manage.py

├─ requirements.txt

├─ Dockerfile

├─ docker-compose.yml

├─ config/ # project settings (env-based)

│ ├─ settings/

│ ├─ urls.py

│ └─ wsgi.py

├─ apps/

│ ├─ core/

│ ├─ accounts/

│ ├─ organizations/

│ ├─ chart\_of\_accounts/

│ ├─ transactions/

│ ├─ invoices/

│ ├─ reporting/

│ ├─ integrations/

│ └─ tasks/

├─ web\_frontend/ # اگر از templates استفاده می‌کنی

├─ api/ # نسخه‌بندی API: v1, v2 ...

├─ docs/

└─ tests/

هر app:

* models.py
* serializers.py (اگر DRF)
* views.py / viewsets.py
* urls.py
* services.py (business logic خارج از views)
* tasks.py (Celery tasks)
* tests/

هدف: keep views thin — منطق در services.py باشه تا تست‌پذیر بمونه.

**۷) مدل‌سازی اولیه (خیلی خلاصه — برای MVP)**

* Organization: id, name, timezone, currency, settings(json)
* User (extends AbstractUser): email, phone, roles
* Account (ChartOfAccount): id, org\_id, code, name, type (asset/liability/equity/revenue/expense), parent\_id
* Transaction: id, org\_id, date, description, lines (FK ➜ TransactionLine), total\_amount
* TransactionLine: transaction\_id, account\_id, debit, credit, metadata(json)
* Invoice: org\_id, customer\_info(json), amount, status, linked\_transactions
* AuditLog: who, what, when, changes (برای ردیابی و حسابرسی)

**۸) API design & versioning**

* از روز اول **API versioning** بذاریم (/api/v1/...)
* اکشن‌های اصلی: CRUD برای accounts/transactions/invoices, report endpoints (aggregate)
* authentication: endpoints برای login/logout/refresh-token
* برای مصرف داخلی و اپلیکیشن موبایل یا SPA، REST کافی است؛ در آینده می‌توان GraphQL اضافه کرد (در صورت نیاز به queryهای پیچیده).

**۹) پس‌زمینه / Scheduled jobs**

* وظایف سنگین مثل: تولید گزارش‌های دوره‌ای، ارسال ایمیل روزانه/ماهانه، بستن حساب دوره‌ای → Celery beat + workers
* نگه داشتن لاگ اجرای تسک‌ها و مکانیزم retry معقول.

**۱۰) لاگینگ، مانیتورینگ و backup**

* لاگ‌ها: structured JSON logs (برای ارسال به ELK یا دیگری)
* Sentry برای error tracking
* پشتیبان‌گیری روزانه از PostgreSQL (dump) و تست بازیابی دوره‌ای
* متریک‌های پایه (latency, queue length, error rate) + داشبورد ساده

**۱۱) مسیر ارتقا به میکروسرویس (قدم به قدم)**

1. طراحی ماژولار و جداسازی لایهٔ business logic (services) از framework-specific code.
2. برداشتن ماژول‌هایی که بالاترین نیاز به مقیاس دارند (مثلاً reporting یا integrations) و تبدیل آن‌ها به سرویس‌های مستقل با API.
3. معرفی یک API Gateway ساده (یا Nginx reverse proxy) و یک Auth Service مرکزی.
4. در صورت نیاز به تبادل رویداد: اضافه کردن message broker مناسب (RabbitMQ یا Kafka) برای event-driven comms.
5. مهاجرت تدریجی دیتابیس: یا schema-per-tenant یا DB-per-tenant برای سازمان‌هایی که دادهٔ خیلی زیادی دارند.
6. کپسوله‌سازی در کانتینرها و مهاجرت به Kubernetes وقتی بار و تعداد سرویس‌ها بالاتر رفت.

**۱۲) نکات عملی /‌ توصیه‌ها برای شروع سریع (MVP)**

* از همون اول تست بنویس: هر ماژول حداقل unit tests و چند integration test داشته باشه.
* اول MVP: ثبت تراکنش، لیست تراکنش‌ها، دسته‌بندی حساب‌ها، صفحهٔ سازمان و احراز هویت. (گزارش‌ها ساده CSV/HTML)
* تحویل سریع: از Django admin برای مدیریت اولیه استفاده کن تا یک UI مدیریتی ساده داشته باشی بدون هزینهٔ اضافه.
* لاگ و audit ساده بذار (هر تغییری که مهمه لاگ بشه).
* تنظیم محیط dev سبک: docker-compose با ۳ سرویس (web, db, redis) — همین کافیه برای شروع.

رفیق، این نقشهٔ پایه‌ایه که هم الان می‌تونه تو رو سریع حرکت بده و هم در آینده اجازهٔ رشد و جداسازی می‌ده.