ADR 001: Estratégia de Arquitetura (Modular Monolith)

- **Status:** Aceito
- **Contexto:** A complexidade operacional de microsserviços (latência de rede, *distributed tracing*, falhas parciais) é desproporcional ao estágio atual do NexusPay. Precisamos de velocidade de entrega e consistência forte (ACID), mas mantendo a possibilidade de escala futura.
- **Decisão:** Adotaremos o padrão **Modular Monolith** utilizando **Spring Modulith**. O sistema será um único deploy, mas com separação física de pacotes por contexto (Auth, Ledger, Notification). Comunicações entre módulos devem ser feitas via interfaces de domínio ou eventos internos.
- **Consequências:**
    - **Positivas:** Facilidade de refatoração, transações em nível de banco de dados (atomicidade garantida) e ambiente de teste simplificado.
    - **Negativas:** Escala vertical única (não podemos escalar apenas o Ledger sem escalar o Auth), exigindo monitoramento rigoroso para evitar acoplamento indevido entre pacotes.

ADR 002: Estratégia de Concorrência (Optimistic Locking)

- **Status:** Aceito
- **Contexto:** O Ledger sofrerá alta concorrência em contas populares (ex: conta de recebimento de uma grande loja). Precisamos garantir que o saldo nunca fique inconsistente sem sacrificar o *throughput* global do banco de dados.
- **Decisão:** Utilizaremos **Optimistic Locking** via controle de versão (@Version no JPA). Em caso de conflito, a aplicação capturará a exceção e aplicará uma estratégia de **Retry com Exponential Backoff + Jitter**.
- **Consequências:**
    - **Positivas:** Maior disponibilidade do banco (não há retenção de locks de linha por tempo indeterminado) e melhor performance para a maioria das transações.
    - **Negativas:** Em cenários de altíssima contenção na mesma conta, a taxa de retentativas pode aumentar a latência percebida pelo usuário final.

ADR 003: Representação de Valores Monetários

- **Status:** Aceito
- **Contexto:** O uso de tipos de ponto flutuante (float, double) em sistemas financeiros causa erros de arredondamento cumulativos que podem gerar prejuízos reais.
- **Decisão:** Todos os valores monetários serão tratados como **Inteiros (Long)** na menor unidade da moeda (centavos). A fórmula para saldo será:
  $$Balance = \sum Credits - \sum Debits$$
  No Java, utilizaremos a classe BigDecimal para cálculos complexos e Long para persistência e transporte.
- **Consequências:**
    - **Positivas:** Precisão aritmética absoluta e facilidade de comparação de valores.
    - **Negativas:** Exige camadas de conversão rigorosas nos DTOs de entrada e saída para garantir que o usuário veja **R$ 10,00** e o banco processe **1000**. Qualquer erro na divisão/multiplicação por 100 pode gerar erros de escala massivos.