MAPEAMENTO COMPLETO DO SISTEMA FINANZA

Desenvolvido por Kalleby Schultz - IFSUL Campus Venâncio Aires

2025

Contents

Chapter 1

📱 MAPEAMENTO COMPLETO DO SISTEMA FINANZA

1.1 📋 ÍNDICE

1.2 🎯 VISÃO GERAL DO SISTEMA

O **Finanza** é um sistema completo de controle financeiro pessoal desenvolvido como projeto interdisciplinar do IFSUL - Campus Venâncio Aires. O sistema implementa uma arquitetura cliente-servidor híbrida com suporte a múltiplas plataformas (Mobile Android e Desktop Java).

1.2.1 Características Principais

- **Arquitetura Cliente-Servidor:** Comunicação via TCP/IP (sockets) na porta 12345
- **Sincronização Bidirecional:** Dados sincronizados entre mobile e servidor em tempo real
- **Offline-First:** Aplicativo mobile funciona offline com sincronização posterior
- **Resolução de Conflitos:** Sistema inteligente de resolução de conflitos por timestamp
- **Segurança:** Criptografia SHA-256 para senhas, comunicação via sockets
- **Multiplataforma:** Mobile (Android) + Desktop (Java Swing) + Servidor (Java)

1.2.2 Tecnologias Utilizadas

Componente	Tecnologia	Versão
Mobile	Android SDK	API 24-36
Linguagem Mobile	Java	11
Banco Local	Room (SQLite)	2.6.1
Desktop	Java Swing	JDK 17+
Servidor	Java	JDK 17+
Banco Servidor	MySQL	8.0+
Comunicação	TCP/IP Sockets	-
Build Mobile	Gradle	8.12.3
Build Desktop	Ant (NetBeans)	-

1.3 📱 VERSÃO MOBILE (Android)

1.3.1 Informações Gerais

- **Package:** com.example.finanza
- **Min SDK:** 24 (Android 7.0)
- **Target SDK:** 36 (Android 14)
- **Compile SDK:** 36
- **Version:** 1.0 (versionCode: 1)
- **Linguagem:** Java 11
- **Build System:** Gradle

1.3.2 Estrutura de Diretórios

1.3.2.1 Código-Fonte Java (25 arquivos)

```
app/src/main/java/com/example/finanza/
├── MainActivity.java              # Atividade principal do app (splash/redirect)
├── db/                           # Camada de persistência local (5 arquivos)
│   ├── AppDatabase.java           # Configuração do banco Room (Singleton)
│   ├── CategoriaDao.java          # DAO para operações de Categoria (15 métodos)
│   ├── ContaDao.java             # DAO para operações de Conta (27 métodos)
│   ├── LancamentoDao.java         # DAO para operações de Lançamento (40+ métodos)
│   └── UsuarioDao.java            # DAO para operações de Usuário (20 métodos)
├── model/                           # Modelos de dados (4 entidades Room)
│   ├── Categoria.java              # Entidade Categoria (receita/despesa)
│   ├── Conta.java                 # Entidade Conta (corrente, poupança, etc)
│   ├── Lancamento.java             # Entidade Lançamento/Transação financeira
│   └── Usuario.java               # Entidade Usuário com suporte a sincronização
├── network/                      # Camada de rede e sincronização (6 arquivos)
│   ├── AuthManager.java            # Gerenciamento de autenticação e sessão
│   ├── ConflictResolutionManager.java  # Resolução de conflitos de sincronização
│   ├── EnhancedSyncService.java     # Serviço avançado de sincronização incremental
│   ├── Protocol.java              # Protocolo de comunicação (50+ comandos)
│   ├── ServerClient.java           # Cliente TCP/IP para comunicação com servidor
```

```
│   └── SyncService.java           # Serviço base de sincronização offline-
first
├── ui/                        # Camada de interface do usuário (8 Activities)
│   ├── AccountsActivity.java          # Gerenciamento de contas bancárias
│   ├── CategoriaActivity.java        # Gerenciamento de categorias personalizadas
│   ├── LoginActivity.java           # Autenticação de usuários (Activity Launcher)
│   ├── MenuActivity.java             # Dashboard principal com visão geral
│   ├── MovementsActivity.java        # Lançamentos e transações financeiras
│   ├── ProfileActivity.java              # Visualização e edição de perfil
│   ├── RegisterActivity.java             # Cadastro de novos usuários
│   └── SettingsActivity.java             # Configurações do aplicativo
└── util/                                 # Utilitários (1 arquivo)
    └── DataIntegrityValidator.java    # Validador de integridade e consistência
```

1.3.2.2 Layouts XML (25 arquivos)

```
app/src/main/res/layout/
├── activity_login.xml                    # Layout da tela de login
├── activity_register.xml                 # Layout da tela de registro
├── activity_main.xml                     # Layout da MainActivity
├── activity_menu.xml                     # Layout do dashboard principal
├── activity_accounts.xml              # Layout de gerenciamento de contas
├── activity_categoria.xml           # Layout de gerenciamento de categorias
├── activity_movements.xml                # Layout de movimentações
├── activity_profile.xml                  # Layout do perfil do usuário
├── activity_settings.xml                 # Layout de configurações
├── activity_base_content.xml             # Layout base reutilizável
├── content_menu.xml                      # Conteúdo do menu (incluído)
├── content_movements.xml            # Conteúdo de movimentações (incluído)
├── content_categoria.xml               # Conteúdo de categorias (incluído)
├── dialog_add_transaction.xml            # Dialog para adicionar transação
├── dialog_add_transaction_movements.xml # Dialog de transação (movimentos)
├── dialog_add_categoria.xml              # Dialog para adicionar categoria
├── dialog_edit_transaction.xml           # Dialog para editar transação
├── dialog_edit_categoria.xml             # Dialog para editar categoria
├── dialog_edit_account.xml               # Dialog para editar conta
├── dialog_edit_profile.xml               # Dialog para editar perfil
├── dialog_delete_transaction.xml       # Dialog de confirmação de exclusão
├── dialog_delete_categoria.xml           # Dialog de exclusão de categoria
├── dialog_delete_account.xml             # Dialog de exclusão de conta
├── dialog_confirm_delete_account.xml    # Confirmação de exclusão de conta
└── dialog_recuperar_senha.xml            # Dialog de recuperação de senha
```

1.3.2.3 Permissões do Manifesto

```xml
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

1.3.3 Dependências e Bibliotecas Mobile

```
dependencies {
    // Material Design e UI
    implementation "com.google.android.material:material:1.12.0"
    implementation "androidx.appcompat:appcompat:1.7.1"
    implementation "androidx.activity:activity:1.10.1"
    implementation "androidx.constraintlayout:constraintlayout:2.2.1"

    // Room Database (SQLite)
    implementation "androidx.room:room-runtime:2.6.1"
    annotationProcessor "androidx.room:room-compiler:2.6.1"

    // Lifecycle Components
    implementation "androidx.lifecycle:lifecycle-viewmodel:2.7.0"
    implementation "androidx.lifecycle:lifecycle-livedata:2.7.0"

    // Testing
    testImplementation "junit:junit:4.13.2"
    androidTestImplementation "androidx.test.ext:junit:1.3.0"
    androidTestImplementation "androidx.test.espresso:espresso-core:3.7.0"
}
```

1.3.4 Fluxo de Funcionalidades - Mobile

1.3.4.1 1. LoginActivity.java

Função: Tela de autenticação do usuário - Recebe email e senha do usuário - Usa `AuthManager.java` para validar credenciais via servidor - Comunica com servidor através de `ServerClient.java` usando `Protocol.java` - Em caso de sucesso, armazena dados do usuário em `UsuarioDao.java` - Redireciona para `MenuActivity.java`

1.3.4.2 2. RegisterActivity.java

Função: Cadastro de novos usuários - Coleta dados do novo usuário (nome, email, senha) - Envia dados ao servidor via `ServerClient.java` - Usa criptografia para senha (implementada em `Protocol.java`) - Após registro bem-sucedido, redireciona para `LoginActivity.java`

1.3.4.3 3. MenuActivity.java (Dashboard)

Função: Tela principal com visão geral financeira - Exibe saldo total de todas as contas - Mostra resumo de receitas e despesas - Busca dados de `ContaDao.java` e `LancamentoDao.java` - Inicia `SyncService.java` para sincronizar dados com servidor - Navega para outras activities (Accounts, Movements, Categories, Profile, Settings)

1.3.4.4 4. AccountsActivity.java

Função: Gerenciamento de contas bancárias - Lista todas as contas do usuário obtidas de `ContaDao.java` - Permite criar novas contas (corrente, poupança, cartão,

investimento) - Editar e excluir contas existentes - Sincroniza alterações com servidor via `EnhancedSyncService.java` - Atualiza saldos baseado em lançamentos de `LancamentoDao.java`

1.3.4.5 5. CategoriaActivity.java

Função: Gerenciamento de categorias de transações - Lista categorias obtidas de `CategoriaDao.java` - Permite criar, editar e excluir categorias - Categorias podem ser de receita ou despesa - Sincroniza com servidor usando `SyncService.java`

1.3.4.6 6. MovementsActivity.java

Função: Gerenciamento de lançamentos/transações - Lista todos os lançamentos de `LancamentoDao.java` - Permite criar novos lançamentos (receita ou despesa) - Editar e excluir lançamentos - Associa lançamento a uma conta (`Conta.java`) e categoria (`Categoria.java`) - Sincroniza alterações com servidor via `EnhancedSyncService.java` - Valida integridade usando `DataIntegrityValidator.java`

1.3.4.7 7. ProfileActivity.java

Função: Visualização e edição de perfil - Exibe dados do usuário de `UsuarioDao.java` - Permite editar nome e email - Permite alterar senha (criptografada) - Sincroniza alterações com servidor

1.3.4.8 8. SettingsActivity.java

Função: Configurações do aplicativo - Configurações de sincronização - Preferências de notificação - Opções de logout

1.3.5 Camada de Dados - Mobile

1.3.5.1 AppDatabase.java

Função: Configuração central do banco Room (SQLite) - Define versão do banco: 1 - Lista todas as entidades (Usuario, Conta, Categoria, Lancamento) - Fornece acesso aos DAOs através do padrão Singleton - Configuração: `@Database(entities = {Usuario.class, Conta.class, Categoria.class, Lancamento.class}, version = 1)` - Implementa padrão Singleton para garantir única instância - Suporta migração de schema quando necessário

1.3.5.2 DAOs (Data Access Objects)

1.3.5.2.1 UsuarioDao.java (20+ métodos) **Função:** CRUD de usuários no banco local

Operações CRUD Básicas: - `inserir(Usuario)`: Insere novo usuário, retorna ID - `atualizar(Usuario)`: Atualiza dados do usuário - `deletar(Usuario)`: Remove usuário do banco - `buscarPorId(int)`: Busca usuário pelo ID - `buscarPorEmail(String)`: Busca usuário pelo email único - `listarTodos()`: Lista todos os usuários

Sincronização: - `buscarPorUuid(String)`: Busca por UUID universal - `obterPendentesSync()`: Retorna usuários com status NEEDS_SYNC ou CONFLICT - `obterModificadosApos(timestamp)`: Usuários modificados após timestamp - `marcarComoSincronizado(id, syncTime)`: Marca como sincronizado - `marcarParaSync(id, timestamp)`: Marca para sincronização - `atualizarMetadataSync(uuid, status, syncTime, hash)`: Atualiza metadados

Transacionais: - `inserirOuAtualizar(Usuario)`: Insere ou atualiza baseado em UUID e timestamp - `inserirSeguro(Usuario)`: Insere com detecção de duplicatas

1.3.5.2.2 ContaDao.java (27+ métodos)

Função: Gerencia contas bancárias no SQLite

Operações CRUD Básicas: - `inserir(Conta)`: Insere nova conta, retorna ID - `atualizar(Conta)`: Atualiza dados da conta - `deletar(Conta)`: Remove conta do banco - `excluirPorUsuario(usuarioId)`: Remove todas as contas de um usuário - `deletarTodosDoUsuario(usuarioId)`: Alias para excluirPorUsuario - `listarPorUsuario(usuarioId)`: Lista contas do usuário - `buscarPorId(id)`: Busca conta pelo ID - `listarTodos()`: Lista todas as contas

Tipos de Conta Suportados: - Conta Corrente - Poupança - Cartão de Crédito - Investimento - Dinheiro

Sincronização: - `buscarPorUuid(String)`: Busca por UUID universal - `obterPendentesSync()`: Contas que precisam sincronização - `obterPendentesSyncPorUsuario(usuarioId)`: Pendentes de um usuário - `obterModificadosApos(timestamp)`: Contas modificadas após data - `obterModificadosAposPorUsuario(usuarioId, timestamp)`: Modificadas por usuário - `marcarComoSincronizado(id, syncTime)`: Marca como sincronizado - `marcarParaSync(id, timestamp)`: Marca para sincronização - `atualizarMetadataSync(uuid, status, syncTime, hash)`: Atualiza metadados - `atualizarStatusSync(uuids, status)`: Atualiza status de múltiplas contas - `obterUltimoTempoSync()`: Retorna timestamp da última sincronização

Detecção de Duplicatas: - `buscarDuplicataPorNomeEUsuario(nome, usuarioId, excludeUuid)`: Busca duplicatas - `buscarPorNomeEUsuario(nome, usuarioId)`: Busca por nome e usuário exatos

Transacionais: - `inserirOuAtualizar(Conta)`: Insere ou atualiza com resolução de conflitos - `inserirSeguro(Conta)`: Insere com detecção de duplicatas - `sincronizarDoServidor(serverId, nome, tipo, saldo, usuarioId)`: Sincroniza do servidor

1.3.5.2.3 CategoriaDao.java (15+ métodos)

Função: Gerencia categorias de transações

Operações CRUD Básicas: - `inserir(Categoria)`: Insere nova categoria - `atualizar(Categoria)`: Atualiza categoria existente - `deletar(Categoria)`: Remove categoria - `excluirPorUsuario(usuarioId)`: Remove todas as categorias do usuário - `listarPorUsuario(usuarioId)`: Lista categorias do usuário - `listarPorUsuarioETipo(usuarioId, tipo)`: Lista por usuário e tipo (receita/despesa) - `buscarPorId(id)`: Busca categoria pelo ID - `listarTodos()`: Lista todas as categorias

Tipos de Categoria: - Receita: Entrada de dinheiro - Despesa: Saída de dinheiro

Sincronização: - `buscarPorUuid(String)`: Busca por UUID - `obterPendentesSync()`: Categorias pendentes de sincronização - `obterModificadosApos(timestamp)`: Modificadas após timestamp - `marcarComoSincronizado(id, syncTime)`: Marca como sincronizado - `marcarParaSync(id, timestamp)`: Marca para sincronização

Transacionais: - `inserirOuAtualizar(Categoria)`: Insere ou atualiza com resolução de conflitos - `inserirSeguro(Categoria)`: Insere com detecção de duplicatas

1.3.5.2.4 LancamentoDao.java (40+ métodos) Função: Gerencia lançamentos/transações financeiras

Operações CRUD Básicas: - `inserir(Lancamento)`: Insere novo lançamento - `atualizar(Lancamento)`: Atualiza lançamento existente - `deletar(Lancamento)`: Remove lançamento (hard delete) - `excluirPorUsuario(usuarioId)`: Remove todos os lançamentos do usuário - `deletarTodosDoUsuario(usuarioId)`: Alias para excluirPorUsuario

Consultas por Usuário: - `listarPorUsuario(usuarioId)`: Lista todos ordenados por data DESC - `listarAtivosPorUsuario(usuarioId)`: Lista apenas não-deletados - `listarUltimasPorUsuario(usuarioId, limit)`: Lista últimos N lançamentos - `listarPorUsuarioPeriodo(usuarioId, inicio, fim)`: Lista por período

Consultas por Conta e Categoria: - `buscarPorId(id)`: Busca lançamento pelo ID - `listarPorConta(contaId)`: Lista lançamentos de uma conta - `buscarPorConta(contaId)`: Alias para listarPorConta - `listarPorCategoria(categoriaId)`: Lista lançamentos de uma categoria

Consultas de Soma e Cálculos: - `somaPorTipo(tipo, usuarioId)`: Soma total por tipo (receita/despesa) - `somaPorTipoConta(tipo, usuarioId, contaId)`: Soma por tipo e conta - `saldoPorConta(contaId, usuarioId)`: Calcula saldo de uma conta - `somaPorContaETipo(contaId, tipo)`: Soma por conta e tipo - `somaPorCategoria(categoriaId, usuarioId)`: Soma total de uma categoria

Busca e Detecção de Duplicatas: - `buscarPorDescricaoOuValor(usuarioId, searchTerm)`: Busca textual com wildcards - `buscarDuplicata(valor, data, descricao, contaId, usuarioId, excludeUuid)`: Busca duplicata exata - `buscarSimilares(valor, data, timeWindow, contaId, usuarioId)`: Busca transações similares

Sincronização: - `buscarPorUuid(String)`: Busca por UUID universal - `obterPendentesSync()`: Lançamentos que precisam sincronização - `obterPendentesSyncPorUsuario(usuarioId)`: Pendentes de um usuário - `obterModificadosApos(timestamp)`: Lançamentos modificados (não deletados) - `obterModificadosAposPorUsuario(usuarioId, timestamp)`: Modificados por usuário - `obterDeletadosApos(timestamp)`: Lançamentos marcados como deletados - `marcarComoSincronizado(id, syncTime)`: Marca como sincronizado - `marcarParaSync(id, timestamp)`: Marca para sincronização - `marcarComoExcluido(id, timestamp)`: Soft delete (marca isDeleted=1) - `atualizarMetadataSync(uuid, status, syncTime, hash)`: Atualiza metadados - `atualizarStatusSync(uuids, status)`: Atualiza status de múltiplos lançamentos - `obterUltimoTempoSync()`: Retorna timestamp da última sincronização

Atualização de IDs de Relacionamento: - `atualizarCategoriaId(antigoId,`

`novoId)`: Atualiza ID da categoria em lançamentos - `atualizarContaId(antigoId, novoId)`: Atualiza ID da conta em lançamentos

Transacionais: - `inserirOuAtualizar(Lancamento)`: Insere ou atualiza com resolução de conflitos por timestamp - `inserirSeguro(Lancamento)`: Insere com detecção inteligente de duplicatas em múltiplas camadas - `excluirSeguro(id)`: Exclui usando soft delete para manter histórico

1.3.6 Camada de Rede - Mobile

1.3.6.1 ServerClient.java

Função: Cliente de comunicação via sockets TCP/IP - **Protocolo:** TCP/IP sobre sockets Java - **Porta Padrão:** 12345 - **Timeout:** Configurável (padrão 30 segundos) - **Características:** - Conecta ao servidor pela porta especificada - Envia comandos usando Protocol.java - Recebe respostas do servidor - Gerencia timeout e reconexão automática - Suporta leitura/escrita de streams - Tratamento de erros de rede - Pool de conexões reutilizáveis

Métodos Principais: - `connect(host, port)`: Estabelece conexão TCP - `sendCommand(command, params)`: Envia comando ao servidor - `receiveResponse()`: Recebe resposta do servidor - `disconnect()`: Fecha conexão - `isConnected()`: Verifica status da conexão

1.3.6.2 Protocol.java (487 linhas - 50+ comandos)

Função: Define o protocolo completo de comunicação cliente-servidor

Separadores:

```
COMMAND_SEPARATOR = "|"     // Separa comando e parâmetros
FIELD_SEPARATOR = ";"       // Separa campos em listas
DATA_SEPARATOR = ","        // Separa dados individuais
```

Códigos de Status:

```
STATUS_OK = "OK"                        // Operação bem-sucedida
STATUS_ERROR = "ERROR"                  // Erro genérico
STATUS_INVALID_DATA = "INVALID_DATA"    // Dados inválidos
STATUS_INVALID_CREDENTIALS = "INVALID_CREDENTIALS"  // Credenciais incorretas
STATUS_USER_EXISTS = "USER_EXISTS"      // Email já cadastrado
STATUS_NOT_FOUND = "NOT_FOUND"          // Recurso não encontrado
STATUS_CONFLICT = "CONFLICT"            // Conflito de sincronização
STATUS_DUPLICATE = "DUPLICATE"          // Duplicata detectada
```

Comandos de Autenticação:

```
CMD_LOGIN = "LOGIN"                     // Login de usuário: LOGIN|email|senha
CMD_REGISTER = "REGISTER"               // Registro: REGISTER|nome|email|senha
CMD_LOGOUT = "LOGOUT"                   // Logout do usuário
CMD_GET_PERFIL = "GET_PERFIL"           // Buscar perfil do usuário
CMD_UPDATE_PERFIL = "UPDATE_PERFIL"     // Atualizar perfil
CMD_CHANGE_PASSWORD = "CHANGE_PASSWORD"  // Trocar senha
CMD_RESET_PASSWORD = "RESET_PASSWORD"    // Recuperar senha
```

Comandos de Dashboard:

```
CMD_GET_DASHBOARD = "GET_DASHBOARD"   // Buscar dados do dashboard
```

Comandos de Contas:

```
CMD_LIST_CONTAS = "LIST_CONTAS"       // Listar todas as contas
CMD_ADD_CONTA = "ADD_CONTA"           // Adicionar conta: nome|tipo|saldo|usuarioId
CMD_UPDATE_CONTA = "UPDATE_CONTA"     // Atualizar conta existente
CMD_DELETE_CONTA = "DELETE_CONTA"     // Excluir conta: contaId
CMD_ADD_CONTA_ENHANCED = "ADD_CONTA_ENHANCED"      // Adicionar com UUID
CMD_UPDATE_CONTA_ENHANCED = "UPDATE_CONTA_ENHANCED"  // Atualizar com metadados
CMD_SYNC_CONTA = "SYNC_CONTA"         // Sincronizar conta específica
```

Comandos de Categorias:

```
CMD_LIST_CATEGORIAS = "LIST_CATEGORIAS"  // Listar todas as categorias
CMD_LIST_CATEGORIAS_TIPO = "LIST_CATEGORIAS_TIPO"  // Listar por tipo (receita/despe
CMD_ADD_CATEGORIA = "ADD_CATEGORIA"      // Adicionar categoria
CMD_UPDATE_CATEGORIA = "UPDATE_CATEGORIA"  // Atualizar categoria
CMD_DELETE_CATEGORIA = "DELETE_CATEGORIA"  // Excluir categoria
CMD_ADD_CATEGORIA_ENHANCED = "ADD_CATEGORIA_ENHANCED"      // Adicionar com UUID
CMD_UPDATE_CATEGORIA_ENHANCED = "UPDATE_CATEGORIA_ENHANCED"  // Atualizar com metada
CMD_SYNC_CATEGORIA = "SYNC_CATEGORIA"    // Sincronizar categoria específica
```

Comandos de Movimentações/Lançamentos:

```
CMD_LIST_MOVIMENTACOES = "LIST_MOVIMENTACOES"  // Listar todas as movimentações
CMD_LIST_MOVIMENTACOES_PERIODO = "LIST_MOVIMENTACOES_PERIODO"  // Listar por período
CMD_LIST_MOVIMENTACOES_CONTA = "LIST_MOVIMENTACOES_CONTA"  // Listar por conta
CMD_ADD_MOVIMENTACAO = "ADD_MOVIMENTACAO"      // Adicionar movimentação
CMD_UPDATE_MOVIMENTACAO = "UPDATE_MOVIMENTACAO"  // Atualizar movimentação
CMD_DELETE_MOVIMENTACAO = "DELETE_MOVIMENTACAO"  // Excluir movimentação
CMD_ADD_MOVIMENTACAO_ENHANCED = "ADD_MOVIMENTACAO_ENHANCED"      // Adicionar com UU
CMD_UPDATE_MOVIMENTACAO_ENHANCED = "UPDATE_MOVIMENTACAO_ENHANCED"  // Atualizar com
CMD_SYNC_MOVIMENTACAO = "SYNC_MOVIMENTACAO"    // Sincronizar movimentação especific
```

Comandos de Sincronização Avançada:

```
CMD_SYNC_STATUS = "SYNC_STATUS"             // Verificar status de sincronização
CMD_INCREMENTAL_SYNC = "INCREMENTAL_SYNC"   // Sincronização incremental por timesta
CMD_LIST_CHANGES_SINCE = "LIST_CHANGES_SINCE"  // Listar mudanças desde timestamp
CMD_RESOLVE_CONFLICT = "RESOLVE_CONFLICT"   // Resolver conflito manual
CMD_BULK_UPLOAD = "BULK_UPLOAD"             // Upload em lote (múltiplas entidades)
CMD_VERIFY_INTEGRITY = "VERIFY_INTEGRITY"   // Verificar integridade dos dados
```

Formato de Mensagens:

Comando de Login:

```
LOGIN|usuario@email.com|senha_hash_sha256
```

Resposta de Sucesso:

```
OK|{"id":1,"nome":"Usuario","email":"usuario@email.com"}
```

Comando de Adicionar Conta:

```
ADD_CONTA|Conta Corrente|corrente|1000.0|1
```

Comando Enhanced (com UUID):

```
ADD_CONTA_ENHANCED|uuid-1234|Nubank|corrente|5000.0|1|timestamp|hash
```

Resposta de Erro:

```
ERROR|Mensagem de erro descritiva
```

1.3.6.3 AuthManager.java

Função: Gerencia autenticação e sessão do usuário

Responsabilidades: - **Login e Logout:** Autenticação via servidor - **Gerenciamento de Sessão:** Mantém usuário logado - **Tokens:** Renovação automática de tokens de sessão - **Armazenamento Seguro:** Salva credenciais de forma segura - **Validação:** Valida formato de email e senha - **Cache:** Mantém dados do usuário em memória

Métodos Principais: - `login(email, senha)`: Autentica usuário no servidor - `register(nome, email, senha)`: Registra novo usuário - `logout()`: Encerra sessão do usuário - `isLoggedIn()`: Verifica se há usuário logado - `getCurrentUser()`: Retorna usuário atual - `updateProfile(usuario)`: Atualiza dados do perfil - `changePassword(senhaAntiga, senhaNova)`: Altera senha

1.3.6.4 SyncService.java

Função: Serviço base de sincronização offline-first

Características: - **Offline-First:** Funciona sem conexão, sincroniza quando disponível - **Fila de Operações:** Mantém fila de operações pendentes - **Sincronização Automática:** Executa periodicamente em background - **Detecção de Conflitos:** Identifica conflitos por timestamp - **Retry Logic:** Tenta novamente em caso de falha

Fluxo de Sincronização: 1. Verifica conectividade com servidor 2. Busca dados pendentes nos DAOs (syncStatus = 2 ou 3) 3. Para cada operação pendente: - Envia ao servidor via ServerClient - Aguarda confirmação - Marca como sincronizado ou trata erro 4. Busca atualizações do servidor 5. Aplica atualizações localmente via DAOs 6. Notifica activities sobre dados atualizados

Métodos Principais: - `startSync()`: Inicia sincronização - `syncContas()`: Sincroniza contas - `syncCategorias()`: Sincroniza categorias - `syncLancamentos()`: Sincroniza lançamentos - `handleSyncError(error)`: Trata erros de sincronização

1.3.6.5 EnhancedSyncService.java

Função: Serviço avançado de sincronização incremental

Recursos Avançados: - **Sincronização Incremental:** Apenas dados modificados desde último sync - **Priorização de Operações:** Prioriza operações críticas - **Resolução de Conflitos Complexos:** Estratégias avançadas de merge - **Compressão de Dados:** Reduz tráfego de rede - **Batch Operations:** Agrupa operações para eficiência - **Delta Sync:** Envia apenas diferenças (delta) dos dados

Estratégias de Sincronização: - **Full Sync:** Sincronização completa (primeira vez) - **Incremental Sync:** Apenas mudanças desde último timestamp - **Delta Sync:** Apenas campos modificados - **Batch Sync:** Múltiplas entidades em uma requisição

Métodos Principais: - `incrementalSync(lastSyncTimestamp)`: Sincronização incremental - `batchUpload(entities)`: Upload em lote - `deltaSync(entity)`: Sincronização delta - `getPendingOperations()`: Retorna operações pendentes - `prioritizeOperations()`: Define prioridade de operações

1.3.6.6 ConflictResolutionManager.java

Função: Resolução de conflitos de sincronização de dados

Estratégias de Resolução:

1. **Last Write Wins (LWW):** Última modificação vence
   - Compara timestamps (lastModified)
   - Mantém versão mais recente
   - Estratégia padrão
2. **Server Wins:** Prioridade do servidor
   - Servidor sempre sobrescreve cliente
   - Usada para dados administrativos
3. **Client Wins:** Prioridade do cliente
   - Cliente sobrescreve servidor
   - Usada para dados locais críticos
4. **Merge:** Combina mudanças
   - Tenta mesclar campos não conflitantes
   - Marca campos conflitantes para revisão manual
5. **Manual Resolution:** Notifica usuário
   - Conflitos críticos requerem decisão do usuário
   - Interface de resolução manual

Detecção de Conflitos: - Compara timestamps de modificação - Verifica hash de dados (serverHash vs localHash) - Identifica tipo de conflito (UPDATE vs DELETE, etc)

Métodos Principais: - `detectConflict(local, server)`: Detecta conflito - `resolveConflict(local, server, strategy)`: Resolve conflito - `mergeEntities(local, server)`: Tenta merge automático - `notifyUserConflict(conflict)`: Notifica usuário - `applyResolution(entity)`: Aplica resolução escolhida

1.3.7 Utilitários - Mobile

1.3.7.1 DataIntegrityValidator.java

Função: Validação de integridade de dados - Valida consistência de saldos - Verifica integridade referencial - Detecta anomalias nos dados

---

1.4 📦 DEPENDÊNCIAS E BIBLIOTECAS

1.4.1 Mobile (Android)

1.4.1.1 UI e Material Design

```
implementation "com.google.android.material:material:1.12.0"
```

- **Descrição:** Material Design Components para Android
- **Uso:** Buttons, TextInputLayout, CardView, FloatingActionButton, Dialogs
- **Features:** Temas Material 3, componentes modernos

```
implementation "androidx.appcompat:appcompat:1.7.1"
```

- **Descrição:** Biblioteca de compatibilidade AndroidX
- **Uso:** AppCompatActivity, Toolbar, ActionBar
- **Features:** Compatibilidade retroativa com versões antigas do Android

```
implementation "androidx.constraintlayout:constraintlayout:2.2.1"
```

- **Descrição:** Layout constraint-based para UI responsiva
- **Uso:** Layouts complexos sem nesting profundo
- **Features:** Performance otimizada, design responsivo

```
implementation "androidx.activity:activity:1.10.1"
```

- **Descrição:** Componentes de Activity do AndroidX
- **Uso:** Gerenciamento de Activities e callbacks
- **Features:** APIs modernas para Activities

1.4.1.2 Persistência de Dados (Room)

```
implementation "androidx.room:room-runtime:2.6.1"
annotationProcessor "androidx.room:room-compiler:2.6.1"
```

- **Descrição:** ORM sobre SQLite
- **Uso:** Camada de abstração para banco de dados local
- **Features:**
  - Annotations (@Entity, @Dao, @Database)
  - Queries SQL verificadas em compile-time
  - LiveData integration
  - Migrations automáticas
  - Transações ACID

Entidades Room: - Usuario, Conta, Categoria, Lancamento

DAOs Implementados: - UsuarioDao (20+ métodos) - ContaDao (27+ métodos) - CategoriaDao (15+ métodos) - LancamentoDao (40+ métodos)

1.4.1.3 Lifecycle Components

```
implementation "androidx.lifecycle:lifecycle-viewmodel:2.7.0"
```

- **Descrição:** ViewModel para gerenciamento de estado
- **Uso:** Manter estado da UI durante mudanças de configuração

- **Features:** Sobrevive a rotações de tela

```
implementation "androidx.lifecycle:lifecycle-livedata:2.7.0"
```

- **Descrição:** LiveData para observação de dados
- **Uso:** Atualização reativa da UI
- **Features:** Lifecycle-aware, evita memory leaks

1.4.1.4 Testing

```
testImplementation "junit:junit:4.13.2"
```

- **Descrição:** Framework de testes unitários
- **Uso:** Testes de lógica de negócio

```
androidTestImplementation "androidx.test.ext:junit:1.3.0"
androidTestImplementation "androidx.test.espresso:espresso-core:3.7.0"
```

- **Descrição:** Framework de testes instrumentados
- **Uso:** Testes de UI e integração

1.4.1.5 Bibliotecas Nativas do Android (Não-Gradle)

Java Sockets (java.net): - `Socket`: Comunicação TCP/IP com servidor - `ServerSocket`: Não usado no mobile - Classe: `ServerClient.java`

JSON Parsing (org.json): - `JSONObject`: Parsing de respostas do servidor - `JSONArray`: Listas de dados - Incluído no Android SDK

Security (java.security): - `MessageDigest`: SHA-256 para hashing de senhas - `SecureRandom`: Geração de UUIDs e tokens - Classe: `AuthManager.java`, `SecurityUtil.java`

Concurrency (java.util.concurrent): - `ExecutorService`: Threads para sincronização - `AsyncTask`: Operações de rede (deprecated, mas ainda usado) - Classe: `SyncService.java`, `EnhancedSyncService.java`

1.4.2 Desktop (Servidor)

1.4.2.1 Bibliotecas Principais

MySQL Connector/J

```
<!-- Deve ser adicionado manualmente ao classpath -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

- **Descrição:** Driver JDBC para MySQL
- **Uso:** Conexão e queries ao banco MySQL
- **Classes:** DatabaseUtil.java, todos os DAOs

Java Standard Library:

java.net.Socket: - Comunicação TCP/IP servidor-cliente - Classes: FinanzaServer.java, ClientHandler.java

java.sql (JDBC): - Connection, PreparedStatement, ResultSet - Classes: Todos os DAOs (UsuarioDAO, ContaDAO, etc)

javax.crypto: - Criptografia e hashing - Classe: SecurityUtil.java

java.util.concurrent: - Multithreading para múltiplos clientes - Classe: FinanzaServer.java (ThreadPool)

1.4.3 Desktop (Cliente Admin)

1.4.3.1 Java Swing (javax.swing)

- **JFrame:** Janelas principais
- **JDialog:** Dialogs modais
- **JTable:** Tabelas de dados
- **JButton, JTextField, JLabel:** Componentes de UI
- **Classes:** LoginView.java, AdminDashboardView.java, EditarUsuarioDialog.java

1.4.3.2 Networking

- **Socket:** Comunicação com servidor
- **Classe:** NetworkClient.java

1.4.4 Comparação de Dependências

Feature	Mobile	Servidor	Cliente
ORM/Database	Room 2.6.1	JDBC Raw SQL	-
UI Framework	Material Design	-	Swing
Networking	java.net.Socket	java.net.ServerSocket	java.net.Socket
JSON Parsing	org.json (Android)	Manual (StringBuilder)	Manual
Build System	Gradle	Ant	Ant
Dependency Mgmt	Gradle	Manual JARs	Manual JARs

1.4.5 Tamanhos de Dependências

Mobile (APK Final): - Tamanho base: ~5 MB - Com dependências: ~8-10 MB - Room: ~1.5 MB - Material Design: ~2 MB - Lifecycle: ~500 KB

Desktop (JAR): - Servidor: ~50 KB (sem MySQL driver) - Servidor + MySQL driver: ~2.5 MB - Cliente: ~40 KB

1.4.6 Requisitos de Versão

Componente	Min Version	Recommended
Android SDK	API 24 (7.0)	API 33+ (13+)

Componente	Min Version	Recommended
Java (Mobile)	Java 11	Java 11
Java (Desktop)	Java 17	Java 17+
MySQL	5.7	8.0+
Gradle	7.0	8.12.3
Android Studio	Arctic Fox	Latest

1.5 🖥️ VERSÃO DESKTOP

1.5.1 Arquitetura Geral

A versão desktop do Finanza é dividida em dois componentes independentes:

1. **ServidorFinanza:** Servidor TCP/IP que gerencia a lógica de negócio e persistência no MySQL
2. **ClienteFinanza:** Interface administrativa Swing para gerenciar usuários do sistema

Ambos se comunicam via sockets TCP/IP na porta 12345.

1.5.2 Cliente Desktop (ClienteFinanza)

1.5.2.1 Informações Gerais

- **Linguagem:** Java 17+
- **Interface:** Java Swing
- **Build System:** Apache Ant (NetBeans)
- **Arquitetura:** MVC (Model-View-Controller)
- **Comunicação:** Socket TCP/IP

1.5.2.2 Estrutura de Diretórios (11 arquivos Java)

1.5.3 Cliente Desktop

```
DESKTOP VERSION/ClienteFinanza/src/
├── MainCliente.java                    # Classe principal do cliente
├── controller/                         # Controladores MVC
│   ├── AuthController.java             # Controle de autenticação
│   └── FinanceController.java         # Controle de operações financeiras
├── view/                               # Interface gráfica Swing
│   ├── AdminDashboardView.java         # Dashboard do administrador
│   ├── EditarUsuarioDialog.java        # Dialog de edição de usuário
│   └── LoginView.java                  # Tela de login
├── model/                              # Modelos de dados
│   ├── Categoria.java                  # Modelo Categoria
│   ├── Conta.java                      # Modelo Conta
│   ├── Movimentacao.java               # Modelo Movimentação
│   └── Usuario.java                    # Modelo Usuário
```

```
└── util/                                   # Utilitários
    └── NetworkClient.java                  # Cliente de rede
```

1.5.4 Servidor Desktop

```
DESKTOP VERSION/ServidorFinanza/src/
├── MainServidor.java                       # Classe principal do servidor
├── server/                                 # Lógica do servidor
│   ├── ClientHandler.java                # Handler para cada cliente conectado
│   ├── FinanzaServer.java                  # Servidor principal TCP/IP
│   └── Protocol.java                       # Protocolo de comunicação
├── dao/                                    # Acesso a dados (MySQL)
│   ├── CategoriaDAO.java                   # DAO Categoria
│   ├── ContaDAO.java                       # DAO Conta
│   ├── MovimentacaoDAO.java                # DAO Movimentação
│   └── UsuarioDAO.java                     # DAO Usuário
├── model/                                  # Modelos de dados
│   ├── Categoria.java                      # Modelo Categoria
│   ├── Conta.java                          # Modelo Conta
│   ├── Movimentacao.java                   # Modelo Movimentação
│   └── Usuario.java                        # Modelo Usuário
└── util/                                   # Utilitários
    ├── DatabaseUtil.java                   # Utilitário de banco de dados
   └── SecurityUtil.java                 # Utilitário de segurança (criptografia)
```

1.5.5 Fluxo de Funcionalidades - Desktop Cliente

1.5.5.1 1. MainCliente.java

Função: Ponto de entrada do cliente desktop - Inicializa a aplicação - Carrega configurações - Exibe `LoginView.java`

1.5.5.2 2. LoginView.java

Função: Interface de login para administradores - Coleta credenciais do administrador - Usa `AuthController.java` para autenticação - Conecta ao servidor via `NetworkClient.java` - Exibe `AdminDashboardView.java` após login

1.5.5.3 3. AdminDashboardView.java

Função: Dashboard administrativo - Lista todos os usuários do sistema - Permite visualizar detalhes de cada usuário - Acessa `EditarUsuarioDialog.java` para editar usuários - Usa `FinanceController.java` para operações - Busca dados do servidor via `NetworkClient.java`

1.5.5.4 4. EditarUsuarioDialog.java

Função: Dialog para edição de usuários - Permite editar nome, email - Permite alterar senha - Salva alterações no servidor via `FinanceController.java`

1.5.5.5 5. AuthController.java

Função: Controla autenticação - Valida credenciais de administrador - Gerencia sessão - Comunica com servidor usando `NetworkClient.java`

1.5.5.6 6. FinanceController.java

Função: Controla operações financeiras - Busca lista de usuários - Atualiza dados de usuários - Gerencia operações CRUD via servidor

1.5.5.7 7. NetworkClient.java

Função: Cliente de comunicação com servidor - Estabelece conexão TCP/IP - Envia comandos usando protocolo definido - Recebe e processa respostas

1.5.6 Fluxo de Funcionalidades - Servidor

1.5.6.1 1. MainServidor.java

Função: Ponto de entrada do servidor - Inicializa `FinanzaServer.java` - Configura porta de escuta (geralmente 12345) - Inicializa conexão com banco MySQL via `DatabaseUtil.java`

1.5.6.2 2. FinanzaServer.java

Função: Servidor TCP/IP principal - Escuta conexões na porta configurada - Aceita conexões de clientes (mobile e desktop) - Cria uma thread `ClientHandler.java` para cada cliente

1.5.6.3 3. ClientHandler.java

Função: Processa requisições de um cliente - Recebe comandos do cliente - Interpreta usando `Protocol.java` - Executa operações via DAOs apropriados - Retorna respostas ao cliente - Gerencia autenticação usando `SecurityUtil.java`

1.5.6.4 4. Protocol.java

Função: Define protocolo de comunicação - Comandos suportados: - LOGIN: Autenticação - REGISTER: Registro de novo usuário - LISTAR_USUARIOS: Lista todos usuários (admin) - ATUALIZAR_USUARIO: Atualiza dados de usuário - SYNC_CONTAS: Sincroniza contas - SYNC_CATEGORIAS: Sincroniza categorias - SYNC_LANCAMENTOS: Sincroniza lançamentos - CREATE/UPDATE/DELETE para cada entidade - Define formato de resposta (SUCCESS, ERROR, DATA)

1.5.6.5 5. DAOs do Servidor

UsuarioDAO.java - Função: Gerencia usuários no MySQL - Métodos: - autenticar(email, senha): Valida login - criar(usuario): Cria novo usuário - atualizar(usuario): Atualiza dados - listarTodos(): Lista usuários (admin) - buscarPorId(id): Busca usuário específico

ContaDAO.java - Função: Gerencia contas no MySQL - Métodos: - criar(conta): Cria nova conta - atualizar(conta): Atualiza conta - deletar(id): Remove conta - listarPorUsuario(usuarioId): Lista contas do usuário - atualizarSaldo(contaId, valor): Atualiza saldo

CategoriaDAO.java - Função: Gerencia categorias no MySQL - Métodos: - criar(categoria): Cria categoria - atualizar(categoria): Atualiza categoria - deletar(id): Remove categoria - listarPorUsuario(usuarioId): Lista categorias do usuário

MovimentacaoDAO.java - Função: Gerencia movimentações no MySQL - Métodos: - criar(movimentacao): Cria lançamento - atualizar(movimentacao): Atualiza lançamento - deletar(id): Remove lançamento - listarPorUsuario(usuarioId): Lista lançamentos - listarPorConta(contaId): Lista por conta - listarPorCategoria(categoriaId): Lista por categoria - listarPorPeriodo(dataInicio, dataFim): Lista por período

1.5.6.6 6. Utilitários do Servidor

DatabaseUtil.java - Função: Gerencia conexões com MySQL - Métodos: - getConnection(): Obtém conexão do pool - closeConnection(): Fecha conexão - inicializarBanco(): Cria tabelas se não existirem - Configurações: host, porta, database, usuário, senha

SecurityUtil.java - Função: Segurança e criptografia - Métodos: - hashSenha(senha): Gera hash SHA-256 da senha - verificarSenha(senha, hash): Valida senha - gerarToken(): Gera token de sessão

---

1.6 🔄 FLUXO DE DADOS COMPLETO

1.6.1 Exemplo 1: Login de Usuário (Mobile)

1. **Usuario** insere credenciais em `LoginActivity.java`
2. `LoginActivity` chama `AuthManager.login(email, senha)`
3. `AuthManager` usa `ServerClient.sendCommand(Protocol.LOGIN, dados)`
4. `ServerClient` envia via socket TCP/IP para servidor
5. **Servidor** recebe em `ClientHandler.processCommand()`
6. `ClientHandler` chama `UsuarioDAO.autenticar(email, senha)`
7. `UsuarioDAO` consulta MySQL, valida senha com `SecurityUtil.verificarSenha()`
8. Resposta SUCCESS retorna para `ClientHandler`
9. `ClientHandler` envia resposta ao `ServerClient`
10. `ServerClient` retorna para `AuthManager`
11. `AuthManager` salva usuário em `UsuarioDao` (local)
12. `LoginActivity` redireciona para `MenuActivity`

1.6.2 Exemplo 2: Criar Lançamento (Mobile)

1. **Usuario** preenche formulário em `MovementsActivity.java`
2. `MovementsActivity` cria objeto `Lancamento`
3. Salva localmente em `LancamentoDao.insert(lancamento)`
4. Chama `EnhancedSyncService.syncLancamento(lancamento)`

5. `EnhancedSyncService` envia ao servidor via `ServerClient`
6. **Servidor** (`ClientHandler`) recebe comando CREATE_LANCAMENTO
7. `ClientHandler` chama `MovimentacaoDAO.criar(movimentacao)`
8. `MovimentacaoDAO` insere no MySQL e atualiza saldo via `ContaDAO`
9. Resposta SUCCESS retorna ao mobile
10. `MovementsActivity` atualiza UI com novo lançamento

1.6.3 Exemplo 3: Administrador Edita Usuário (Desktop)

1. **Admin** seleciona usuário em `AdminDashboardView.java`
2. Clica em "Editar", abre `EditarUsuarioDialog.java`
3. Admin altera dados e clica "Salvar"
4. `EditarUsuarioDialog` chama `FinanceController.atualizarUsuario(usuario)`
5. `FinanceController` usa `NetworkClient.sendCommand(Protocol.ATUALIZAR_USUARIO, dados)`
6. **Servidor** (`ClientHandler`) recebe e chama `UsuarioDAO.atualizar(usuario)`
7. `UsuarioDAO` atualiza MySQL
8. Resposta SUCCESS retorna ao desktop
9. `AdminDashboardView` atualiza lista de usuários

1.6.4 Exemplo 4: Sincronização Automática (Mobile)

1. `MenuActivity` inicia `SyncService` em background
2. `SyncService` verifica dados pendentes em todos os DAOs
3. Para cada operação pendente:
   - Envia ao servidor via `ServerClient`
   - Aguarda confirmação
   - Marca como sincronizado ou trata erro
4. Busca atualizações do servidor
5. Aplica atualizações localmente via DAOs
6. `ConflictResolutionManager` resolve conflitos se houver
7. Notifica activities sobre dados atualizados

1.7 🎨 CAMADA DE INTERFACE

1.7.1 Mobile - Activities e Layouts

- **activity_login.xml** → LoginActivity.java
- **activity_register.xml** → RegisterActivity.java
- **activity_menu.xml** → MenuActivity.java (Dashboard)
- **activity_accounts.xml** → AccountsActivity.java
- **activity_categoria.xml** → CategoriaActivity.java
- **activity_movements.xml** → MovementsActivity.java
- **activity_profile.xml** → ProfileActivity.java
- **activity_settings.xml** → SettingsActivity.java

1.7.2 Desktop - Views Swing

- **LoginView.java** → JFrame de login
- **AdminDashboardView.java** → JFrame principal com JTable de usuários

- **EditarUsuarioDialog.java** → JDialog modal para edição

1.8 🗄️ BANCO DE DADOS

1.8.1 Mobile - Room (SQLite)

1.8.1.1 Configuração

- **ORM:** Room Persistence Library 2.6.1
- **Banco:** SQLite (local no dispositivo)
- **Versão do Schema:** 1
- **Localização:** `/data/data/com.example.finanza/databases/finanza.db`

1.8.1.2 Tabelas e Campos Completos

```
CREATE TABLE usuario (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    uuid TEXT NOT NULL UNIQUE,                -- UUID universal para sincronização
    nome TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    senhaHash TEXT NOT NULL,                  -- SHA-256 hash
    dataCriacao INTEGER NOT NULL,             -- Timestamp em milisegundos
    lastModified INTEGER NOT NULL,            -- Timestamp última modificação
    syncStatus INTEGER NOT NULL DEFAULT 0,    -- 0=LOCAL, 1=SYNCED, 2=NEEDS_SYNC, 3=C
    lastSyncTime INTEGER,                     -- Timestamp última sincronização
    serverHash TEXT                           -- Hash MD5 dos dados no servidor
);

-- Índices
CREATE INDEX index_Usuario_uuid ON usuario(uuid);
CREATE INDEX index_Usuario_syncStatus ON usuario(syncStatus);
CREATE UNIQUE INDEX index_Usuario_email ON usuario(email);
```

1.8.1.2.1 Tabela: usuario **Estados de Sincronização:** - `0 = LOCAL_ONLY`: Dados apenas locais, não sincronizados - `1 = SYNCED`: Dados sincronizados com servidor - `2 = NEEDS_SYNC`: Dados modificados, necessitam sincronização - `3 = CONFLICT`: Conflito detectado durante sincronização

```
CREATE TABLE conta (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    uuid TEXT NOT NULL UNIQUE,
    nome TEXT NOT NULL,
    tipo TEXT NOT NULL,                       -- 'corrente', 'poupanca', 'cartao', 'i
    saldoInicial REAL NOT NULL DEFAULT 0.0,
    saldoAtual REAL NOT NULL DEFAULT 0.0,
    usuarioId INTEGER NOT NULL,
    dataCriacao INTEGER NOT NULL,
    lastModified INTEGER NOT NULL,
```

```
    syncStatus INTEGER NOT NULL DEFAULT 0,
    lastSyncTime INTEGER,
    serverHash TEXT,
    FOREIGN KEY (usuarioId) REFERENCES usuario(id) ON DELETE CASCADE
);

-- Índices
CREATE INDEX index_Conta_uuid ON conta(uuid);
CREATE INDEX index_Conta_syncStatus ON conta(syncStatus);
CREATE INDEX index_Conta_usuarioId ON conta(usuarioId);
CREATE INDEX index_Conta_nome_usuarioId ON conta(nome, usuarioId);
```

1.8.1.2.2 Tabela: conta **Tipos de Conta:** - `corrente`: Conta corrente bancária - `poupanca`: Conta poupança - `cartao`: Cartão de crédito - `investimento`: Conta de investimentos - `dinheiro`: Dinheiro em espécie

```
CREATE TABLE categoria (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    uuid TEXT NOT NULL UNIQUE,
    nome TEXT NOT NULL,
    tipo TEXT NOT NULL,                    -- 'receita' ou 'despesa'
    usuarioId INTEGER NOT NULL,
    dataCriacao INTEGER NOT NULL,
    lastModified INTEGER NOT NULL,
    syncStatus INTEGER NOT NULL DEFAULT 0,
    lastSyncTime INTEGER,
    serverHash TEXT,
    FOREIGN KEY (usuarioId) REFERENCES usuario(id) ON DELETE CASCADE
);

-- Índices
CREATE INDEX index_Categoria_uuid ON categoria(uuid);
CREATE INDEX index_Categoria_syncStatus ON categoria(syncStatus);
CREATE INDEX index_Categoria_usuarioId ON categoria(usuarioId);
CREATE INDEX index_Categoria_tipo ON categoria(tipo);
CREATE INDEX index_Categoria_nome_usuarioId_tipo ON categoria(nome, usuarioId, tipo)
```

1.8.1.2.3 Tabela: categoria **Tipos de Categoria:** - `receita`: Categoria de entrada de dinheiro - `despesa`: Categoria de saída de dinheiro

Categorias Padrão de Despesa: - Alimentação - Transporte - Moradia - Saúde - Educação - Lazer

Categorias Padrão de Receita: - Salário - Freelance - Investimentos - Outros

```
CREATE TABLE lancamento (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    uuid TEXT NOT NULL UNIQUE,
    valor REAL NOT NULL,
    data INTEGER NOT NULL,                 -- Timestamp da transação
```

```
    descricao TEXT,
    tipo TEXT NOT NULL,                    -- 'receita' ou 'despesa'
    contaId INTEGER NOT NULL,
    categoriaId INTEGER NOT NULL,
    usuarioId INTEGER NOT NULL,
    dataCriacao INTEGER NOT NULL,
    lastModified INTEGER NOT NULL,
    syncStatus INTEGER NOT NULL DEFAULT 0,
    lastSyncTime INTEGER,
    serverHash TEXT,
    isDeleted INTEGER NOT NULL DEFAULT 0,    -- Soft delete: 0=ativo, 1=deletado
    FOREIGN KEY (contaId) REFERENCES conta(id) ON DELETE CASCADE,
    FOREIGN KEY (categoriaId) REFERENCES categoria(id) ON DELETE CASCADE,
    FOREIGN KEY (usuarioId) REFERENCES usuario(id) ON DELETE CASCADE
);

-- Índices
CREATE INDEX index_Lancamento_uuid ON lancamento(uuid);
CREATE INDEX index_Lancamento_syncStatus ON lancamento(syncStatus);
CREATE INDEX index_Lancamento_usuarioId ON lancamento(usuarioId);
CREATE INDEX index_Lancamento_contaId ON lancamento(contaId);
CREATE INDEX index_Lancamento_categoriaId ON lancamento(categoriaId);
CREATE INDEX index_Lancamento_data ON lancamento(data);
CREATE INDEX index_Lancamento_tipo ON lancamento(tipo);
CREATE INDEX index_Lancamento_isDeleted ON lancamento(isDeleted);
```

1.8.1.2.4 Tabela: lancamento **Soft Delete:** O campo `isDeleted` implementa exclusão lógica: - `0`: Lançamento ativo - `1`: Lançamento deletado (mantido para sincronização)

1.8.2 Servidor - MySQL

1.8.2.1 Configuração

- **Banco:** MySQL 8.0+
- **Nome do Banco:** finanza_db
- **Charset:** utf8mb4
- **Collation:** utf8mb4_unicode_ci
- **Engine:** InnoDB (transações ACID)

1.8.2.2 Tabelas e Campos Completos

```
CREATE TABLE IF NOT EXISTS usuario (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(150) UNIQUE NOT NULL,
    senha_hash VARCHAR(255) NOT NULL,        -- SHA-256 Base64
    tipo_usuario ENUM('admin', 'usuario') NOT NULL DEFAULT 'usuario',
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    data_atualizacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    INDEX idx_email (email),
```

```
    INDEX idx_tipo_usuario (tipo_usuario)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.8.2.2.1 Tabela: usuario Tipos de Usuário: - `admin`: Administrador com acesso total via desktop - `usuario`: Usuário comum com acesso mobile

Usuários Padrão: - Admin: `admin@finanza.com` / senha: `admin123` - Teste: `teste1@gmail.com` / senha: `teste123`

```
CREATE TABLE IF NOT EXISTS conta (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tipo ENUM('corrente', 'poupanca', 'cartao', 'investimento', 'dinheiro') NOT NULL
    saldo_inicial DECIMAL(10,2) DEFAULT 0.00,
    id_usuario INT NOT NULL,
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (id_usuario) REFERENCES usuario(id) ON DELETE CASCADE,
    INDEX idx_conta_usuario (id_usuario),
    INDEX idx_conta_tipo (tipo)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.8.2.2.2 Tabela: conta Precisão Decimal: - `DECIMAL(10,2)`: Até 99.999.999,99 (10 dígitos, 2 casas decimais)

```
CREATE TABLE IF NOT EXISTS categoria (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tipo ENUM('receita', 'despesa') NOT NULL,
    id_usuario INT NOT NULL,
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (id_usuario) REFERENCES usuario(id) ON DELETE CASCADE,
    INDEX idx_categoria_usuario (id_usuario),
    INDEX idx_categoria_tipo (tipo),
    UNIQUE INDEX idx_categoria_nome_usuario (nome, id_usuario, tipo)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.8.2.2.3 Tabela: categoria

```
CREATE TABLE IF NOT EXISTS movimentacao (
    id INT AUTO_INCREMENT PRIMARY KEY,
    valor DECIMAL(10,2) NOT NULL,
    data DATE NOT NULL,
    descricao TEXT,
    tipo ENUM('receita', 'despesa') NOT NULL,
    id_conta INT NOT NULL,
    id_categoria INT NOT NULL,
    id_usuario INT NOT NULL,
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    data_atualizacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    FOREIGN KEY (id_conta) REFERENCES conta(id) ON DELETE CASCADE,
    FOREIGN KEY (id_categoria) REFERENCES categoria(id) ON DELETE CASCADE,
    FOREIGN KEY (id_usuario) REFERENCES usuario(id) ON DELETE CASCADE,
    INDEX idx_movimentacao_data (data),
    INDEX idx_movimentacao_usuario (id_usuario),
    INDEX idx_movimentacao_conta (id_conta),
    INDEX idx_movimentacao_categoria (id_categoria),
    INDEX idx_movimentacao_tipo (tipo)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.8.2.2.4 Tabela: movimentacao Índices para Performance: - `idx_movimentacao_data`: Otimiza consultas por período - `idx_movimentacao_usuario`: Otimiza consultas por usuário - `idx_movimentacao_conta`: Otimiza consultas por conta - `idx_movimentacao_categoria`: Otimiza consultas por categoria - `idx_movimentacao_tipo`: Otimiza somas por tipo

1.8.2.3 Script de Migração

1.8.2.3.1 migration_add_tipo_usuario.sql Adiciona campo `tipo_usuario` para diferenciar admins e usuários comuns:

```
ALTER TABLE usuario ADD COLUMN tipo_usuario ENUM('admin', 'usuario') NOT NULL DEFAUL
UPDATE usuario SET tipo_usuario = 'admin' WHERE email = 'admin@finanza.com';
```

1.8.3 Relacionamentos e Integridade Referencial

```
usuario (1) ──< (N) conta
usuario (1) ──< (N) categoria
usuario (1) ──< (N) movimentacao
conta (1) ──< (N) movimentacao
categoria (1) ──< (N) movimentacao
```

Cascade Delete: - Ao deletar usuário: Remove todas suas contas, categorias e movimentações - Ao deletar conta: Remove todas as movimentações da conta - Ao deletar categoria: Remove todas as movimentações da categoria

1.8.4 Backup e Recuperação

Mobile (Room/SQLite): - Backup automático via Android Backup Service - Export manual para arquivo .db - Sincronização com servidor funciona como backup

Servidor (MySQL): - Backup diário recomendado via mysqldump - Replicação master-slave para alta disponibilidade - Binary logs para point-in-time recovery

Comando de Backup MySQL:

```
mysqldump -u root -p finanza_db > backup_finanza_$(date +%Y%m%d).sql
```

1.9 🔌 COMUNICAÇÃO

1.9.1 Protocolo TCP/IP

- **Porta padrão:** 12345
- **Formato:** Texto com delimitadores
- **Estrutura:** COMANDO|PARAM1|PARAM2|...
- **Resposta:** SUCCESS|dados ou ERROR|mensagem

1.9.2 Comandos Principais

- LOGIN|email|senha
- REGISTER|nome|email|senha
- SYNC_CONTAS|usuario_id
- SYNC_CATEGORIAS|usuario_id
- SYNC_LANCAMENTOS|usuario_id
- CREATE_CONTA|dados
- UPDATE_CONTA|dados
- DELETE_CONTA|id
- (similar para categorias e lançamentos)

1.10 🔒 SEGURANÇA

- **Senhas:** Hash SHA-256 via SecurityUtil
- **Comunicação:** Socket TCP/IP (pode ser atualizado para SSL/TLS)
- **Autenticação:** Email + senha hash
- **Sessão:** Mantida enquanto conexão ativa

1.11 🚀 EXECUÇÃO

1.11.1 Mobile

1.11.1.1 Requisitos

- **Android Studio:** Arctic Fox ou superior
- **JDK:** Java 11 ou superior
- **Android SDK:** API 24-36
- **Gradle:** 8.12.3 (incluído no wrapper)
- **Dispositivo/Emulador:** Android 7.0+ (API 24+)

1.11.1.2 Passos de Instalação

1. **Clonar o Repositório**

```
git clone https://github.com/KallebySchultz/FinanzaCompleto.git
cd FinanzaCompleto
```

2. **Abrir no Android Studio**

- File > Open > Selecionar pasta do projeto
- Aguardar sincronização do Gradle
- Resolver dependências automaticamente

3. **Configurar IP do Servidor**

   Editar `app/src/main/java/com/example/finanza/network/ServerClient.java`:

```
// Linha 15-20 (aproximadamente)
private static final String SERVER_HOST = "192.168.1.100"; // Substituir pelo IP
private static final int SERVER_PORT = 12345;
```

   Importante: Use o IP da máquina onde o servidor está rodando:

   - Localhost não funciona no emulador (usar 10.0.2.2)
   - Dispositivo físico: IP da rede local (192.168.x.x)

4. **Compilar o Projeto**

   - Build > Make Project (Ctrl+F9)
   - Verificar erros no Build Output
   - Garantir que todas as dependências foram baixadas

5. **Executar no Dispositivo/Emulador**

   - Run > Run 'app' (Shift+F10)
   - Selecionar dispositivo/emulador
   - Aguardar instalação do APK

6. **Gerar APK para Distribuição**

```
./gradlew assembleRelease
# APK gerado em: app/build/outputs/apk/release/app-release-unsigned.apk
```

1.11.2 Desktop (Servidor)

1.11.2.1 Requisitos

- **JDK:** Java 17 ou superior
- **MySQL:** 8.0 ou superior
- **IDE:** NetBeans, IntelliJ IDEA ou Eclipse
- **SO:** Windows, Linux ou macOS

1.11.2.2 Passos de Instalação

1. **Configurar MySQL**

```
# Instalar MySQL
sudo apt-get install mysql-server  # Linux
brew install mysql                 # macOS
# Windows: Baixar instalador oficial

# Acessar MySQL
mysql -u root -p
```

2. **Executar Script de Criação do Banco**

```
-- No MySQL shell
source /caminho/para/DESKTOP\ VERSION/banco/script_inicial.sql;
-- ou
mysql -u root -p < "DESKTOP VERSION/banco/script_inicial.sql"
```

3. **Configurar Conexão do Servidor**

   Editar `DESKTOP VERSION/ServidorFinanza/src/util/DatabaseUtil.java`:

   ```
   // Linhas 10-15 (aproximadamente)
   private static final String URL = "jdbc:mysql://localhost:3306/finanza_db";
   private static final String USER = "root";
   private static final String PASSWORD = "sua_senha_mysql";
   ```

4. **Compilar e Executar o Servidor**

   Via NetBeans:

   - File > Open Project > Selecionar ServidorFinanza
   - Clean and Build (Shift+F11)
   - Run Project (F6)

   Via Linha de Comando:

   ```
   cd "DESKTOP VERSION/ServidorFinanza"
   ant clean
   ant jar
   java -jar dist/ServidorFinanza.jar
   ```

5. **Verificar Servidor em Execução**

   O servidor estará escutando na porta 12345:

   ```
   [INFO] Servidor Finanza iniciado na porta 12345
   [INFO] Aguardando conexões de clientes...
   ```

1.11.3 Desktop (Cliente Admin)

1.11.3.1 Passos de Instalação

1. **Configurar IP do Servidor**

   Editar `DESKTOP VERSION/ClienteFinanza/src/util/NetworkClient.java`:

   ```
   private static final String SERVER_HOST = "localhost"; // ou IP do servidor
   private static final int SERVER_PORT = 12345;
   ```

2. **Compilar e Executar**

   Via NetBeans:

   - File > Open Project > Selecionar ClienteFinanza
   - Clean and Build
   - Run Project

   Via Linha de Comando:

   ```
   cd "DESKTOP VERSION/ClienteFinanza"
   ant clean
   ```

```
ant jar
java -jar dist/ClienteFinanza.jar
```

3. **Fazer Login como Admin**
   - Email: `admin@finanza.com`
   - Senha: `admin123`

1.12 □ CONFIGURAÇÃO E INSTALAÇÃO

1.12.1 Configuração de Rede

1.12.1.1 Firewall (Linux/Windows)

```
# Linux - Permitir porta 12345
sudo ufw allow 12345/tcp

# Windows - Adicionar regra no Firewall do Windows
# Painel de Controle > Firewall > Regras de Entrada > Nova Regra
# Tipo: Porta
# Protocolo: TCP
# Porta: 12345
```

1.12.1.2 Verificar Conectividade

```
# Testar se servidor está escutando
netstat -an | grep 12345
# ou
lsof -i :12345

# Testar conexão do cliente
telnet IP_DO_SERVIDOR 12345
# ou
nc -zv IP_DO_SERVIDOR 12345
```

1.12.2 Variáveis de Ambiente

1.12.2.1 Servidor

```
# Linux/macOS
export FINANZA_DB_HOST=localhost
export FINANZA_DB_PORT=3306
export FINANZA_DB_NAME=finanza_db
export FINANZA_DB_USER=root
export FINANZA_DB_PASSWORD=senha
export FINANZA_SERVER_PORT=12345

# Windows
set FINANZA_DB_HOST=localhost
```

```
set FINANZA_DB_PORT=3306
# ... (continuar)
```

1.12.3 Logs e Depuração

1.12.3.1 Mobile (Logcat)

```
# Filtrar logs do Finanza
adb logcat | grep Finanza

# Ver apenas erros
adb logcat *:E | grep Finanza

# Salvar logs em arquivo
adb logcat > finanza_mobile.log
```

1.12.3.2 Servidor (server.log)

```
# Visualizar logs em tempo real
tail -f DESKTOP\ VERSION/ServidorFinanza/server.log

# Buscar erros
grep ERROR DESKTOP\ VERSION/ServidorFinanza/server.log
```

1.13 TROUBLESHOOTING

1.13.1 Problemas Comuns e Soluções

1.13.1.1 1. Mobile não conecta ao servidor

Sintomas: - “Erro de conexão” ao fazer login - Timeout ao sincronizar - “Servidor indisponível”

Soluções:

```
1. Verificar se servidor está rodando (porta 12345)
2. Verificar IP configurado no ServerClient.java
   - Emulador: usar 10.0.2.2 (não localhost)
   - Dispositivo físico: usar IP da rede local
3. Verificar firewall não está bloqueando porta 12345
4. Testar conectividade: ping IP_DO_SERVIDOR
5. Verificar dispositivo está na mesma rede do servidor
```

Código para Debug:

```
// Adicionar em ServerClient.java
Log.d("Finanza", "Tentando conectar: " + SERVER_HOST + ":" + SERVER_PORT);
Log.d("Finanza", "Timeout configurado: " + TIMEOUT_MS + "ms");
```

1.13.1.2 2. Erro de autenticação/senha incorreta

Sintomas: - “Credenciais inválidas” - Login não funciona com senha correta

Soluções:

```
1. Verificar se senha está sendo hashada corretamente (SHA-256)
2. Conferir encoding (Base64 vs Hex)
3. Verificar se usuário existe no banco:
   SELECT * FROM usuario WHERE email = 'seu@email.com';
4. Resetar senha do usuário admin:
  UPDATE usuario SET senha_hash = 'jZae727K08KaOmKSgOaGzww/XVqGr/PKEgIMkjrcbJI='
   WHERE email = 'admin@finanza.com';
   (senha: admin123)
```

1.13.1.3 3. Erro de sincronização/conflitos

Sintomas: - Dados não sincronizam - “Conflito detectado” - Dados duplicados

Soluções:

```
1. Limpar cache do app: Settings > Apps > Finanza > Clear Data
2. Verificar status de sincronização no banco:
   SELECT uuid, syncStatus FROM conta WHERE syncStatus != 1;
3. Forçar resync completo (deletar e recriar dados)
4. Verificar UUIDs únicos:
   SELECT uuid, COUNT(*) FROM conta GROUP BY uuid HAVING COUNT(*) > 1;
5. Resolver conflitos manualmente via ConflictResolutionManager
```

1.13.1.4 4. Banco de dados MySQL não conecta

Sintomas: - “Access denied for user” - “Unknown database ‘finanza_db’ ” - “Communications link failure”

Soluções:

```
1. Verificar MySQL está rodando:
   sudo systemctl status mysql    # Linux
   brew services list              # macOS
   services.msc                    # Windows

2. Verificar credenciais no DatabaseUtil.java

3. Criar banco se não existe:
   mysql -u root -p
   CREATE DATABASE finanza_db;

4. Conceder permissões:
   GRANT ALL PRIVILEGES ON finanza_db.* TO 'root'@'localhost';
   FLUSH PRIVILEGES;

5. Verificar firewall MySQL:
   sudo ufw allow 3306/tcp
```

1.13.1.5 5. Gradle build falha no mobile

Sintomas: - “Could not resolve dependencies” - “Failed to download” - “Unsupported class file major version”

Soluções:

```
1. Limpar e rebuildar:
   ./gradlew clean
   ./gradlew build --refresh-dependencies

2. Sincronizar projeto com Gradle:
   File > Sync Project with Gradle Files

3. Invalidar cache:
   File > Invalidate Caches and Restart

4. Verificar versão do JDK (deve ser Java 11):
   java -version

5. Atualizar Gradle wrapper:
   ./gradlew wrapper --gradle-version=8.12.3
```

1.13.1.6 6. Room database migration error

Sintomas: - “IllegalStateException: Room cannot verify the data integrity” - “Migration didn’t properly handle”

Soluções:

```
1. Desinstalar e reinstalar app (perde dados locais)
2. Implementar estratégia de migração:
   .fallbackToDestructiveMigration()

3. Ou forçar sync completo após reinstalação
```

1.13.1.7 7. Servidor não aceita conexões

Sintomas: - Servidor inicia mas não aceita clientes - “Connection refused”

Soluções:

```
1. Verificar porta não está em uso:
   netstat -an | grep 12345
   lsof -i :12345

2. Matar processo na porta:
   kill -9 $(lsof -t -i:12345)

3. Verificar bind address (0.0.0.0 vs localhost)

4. Verificar logs do servidor para erros

5. Testar com cliente de teste:
   telnet localhost 12345
```

1.13.1.8 8. Lentidão na sincronização

Sintomas: - Sincronização muito lenta - App congela durante sync

Soluções:

```
1. Usar sincronização incremental ao invés de full sync
2. Reduzir frequência de sincronização automática
3. Sincronizar em background thread:
   new Thread(() -> syncService.startSync()).start();
4. Otimizar queries SQL:
   - Adicionar índices
   - Usar transações em batch
5. Implementar paginação para listas grandes
```

1.13.1.9 9. APK não instala no dispositivo

Sintomas: - "App not installed" - "Parse error"

Soluções:

```
1. Habilitar "Unknown sources" nas configurações
2. Verificar assinatura do APK
3. Desinstalar versão anterior
4. Verificar compatibilidade da versão Android
5. Limpar cache do Package Installer
```

1.13.1.10 10. Dados não aparecem após login

Sintomas: - Login bem-sucedido mas telas vazias - Dados não carregam do servidor

Soluções:

```
1. Verificar logs para erros de parsing
2. Conferir formato de resposta do servidor
3. Verificar se DAOs estão retornando dados:
   List<Conta> contas = contaDao.listarPorUsuario(usuarioId);
   Log.d("Finanza", "Contas encontradas: " + contas.size());
4. Forçar refresh da UI
5. Verificar se usuário tem dados no servidor:
   SELECT * FROM conta WHERE id_usuario = X;
```

1.13.2 Logs Úteis para Debug

1.13.2.1 Mobile

```
// Em cada Activity
private static final String TAG = "Finanza_NomeDaActivity";
Log.d(TAG, "Método executado");
Log.e(TAG, "Erro: " + e.getMessage(), e);

// Sync
Log.d("FinanzaSync", "Iniciando sincronização...");
```

```
Log.d("FinanzaSync", "Pendentes: " + pendentes.size());
Log.d("FinanzaSync", "Sincronização concluída com sucesso");
```

1.13.2.2 Servidor

```
// Em ClientHandler
System.out.println("[" + new Date() + "] Comando recebido: " + command);
System.out.println("[" + new Date() + "] Resposta enviada: " + response);
System.err.println("[ERROR] " + e.getMessage());
```

1.13.3 Ferramentas de Diagnóstico

1.13.3.1 Android Debug Bridge (ADB)

```
# Listar dispositivos conectados
adb devices

# Instalar APK
adb install app-debug.apk

# Visualizar banco de dados
adb shell
cd /data/data/com.example.finanza/databases
sqlite3 finanza.db
.tables
SELECT * FROM usuario;

# Pull banco para análise
adb pull /data/data/com.example.finanza/databases/finanza.db
```

1.13.3.2 MySQL Debug

```
-- Verificar dados
SELECT COUNT(*) FROM usuario;
SELECT COUNT(*) FROM conta;
SELECT COUNT(*) FROM movimentacao;

-- Verificar índices
SHOW INDEX FROM movimentacao;

-- Analisar queries lentas
SHOW PROCESSLIST;
EXPLAIN SELECT * FROM movimentacao WHERE id_usuario = 1;

-- Habilitar query log
SET GLOBAL general_log = 'ON';
SET GLOBAL log_output = 'TABLE';
SELECT * FROM mysql.general_log;
```

1.14 🤝 SUPORTE E CONTRIBUIÇÃO

1.14.1 Contato

- **Desenvolvedor:** Kalleby Schultz
- **Instituição:** IFSUL - Campus Venâncio Aires
- **Projeto:** Trabalho Interdisciplinar 4º ano - Técnico em Informática

1.14.2 Contribuindo

Este é um projeto acadêmico. Sugestões e melhorias são bem-vindas através de: - Issues no GitHub - Pull Requests com melhorias - Documentação adicional

1.14.3 Licença

Projeto desenvolvido para fins acadêmicos e educacionais. Uso livre para aprendizado e referência.
