DOCUMENTAÇÃO BASE DO SISTEMA — Lead SaaS (White-Label)

Versão: 1.0

Sumário executivo

Este documento define a **Documentação Base do Sistema** para o produto SaaS de 360º de marketing (captura, tratamento e automação de leads) em modelo **white-label**. Todas as decisões de arquitetura, convenções, padrões e entregáveis descritos aqui são mandatórios durante o desenvolvimento: qualquer desvio deverá ser sinalizado antes da implementação.

Princípios norteadores (ordem de prioridade): **Consistência > Clareza > Escalabilidade > Performance.**

---

1. Objetivo do sistema

Construir uma plataforma SaaS white-label para captura, enriquecimento, automação e gestão de leads que permita: - Receber leads de múltiplas fontes (formulários, webhooks, Google Maps, Instagram, importações e APIs de terceiros). - Tratar, deduplicar e enriquecer leads automaticamente (CNPJ, endereço, dados públicos). - Orquestrar automações e notificações (email, SMS, webhook, WhatsApp) por regras configuráveis por conta. - Fornecer painel web personalizável por cliente (branding) para gerir leads, pipelines e relatórios. - Expor API para integrações e permitir multi-tenant seguro por `Account`.

Público-alvo: agências de marketing, times de vendas e empresas que desejam centralizar prospecção e nutrição de leads com marca própria (white-label).

---

2. Regras de negócio principais

1. **Multi-tenant:** todos os dados são ligados a uma `Account`. Usuários só acessam dados da sua `Account`.
2. **Deduplicação:** ao receber um lead, aplicar regras (ordem): por `email` → por `phone` → por `cnpj` → por combinação `nome + telefone` dentro de janela de 30 dias. Se empate, atualizar o lead existente (merge) e criar `LeadEvent` de merge.
3. **Enriquecimento:** todos os leads passam por pipeline de enriquecimento assíncrono (worker): consulta CNPJ, Google Places, geocoding, scraping social quando autorizado.
4. **Fluxo de automação:** gatilhos (on_create, on_update, on_tag, on_stage_change) disparam ações (send_email, send_sms, webhook, add_tag, assign_user). Regras configuráveis por conta via interface.
5. **Branding:** cada `Account` pode customizar logo, paleta de cores, subdomínio (opcional) e templates de email. Assets armazenados por `Account` e servidos via CDN.

6. **Retenção / auditoria:** manter histórico de alterações (`lead_history`) por 90 dias padrão (configurável). Logs de auditoria para ações críticas.
7. **Segurança / permissões:** RBAC mínimo: `superadmin` (plataforma), `admin` (account), `manager`, `agent`, `readonly`. Autenticação JWT + opção SSO (OAuth2) por conta.
8. **Webhooks de recebimento:** endpoints públicos protegidos por token HMAC e validação de origem + rate limiting.

---

3. Componentes do backend

Arquitetura: **Monolito modular** com camadas claras e um processo worker para jobs assíncronos.

3.1 Camadas e responsabilidade

- **API (Express)**: exposição de rotas REST/JSON e healthchecks.
- **Controllers**: validação de entrada (DTOs), orquestração de serviços, resposta padronizada.
- **Services**: lógica de negócio (leadService, automationService, enrichmentService, userService, accountService).
- **Repositories (Prisma)**: abstração de acesso a banco; métodos CRUD e queries especializadas.
- **Workers**: processamento assíncrono (BullMQ) para scraping, enriquecimento, envio de notificações, retries e backoff.
- **Integrations**: módulos específicos por provedor (GoogleMapsClient, CnpjClient, InstagramClient, EmailClient, SmsClient).
- **Middlewares**: autenticação JWT, autorização por roles, validação de schema (zod), logging, rate limiting, error handler.
- **Utils / Helpers**: formatação, deduplication utilities, normalizadores de telefone/email.

3.2 Principais serviços a implementar

- `LeadService` (criar, atualizar, merge, buscar, aplicar tags, mover estágio)
- `AutomationService` (avaliar gatilhos, enfileirar ações)
- `EnrichmentService` (pipeline síncrono para dados rápidos; delega jobs pesados ao worker)
- `IntegrationService` (abstrai chamadas externas e tokens)
- `AuthService` (token, refresh, SSO)
- `BrandingService` (templates, assets)
- `AuditService` (grava logs de ação de usuários)

---

4. Componentes do frontend

Stack: **React + Vite** com arquitetura de componentes, state management (Zustand ou Redux Toolkit), roteamento React Router, e biblioteca de componentes (opcional: Radix + shadcn/ui ou Material).

4.1 Áreas principais

- **Landing / Auth**: login, signup (multi-tenant invite), reset password.
- **Admin (Account settings)**: branding, usuários, integrações, billing info.
- **Leads**: listagem (filtros dinâmicos), visualização detalhada (timeline de eventos), edição, merge manual.

- **Pipelines / Automations**: builder visual de regras (drag & drop simples) e histórico de execuções.
- **Imports & Sources**: configuração de fontes (form builders, webhooks, Google Maps scraping, Instagram scraping).
- **Relatórios**: métricas de captura, conversão, tempo médio de resposta.

4.2 Themability (white-label)

- Tokens de design (CSS variables / Tailwind config) por `Account` com fallback global.
- Possibilidade de registro de subdomínio CNAME + emissão de certificados TLS (Let's Encrypt) via infraestrutura.

---

5. Estrutura de diretórios proposta

Aplica tanto ao repositório monorepo atual quanto a uma possível separação futura.

```
/ (root)
├─ backend/
│  ├─ src/
│  │  ├─ api/
│  │  │  ├─ routes/
│  │  │  │  ├─ auth.routes.ts
│  │  │  │  ├─ user.routes.ts
│  │  │  │  ├─ lead.routes.ts
│  │  │  │  └─ integration.routes.ts
│  │  ├─ controllers/
│  │  ├─ services/
│  │  ├─ repositories/
│  │  ├─ workers/
│  │  ├─ integrations/
│  │  ├─ middlewares/
│  │  ├─ utils/
│  │  ├─ lib/
│  │  │  ├─ prisma.ts
│  │  │  ├─ logger.ts
│  │  └─ server.ts
│  ├─ prisma/
│  │  ├─ schema.prisma
│  │  └─ migrations/
│  ├─ Dockerfile
│  └─ package.json
├─ frontend/
│  ├─ src/
│  │  ├─ pages/
│  │  ├─ components/
│  │  ├─ hooks/
│  │  ├─ services/api.ts
│  │  ├─ theme/
```

```
│   │   └─ app.tsx
│   ├─ package.json
│   └─ vite.config.ts
├─ infra/
│   ├─ docker-compose.yml
│   ├─ k8s/ (se aplicável)
│   └─ scripts/
├─ docs/
│   ├─ architecture.md
│   └─ api-spec.yaml (OpenAPI)
└─ README.md
```

---

6. Estrutura de rotas (REST) proposta — principais endpoints

Todas as rotas devem seguir versão: `/api/v1/...`

Autenticação - `POST /api/v1/auth/login` — body: `{ email, password }` → returns `{ accessToken, refreshToken }` - `POST /api/v1/auth/refresh` — refresh token - `POST /api/v1/auth/logout`

Usuários e contas - `GET /api/v1/accounts/:accountId` — admin only - `POST /api/v1/accounts` — cria conta (superadmin) - `GET /api/v1/users` — listar usuários da conta - `POST /api/v1/users` — criar usuário - `PUT /api/v1/users/:id` — atualizar

Leads - `POST /api/v1/leads` — criar lead (público — protegido por token HMAC opcional) - `GET /api/v1/leads` — listagem com filtros (accountId implícito) - `GET /api/v1/leads/:id` — detalhe - `PUT /api/v1/leads/:id` — atualizar - `POST /api/v1/leads/:id/merge` — merge manual - `POST /api/v1/leads/import` — import CSV

Automations & Pipelines - `GET/POST/PUT /api/v1/automations` - `POST /api/v1/automations/:id/execute` — execução manual

Integrations - `POST /api/v1/webhooks/form` — ponto de entrada para forms - `POST /api/v1/integrations/google-maps/sync` - `POST /api/v1/integrations/instagram/scrape`

Branding - `GET /api/v1/branding` — retorna configurações - `PUT /api/v1/branding` — atualiza

Admin / Health - `GET /api/v1/health` — healthcheck - `GET /api/v1/metrics` — métricas (auth restricted)

Todos os responses devem obedecer o envelope padrão `{ success: boolean, data: any, error?: { code, message } }`.

---

7. Fluxos essenciais (detalhados)

7.1 Login / sessão

1. Usuário envia `POST /auth/login`.
2. `AuthController` valida credenciais via `AuthService`.
3. `AuthService` retorna `accessToken` JWT curto e `refreshToken` salvo no DB (hash). JWT contém `userId`, `accountId`, `roles`.
4. Middleware `authMiddleware` valida JWT em rotas protegidas e injeta `req.context = { user, accountId }`.

7.2 Cadastro de lead (via webhook/form)

1. Request chega em `POST /webhooks/form` com `source` e `payload`.
2. `WebhookController` normaliza payload via `NormalizationService` (mapeia campos, normaliza telefone/email).
3. `LeadService.createOrMerge()` aplica deduplicação síncrona mínima; se merge ocorrer, cria `LeadEvent`.
4. `LeadService` grava lead inicial e enfileira job de `EnrichmentWorker` (BullMQ).
5. `EnrichmentWorker` executa chamadas externas, atualiza lead com dados enriquecidos e publica eventos para `AutomationService`.
6. `AutomationService` avalia regras e enfileira ações (email, webhook target, SMS).

7.3 Enriquecimento e automações (assíncrono)

- Jobs idempotentes com idempotencyKey; retries com backoff exponencial; dead-letter queue para falhas repetidas.

---

8. Modelos de dados (visão lógica) — proposta (Prisma / PostgreSQL)

8.1 Principais entidades (resumo)

- **Account**: configurações da empresa (branding, quotas, plan)
- **User**: pertence a Account; roles
- **Lead**: dados do lead (nome, email, phone, cnpj, dataRecebimento, source, json extras)
- **LeadEvent**: histórico de mudanças e interações
- **Tag**: tag por account (muitos-para-muitos com Lead)
- **Automation**: regras configuradas
- **Integration**: credenciais e config por account
- **AuditLog**: ações críticas realizadas por usuários

8.2 Exemplo de schema (Prisma simplified)

```
model Account {
  id        String   @id @default(uuid())
  name      String
  subdomain String?
  branding  Json?
```

```
  plan      String
  users     User[]
  leads     Lead[]
  createdAt DateTime @default(now())
}

model User {
  id        String  @id @default(uuid())
  email     String  @unique
  password  String
  name      String
  role      String
  account   Account @relation(fields: [accountId], references: [id])
  accountId String
  createdAt DateTime @default(now())
}

model Lead {
  id          String   @id @default(uuid())
  account     Account  @relation(fields: [accountId], references: [id])
  accountId   String
  name        String?
  email       String?
  phone       String?
  cnpj        String?
  source      String
  stage       String?
  score       Int      @default(0)
  data        Json?    // campos flexíveis
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  events      LeadEvent[]
  tags        Tag[]    @relation("LeadTags")
}

model LeadEvent {
  id        String  @id @default(uuid())
  lead      Lead    @relation(fields: [leadId], references: [id])
  leadId    String
  type      String
  payload   Json?
  createdAt DateTime @default(now())
}

model Tag {
  id        String @id @default(uuid())
  name      String
  account   Account @relation(fields: [accountId], references: [id])
  accountId String
  leads     Lead[]  @relation("LeadTags")
}
```

```
model Automation {
  id        String @id @default(uuid())
  accountId String
  account   Account @relation(fields: [accountId], references: [id])
  name      String
  trigger   Json  // representação da regra
  actions   Json  // passos a executar
}

model AuditLog {
  id        String  @id @default(uuid())
  accountId String
  userId    String?
  action    String
  payload   Json?
  createdAt DateTime @default(now())
}
```

Observações: usar `jsonb` (Prisma `Json`) em Postgres para `data`, `branding`, `trigger` e `actions` para flexibilidade. Indexar campos frequentemente consultados (`email`, `phone`, `cnpj`) e usar GIN index para `jsonb` quando necessário.

---

9. Serviços, helpers e middlewares (lista final)

Helpers/Utils

- `normalizePhone(number, country)`
- `normalizeEmail(email)`
- `dedupeKeyFromLead(lead)` → retorna chaves usadas na dedup

Middlewares

- `authMiddleware` — valida JWT e injeta contexto
- `authorize(roles[])` — verifica permissões
- `validateBody(schema)` — valida via `zod`
- `rateLimiter` — por IP e por account
- `errorHandler` — formata erros e envia para Sentry

Background workers

- `EnrichmentWorker` (CNPJ, Google, Social)
- `NotificationWorker` (envio email/SMS/webhook)
- `ScraperWorker` (Instagram, Google Maps)

---

10. Tecnologias recomendadas

- **Backend:** Node.js 18+, TypeScript, Express 4/5

- **ORM:** Prisma (Postgres)
- **DB:** PostgreSQL 14+
- **Fila:** Redis + BullMQ
- **Frontend:** React 18+, Vite, TypeScript
- **Auth:** JWT short lived + refresh tokens (store refresh token hash), opcional SSO OAuth2
- **Logger:** pino ou winston
- **Testes:** Jest + Supertest (backend), Vitest (frontend)
- **CI/CD:** GitHub Actions (build, lint, test, deploy)
- **Container:** Docker
- **Observability:** Sentry (errors), Prometheus/Grafana (metrics)

---

11. Padrões obrigatórios de escrita de código

- **TypeScript estrito** (`"strict": true`) em `tsconfig.json` .
- **ESLint + Prettier** com regras compartilhadas; `precommit` hooks (husky) para lint/staged.
- **Arquitetura em camadas:** controllers → services → repositories. Controllers apenas orchestrate, sem lógicas de negócio.
- **Injeção de dependências** leve (ex.: `typedi` ou pattern de construção manual) para facilitar testes.
- **Tratamento de erros centralizado:** lançar `AppError` com `code` , `status` . Middleware converte para http response.
- **DTOs e validação:** todo body/query/params validados via `zod` (ou `joi`) antes do controller.
- **Testes:** cobertura mínima crítica 70% (unit para services, integration para endpoints críticos).

---

12. Convenções (nomenclatura e estrutura)

- Arquivos: `kebab-case` para rotas e utilitários (`lead-service.ts`), `PascalCase` para classes (`LeadService`).
- Controllers: `controllers/lead.controller.ts` export default class `LeadController` com métodos públicos `create` , `list` , `get` , `update` .
- Services: `services/lead.service.ts` export default class `LeadService` .
- Repositories: `repositories/lead.repository.ts` com métodos `findById` , `findByEmail` , `create` , `update` .
- Middlewares: `middlewares/auth.middleware.ts` export function `authMiddleware` .
- Tests: espelhos de pastas: `__tests__/services/lead.service.spec.ts` .

---

13. Segurança e boas práticas

- Armazenar segredos em vault (ex.: AWS Secrets Manager) ou pelo menos `.env` com variáveis no CI.
- JWT secret obrigatório; rotacionamento de chaves suportado.
- HTTPS em todas as comunicações e HSTS.
- Rate limiting para endpoints públicos e proteção contra brute force.
- Sanitização de inputs e proteção contra SQL injection (Prisma já ajuda) e XSS no frontend.
- Criptografar informações sensíveis (ex.: tokens de integração) em repouso.

---

14. Migração e plano de transição (sqlite → Postgres)

1. Adotar Postgres em staging; migrar Prisma datasource e executar `prisma migrate deploy`.
2. Validar integridade de dados: export sqlite → transform → importar em Postgres (scripts fornecidos).
3. Testes de concorrência para endpoints de ingestão de leads.
4. Habilitar RLS se optar por política reforçada por tenant.

---

15. Observability e operações

- Integrar `pino` para logs estruturados; enviar para CloudWatch/ELK.
- Métricas: expor `/metrics` compatível com Prometheus (requests, queue length, job failures).
- Alertas: erro 5xx alto, fila de dead letter > threshold, uso de CPU/DB.

---

16. Plano inicial de entregas (sprints) — 6 sprints de 2 semanas

Sprint 0 (setup): mono repo cleanup, remover node_modules, configurar ESLint/Prettier, CI (lint/test), Docker dev, migrar env secrets pattern.

Sprint 1 (core auth + accounts): Auth (JWT), Users, Accounts CRUD, basic RBAC, DB Postgres staging.

Sprint 2 (Leads ingest básico): endpoints de lead create/list/get, dedup sync básico, LeadEvent, tests.

Sprint 3 (Workers + Enrichment): Redis + BullMQ, EnrichmentWorker, NotificationWorker, sample integration Google Maps (mock).

Sprint 4 (Automations & UI): Automations CRUD, executor básico, frontend pages: leads list, lead detail, auth.

Sprint 5 (Branding + Integrations): Branding UI, email templates, Instagram & CNPJ integration, rate limits.

Sprint 6 (Hardening & observability): Sentry, metrics, backup policy, load tests, security review.

---

17. Contratos de API e documentação

- Gerar OpenAPI (Swagger) automatizado a partir das rotas (ex.: `ts-openapi` ou `swagger-jsdoc`).
- Postman collection exportável.

---

18. Checklist de decisões pendentes (para você ou eu decidir)

1. Confirmação do SGBD: **Postgres** (recomendado) — confirmar.
2. Provedor de email preferido (SendGrid / Mailgun) — definir.

3. Mensageria SMS/WhatsApp — qual provedor (Twilio / Zenvia / Gupshup)?
4. Política de retenção de dados padrão (90 dias sugerido).
5. Número esperado de tenants / leads por mês (para sizing infra).
6. Deseja suporte nativo a subdomínio personalizado (CNAME/SSL opex)?

---

19. Próximos passos imediatos (o que eu vou gerar agora se você confirmar)

1. Versão final deste documento em `docs/documentacao-base.md` no repositório.
2. OpenAPI minimal com os endpoints críticos (auth, leads, webhooks).
3. Scripts de migração de sqlite → postgres.
4. Exemplos de `Dockerfile` e `docker-compose.yml` para dev com Postgres e Redis.
5. Lista de tarefas detalhada (issues/epics) ordenadas por prioridade.

---

20. Contato e responsabilidades

Do your instructions: A partir de agora, seguirei esta DOCUMENTAÇÃO BASE como regra. Antes de gerar qualquer arquivo novo, validarei internamente a conformidade com o documento. Se você solicitar algo que conflite, eu avisarei e proporei alternativa.

Documento gerado automaticamente a partir da análise do ZIP enviado.

Se concordar, eu já posso: **(A)** adicionar `docs/documentacao-base.md` ao repositório com o conteúdo acima, **(B)** gerar o `docker-compose.yml` dev com Postgres e Redis, e **(C)** gerar a OpenAPI inicial. Diga apenas: “ok, gere os arquivos iniciais” para que eu proceda (ou eu sigo produzindo sem aguardar — sua preferência).

ARTEFATOS GERADOS AUTOMATICAMENTE

A seguir eu adicionei exemplos prontos e didáticos para implementação profissional. Os arquivos abaixo foram inseridos neste documento para você copiar/colar no repositório ou salvar como arquivos no seu projeto. Cada arquivo vem acompanhado de instruções CLI para execução local e notas de deploy.

1) docker-compose.yml (desenvolvimento)

O `docker-compose.yml` fornece um ambiente local com backend, frontend, PostgreSQL e Redis. Use-o para desenvolvimento e testes.

```
version: '3.8'
services:
  postgres:
```

```
    image: postgres:14
    environment:
      POSTGRES_USER: lead_admin
      POSTGRES_PASSWORD: lead_password
      POSTGRES_DB: lead_db
    ports:
      - '5432:5432'
    volumes:
      - db_data:/var/lib/postgresql/data

  redis:
    image: redis:7
    ports:
      - '6379:6379'

  backend:
    build: ./backend
    command: npm run dev
    volumes:
      - ./backend:/usr/src/app
      - /usr/src/app/node_modules
    ports:
      - '4000:4000'
    environment:
      DATABASE_URL: 'postgresql://lead_admin:lead_password@postgres:5432/
lead_db'
      REDIS_URL: 'redis://redis:6379'
      JWT_SECRET: 'change_me_in_production'
    depends_on:
      - postgres
      - redis

  frontend:
    build: ./frontend
    command: npm run dev -- --host
    volumes:
      - ./frontend:/usr/src/app
      - /usr/src/app/node_modules
    ports:
      - '3000:5173'
    environment:
      VITE_API_URL: 'http://localhost:4000/api'

volumes:
  db_data:
```

Como usar (local) 1. Remova `node_modules` do ZIP e garanta que `backend` e `frontend` possuam `package.json` corretos. 2. No terminal: `docker-compose up --build`. 3. Acesse frontend em `http://localhost:3000` e backend em `http://localhost:4000`.

Nota: altere `JWT_SECRET` e outras variáveis sensíveis em `.env` para produção. Em CI use secrets do provedor.

---

2) prisma/schema.prisma (configuração para Postgres)

Abaixo está a versão do `schema.prisma` compatível com PostgreSQL, derivada da modelagem da documentação.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Account {
  id        String   @id @default(uuid())
  name      String
  subdomain String?
  branding  Json?
  plan      String
  users     User[]
  leads     Lead[]
  createdAt DateTime @default(now())
}

model User {
  id        String  @id @default(uuid())
  email     String  @unique
  password  String
  name      String
  role      String
  account   Account @relation(fields: [accountId], references: [id])
  accountId String
  createdAt DateTime @default(now())
}

model Lead {
  id          String   @id @default(uuid())
  account     Account  @relation(fields: [accountId], references: [id])
  accountId   String
  name        String?
  email       String?  @db.VarChar(255)
  phone       String?
  cnpj        String?
  source      String
  stage       String?
```

```
  score       Int      @default(0)
  data        Json?
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  events      LeadEvent[]
  tags        Tag[]    @relation("LeadTags")
}

model LeadEvent {
  id        String  @id @default(uuid())
  lead      Lead    @relation(fields: [leadId], references: [id])
  leadId    String
  type      String
  payload   Json?
  createdAt DateTime @default(now())
}

model Tag {
  id        String @id @default(uuid())
  name      String
  account   Account @relation(fields: [accountId], references: [id])
  accountId String
  leads     Lead[]  @relation("LeadTags")
}

model Automation {
  id        String @id @default(uuid())
  accountId String
  account   Account @relation(fields: [accountId], references: [id])
  name      String
  trigger   Json
  actions   Json
}

model AuditLog {
  id        String  @id @default(uuid())
  accountId String
  userId    String?
  action    String
  payload   Json?
  createdAt DateTime @default(now())
}
```

Como aplicar 1. Defina `DATABASE_URL` no `.env` (ex.: `postgresql://user:pass@localhost:5432/lead_db`). 2. No backend: `npx prisma migrate dev --name init`. 3. Gere o client: `npx prisma generate`.

---

3) OpenAPI minimal (api-spec.yaml) — endpoints críticos

Abaixo há um OpenAPI 3.0 mínimo com os endpoints de autenticação, leads e webhooks.

```
openapi: 3.0.3
info:
  title: Lead SaaS API
  version: 1.0.0
servers:
  - url: http://localhost:4000/api/v1
paths:
  /auth/login:
    post:
      summary: Login
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
                password:
                  type: string
      responses:
        '200':
          description: OK
  /leads:
    post:
      summary: Criar lead
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                name:
                  type: string
                email:
                  type: string
                phone:
                  type: string
                source:
                  type: string
      responses:
        '201':
          description: Created
    get:
```

```
      summary: Listar leads
      parameters:
        - in: query
          name: page
          schema:
            type: integer
      responses:
        '200':
          description: OK
  /webhooks/form:
    post:
      summary: Recebe dados de formulários
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
      responses:
        '200':
          description: Accepted
```

Como usar - Salve esse YAML como `docs/api-spec.yaml` ou importe no Swagger UI / Postman.

---

4) Passo a passo didático de implementação (checklist detalhado)

Abaixo há um roteiro prático, com comandos e prioridades, para avançar profissionalmente e versionar tudo corretamente.

Preparação do repositório (Sprint 0)

1. Limpeza
2. Remova `node_modules` do repositório: `git rm -r --cached backend/node_modules frontend/node_modules` e adicione `node_modules/` ao `.gitignore`.
3. Crie arquivos de ambiente example
4. `backend/.env.example` com variáveis: `DATABASE_URL`, `REDIS_URL`, `JWT_SECRET`, `PORT`.
5. Padronize `package.json` scripts
6. Backend: `dev`, `build`, `start`, `lint`, `test`.
7. Frontend: `dev`, `build`, `preview`, `lint`.
8. Configure ESLint + Prettier + Husky
9. `npx eslint --init`, `npm i -D prettier husky lint-staged`, adicione `pre-commit` hook para `npm run lint`.

Infra local (docker)

1. Copie o `docker-compose.yml` para a raiz.
2. `docker-compose up --build`.

3. Acesse e verifique logs: `docker-compose logs -f backend`.

Banco e Prisma

1. Ajuste `DATABASE_URL` no `backend/.env`.
2. Rode as migrations: `npx prisma migrate dev --name init`.
3. Gere client: `npx prisma generate`.
4. Se tiver dados no sqlite, exporte e importe (eu posso gerar script de migração se desejar).

Backend — estruturar camadas

1. Crie pastas: `src/controllers`, `src/services`, `src/repositories`, `src/middlewares`, `src/workers`, `src/integrations`.
2. Mova rotas existentes para `src/api/routes` e refatore para chamar `controllers`.
3. Implemente `AuthService` e `authMiddleware` (JWT verifying using `JWT_SECRET`).
4. Configure logger (`pino`) e error handler.

Workers e filas

1. Adicione dependência `bullmq` e configure conexão com `REDIS_URL`.
2. Implemente `EnrichmentWorker` com job types: `enrich:lead`, `notify:lead`.
3. Garanta idempotência dos jobs usando `jobId` baseado no lead id + tipo.

Frontend

1. Garanta que `services/api.ts` use `VITE_API_URL`.
2. Crie páginas: Login, Leads List, Lead Detail, Account Settings.
3. Implemente theming via CSS variables ou Tailwind config dinâmico (carregar branding via `GET /branding`).

Testes e CI

1. Configure `Jest` / `Supertest` para backend e `Vitest` para frontend.
2. Configure GitHub Actions: `lint`, `test`, `build` para branch `main`.

Deploy (básico)

1. Em produção, utilize container registry e orquestrador (ECS / K8s) e gerencie secrets via provider.
2. Configure backup de Postgres (pg_dump agendado ou serviço gerenciado).
3. Monitore com Sentry e Prometheus.

---

5) Próximos artefatos que posso gerar automaticamente

- Script para migrar dados do sqlite para Postgres.
- Exemplos de controllers (AuthController, LeadController) e services correspondentes.
- Configuração de BullMQ com exemplos de workers.
- Playbook de segurança (checklist de hardening) e script de rotação de chaves.

Se concordar, eu agora vou: **gerar e adicionar** ao repositório os seguintes arquivos/artefatos como arquivos separados dentro do documento (para você copiar): - `docker-compose.yml` - `backend/prisma/schema.prisma` - `docs/api-spec.yaml` - `docs/IMPLEMENTATION_CHECKLIST.md` (com os passos acima formatados)

Responda apenas: “ok, gere os arquivos” para eu finalizar a geração ou “gere apenas X” se preferir um arquivo específico primeiro.