Guia de Implementação — Sistema de Transferências (Laravel 10 + Sanctum)

Este guia descreve, passo a passo, como implementar no repositório atual os requisitos:

- Cadastro com Nome Completo, CPF, E-mail e Senha (CPF e E-mail únicos)
- Dois papéis: Usuário (envia e recebe) e Lojista (apenas recebe)
- Transferências entre usuários e para lojistas
- Validação de saldo, transação atômica com rollback
- Autorização externa (GET `https://util.devi.tools/api/v2/authorize`)
- Notificação externa (POST `https://util.devi.tools/api/v1/notify`) com tolerância a falhas

---

1. Pré-requisitos

- PHP 8.1+
- Composer
- MySQL 8+
- Node.js 18+ (apenas para ferramentas auxiliares do projeto)

2. Configuração do ambiente

1. Copie o arquivo `.env` e configure MySQL:

   ```
   cp .env.example .env
   ```

   Edite as variáveis:

   ```
   DB_CONNECTION=mysql
   DB_HOST=127.0.0.1
   DB_PORT=3306
   DB_DATABASE=suabase
   DB_USERNAME=seuusuario
   DB_PASSWORD=suasenha
   ```

2. Instale dependências e gere a chave:

   ```
   composer install
   php artisan key:generate
   ```

3. Modelo de dados e migrations

O repositório já contém:

- `users`: `fullName`, `email` (único), `password`, `cpf` (único)
- `wallets`: `user_id`, `balance`
- `transactions`: `wallet_id`, `type` (`deposit` | `transfer`), `amount`, `receiver_wallet_id`

3.1. Adicionar papel do usuário (`role`)

Crie uma migration para adicionar a coluna `role` (valores: `user`, `merchant`):

```
php artisan make:migration add_role_to_users_table --table=users
```

Edite a migration gerada para conter:

```
Schema::table('users', function (Blueprint $table) {
    $table->enum('role', ['user', 'merchant'])->default('user')->after('cpf');
});
```

E no `down()` remova a coluna `role`.

3.2. Índices recomendados

- `wallets.user_id` index (já criado por `foreignId`)
- `transactions.wallet_id` e `transactions.receiver_wallet_id` índices (opcional para performance)

3.3. Rodar migrations

```
php artisan migrate
```

4. Models e relacionamentos

Atualize/garanta os relacionamentos (em inglês, padrão do projeto):

- `App\Models\User`:
  - `hasOne(Wallet::class)`
- `App\Models\Wallet`:
  - `belongsTo(User::class)`
  - `hasMany(Transaction::class)`
- `App\Models\Transaction` (já possui `wallet()` e `receiverWallet()`)

Exemplo (resumo):

```
// app/Models/User.php
public function wallet()
{
    return $this->hasOne(\App\Models\Wallet::class);
}

// app/Models/Wallet.php
public function user()
{
    return $this->belongsTo(\App\Models\User::class);
}
public function transactions()
{
```

```
    return $this->hasMany(\App\Models\Transaction::class);
}
```

5. Autenticação (Sanctum)

- O projeto já utiliza Sanctum. Garanta que o login retorna um token e que rotas sensíveis usem `auth:sanctum`.
- Evite logar credenciais; trate usuário inexistente com resposta 401 genérica.

6. Criação automática de carteira

Ao criar um usuário, crie a carteira com saldo `0.00`. Opções:

- Via `Observer` de `User` (evento `created`)
- Via `Listener` de `Registered`
- Ou diretamente no `UserService` após `create`

Exemplo simples (no serviço após criar user):

```
$wallet = \App\Models\Wallet::create(['user_id' => $user->id, 'balance' => 0]);
```

7. Regras de negócio de transferência

- Apenas usuários com `role = user` podem ENVIAR.
- Lojistas (`role = merchant`) apenas RECEBEM.
- Validar saldo do remetente.
- Operação deve ser transacional com rollback em qualquer falha.
- Antes de debitar/creditar, consultar serviço autorizador externo (GET) com timeout 3s e até 2 tentativas.
- Após efetivar transferência, enviar notificação (POST). Se falhar, não reverter a transferência; registrar e agendar retry assíncrono.

8. Implementação do serviço de transferência

Crie/implemente `App\Services\TransferService` com:

1. Busca das carteiras de remetente e destinatário com lock pessimista:

```
$sender = Wallet::where('user_id', $user->id)->lockForUpdate()->firstOrFail();
$receiver = Wallet::where('id', $receiverWalletId)->lockForUpdate()->firstOrFail();
```

2. Regras de papel: bloquear envio se `$user->role === 'merchant'`.
3. Validação de saldo: `$sender->balance >= $amount`.
4. Autorização externa (GET `https://util.devi.tools/api/v2/authorize`):
   - Timeout 3s; 2 retries com backoff.
   - Se resposta não autorizar, lançar exceção de domínio (422).
5. Transação de banco (DB::transaction):
   - Debitar remetente, creditar destinatário
   - Criar `Transaction` do tipo `transfer`

6. Notificação externa (POST `https://util.devi.tools/api/v1/notify`):
    - Executar fora da transação
    - Em caso de falha, despachar Job para retry

Pseudocódigo resumido:

```
DB::transaction(function () use ($user, $receiverWalletId, $amount) {
    // lock carteiras
    // validações de papel e saldo
    // chama autorizador externo (GET)
    // debitar/creditar e salvar
    // registrar Transaction
});
// enviar notificação (POST); se falhar, agendar retry
```

9. Integrações externas

9.1. Autorizador (GET)

- Endpoint: `https://util.devi.tools/api/v2/authorize`
- Timeout: 3s
- Retries: 2 (total de 3 tentativas)
- Em caso de falha de rede/timeout após todas tentativas, retornar 503; se resposta negar, retornar 422

9.2. Notificação (POST)

- Endpoint: `https://util.devi.tools/api/v1/notify`
- Tolerância a falhas: se indisponível, manter transferência e agendar retry (fila)
- Configurar `QUEUE_CONNECTION=database` (ou outro provedor) e rodar `php artisan queue:work`

10. Rotas e controllers

- Adicionar rota protegida para transferência:

```
// routes/api.php
Route::middleware('auth:sanctum')->group(function () {
    Route::post('/transfers', [\App\Http\Controllers\TransferController::class,
'transfer']);
});
```

- `TransferController@transfer` deve:
    - Validar com `TransferRequest`
    - Chamar o `TransferService`
    - Responder `201` com os dados da transação ao sucesso; 4xx/5xx adequados ao erro

11. Validação

`App\Http\Requests\TransferRequest` já valida `receiver_wallet_id` e `amount`. Adicione validação de `amount > 0` (já presente `min:0.01`).

Para cadastro de usuário, crie `UserRequest` para validar:

- `fullName` required
- `cpf` required|unique:users,cpf|formato (regex)
- `email` required|email|unique:users,email
- `password` required|min:8
- `role` in:user,merchant (se permitir informar no cadastro)

12. Testes (sugestões)

- Unit: `TransferService` (saldo insuficiente, merchant tentando enviar, autorizador nega, notificação falha)
- Feature: fluxo completo de transferência autenticada
- Concorrência: tentativas simultâneas de débito na mesma carteira (lock deve evitar saldo negativo)

13. Observabilidade e erros comuns

- Logar falhas do autorizador/notify com correlação da transação
- Metricas: quantidade de transferências, latência do autorizador, taxa de erro de notificação
- Erros comuns:
  - Ausência de transação DB: pode gerar inconsistência
  - Falta de `lockForUpdate`: condições de corrida e saldo negativo
  - Reverter transferência por falha de notificação (não deve reverter)

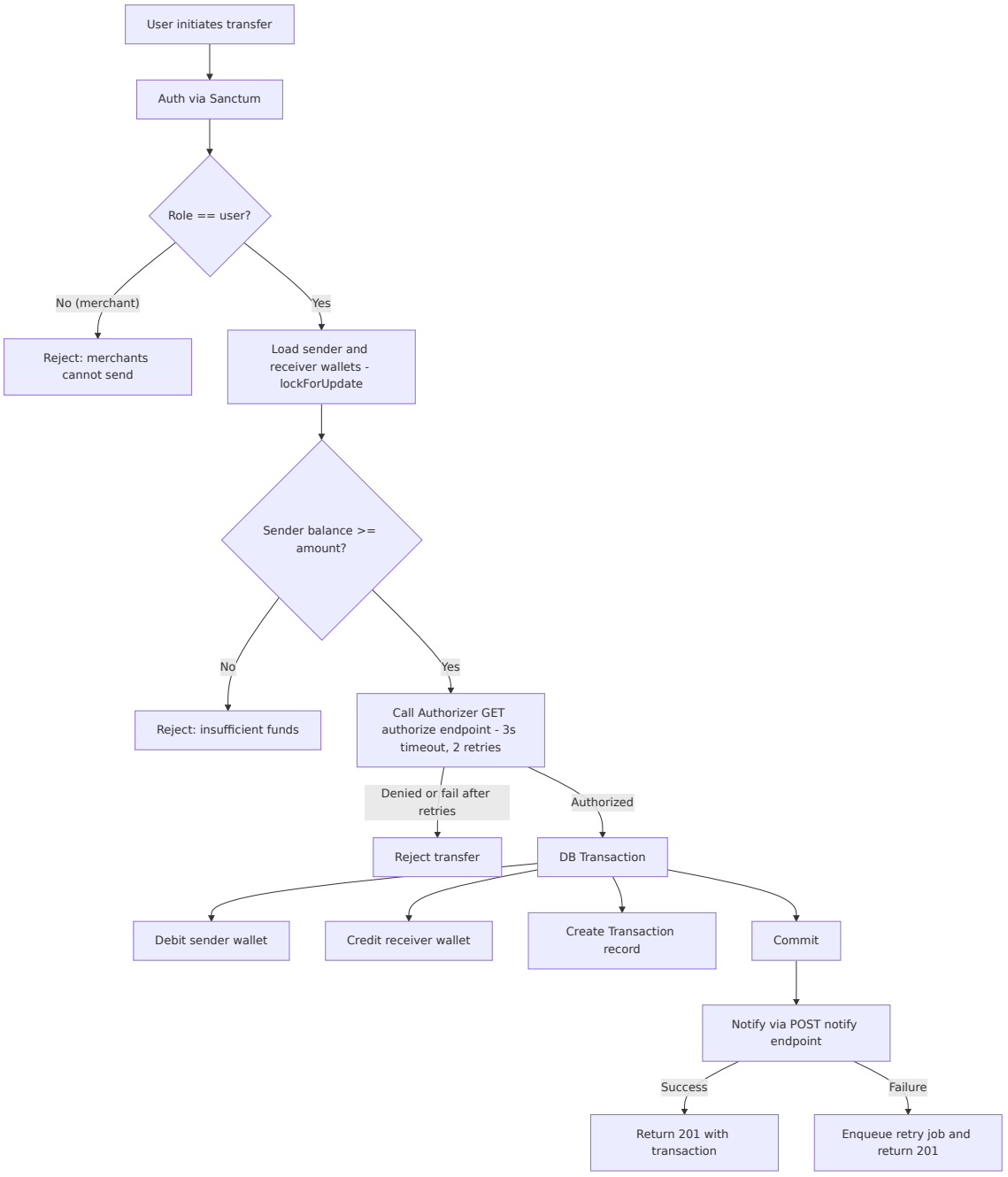
14. Endpoints (exemplos)

- Login: `POST /api/login` → `{ email, password }`
- Criar usuário: `POST /api/users` → `{ fullName, cpf, email, password, role }`
- Transferir: `POST /api/transfers` (com Bearer Token)

```
{
  "receiver_wallet_id": 2,
  "amount": 100.00
}
```

15. Diagrama do fluxo

O diagrama abaixo ilustra o fluxo da transferência (arquivo `docs/diagram.svg` gerado a partir de `docs/diagram.mmd`):



---

16. Como gerar os artefatos (opcional para documentação)

Para gerar o diagrama e este PDF localmente com Node.js:

```
# Gerar SVG a partir do Mermaid
npx @mermaid-js/mermaid-cli -i docs/diagram.mmd -o docs/diagram.svg

# Gerar PDF a partir do Markdown
npx md-to-pdf docs/guia-implementacao.md --output docs/guia-implementacao.pdf
```

Pronto! Com essas etapas, o sistema atenderá aos requisitos com segurança, consistência e tolerância a falhas.