Teste Técnico - Integração com Sistema Legado

📖 Contexto

Uma empresa possui um **sistema legado** que armazena dados de usuários. Este sistema foi desenvolvido há anos e apresenta diversas instabilidades. A equipe técnica decidiu criar um **novo serviço** que será responsável por:

1. Sincronizar os dados do sistema legado;
2. Manter uma base de dados própria, confiável e otimizada;
3. Disponibilizar endpoints modernos para consulta e manipulação dos dados;

Sua missão é desenvolver este novo serviço.

🎯 Objetivo do Teste

Avaliar suas habilidades em:

Competência	O que será observado
Resolução de Problemas	Como você lida com cenários de erro, dados inconsistentes e sistemas instáveis
Arquitetura de Software	Aplicação de DDD, separação de responsabilidades e organização do código
Design Patterns	Uso adequado de padrões como Repository, Service Layer, Factory, etc.
Boas Práticas	Clean Code, tratamento de erros, logs, validações e documentação
Resiliência	Implementação de retry, circuit breaker, fallbacks e idempotência

---

🔧 O que você precisa fazer

1. Configurar e Executar o Sistema Legado

O sistema legado está na pasta `api/`. Para executá-lo:

```
cd api
cp env.example .env          # Configurar variáveis de ambiente
docker build -t api-legada .
docker run -m 128m --network=host api-legada
```

A API estará disponível em `http://localhost:3001`.

2. Criar o Novo Serviço

Desenvolva um novo serviço web em **Node.js + Express** (ou framework de sua preferência em Node.js) que irá:

- Conectar-se ao sistema legado para sincronizar dados;
- Manter seu próprio banco de dados (sugestão: SQLite);
- Expor endpoints REST para consulta e manipulação de usuários;

📡 Sistema Legado - Detalhes Técnicos

Endpoint Disponível

Método	Rota	Descrição
`GET`	`/external/users`	Retorna todos os usuários via streaming

Autenticação

Todas as requisições devem incluir o header:

```
x-api-key: YOUR_API_KEY
```

> ⚠️ Importante: Crie o arquivo .env na pasta api/ a partir do template fornecido:
>
> ```
> cd api
> cp env.example .env
> ```
>
> O arquivo `env.example` já contém uma API_KEY configurada para testes.

Formato dos Dados Retornados

Os dados são enviados via **streaming** em lotes de 100 registros. Cada lote é um array JSON:

```
[
  {
    "id": 1,
    "userName": "john_doe",
    "email": "john@example.com",
    "createdAt": "2024-01-15T10:30:00.000Z",
    "deleted": false
  },
  {
    "id": 2,
    "userName": "jane_doe",
    "email": "jane@example.com",
    "createdAt": "2024-01-16T14:20:00.000Z",
    "deleted": true
  }
]
```

> Nota: Os lotes são enviados concatenados no stream. Você precisará tratar o parsing adequadamente, pois o response completo não é um JSON válido por padrão (são múltiplos arrays concatenados).

⚠️ Comportamentos Instáveis (Simulados)

O sistema legado **intencionalmente** apresenta os seguintes problemas:

Problema	Probabilidade	Descrição
Erro 500	20%	Internal Server Error no início da requisição
Erro 429	20%	Too Many Requests (rate limiting)
Dados Corrompidos	20%	JSON inválido inserido no meio do stream
Duplicatas	-	Mesmo `user_name` pode aparecer múltiplas vezes
Soft Delete	-	Usuários deletados (`deleted: true`) são retornados

📋 Requisitos do Novo Serviço

3.1 Sincronização com Sistema Legado

Implemente um mecanismo de sincronização que:

☐ Consuma o endpoint `/external/users` do sistema legado;

☐ Seja **idempotente** (executar múltiplas vezes não deve causar inconsistências);

☐ Trate adequadamente os erros e instabilidades do sistema legado;

☐ Registre logs/histórico das execuções (sucesso, falha, quantidade de registros, etc.);

☐ Realize **deduplicação** por `user_name`:

- Se existirem duplicatas, manter o registro com `created_at` mais recente;

3.2 Endpoints Obrigatórios

Sincronização

Método	Rota	Descrição
`POST`	`/sync`	Dispara a sincronização com o sistema legado

Consulta

Método	Rota	Descrição
`GET`	`/users`	Lista todos os usuários (com paginação)
`GET`	`/users/:user_name`	Busca usuário pelo `user_name`

CRUD

Método	Rota	Descrição
`POST`	`/users`	Cadastra novo usuário
`PUT`	`/users/:id`	Atualiza usuário existente
`DELETE`	`/users/:id`	Remove usuário (soft-delete)

Exportação

Método	Rota	Descrição
`GET`	`/users/export/csv`	Exporta usuários em formato CSV

> Filtros sugeridos para exportação: data de criação (created_from, created_to)

3.3 Regras de Negócio

1. **Soft Delete:** Todos os endpoints devem retornar **somente usuários não deletados** (`deleted = false`);
2. **Isolamento:** As operações CRUD do novo serviço **não impactam** o sistema legado;
3. **Unicidade:** O campo `user_name` deve ser único no novo serviço;

3.4 Arquitetura

- Utilize **DDD (Domain-Driven Design)** como base da estrutura do projeto;
- Organize o código em camadas claras (Domain, Application, Infrastructure, Presentation);

3.5 Docker

- O serviço deve rodar em container Docker com **recursos limitados** (similar a `m 128m`);
- Forneça um `Dockerfile` e instruções de execução;

📝 Documentação AWS (Obrigatório)

Crie um documento (`AWS_ARCHITECTURE.md` ou similar) descrevendo:

1. **Arquitetura proposta** para executar a sincronização em ambiente AWS;
2. **Serviços utilizados** e justificativa de cada escolha;
3. **Diagrama** da solução (pode ser em texto/ASCII ou imagem);

Considere os seguintes serviços:

- **Compute:** Lambda, ECS, EC2
- **Orquestração:** EventBridge, Step Functions, SQS
- **Banco de Dados:** RDS, DynamoDB, Aurora

- **Armazenamento:** S3
- **Monitoramento:** CloudWatch, X-Ray

Perguntas para guiar sua documentação:

- Como garantir que a sincronização seja executada periodicamente?
- Como tratar falhas e garantir retry automático?
- Como escalar a solução para grandes volumes de dados?
- Qual estratégia de banco de dados para alta disponibilidade?

⭐ Requisitos Bônus (Diferenciais)

Otimização do Sistema Legado

Se você identificar oportunidades de melhoria no código da pasta `api/`:

- [ ] Documente os problemas identificados;
- [ ] Implemente as otimizações;
- [ ] Realize testes comparativos (antes vs depois);
- [ ] Documente os resultados em um arquivo `OPTIMIZATIONS.md`;

Outros Diferenciais

- [ ] Testes unitários e/ou de integração;
- [ ] Documentação da API (Swagger/OpenAPI);
- [ ] Implementação de rate limiting no novo serviço;
- [ ] Health check endpoint;
- [ ] Métricas e observabilidade;

📦 Entregáveis

1. **Código-fonte** do novo serviço em repositório Git;
2. **README.md** com instruções de instalação e execução;
3. **Dockerfile** funcional;
4. **Documentação AWS** (`AWS_ARCHITECTURE.md`);

5. *(Opcional)* `OPTIMIZATIONS.md` com melhorias do sistema legado;

✅ Checklist de Desenvolvimento

Essencial

- [ ] Novo projeto Node.js configurado;
- [ ] Banco de dados SQLite configurado;
- [ ] Endpoint de sincronização (idempotente);
- [ ] Tratamento de erros do sistema legado (retry, fallback);
- [ ] Deduplicação por `user_name` (usar `created_at` mais recente);
- [ ] CRUD completo de usuários;
- [ ] Endpoint de busca por `user_name`;
- [ ] Exportação CSV com filtros;
- [ ] Todos endpoints retornam apenas usuários não deletados;
- [ ] Estrutura DDD;
- [ ] Docker funcional com limite de memória;
- [ ] Documentação AWS;

Diferenciais

- [ ] Testes automatizados;
- [ ] Documentação Swagger;
- [ ] Otimização do sistema legado;
- [ ] Health check e métricas;

🔗 Referências

- **Sistema Legado:** Consulte `api/README.md` para detalhes técnicos completos;
- **Express.js:** https://expressjs.com/
- **SQLite (better-sqlite3):** https://github.com/WiseLibs/better-sqlite3
- **DDD:** https://martinfowler.com/bliki/DomainDrivenDesign.html

---

❓ Dúvidas

Se tiver dúvidas sobre os requisitos, documente suas decisões e premissas no README do seu projeto. Não há problema em fazer suposições razoáveis, desde que estejam documentadas.

Boa sorte! 🚀

teste_tecnico.zip