﻿# SRVGold – Documentação Técnica

**1. Visão Geral**

O projeto \*\*SRVGold\*\* é uma API multi-tenant construída com [NestJS](https://nestjs.com/) e [Prisma](https://www.prisma.io/) para atender diferentes clientes (tenants) utilizando o mesmo código base, mas bancos de dados isolados. A aplicação inclui infraestrutura para criação de novos tenants, roteamento dinâmico de conexões e módulos de domínio que manipulam entidades legadas do ERP.

**2. Arquitetura e Tecnologias**

• \*\*Runtime:\*\* Node.js 20 (requer >= 18).

• \*\*Framework:\*\* NestJS 11 com organização modular (controllers, services e DTOs).

• \*\*ORM:\*\* Prisma 6 com dois schemas:

- `prisma/schema.prisma`: banco "main" com a tabela `Tenant`.

- `prisma/erp.prisma`: espelha a base legada (centenas de tabelas) para cada tenant.

• \*\*Banco de Dados:\*\* PostgreSQL multi-base. Existe um banco principal com metadata (`Tenant`) e um banco por cliente (nomeado `tenant\_<slug>`), clonado a partir do template `goldpdv`.

• \*\*Configuração:\*\* Variáveis de ambiente carregadas via `@nestjs/config`. O `TenantPrismaManager` monta as strings de conexão dinamicamente.

• \*\*Middleware:\*\* `TenantContextMiddleware` injeta automaticamente o `PrismaClient` do tenant com base no header `x-tenant` ou no subdomínio do host.

**3. Estrutura de Pastas e Responsabilidades**

src/

├── app.module.ts

├── main.ts

├── prisma.service.ts

├── tenant-prisma.manager.ts

├── middleware/

│ └── tenant-context.middleware.ts

├── tenants/

│ ├── tenants.controller.ts

│ ├── tenants.module.ts

│ └── tenants.service.ts

├── t\_cli/

├── t\_emp/

├── t\_usere/

└── t\_users/

| Pasta / Arquivo | Importância para o projeto |

|-----------------|----------------------------|

| `src/app.module.ts` | Declara todos os módulos, providers globais e aplica o middleware multi-tenant; é o ponto central da configuração Nest. |

| `src/main.ts` | Responsável pelo bootstrap da aplicação, incluindo CORS e definição da porta HTTP. |

| `src/prisma.service.ts` | Exposição do `PrismaClient` conectado ao banco principal; base para consultas à tabela `Tenant`. |

| `src/tenant-prisma.manager.ts` | Cria e cacheia `PrismaClient` por tenant; evita recriação de conexões e garante isolamento entre bancos. |

| `src/middleware/tenant-context.middleware.ts` | Resolve o tenant a partir do request e injeta o client correto; sem ele os services não conseguem acessar o banco certo. |

| `src/tenants/` | Endpoint, serviço e módulo para criação de novos tenants físicos e registro na tabela `Tenant`. |

| `src/t\_cli/`, `src/t\_emp/`, `src/t\_usere/`, `src/t\_users/` | Módulos de domínio que expõem CRUDs sobre entidades legadas; demonstram o padrão de implementação que deve ser seguido por novos módulos multi-tenant. |

| `prisma/` | Contém os schemas principais (`schema.prisma` e `erp.prisma`) e o código gerado em `prisma/generated/erp`, essenciais para o acesso tipado aos bancos. |

| `scripts/` | Automação em Node/TS, incluindo `migrate-tenants.ts` e `export-doc-pdf.ts`, úteis para manutenção dos bancos e geração de documentação. |

| `docs/` | Repositório dos artefatos de documentação (Markdown, PDF, DOCX). |

| `test/` | Base para testes automáticos (unitários e e2e) seguindo o padrão do Nest. |

Manter esta estrutura organizada facilita a escalabilidade do projeto, pois novos módulos podem ser criados replicando o padrão existente e o comportamento multi-tenant permanece transparente.

**4. Configuração de Ambiente**

1. \*\*Instalar dependências\*\*

npm install

2. \*\*Gerar clientes Prisma\*\* (necessário após alterar schemas ou instalar o projeto):

npx prisma generate

3. \*\*Variáveis de ambiente\*\* (arquivo `.env` na raiz):

- `DATABASE\_URL`: conexão com o banco principal (main).

- `ERP\_TEMPLATE\_URL`: conexão com o banco template utilizada por scripts.

- `BASE\_PG\_URL`: prefixo da URL PostgreSQL para montar conexões dos tenants.

- `PG\_HOST`, `PG\_PORT`, `PG\_USER`, `PG\_PASSWORD`: credenciais administrativas usadas para criar/clonar bancos.

- `PG\_SUPER\_DB` (opcional, padrão `postgres`): banco acessado pelo usuário administrativo.

- `PG\_TEMPLATE` (opcional, padrão `goldpdv`): nome do banco template clonado.

- `PORT` (opcional): porta HTTP exposta pelo Nest (padrão 3000).

> \*\*Segurança:\*\* mantenha credenciais fora do controle de versão (adicione `.env` ao `.gitignore`) e rotacione senhas em ambientes compartilhados.

**5. Execução e Scripts Úteis**

• Desenvolvimento (watch mode)

npm run start:dev

• Execução simples (sem hot-reload)

npm run start

• Build de produção + execução

npm run build && npm run start:prod

• Testes

npm run test # unit

npm run test:e2e # end-to-end

npm run test:cov # cobertura

• Migração dos tenants existentes (sincroniza o schema ERP em todos os bancos)

npm run migrate:tenants

O script `scripts/migrate-tenants.ts` lê todos os registros de `Tenant` no banco main, monta URLs com `BASE\_PG\_URL` e executa `prisma db push --schema=prisma/erp.prisma` para cada banco físico.

**6. Multi-Tenancy em Detalhes**

1. \*\*Registro de tenant\*\* – `POST /tenants` chama `TenantsService.createTenant`, que:

- evita slug duplicado consultando o banco main;

- cria um banco novo via `pg.Pool` clonando o template (`CREATE DATABASE ... WITH TEMPLATE ...`);

- registra `name`, `slug` e `db\_name` na tabela `Tenant`.

2. \*\*Resolução do tenant em runtime\*\*

- `TenantContextMiddleware` lê `x-tenant` ou o subdomínio, busca o registro no banco main (`PrismaService`) e anexa ao request:

- `tenantId` (ID numérico do tenant);

- `tenantClient` (`PrismaClient` da base do tenant, obtido pelo `TenantPrismaManager`).

- Controllers recebem `RequestWithPrisma` e delegam aos services utilizando o cliente correto.

3. \*\*Gerenciamento de conexões\*\*

- `TenantPrismaManager` memoiza `Promise<PrismaClient>` por `dbName`, evitando condições de corrida ao criar múltiplos clientes simultâneos.

- O hook `onModuleDestroy` fecha essas conexões de forma controlada quando a aplicação é encerrada.

**7. Módulos de Domínio (exemplos)**

• \*\*t\_users\*\* (`src/t\_users`)

- CRUD completo sobre `t\_users` (entidade legada com atributos como `cdusu`, `deusu`, `email`).

- DTOs (`CreateTUsersDto`, `UpdateTUsersDto`) com validações `class-validator`.

• \*\*t\_usere\*\* (`src/t\_usere`)

- Controlador e serviço expõem operações CRUD sobre `t\_usere` (relaciona usuários a empresas).

- DTOs alinhados com o schema Prisma: `id` é UUID opcional, `codusu` e `codemp` são obrigatórios, `autocod` é opcional.

• \*\*t\_cli\*\*, \*\*t\_emp\*\*, \*\*t\_comanda\*\* e outros

- Seguem o padrão Nest: módulo registra controller/service; controller recebe `RequestWithPrisma` e encaminha ao service; DTOs usam `class-validator`.

**8. Camada de Dados**

• \*\*Banco principal (`Tenant`)\*\*: gerenciado por `PrismaService` (extende `PrismaClient`). Schema definido em `prisma/schema.prisma`.

• \*\*Banco ERP\*\*: schema extenso (`prisma/erp.prisma`) com centenas de modelos. O cliente é gerado em `prisma/generated/erp` via `prisma generate`.

• \*\*Acesso\*\*: serviços trabalham com o `PrismaClient` passado pelo middleware, garantindo isolamento por tenant.

**9. Observabilidade e Logs**

• `TenantPrismaManager` configura logs de nível `warn` para o Prisma multi-tenant (ajustável conforme necessidade).

• Recomenda-se integrar `nestjs-pino` ou interceptors de logging para registrar `tenantId` em cada requisição.

**10. Boas Práticas e Próximos Passos**

• Adicionar guards/interceptors para controle de acesso específico por tenant e auditoria.

• Considerar o uso de `prisma migrate` para versionar mudanças no schema ERP de forma rastreável.

• Expandir cobertura de testes unitários e e2e para middleware e fluxos multi-tenant.

• Criar pipelines CI/CD que executem build, testes e o script de migração em ambientes de staging.

• Armazenar segredos em \_secret managers\_ (AWS Secrets Manager, Vault, etc.) e restringir acesso ao banco template.

**11. APIs Disponibilizadas**

Todas as rotas (exceto `GET /`) exigem o header `x-tenant` com o slug do tenant ou uma requisição via subdomínio configurado.

| Módulo | Método | Rota | Descrição |

|--------|--------|------|-----------|

| Core | `GET` | `/` | Resposta de saúde simples ("hello world"). |

| Tenants | `POST` | `/tenants` | Cria um novo tenant físico (clona banco template) e registra nome/slug no banco principal. |

| t\_users | `POST` | `/t\_users` | Cria usuário ERP no banco do tenant. |

| t\_users | `GET` | `/t\_users` | Lista usuários do tenant. |

| t\_users | `GET` | `/t\_users/:id` | Retorna usuário pelo ID UUID. |

| t\_users | `PUT` | `/t\_users/:id` | Atualiza todos os campos do usuário. |

| t\_users | `PATCH` | `/t\_users/:id` | Atualização parcial. |

| t\_users | `DELETE` | `/t\_users/:id` | Remove usuário. |

| t\_usere | `POST` | `/t\_usere` | Cria vínculo usuário ↔ empresa. |

| t\_usere | `GET` | `/t\_usere` | Lista vínculos. |

| t\_usere | `GET` | `/t\_usere/:id` | Detalha vínculo. |

| t\_usere | `PUT` | `/t\_usere/:id` | Atualiza vínculo. |

| t\_usere | `PATCH` | `/t\_usere/:id` | Atualização parcial. |

| t\_usere | `DELETE` | `/t\_usere/:id` | Remove vínculo. |

| t\_cli | `POST` | `/t\_cli` | Cria cliente. |

| t\_cli | `GET` | `/t\_cli` | Lista clientes. |

| t\_cli | `GET` | `/t\_cli/:id` | Detalha cliente. |

| t\_cli | `PUT` | `/t\_cli/:id` | Atualiza cliente (full). |

| t\_cli | `PATCH` | `/t\_cli/:id` | Atualiza cliente (parcial). |

| t\_cli | `DELETE` | `/t\_cli/:id` | Remove cliente. |

| t\_emp | `POST` | `/t\_emp` | Cria empresa. |

| t\_emp | `GET` | `/t\_emp` | Lista empresas. |

| t\_emp | `GET` | `/t\_emp/:id` | Detalha empresa. |

| t\_emp | `PUT` | `/t\_emp/:id` | Atualiza empresa (full). |

| t\_emp | `PATCH` | `/t\_emp/:id` | Atualiza empresa (parcial). |

| t\_emp | `DELETE` | `/t\_emp/:id` | Remove empresa. |

---

Documento atualizado automaticamente pelo Codex CLI.

**Diagnóstico**

* prisma/erp.prisma:811 expõe o espelho bruto da base legada, com ~300 modelos sem relacionamentos explícitos, datas/flags representadas como String e ausência de @@index/@relation, o que mantém o legado mas trava integridade e performance.
* O código atual só manipula poucos domínios (t\_cli, t\_emp, t\_users, t\_usere, t\_comanda, etc.), identificáveis pelos DTOs em src/t\_cli/dto/CreateTCliDto.ts:13 e serviços como src/t\_cli/TCli.service.ts:12, então há margem para reduzir drasticamente o schema exposto ao Prisma.
* A documentação (docs/technical-documentation.md) confirma que todos os tenants partilham o mesmo schema ERP e que o script npm run migrate:tenants precisa ser executado sempre que o schema mudar (ver Docs.txt). Isso exige um plano controlado de refino + propagação.

**Plano de Normalização (erp.prisma)**

* Mapear uso real: rode rg "client\\." src (ou equivalente) e catalogue quais modelos/colunas aparecem nos serviços/DTOs. Classifique cada tabela em “usada”, “possivelmente usada” (ex.: referenciada indiretamente) ou “legado”.
* Desenhar o modelo canônico para cada entidade que ficará: para t\_cli, por exemplo, mantenha id, cdcli, cdemp, campos de contato essenciais e troque datacadcli, dataultalt, dtposvenda0\* para DateTime; transforme flags como ativocli/mensalsn em Boolean; exponha Decimal para valores monetários; renomeie colunas legadas com @@map quando precisar preservar o nome físico. Documente tudo em um dicionário (pasta docs/ ou Notion).
* Reforçar integridade: declare relações (@relation) sempre que houver chave estrangeira na base (ex.: t\_cli.cdemp -> t\_emp.cdemp, t\_usere.cdusua -> t\_users.id). Inclua @@unique para chaves naturais (cdcli), índices para filtros (@@index([cdemp, ativocli])) e políticas de cascata (onUpdate: Cascade).
* Consolidar tipos: padronize UUIDs com @db.Uuid, troque colunas numéricas armazenadas em texto por Int/Decimal, remova campos duplicados (recebernewsn/recebernewssn). Para colunas obrigatórias, adicione @default(...)/@updatedAt.
* Implementar em fases: crie uma cópia prisma/erp.legacy.prisma para comparação, então refatore erp.prisma entidade por entidade. Cada rodada gera migração (npx prisma migrate dev --schema=prisma/erp.prisma --name normalize\_t\_cli), atualiza o Client e sincroniza tenants (npm run migrate:tenants). Execute npm run test:e2e (ou testes focados) após cada lote.
* Migração de dados: para colunas convertidas (ex.: String → DateTime), escreva SQL no próprio migration step fazendo ALTER TABLE ... USING to\_timestamp. Para booleans, normalize valores ('S'/'N') antes de trocar o tipo.
* Limpeza: somente após validar com stakeholders elimine tabelas marcadas como “legado”. Se ainda houver dúvidas, mantenha-as em schema separado ou gere views para compatibilidade.

**Replicação no schema.prisma (main)**

* Harmonize as decisões tomadas no ERP com o banco principal: se os tenants deixam de usar IDs numéricos, considere migrar Tenant.id para UUID e adicione colunas que hoje só existem no ERP (ex.: db\_host, db\_schema) para manter rastreabilidade centralizada.
* Replique convenções de nomes/tipos (timestamps com DateTime, flags booleanas, índices nas colunas de busca slug/db\_name). Crie migração correspondente (npx prisma migrate dev --schema=prisma/schema.prisma --name normalize\_tenant).
* Regerar o client (npx prisma generate --schema=prisma/schema.prisma) e atualizar serviços que consomem o main DB, garantindo que DTOs e validações reflitam os novos formatos.

**Próximos Passos**

1. Inventariar tabelas/colunas usadas e aprovar o escopo mínimo com o time funcional.
2. Prototipar a versão normalizada de t\_cli/t\_emp num branch, criar migração e validar em um tenant de teste.
3. Ajustar DTOs/serviços (ex.: novos Date/boolean) e atualizar testes automáticos antes de propagar para todos os bancos.