Documentação Técnica - Sistema de Cafeteria CLI

---

1. Visão Geral do Sistema
O Sistema de Cafeteria CLI é uma aplicação de linha de comando desenvolvida em Node.js para gerenciar operações de uma cafeteria, incluindo:

- 📝 Vendas e controle de estoque
- 👤 Gestão de clientes e usuários
- 📦 Cadastro e controle de produtos
- 📊 Relatórios e gestão de caixa
- ⚠️ Alertas de estoque baixo e validade de produtos

O sistema foi projetado para ser simples, robusto e fácil de manter, com interface de texto intuitiva para uso em ambientes de produção reais.

---

2. Arquitetura e Estrutura de Pastas

```
cafeteria-cli/
├── app.js              # Arquivo principal - ponto de entrada
├── db.js               # Conexão com o banco de dados MySQL
├── estrutura.sql       # Script SQL para criar a estrutura do banco
├── package.json        # Dependências e scripts do projeto
├── utils/
│   └── caixa.js        # Utilitários para gestão de caixa
└── comandos/
    ├── auth.js         # Sistema de autenticação e gestão de usuários
    ├── venda.js        # Gestão de vendas
    ├── cliente.js      # Gestão de clientes
    ├── produto.js      # Gestão de produtos
    ├── estoque.js      # Controle de estoque
    ├── relatorio.js    # Relatórios e gestão de caixa
    ├── log.js          # Registro de logs do sistema
    └── pedido.js       # Gestão detalhada de pedidos
```

Padrões de Arquitetura Aplicados:
MVC Simplificado: Separação clara entre interface (CLI), lógica de negócio (comandos) e dados (banco)
Modularidade: Cada funcionalidade em seu próprio arquivo/módulo
Tratamento de Erros: Try-catch em todas as operações críticas
Validação de Dados: Checagem rigorosa antes de operações no banco

---

3. Detalhamento dos Arquivos Principais

➔ 3.1 ***app.js*** - Arquivo Principal
Responsabilidade: Controlador central do sistema, gerenciando autenticação e navegação entre módulos.

Principais Funções:

- ***menuAutenticacao()***: Interface de login do sistema
- ***menuPrincipal(usuariold)***: Menu principal após login, com opções baseadas nas permissões do usuário
- ***exibirMenuCliente(cliente)***: Exibe o menu principal com as opções de navegação
- ***menuSair()***: Finaliza o sistema de forma segura

Características Importantes:

- Usa readline-sync para interação síncrona com o usuário
- Mantém estado de sessão através do usuarioId
- Trata erros de autenticação e permissões
- Interface responsiva com formatação visual clara

➔ 3.2 ***db.js*** - Conexão com Banco de Dados

Responsabilidade: Gerenciamento da conexão com o banco de dados MySQL.

Configuração:
```
const mysql = require('mysql2/promise');
const pool = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'cafeteria',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});
```

Principais Características:

- Configuração otimizada para XAMPP (senha vazia padrão)
- Usa pool de conexões para melhor performance
- Suporte a queries assíncronas com promises
- Tratamento de erros de conexão
- Compatibilidade com MySQL 8.0+

➔ 3.3 ***utils/caixa.js*** - Utilitários de Caixa

Responsabilidade: Funções auxiliares para gestão de caixas do sistema.

Principais Funções:

- ***getAberto(usuariold)***: Verifica se o usuário tem um caixa aberto
- ***getAbertoQualquer()***: Retorna qualquer caixa aberto no sistema
- ***verificarCaixa(usuariold)***: Válida se há caixa aberto antes de operações
- ***Importância***: Garante a integridade das operações financeiras, impedindo vendas sem caixa aberta.

➔ 3.4 ***comandos/venda.js*** - Gestão de Vendas

Responsabilidade: Processo completo de venda, desde seleção de produtos até geração de comprovante.

Fluxo de Operação:

1. Verifica se há caixa aberto
2. Seleciona cliente (cadastrado ou avulso)
3. Lista produtos disponíveis em estoque
4. Permite seleção de múltiplos itens
5. Calcular total e aplica desconto (máximo 10%)

6. Confirma venda e atualiza estoque
7. Gera comprovante não fiscal

Principais Correções Aplicadas:

- Correção de colunas inexistentes: Remoção de usuario_id e motivo nas queries de estoque
- Ajuste de nomes de colunas: Padronização para preço em vez de preco_venda
- Remoção de condições inválidas: Eliminação de ativo = true onde não existia a coluna
- Validação de estoque: Checagem antes de adicionar itens à venda

➔ 3.5 ***comandos/cliente.js*** - Gestão de Clientes

Responsabilidade: Cadastro, listagem e edição de clientes no sistema.

Funcionalidades:

- ***listar():*** Exibe todos os clientes cadastrados
- ***cadastrar()***: Cria novo cliente com validação de CPF único
- ***editar()***: Permite atualização de dados do cliente
- ***buscarPorCPF()***: Busca cliente por CPF com formatação automática
- ***excluir(usuariold)***: Remove cliente com verificação de dependências (vendas associadas) e opção de inativação

Adaptações para Banco Real:

- Estrutura simplificada: Apenas id, nome e cpf (conforme estrutura real do banco)
- Validação de CPF: Verificação de duplicidade antes de cadastrar
- Formatação de CPF: Exibição amigável com máscara 000.000.000-00
- Segurança em exclusões: Verificação de vendas históricas antes da exclusão

➔ 3.6 ***comandos/produto.js*** - Gestão de Produtos

Responsabilidade: Cadastro, edição e controle de categorias de produtos.

Funcionalidades Principais:

- ***listar(): Exibe produtos com detalhes de categoria e estoque***
- ***cadastrar(): Cria novo produto com seleção de categoria existente***
- ***editar(): Atualiza informações do produto***
- ***atualizarEstoque(): Ajustar manualmente o estoque de produtos***
- ***excluir(usuariold): Remove produto com verificação de estoque zero e histórico de vendas***
- ***excluirCategoria(usuariold): Remove categorias sem produtos associados***

Soluções de Qualidade:

- Prevenção de duplicação de categorias: Verificação antes de criar nova categoria
- Padronização de nomes: Formatação automática (primeira letra maiúscula)
- Seleção inteligente de categorias: Lista existentes ou permite criar nova
- Tratamento de estoque negativo: Validação antes de atualizar

➔ 3.7 ***comandos/estoque.js*** - Controle de Estoque

Responsabilidade: Movimentação de estoque, histórico e alertas.

Funcionalidades:

- ***listar()***: Exibe estoque atual de todos os produtos
- ***movimentar()***: Registra entrada/saída manual de produtos
- ***historico()***: Mostra movimentações recentes com detecção dinâmica de colunas
- ***alertaEstoqueBaixo()***: Identifica produtos com estoque crítico

Inovações Técnicas:

- Detecção dinâmica de estrutura do banco: Adapta-se a diferentes nomes de colunas
- Múltiplos fallbacks: Mantém funcionalidade mesmo com erros no banco
- Tratamento seguro de valores undefined: Nunca quebra por dados faltantes
- Geração de pedidos de reposição: Sugestão automática de quantidades

➔ 3.8 ***comandos/relatorio.js*** - Relatórios e Caixa

Responsabilidade: Gestão financeira e relatórios do sistema.

Funcionalidades:

- ***abrirCaixa()***: Inicializa novo caixa com valor inicial
- ***fecharCaixa()***: Finaliza caixa com relatório detalhado
- ***vendasDoDia()***: Lista todas as vendas do dia atual
- ***alertas()***: Agrega alertas de estoque e validade

Versão Estável Implementada:

- Menos dependências: Versão mínima funcional para evitar erros
- Mensagens claras: Indicação de funcionalidades em desenvolvimento
- Foco no essencial: Abertura de caixa 100% funcional
- Evolução gradual: Base para implementação futura de relatórios

➔ 3.9 ***comandos/log.js*** - Registro de Logs

Responsabilidade: Auditoria de todas as operações do sistema.

Funcionalidades:

- ***registrar(usuarioId, acao, detalhes)***: Registra ação no banco
- ***listar()***: Exibe histórico de operações (em desenvolvimento)

Importância para Produção:

- Rastreabilidade completa de todas as operações
- Identificação de usuários responsáveis por cada ação
- Suporte para auditorias e investigação de problemas

---

4. Mudanças Significativas e Suas Razões

➔ 4.1 Correção de Discrepâncias Banco de Dados ou Código

Problema: O código assumia uma estrutura de banco de dados que não existia na prática.

Mudanças Realizadas:

```javascript
// Antes (quebrava):
SELECT id, nome, preco_venda FROM produtos WHERE ativo = true

// Depois (funciona):
SELECT id, nome, preco FROM produtos WHERE quantidade_estoque > 0
```

Razão: O banco de dados real não tinha as colunas preco_venda, ativo, usuario_id e motivo em várias tabelas.

➔ 4.2 Padronização de Interface de Módulos

Problema: Alguns módulos usavam this.funcao() enquanto outros usavam funções independentes.

Mudança Realizada:

```javascript
// Antes (inconsistente):
else if (op === '2') await this.fecharCaixa(usuario.id);

// Depois (padronizado):
case '2':
  await fecharCaixa(usuarioId);
  break;
```

Razão: Criar um padrão consistente onde todos os módulos seguem a mesma estrutura:

1. Funções independentes para cada operação
2. Um único menu(usuarioId) como ponto de entrada
3. Exportação consistente com module.exports = { menu: menu }

➔ 4.3 Tratamento Robusto de Erros

Problema: O sistema quebrou completamente com um único erro de banco de dados.

Mudança Realizada:

```javascript
const [columns] = await db.execute('SHOW COLUMNS FROM produtos');
const hasValidade = columns.some(col =>
  col.Field.toLowerCase().includes('validade')
);
```

Razão: Garantir que o sistema permaneça utilizável mesmo em situações de erro, com mensagens claras para o usuário.

➔ 4.4 Detecção Dinâmica de Estrutura do Banco

Problema: O sistema falhava em ambientes com estruturas de banco diferentes.

Mudança Realizada:

```javascript
const [columns] = await db.execute('SHOW COLUMNS FROM produtos');
const hasValidade = columns.some(col =>
  col.Field.toLowerCase().includes('validade')
);
```

Razão: Tornar o sistema adaptável a diferentes versões do banco de dados, aumentando sua portabilidade.

➔ 4.5 Simplificação Progressiva para Estabilidade

Problema: Funcionalidades complexas causavam erros na produção.

Mudança Realizada:

```javascript
// Versão complexa (quebrava):
// Implementação completa de fecharCaixa

// Versão simplificada (funcional):
async function fecharCaixa(usuarioId) {
  console.log('\n⚠️ FUNCIONALIDADE EM DESENVOLVIMENTO');
  console.log('Para fechar o caixa, use o menu principal...');
}
```

Razão: Priorizar a estabilidade do sistema para operações essenciais, implementando funcionalidades complexas gradualmente.

---

5. Decisões de Design e Padrões

➔ 5.1 Interface de Texto (CLI) vs Interface Gráfica

Decisão: Optar por CLI em vez de interface gráfica web.

Razões:

- Simplicidade de implantação: Não requer servidor web ou configuração complexa
- Performance: Resposta imediata sem latência de rede
- Confiabilidade: Menos dependências e pontos de falha
- Ambiente de uso: Ideal para terminais de caixa em ambientes com internet instável

➔ 5.2 Banco de Dados Relacional (MySQL)

Decisão: Usar MySQL em vez de bancos NoSQL.

Razões:

- Integridade de dados: Transações ACID para operações financeiras
- Relacionamentos claros: Clientes, produtos, vendas têm relações bem definidas
- Familiaridade: Equipes de cafeteria geralmente conhecem SQL
- Reporting: Consultas complexas para relatórios são mais simples em SQL

➔ 5.3 Arquitetura Modular

Decisão: Separar funcionalidades em módulos independentes.

Razões:

Manutenibilidade: Problemas em um módulo não afetam outros
Equipe: Diferentes desenvolvedores podem trabalhar em módulos simultaneamente
Escalabilidade: Módulos podem ser substituídos ou melhorados individualmente
Testabilidade: Cada módulo pode ser testado isoladamente

➔ 5.4 Tratamiento de Errores Pragmático

Decisão: Focar em mensagens claras para o usuário final em vez de logs técnicos detalhados.

Razões:

- Público-alvo: Funcionários de cafeteria não são técnicos
- Resolução rápida: Mensagens diretas ajudam a resolver problemas sem suporte
- Experiência do usuário: Manter o sistema utilizável mesmo durante erros
- Priorização: Corrigir erros que impedem operações essenciais primeiro

---

6. Como Executar e Manter o Sistema

➔ 6.1 Requisitos de Sistema
- Node.js v14+
- MySQL 5.7+ ou MariaDB 10.3+
- Windows/Linux/macOS (testado em Windows 10/11)

➔ 6.2 Instalação

```bash
# Clonar repositório
git clone https://github.com/seuusuario/cafeteria-cli.git
cd cafeteria-cli

# Instalar dependências
npm install

# Configurar banco de dados
# 1. Criar banco 'cafeteria'
# 2. Importar estrutura do arquivo estrutura.sql
# 3. Configurar credenciais em db.js

# Executar sistema
node app.js
```

➔ 6.3 Estrutura do Banco de Dados

Tabelas essenciais do sistema:

- usuarios: id, nome, email, senha, cargo, ativo
- clientes: id, nome, cpf
- categorias: id, nome
- produtos: id, nome, preco, quantidade_estoque, categoria_id
- caixas: id, usuario_id, usuario_nome, valor_inicial, data_abertura, data_fechamento, valor_fechamento, status
- vendas: id, caixa_id, cliente_id, valor_total, desconto, data_criacao, status
- itens_venda: id, venda_id, produto_id, quantidade, preco_unitario
- movimentacao_estoque: id, produto_id, tipo, quantidade, data_movimentacao, motivo
- logs: id, usuario_id, acao, detalhes, data_hora

➔ 6.4 Manutenção e Atualizações

- Backup diário: Configure backup automático do banco de dados
- Atualização de dependências: npm update mensalmente
- Monitoramento de erros: Verifique logs regularmente
- Expansão de funcionalidades:
    1. Implementar fechamento de caixa completo
    2. Adicionar relatórios de vendas detalhados
    3. Implementar sistema de alertas de validade
    4. Integrar com impressora térmica para comprovantes

➔ 6.5 Solução de Problemas Comuns

Problema: Erro de conexão com o banco de dados
Solução: Verifique credenciais em db.js e se o serviço MySQL está rodando

Problema: "Module not found" para readline-sync
Solução: Execute npm install readline-sync

Problema: Funções de relatório não funcionando
Solução: Use a versão estável fornecida e aguarde atualizações futuras

Problema: Erros de coluna inexistentes
Solução: Adapte as queries conforme a estrutura real do seu banco, usando as técnicas de detecção dinâmica implementadas

---

7. Conclusão e Próximos Passos

➔ 7.1 Estado Atual do Sistema

O sistema está 100% funcional para operações essenciais:

✅ Login e autenticação de usuários
✅ Abertura de caixa
✅ Registro de vendas com controle de estoque
✅ Gestão básica de clientes e produtos
✅ Controle de estoque com alertas

➔ 7.2 Melhorias Planejadas

- Relatórios completos: Fechamento de caixa com conciliação financeira
- Integração com SEFAZ: Geração de NFC-e para vendas
- Interface mobile: App para pedidos via QR code
- Sincronização offline: Funcionamento sem internet com sincronização posterior
- Análise de dados: Insights de vendas e sugestões de estoque

➔ 7.3 Recomendações para Produção

- Treinamento de usuários: Realize sessões de treinamento com os funcionários
- Backup diário: Configure backup automático do banco de dados
- Atualizações graduais: Implementa novas funcionalidades uma por vez
- Suporte local: Designe um funcionário como responsável técnico local
- Monitoramento: Verifique logs regularmente para identificar padrões de uso

---

8. Apêndice: Diagramas de Fluxo
Fluxo de Venda

```
Início → Selecionar Cliente → Listar Produtos → Adicionar Itens →
Calcular Total → Aplicar Desconto → Confirmar Venda →
Atualizar Estoque → Gerar Comprovante → Fim
```

Fluxo de Estoque

```
Início → Listar Estoque Atual → Selecionar Ação →
[Entrada] → Selecionar Produto → Digitar Quantidade →
Atualizar Estoque → Registrar Movimentação → Fim
[Saída] → Selecionar Produto → Digitar Quantidade →
Verificar Estoque → Atualizar Estoque → Registrar Movimentação → Fim
```

Fluxo de Caixa

```
Início → Verificar Caixa Aberto →
[Nenhum aberto] → Abrir Novo Caixa → Registrar Valor Inicial → Fim
[Já aberto] → Exibir Opções →
[Fechar Caixa] → Calcular Total → Registrar Fechamento → Gerar Relatório → Fim
[Continuar] → Retornar ao Menu Principal → Fim
```

Documento preparado por: Equipe de Desenvolvimento para S.A
Última atualização: 5 de dezembro de 2025
Versão do sistema: v Beta 1.0.2 (Estável para Teste Locais)