Título: Documento de Arquitetura – Sistema de Rifas Digitais
Autor: Marcos Morais
Data: 05/11/2025

1. Problema

- Rifas tradicionais são geridas manualmente, sem controle digital eficiente.
- Falta transparência e confiabilidade na apuração dos vencedores.
- Não há integração direta com resultados oficiais da Loteria Federal da Caixa.

---

2. Objetivo

Construir um sistema de rifas digital que permita:

- Cadastro de itens rifados (carro, moto, terreno etc).
- Geração e venda de bilhetes numerados.
- Configuração para usar ou não a API da Loteria Federal.
- Apuração automática (via API) ou manual do sorteio.
- Relatórios e auditoria para transparência pública.

---

3. Hipótese

- Se o sistema oferecer flexibilidade (API ou inserção manual), então:
    - Organizadores terão autonomia mesmo sem integração oficial.
    - Participantes confiarão mais no processo.
    - O sistema poderá escalar para diferentes contextos (rifas locais a nacionais).

---

4. Solução — Visão de Alto Nível

Módulos principais:

- **Módulo de Cadastro de Rifas:** registrar item, quantidade de bilhetes, data do sorteio, regras.
- **Módulo de Bilhetes:** geração automática de bilhetes numerados e controle de estados (disponível / reservado / vendido).
- **Módulo de Participantes:** cadastro de compradores, verificação básica (CPF) e histórico de bilhetes.
- **Módulo de Pagamentos/Transações:** integração com gateways (Pix, cartão), registro e conciliação.
- **Módulo de Sorteio / Apuração:** integração com API da Loteria Federal (quando habilitado) ou entrada manual do número sorteado.
- **Módulo de Relatórios / Auditoria:** histórico completo de rifas, transações, bilhetes e apurações com assinatura temporal.
  Frontend (Web / Mobile): interfaces separadas para Organizadores (Admin) e Participantes (compra e acompanhamento).
- **Infra / Observabilidade:** métricas, logs estruturados, rastreabilidade e backups

---

5. Casos de Uso (UML simplificado)

Atores: Administrador (Organizador), Participante, Sistema, Fonte Externa (API Loteria Federal)

Casos de Uso Principais:

- **UC01 – Cadastrar Rifa** — Administrador: cria rifa com item, quantidade de bilhetes, preço e data.
- **UC02 – Gerar Bilhetes** — Sistema: gera bilhetes numerados automaticamente.
- **UC03 – Comprar Bilhete** — Participante: seleciona bilhete, paga, recebe confirmação.
- **UC04 – Configurar Origem do Sorteio** — Administrador: define API da Loteria ou manual.
- **UC05 – Executar Sorteio** — Sistema: consulta API ou recebe manualmente o número sorteado.
- **UC06 – Identificar Bilhete Vencedor** — Sistema: cruza número sorteado com bilhetes emitidos.
- **UC07 – Emitir Relatório de Transparência** — Sistema: publica relatório com origem do sorteio e prova de apuração.

---

6. Modelo de Dados (resumo)

Entidades principais e atributos (resumo):

- **Rifa:** id, título, descrição, item, preco_bilhete, quantidade_bilhetes, data_sorteio, concurso, usar_api_loteria (bool), status.
- **Bilhete:** id, numero (único por rifa), estado (disponível/reservado/vendido), rifa_id, participante_id (opcional).
- **Participante:** id, nome, cpf, email, telefone, endereco (opcional).
- **Transacao:** id, rifa_id, participante_id, valor_total, status (pendente/pago/cancelado), relacionamento com bilhetes (N..N).
- **Apuracao:** id, rifa_id, fonte (API/Manual), concurso, numero_sorteado, bilhete_vencedor_id, participante_vencedor_id, data_registro.

Relacionamentos:

- **Rifa 1..N Bilhetes**
- **Participante 1..N Bilhetes (via compra)**
- **Participante 1..N Transações**
- **Rifa 1..1 Apuração** (uma apuração vinculada à rifa)

---

7. Quadro de Tecnologias Possíveis (decisão adotada: Django + PostgreSQL + Next.js)

Tecnologia	Uso no Sistema	Pontos Positivos	Pontos Negativos
Django (Python)	Backend/API + Painel Administrativo	- Framework completo (ORM, autenticação, admin pronto) - Produtividade alta - Boa integração com Django REST Framework	- Performance inferior em cenários de alta concorrência - Menos flexível para arquiteturas altamente customizadas
Node.js (JavaScript)	Backend/API REST	- Alta performance em operações I/O - Ecossistema rico (npm) - Comunidade ativa	- Menos adequado para operações CPU-intensivas - Requer boas práticas para evitar problemas de concorrência

.NET Core (C#)	Backend/API REST	- Performance robusta - Suporte corporativo da Microsoft - Ferramentas maduras de segurança e logging	- Curva de aprendizado maior para quem não vem do mundo Microsoft - Menos popular em startups comparado a Node.js
Java Spring Boot	Backend/API REST	- Framework consolidado e estável - Excelente para sistemas complexos e corporativos - Grande comunidade	- Verbosidade do Java - Consumo de recursos maior que alternativas mais leves
Next.js (React)	Frontend Web	- Renderização híbrida (SSR/SSG) - Excelente para SEO - Integração nativa com React - Ótimo para aplicações escaláveis	- Complexidade maior que React puro - Requer conhecimento de SSR/SSG para aproveitar bem
React	Frontend Web	- Biblioteca flexível e popular - Grande ecossistema de componentes - Comunidade ativa	- Necessidade de configurar várias ferramentas adicionais (roteamento, estado) - Curva de aprendizado para iniciantes
Flutter	Frontend Mobile	- Código único para iOS e Android - Alta performance (compilado nativo) - UI moderna e rica	- Aplicativos podem ficar pesados - Comunidade menor que React Native
PostgreSQL	Banco de Dados Relacional	- Suporte avançado a queries complexas - Estabilidade e confiabilidade - Open source	- Configuração inicial pode ser mais complexa - Menos flexível para dados não estruturados
MongoDB	Banco de Dados NoSQL	- Flexível para dados não estruturados - Escalabilidade horizontal fácil - Modelo de documentos intuitivo	- Menos adequado para transações complexas - Consistência eventual em clusters distribuídos

---

8. Considerações Arquiteturais

- **Flexibilidade:** operar com ou sem API da Loteria; origem do número visível no relatório.
- **Transparência:** relatórios com carimbo temporal e origem da apuração.
- **Escalabilidade:** particionar rifas de alto volume, usar caches e filas para controle de concorrência.
- **Segurança / LGPD:** proteção de dados pessoais (CPF, contato), criptografia em repouso/transporte, consentimento explícito.
- **Concorrência em vendas:** usar lock otimista/pessimista (Redis + transações DB) para evitar dupla venda de bilhetes.
- **Auditoria:** registrar eventos imutáveis (apuração, mudança de status) com timestamps.

9. Próximos passos sugeridos (opções)

1. Diagrama de Componentes UML (Django API + Next.js + PostgreSQL + Redis + Gateway + API Loteria).
2. Diagrama ER detalhado + scripts de migração iniciais (Django migrations).
3. Casos de uso detalhados (fluxos alternativos, exceções).
4. Documento de Requisitos Não-Funcionais (SLA, RTO, backups, compliance LGPD).