📋 MAPEAMENTO COMPLETO

Sistema de Controle Financeiro Finanza

IFSUL - Campus Venâncio Aires

Documentação Técnica Detalhada com Fluxos Completos de Código

📋 MAPEAMENTO COMPLETO DO SISTEMA FINANZA

📑 ÍNDICE

🎓 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).

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)

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)	-

📱 VERSÃO MOBILE (Android)

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

Estrutura de Diretórios

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
```

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
```

Permissões do Manifesto

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

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"
}
```

Fluxo de Funcionalidades - Mobile

1. LoginActivity.java

Função: Tela de autenticação do usuário

Fluxo Detalhado de Execução: 1. **Interface do Usuário (activity_login.xml)** - Usuário insere email no campo `EditText etEmail` - Usuário insere senha no campo `EditText etSenha` - Usuário clica no botão `Button btnLogin`

1. **Evento onClick do botão (LoginActivity.java)**
2. Método `btnLogin.setOnClickListener()` é acionado
3. Valida campos não vazios
4. Extrai texto dos campos: `String email = etEmail.getText().toString()`
5. Extrai senha: `String senha = etSenha.getText().toString()`
6. **Chamada ao AuthManager (AuthManager.java)**
7. **LoginActivity chama:** `AuthManager.getInstance().login(email, senha)`
8. **AuthManager executa:** Hash SHA-256 da senha usando `SecurityUtil.hashPassword(senha)`
9. **AuthManager cria:** String de comando formatada usando `Protocol.CMD_LOGIN`

10. **Comunicação com Servidor (ServerClient.java)**
    11. **AuthManager chama:** `ServerClient.sendCommand(Protocol.CMD_LOGIN, email + "|" + senhaHash)`
    12. **ServerClient executa:** Abre socket TCP na porta 12345
    13. **ServerClient envia:** Comando formatado "LOGIN|email@exemplo.com|hash_sha256"
    14. **ServerClient aguarda:** Resposta do servidor via `BufferedReader.readLine()`
15. **Processamento da Resposta**
    16. **Se resposta == "OK|{dados_usuario}":**
        17. Parse JSON dos dados do usuário
        18. Cria objeto `Usuario` com os dados recebidos
        19. **AuthManager chama:** `UsuarioDao.inserir(usuario)` para salvar localmente
        20. **AuthManager retorna:** true para LoginActivity
    21. **Se resposta == "ERROR|mensagem":**
        22. **AuthManager retorna:** false
        23. LoginActivity exibe Toast com erro
24. **Redirecionamento (LoginActivity.java)**
    25. **LoginActivity executa:** `Intent intent = new Intent(this, MenuActivity.class)`
    26. **LoginActivity chama:** `intent.putExtra("usuario_id", usuario.getId())`
    27. **LoginActivity inicia:** `startActivity(intent)`
    28. **LoginActivity finaliza:** `finish()` para remover da pilha

Resumo da Cadeia de Chamadas:

```
LoginActivity.onClick()
  → AuthManager.login()
    → SecurityUtil.hashPassword()
    → ServerClient.sendCommand()
      → ServerSocket envia via TCP
      → Servidor processa (ClientHandler)
      → Resposta retorna
    → UsuarioDao.inserir()
  → Intent(MenuActivity)
  → startActivity()
```

2. RegisterActivity.java

Função: Cadastro de novos usuários

Fluxo Detalhado de Execução: 1. **Interface do Usuário (activity_register.xml)** - Usuário insere nome no `EditText etNome` - Usuário insere email no `EditText etEmail` - Usuário insere senha no `EditText etSenha` - Usuário confirma senha no `EditText etConfirmarSenha` - Usuário clica no botão `Button btnRegistrar`

1. **Validação de Dados (RegisterActivity.java)**
    2. Método `btnRegistrar.setOnClickListener()` é acionado
    3. **Valida:** Campos não vazios - `if (nome.isEmpty() || email.isEmpty() || senha.isEmpty())`
    4. **Valida:** Formato de email - `if (!Patterns.EMAIL_ADDRESS.matcher(email).matches())`
    5. **Valida:** Senhas coincidem - `if (!senha.equals(confirmarSenha))`
    6. **Valida:** Tamanho mínimo da senha - `if (senha.length() < 6)`
7. **Preparação dos Dados**
    8. Extrai valores: `String nome = etNome.getText().toString().trim()`
    9. Gera UUID único: `String uuid = UUID.randomUUID().toString()`
    10. **Chama SecurityUtil:** `String senhaHash = SecurityUtil.hashPassword(senha)`
    11. Obtém timestamp: `long timestamp = System.currentTimeMillis()`
12. **Criação do Comando (Protocol.java)**
    13. **RegisterActivity monta:** Parâmetros separados por "|"
    14. Formato: `"REGISTER|" + nome + "|" + email + "|" + senhaHash + "|" + uuid + "|" + timestamp`
15. **Envio ao Servidor (ServerClient.java)**
    16. **RegisterActivity chama:** `ServerClient.sendCommand(Protocol.CMD_REGISTER, params)`
    17. **ServerClient executa:**
        18. Estabelece conexão TCP com servidor (porta 12345)
        19. Envia comando via `PrintWriter.println()`
        20. Aguarda resposta via `BufferedReader.readLine()`
        21. Define timeout de 30 segundos

22. **Processamento da Resposta**

23. **Se resposta == "OK|id_usuario":**
    24. Exibe Toast: "Cadastro realizado com sucesso!"
    25. Cria Intent para LoginActivity
    26. Preenche email automaticamente: `intent.putExtra("email", email)`
    27. **RegisterActivity chama:** `startActivity(intent)`
    28. **RegisterActivity finaliza:** `finish()`

29. **Se resposta == "ERROR|USER_EXISTS":**
    30. Exibe Toast: "Email já cadastrado no sistema"
    31. Mantém na tela de registro

32. **Se resposta == "ERROR|mensagem":**
    33. Exibe Toast com a mensagem de erro
    34. Permite nova tentativa

Resumo da Cadeia de Chamadas:

```
RegisterActivity.onClick()
  → Validações locais
  → UUID.randomUUID()
  → SecurityUtil.hashPassword()
  → Protocol.formatCommand(CMD_REGISTER)
  → ServerClient.sendCommand()
    → Socket TCP conecta ao servidor
    → Envia: "REGISTER|nome|email|hash|uuid|timestamp"
    → Aguarda resposta
  → Parse da resposta
  → Intent(LoginActivity)
  → startActivity() e finish()
```

3. MenuActivity.java (Dashboard)

Função: Tela principal com visão geral financeira

Fluxo Detalhado de Execução:

1. **Inicialização da Activity (onCreate)**

2. **MenuActivity recebe:** Intent com `usuario_id` via `getIntent().getIntExtra("usuario_id", -1)`
3. **MenuActivity carrega:** Layout `activity_menu.xml` com `setContentView()`
4. **MenuActivity obtém:** Referências dos componentes (TextViews, Buttons, CardViews)

5. **MenuActivity inicializa:** Instância do banco Room: `AppDatabase.getInstance(this)`

6. **Busca de Dados Financeiros**

7. **MenuActivity obtém DAOs:**

```
ContaDao contaDao = AppDatabase.getInstance(this).contaDao();
LancamentoDao lancamentoDao = AppDatabase.getInstance(this).lancamentoDao();
```

8. **MenuActivity chama:** `contaDao.listarPorUsuario(usuarioId)`

9. **ContaDao executa:** Query SQL `SELECT * FROM conta WHERE usuarioId = ?`

10. **Room retorna:** `List<Conta>` com todas as contas do usuário

11. **MenuActivity calcula:** Saldo total iterando sobre as contas

```
double saldoTotal = 0.0;
for (Conta conta : contas) {
    saldoTotal += conta.getSaldoAtual();
}
```

12. **Cálculo de Receitas e Despesas**

13. **MenuActivity chama:** `lancamentoDao.somaPorTipo("receita", usuarioId)`

14. **LancamentoDao executa:** `SELECT SUM(valor) FROM lancamento WHERE tipo = 'receita' AND usuarioId = ? AND isDeleted = 0`

15. **Room retorna:** Double com soma total de receitas

16. **MenuActivity chama:** `lancamentoDao.somaPorTipo("despesa", usuarioId)`

17. **LancamentoDao executa:** `SELECT SUM(valor) FROM lancamento WHERE tipo = 'despesa' AND usuarioId = ? AND isDeleted = 0`

18. **Room retorna:** Double com soma total de despesas

19. **Atualização da Interface**

20. **MenuActivity atualiza TextViews:**

```
tvSaldoTotal.setText(String.format("R$ %.2f", saldoTotal));
tvTotalReceitas.setText(String.format("R$ %.2f", totalReceitas));
tvTotalDespesas.setText(String.format("R$ %.2f", totalDespesas));
tvSaldoMensal.setText(String.format("R$ %.2f", totalReceitas - totalDespesas));
```

21.
Define cores: Verde para positivo, Vermelho para negativo

22.
Inicialização da Sincronização (SyncService.java)

23. **MenuActivity cria thread:** `new Thread(() -> { ... }).start()`
24. **Thread executa:** `SyncService.startSync(usuarioId, context)`
25. **SyncService executa em background:**
    26. Verifica conectividade de rede
    27. Busca dados pendentes: `contaDao.obterPendentesSyncPorUsuario(usuarioId)`
    28. Para cada registro pendente, envia ao servidor
    29. Busca atualizações do servidor desde último sync
    30. Aplica atualizações localmente
    31. Notifica via `runOnUiThread()` para atualizar UI

32.
Configuração de Navegação

33.
Botão Contas (btnContas):

34. onClick → `Intent intent = new Intent(MenuActivity.this, AccountsActivity.class)`
35. Passa usuarioId → `intent.putExtra("usuario_id", usuarioId)`
36. Inicia Activity → `startActivity(intent)`

37.
Botão Movimentações (btnMovimentos):

38. onClick → Inicia `MovementsActivity.class`

39.
Botão Categorias (btnCategorias):

40. onClick → Inicia `CategoriaActivity.class`

41.
Botão Perfil (btnPerfil):

42. onClick → Inicia `ProfileActivity.class`

43. **Botão Configurações (btnSettings):**

    44. onClick → Inicia `SettingsActivity.class`

45. **Atualização ao Retomar (onResume)**

    46. **MenuActivity.onResume() chama:** Métodos de atualização novamente
    47. Recarrega dados do banco para refletir mudanças feitas em outras telas
    48. Recalcula saldos e totais
    49. Atualiza interface

Resumo da Cadeia de Chamadas:

```
MenuActivity.onCreate()
  → AppDatabase.getInstance()
    → ContaDao.listarPorUsuario()
      → Room executa SELECT em SQLite
    → LancamentoDao.somaPorTipo("receita")
      → Room executa SUM query
    → LancamentoDao.somaPorTipo("despesa")
      → Room executa SUM query
  → Atualiza TextViews da UI
  → new Thread().start()
    → SyncService.startSync()
      → ContaDao.obterPendentesSyncPorUsuario()
      → ServerClient.sendCommand() para cada pendente
      → Atualiza registros locais
      → runOnUiThread() atualiza UI
  → Configura listeners dos botões
    → onClick → startActivity(nova Activity)
```

4. AccountsActivity.java

Função: Gerenciamento de contas bancárias

Fluxo Detalhado de Execução:

1. **Inicialização e Carregamento de Dados (onCreate)**

    2. **AccountsActivity recebe:** `usuario_id` do Intent
    3. **AccountsActivity inicializa:** RecyclerView para lista de contas
    4. **AccountsActivity obtém DAO:** `ContaDao contaDao = AppDatabase.getInstance(this).contaDao()`

5. **AccountsActivity chama:** `contaDao.listarPorUsuario(usuarioId)`
6. **ContaDao executa:** `SELECT * FROM conta WHERE usuarioId = ? AND syncStatus != 4 ORDER BY dataCriacao DESC`
7. **Room retorna:** `List<Conta>` ordenada por data de criação
8. **AccountsActivity cria:** Adapter do RecyclerView com lista de contas
9. **AccountsActivity define:** Adapter no RecyclerView: `recyclerView.setAdapter(adapter)`

10. **Adicionar Nova Conta (FloatingActionButton)**

11. **Usuário clica:** FAB "+" no canto inferior direito
12. **AccountsActivity exibe:** Dialog personalizado `dialog_add_account.xml`
13. **Dialog contém:**

14. EditText para nome da conta
15. Spinner para tipo (corrente, poupança, cartão, investimento, dinheiro)
16. EditText para saldo inicial
17. Botões Cancelar e Salvar

18. **Quando usuário clica "Salvar":**

19. **AccountsActivity valida:** Campos não vazios
20. **AccountsActivity valida:** Saldo é número válido
21. **AccountsActivity cria objeto Conta:**

```
Conta novaConta = new Conta();
novaConta.setUuid(UUID.randomUUID().toString());
novaConta.setNome(nome);
novaConta.setTipo(tipo);
novaConta.setSaldoInicial(saldo);
novaConta.setSaldoAtual(saldo);
novaConta.setUsuarioId(usuarioId);
novaConta.setDataCriacao(System.currentTimeMillis());
novaConta.setLastModified(System.currentTimeMillis());
novaConta.setSyncStatus(2); // NEEDS_SYNC
```

22. **AccountsActivity executa em thread:**

```
new Thread(() -> {
    long id = contaDao.inserir(novaConta);
    runOnUiThread(() -> {
        // Atualiza lista
        recarregarContas();
        // Inicia sincronização
        sincronizarConta(novaConta);
    });
}).start();
```

23.
Editar Conta Existente

24. **Usuário clica:** Em uma conta na lista
25. **RecyclerView.Adapter chama:** `onItemClick(conta)`
26. **AccountsActivity exibe:** Dialog `dialog_edit_account.xml` preenchido
27.
Dialog carrega: Dados da conta selecionada
28.
Quando usuário salva alterações:
29.
AccountsActivity atualiza campos:

```
conta.setNome(novoNome);
conta.setSaldoInicial(novoSaldo);
conta.setLastModified(System.currentTimeMillis());
conta.setSyncStatus(2); // Marca para sincronização
```

30.
AccountsActivity chama: `contaDao.atualizar(conta)`
31. **ContaDao executa:** `UPDATE conta SET nome=?, saldoInicial=?, lastModified=?, syncStatus=? WHERE id=?`
32. **Room confirma:** Atualização bem-sucedida
33. **AccountsActivity recarrega:** Lista de contas
34. **AccountsActivity inicia:** Sincronização via `EnhancedSyncService`

35.
Excluir Conta

36. **Usuário mantém pressionado:** Item da conta na lista (long click)
37. **AccountsActivity exibe:** Dialog de confirmação `dialog_delete_account.xml`
38.

Dialog mostra: "Tem certeza que deseja excluir a conta X? Todas as movimentações serão removidas."

39.
Se usuário confirma:

40.
AccountsActivity verifica: Se há lançamentos associados

```
int countLancamentos = lancamentoDao.listarPorConta(conta.getId()).size();
```

41.
Se houver lançamentos:

42. Exibe aviso adicional: "Esta conta possui X movimentações"
43. Solicita confirmação extra

44.
AccountsActivity executa exclusão:

```
new Thread(() -> {
    // Exclui lançamentos da conta (CASCADE)
    lancamentoDao.excluirPorConta(conta.getId());
    // Exclui a conta
    contaDao.deletar(conta);
    // Sincroniza exclusão com servidor
    sincronizarExclusao(conta.getUuid());
    runOnUiThread(() -> recarregarContas());
}).start();
```

45.
Cálculo e Atualização de Saldos

46.
Para cada conta na lista:

47.
AccountsActivity chama: `lancamentoDao.saldoPorConta(contaId, usuarioId)`

48. **LancamentoDao executa:**

```
SELECT
  SUM(CASE WHEN tipo='receita' THEN valor ELSE 0 END) -
  SUM(CASE WHEN tipo='despesa' THEN valor ELSE 0 END)
FROM lancamento
WHERE contaId = ? AND usuarioId = ? AND isDeleted = 0
```

49.

AccountsActivity calcula: `saldoAtual = saldoInicial + saldoMovimentacoes`

50. **Se saldo mudou:**
    51. **AccountsActivity atualiza:** `conta.setSaldoAtual(novoSaldo)`
    52. **AccountsActivity chama:** `contaDao.atualizar(conta)`

53. **Adapter exibe:** Saldo atualizado com formatação de moeda

54. **Adapter aplica cores:** Verde para positivo, Vermelho para negativo

55. **Sincronização com Servidor (EnhancedSyncService)**

56. **AccountsActivity inicia:** Sincronização em background

57. **EnhancedSyncService.syncContas() executa:**

    58. **Busca pendentes:** `contaDao.obterPendentesSyncPorUsuario(usuarioId)`

    59. **Para cada conta pendente:**

        60. Verifica se é nova (syncStatus = 2) ou atualizada

        61. **Chama ServerClient:**

```
String comando = conta.getId() == 0 ?
    Protocol.CMD_ADD_CONTA_ENHANCED :
    Protocol.CMD_UPDATE_CONTA_ENHANCED;
String params = formatarContaParaSync(conta);
String resposta = ServerClient.sendCommand(comando, params);
```

        62. **Se resposta OK:**

            63. Parse do serverId: `int serverId = Integer.parseInt(resposta.split("|")[1])`

            64. **Atualiza metadados:** `contaDao.marcarComoSincronizado(conta.getId(), timestamp)`

            65. **Marca:** `conta.setSyncStatus(1)` // SYNCED

    66. **Busca atualizações do servidor:**

        67. **Chama:** `ServerClient.sendCommand(CMD_LIST_CHANGES_SINCE, lastSyncTime)`

68. **Recebe:** Lista de contas modificadas no servidor
69. **Para cada conta do servidor:**
    70. Verifica se existe localmente por UUID
    71. Se existir, aplica merge usando `ConflictResolutionManager`
    72. Se não existir, insere nova conta localmente

73. **Notifica UI:** Via callback ou broadcast para atualizar lista

Resumo da Cadeia de Chamadas: **Adicionar Conta:**

```
AccountsActivity.onClick(FAB)
  → Dialog.show()
  → Dialog.onClick(Salvar)
    → new Conta(dados)
    → UUID.randomUUID()
    → new Thread().start()
      → ContaDao.inserir(conta)
        → Room INSERT INTO conta
      → runOnUiThread()
        → recarregarContas()
        → EnhancedSyncService.syncConta()
          → ServerClient.sendCommand(ADD_CONTA_ENHANCED)
            → Socket TCP envia dados
          → ContaDao.marcarComoSincronizado()
```

Editar Conta:

```
RecyclerView.onClick(item)
  → Dialog.show(conta)
  → Dialog.onClick(Salvar)
    → conta.setNome/setSaldo/setLastModified()
    → conta.setSyncStatus(NEEDS_SYNC)
    → ContaDao.atualizar(conta)
      → Room UPDATE conta
    → EnhancedSyncService.syncConta()
      → ServerClient.sendCommand(UPDATE_CONTA_ENHANCED)
```

Excluir Conta:

```
RecyclerView.onLongClick(item)
  → Dialog.show(confirmação)
  → Dialog.onClick(Confirmar)
    → new Thread().start()
      → LancamentoDao.excluirPorConta()
      → ContaDao.deletar(conta)
        → Room DELETE FROM conta
      → ServerClient.sendCommand(DELETE_CONTA)
      → runOnUiThread() → recarregarContas()
```

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`

6. MovementsActivity.java

Função: Gerenciamento de lançamentos/transações

Fluxo Detalhado de Execução:

1. **Inicialização e Carregamento (onCreate)**
2. **MovementsActivity recebe:** `usuario_id` via Intent
3. **MovementsActivity inicializa:** RecyclerView, DAOs e Adapters
4. **MovementsActivity carrega dados:**

```
lancamentoDao = AppDatabase.getInstance(this).lancamentoDao();
contaDao = AppDatabase.getInstance(this).contaDao();
categoriaDao = AppDatabase.getInstance(this).categoriaDao();

// Carrega lançamentos ativos (não deletados)
List<Lancamento> lancamentos = lancamentoDao.listarAtivosPorUsuario(usuarioId);
```

5. **LancamentoDao executa:** `SELECT * FROM lancamento WHERE usuarioId = ? AND isDeleted = 0 ORDER BY data DESC`
6. **Room retorna:** Lista ordenada por data (mais recentes primeiro)
7. **Exibição na Interface**
8. **MovementsActivity cria:** Adapter customizado para RecyclerView
9. **Para cada lançamento, Adapter exibe:**
10. Descrição do lançamento

11. Valor formatado (R$ X,XX)
12. Data formatada (dd/MM/yyyy)
13. Nome da categoria (busca em memória ou cache)
14. Nome da conta (busca em memória ou cache)
15. Cor e ícone baseado no tipo (verde=receita, vermelho=despesa)

16. **Adapter agrupa por:** Mês/Ano para facilitar visualização

17. **Adicionar Novo Lançamento (FAB)**

18. **Usuário clica:** FloatingActionButton "+"
19. **MovementsActivity exibe:** Dialog `dialog_add_transaction_movements.xml`
20. **Dialog contém campos:**

21. EditText: Descrição
22. EditText: Valor (teclado numérico)
23. DatePicker: Data da transação
24. RadioGroup: Tipo (Receita/Despesa)
25. Spinner: Conta (carrega de `contaDao.listarPorUsuario()`)
26. Spinner: Categoria (filtra por tipo selecionado)

27. **Quando usuário seleciona tipo:**

28. **Dialog chama:** `categoriaDao.listarPorUsuarioETipo(usuarioId, tipo)`
29. **CategoriaDao executa:** `SELECT * FROM categoria WHERE usuarioId = ? AND tipo = ?`
30. **Room retorna:** Categorias filtradas (ex: só receitas)

31. **Dialog atualiza:** Spinner de categorias dinamicamente

32. **Quando usuário clica "Salvar":**

33. **MovementsActivity valida dados:**

```
if (descricao.isEmpty()) {
    Toast.show("Descrição é obrigatória");
    return;
}
if (valor <= 0) {
    Toast.show("Valor deve ser maior que zero");
    return;
}
if (contaSelecionada == null || categoriaSelecionada == null) {
    Toast.show("Selecione conta e categoria");
    return;
}
```

34.
MovementsActivity cria objeto:

```
Lancamento lancamento = new Lancamento();
lancamento.setUuid(UUID.randomUUID().toString());
lancamento.setDescricao(descricao);
lancamento.setValor(valor);
lancamento.setData(dataSelecionada.getTime());
lancamento.setTipo(tipo); // "receita" ou "despesa"
lancamento.setContaId(contaSelecionada.getId());
lancamento.setCategoriaId(categoriaSelecionada.getId());
lancamento.setUsuarioId(usuarioId);
lancamento.setDataCriacao(System.currentTimeMillis());
lancamento.setLastModified(System.currentTimeMillis());
lancamento.setSyncStatus(2); // NEEDS_SYNC
lancamento.setIsDeleted(0); // Ativo
```

35.
Validação de Integridade (DataIntegrityValidator)

36. **MovementsActivity chama:** `DataIntegrityValidator.validarLancamento(lancamento)`
37. **DataIntegrityValidator executa:**
    38. Verifica valor é positivo
    39. Verifica conta existe: `contaDao.buscarPorId(lancamento.getContaId())`
    40. Verifica categoria existe: `categoriaDao.buscarPorId(lancamento.getCategoriaId())`
    41. Verifica data não é futura (opcional)
    42. **Busca duplicatas:** `lancamentoDao.buscarDuplicata(valor, data, descricao, contaId, usuarioId)`
        43. **LancamentoDao executa:** Query complexa verificando similaridade
        44. Se encontrar duplicata exata nos últimos 5 minutos, retorna aviso
45. **Se validação falhar:** Exibe erro e impede salvamento

46. **Se passar:** Continua para inserção

47.
Inserção no Banco de Dados

48.
MovementsActivity executa em thread:

```
new Thread(() -> {
    try {
        long id = lancamentoDao.inserirSeguro(lancamento);

        // Atualiza saldo da conta
        Conta conta = contaDao.buscarPorId(lancamento.getContaId());
        double novoSaldo = conta.getSaldoAtual();
        if (lancamento.getTipo().equals("receita")) {
            novoSaldo += lancamento.getValor();
        } else {
            novoSaldo -= lancamento.getValor();
        }
        conta.setSaldoAtual(novoSaldo);
        conta.setLastModified(System.currentTimeMillis());
        conta.setSyncStatus(2); // Marca para sync
        contaDao.atualizar(conta);

        runOnUiThread(() -> {
            Toast.show("Lançamento adicionado com sucesso!");
            recarregarLancamentos();
            sincronizarDados();
        });
    } catch (Exception e) {
        runOnUiThread(() -> {
            Toast.show("Erro ao salvar: " + e.getMessage());
        });
    }
}).start();
```

49.
Editar Lançamento Existente

50. **Usuário clica:** Em um item da lista
51. **MovementsActivity exibe:** Dialog `dialog_edit_transaction.xml` preenchido
52. **Dialog carrega:** Dados do lançamento selecionado
53.
Dialog permite: Alterar todos os campos exceto UUID
54.
Quando salva alterações:

    55.
    MovementsActivity calcula: Diferença de valores para ajustar saldo

```
double valorAntigo = lancamentoOriginal.getValor();
double valorNovo = lancamento.getValor();
double diferenca = valorNovo - valorAntigo;
```

56.
Se mudou de conta:

57. Reverte impacto na conta antiga
58. Aplica impacto na conta nova

59.
Atualiza lançamento:

```
lancamento.setLastModified(System.currentTimeMillis());
lancamento.setSyncStatus(2);
lancamentoDao.atualizar(lancamento);
```

60.
Atualiza saldos das contas afetadas

61. **Recarrega lista e sincroniza**

62.
Excluir Lançamento (Soft Delete)

63. **Usuário long-press:** Item da lista
64. **MovementsActivity exibe:** Dialog de confirmação
65.
Dialog mostra: "Tem certeza que deseja excluir esta movimentação?"
66.
Se confirma:

```
new Thread(() -> {
    // Soft delete - marca como deletado
    lancamentoDao.marcarComoExcluido(lancamento.getId(),
                                     System.currentTimeMillis());

    // Reverte impacto no saldo da conta
    Conta conta = contaDao.buscarPorId(lancamento.getContaId());
    if (lancamento.getTipo().equals("receita")) {
        conta.setSaldoAtual(conta.getSaldoAtual() - lancamento.getValor());
    } else {
        conta.setSaldoAtual(conta.getSaldoAtual() + lancamento.getValor());
    }
    conta.setLastModified(System.currentTimeMillis());
    conta.setSyncStatus(2);
    contaDao.atualizar(conta);

    runOnUiThread(() -> {
        recarregarLancamentos(); // Remove da lista (isDeleted=1 é filtrado)
        sincronizarDados();
    });
}).start();
```

67.

Filtros e Busca

68. **Usuário digita:** SearchView no topo da tela
69. **MovementsActivity chama:** `lancamentoDao.buscarPorDescricaoOuValor(usuarioId, searchTerm)`
70. **LancamentoDao executa:** `SELECT * FROM lancamento WHERE (descricao LIKE '%?%' OR valor LIKE '%?%') AND usuarioId = ?`

71.

Adapter atualiza: Lista com resultados filtrados

72.

Filtro por tipo (Chips/Tabs):

73. Usuário seleciona: "Todas", "Receitas" ou "Despesas"
74. Recarrega lista com filtro apropriado

75.

Filtro por período:

76. Usuário seleciona: "Este mês", "Últimos 30 dias", "Personalizado"
77. **MovementsActivity chama:** `lancamentoDao.listarPorUsuarioPeriodo(usuarioId, dataInicio, dataFim)`

78.

Sincronização (EnhancedSyncService)

79. **MovementsActivity chama:** `EnhancedSyncService.syncLancamentos(usuarioId)`
80. **EnhancedSyncService executa:**
    81. **Busca pendentes:** `lancamentoDao.obterPendentesSyncPorUsuario(usuarioId)`
    82. **Agrupa em batch:** Para envio eficiente (até 50 lançamentos por vez)
    83. **Envia ao servidor:**

```
String comando = Protocol.CMD_BULK_UPLOAD;
String params = serializarLancamentos(lancamentosPendentes);
String resposta = ServerClient.sendCommand(comando, params);
```

    84. **Processa resposta:**
        85. Parse dos IDs retornados pelo servidor
        86. Atualiza metadados locais
        87. Marca como sincronizado: `lancamentoDao.atualizarStatusSync(uuids, 1)`
    88. **Busca mudanças do servidor:**
        89. Obtém timestamp do último sync: `lancamentoDao.obterUltimoTempoSync()`
        90. **Chama servidor:** `CMD_LIST_CHANGES_SINCE|lastSyncTime`
        91. **Recebe:** Lista de lançamentos novos/modificados
        92. **Para cada lançamento:**
            93. Busca local por UUID
            94. Se não existe, insere
            95. Se existe e timestamps diferentes, resolve conflito via `ConflictResolutionManager`

Resumo das Cadeias de Chamadas:

Adicionar Lançamento:

```
MovementsActivity.onClick(FAB)
  → Dialog.show()
  → Usuario preenche dados
  → Dialog.onClick(Salvar)
    → MovementsActivity.validarCampos()
    → DataIntegrityValidator.validarLancamento()
      → ContaDao.buscarPorId()
      → CategoriaDao.buscarPorId()
      → LancamentoDao.buscarDuplicata()
    → new Thread().start()
      → LancamentoDao.inserirSeguro()
        → Room INSERT INTO lancamento
      → ContaDao.buscarPorId()
      → Conta.setSaldoAtual(novoSaldo)
      → ContaDao.atualizar()
      → runOnUiThread()
        → recarregarLancamentos()
        → EnhancedSyncService.syncLancamentos()
          → ServerClient.sendCommand(BULK_UPLOAD)
```

Editar Lançamento:

```
RecyclerView.onClick(item)
  → Dialog.show(lancamento)
  → Usuario edita dados
  → Dialog.onClick(Salvar)
    → MovementsActivity.calcularDiferencaSaldo()
    → Lancamento.setLastModified()
    → new Thread().start()
      → LancamentoDao.atualizar()
      → ContaDao.atualizarSaldo(antiga)
      → ContaDao.atualizarSaldo(nova)
      → EnhancedSyncService.syncLancamentos()
```

Excluir Lançamento:

```
RecyclerView.onLongClick(item)
  → Dialog.show(confirmação)
  → Dialog.onClick(Confirmar)
    → new Thread().start()
      → LancamentoDao.marcarComoExcluido()
        → Room UPDATE lancamento SET isDeleted=1
      → ContaDao.buscarPorId()
      → Conta.ajustarSaldo(reverter)
      → ContaDao.atualizar()
      → runOnUiThread()
        → recarregarLancamentos() // Filtra isDeleted=0
        → EnhancedSyncService.syncLancamentos()
```

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

8. SettingsActivity.java

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

Camada de Dados - Mobile

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

DAOs (Data Access Objects)

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

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

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

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

Camada de Rede - Mobile

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

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/despesa)
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 metadados
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 UUID
CMD_UPDATE_MOVIMENTACAO_ENHANCED = "UPDATE_MOVIMENTACAO_ENHANCED"  // Atualizar com metadados
CMD_SYNC_MOVIMENTACAO = "SYNC_MOVIMENTACAO"    // Sincronizar movimentação específica
```

Comandos de Sincronização Avançada:

```
CMD_SYNC_STATUS = "SYNC_STATUS"             // Verificar status de sincronização
CMD_INCREMENTAL_SYNC = "INCREMENTAL_SYNC"   // Sincronização incremental por timestamp
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
```

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

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

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

ConflictResolutionManager.java

Função: Resolução de conflitos de sincronização de dados

Estratégias de Resolução:

1. **Last Write Wins (LWW):** Última modificação vence
    2. Compara timestamps (lastModified)
    3. Mantém versão mais recente
    4. Estratégia padrão
5. **Server Wins:** Prioridade do servidor
    6. Servidor sempre sobrescreve cliente
    7. Usada para dados administrativos
8. **Client Wins:** Prioridade do cliente
    9. Cliente sobrescreve servidor
    10. Usada para dados locais críticos
11. **Merge:** Combina mudanças
    12. Tenta mesclar campos não conflitantes
    13. Marca campos conflitantes para revisão manual
14. **Manual Resolution:** Notifica usuário
    15. Conflitos críticos requerem decisão do usuário
    16. 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

Utilitários - Mobile

DataIntegrityValidator.java

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

---

📚 DEPENDÊNCIAS E BIBLIOTECAS

Mobile (Android)

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

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)

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

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

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`

Desktop (Servidor)

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)

Desktop (Cliente Admin)

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

Networking

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

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

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

Requisitos de Versão

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

💻 VERSÃO DESKTOP

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.

Cliente Desktop (ClienteFinanza)

Informações Gerais

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

Estrutura de Diretórios (11 arquivos Java)

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
```

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)
```

Fluxo de Funcionalidades - Desktop Cliente

1. MainCliente.java

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

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

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`

4. EditarUsuarioDialog.java

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

5. AuthController.java

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

6. FinanceController.java

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

7. NetworkClient.java

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

Fluxo de Funcionalidades - Servidor

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`

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

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`

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)

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

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

---

🔄 FLUXO DE DADOS COMPLETO

Este capítulo detalha o fluxo completo de dados através de todo o sistema, desde a interface do usuário até o banco de dados e vice-versa. Cada exemplo mostra **exatamente** quais arquivos são chamados, quais métodos são executados, e como os dados fluem entre as camadas.

Exemplo 1: Login de Usuário Completo (Mobile → Servidor → Mobile)

Contexto: Usuário abre o app e faz login com email e senha.

PASSO 1: Interface do Usuário (LoginActivity.java)

Arquivo: `app/src/main/java/com/example/finanza/ui/LoginActivity.java` **Layout:** `app/src/main/res/layout/activity_login.xml`

```
// Linha 45-50: Usuário clica no botão de login
btnLogin.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String email = etEmail.getText().toString().trim();
        String senha = etSenha.getText().toString().trim();

        // Validação básica
        if (email.isEmpty() || senha.isEmpty()) {
            Toast.makeText(LoginActivity.this,
                           "Preencha todos os campos",
                           Toast.LENGTH_SHORT).show();
            return;
        }

        // Chama método de login
        realizarLogin(email, senha);
    }
});
```

O que acontece: - Extrai texto dos campos EditText - Valida se não estão vazios - Chama método `realizarLogin(email, senha)`

PASSO 2: Chamada ao AuthManager (AuthManager.java)

Arquivo: `app/src/main/java/com/example/finanza/network/AuthManager.java`

```
// LoginActivity.java - Linha 80-100
private void realizarLogin(String email, String senha) {
    // Exibe ProgressDialog
    ProgressDialog dialog = ProgressDialog.show(this,
                                                "Autenticando",
                                                "Aguarde...");

    // Executa em thread para não bloquear UI
    new Thread(() -> {
        // CHAMA AuthManager
        boolean sucesso = AuthManager.getInstance(this)
                                     .login(email, senha);

        runOnUiThread(() -> {
            dialog.dismiss();
            if (sucesso) {
                irParaMenu();
            } else {
                Toast.makeText(this,
                               "Credenciais inválidas",
                               Toast.LENGTH_SHORT).show();
            }
        });
    }).start();
}
```

```
// AuthManager.java - Linha 35-70: Implementação do login
public boolean login(String email, String senha) {
    try {
        // 1. Hash da senha usando SHA-256
        String senhaHash = SecurityUtil.hashPassword(senha);

        // 2. Monta comando do protocolo
        String comando = Protocol.CMD_LOGIN; // "LOGIN"
        String parametros = email + "|" + senhaHash;

        // 3. CHAMA ServerClient para enviar ao servidor
        String resposta = ServerClient.getInstance()
                                      .sendCommand(comando, parametros);

        // 4. Processa resposta
        if (resposta != null && resposta.startsWith("OK|")) {
            // Parse dos dados do usuário
            String jsonUsuario = resposta.substring(3);
            JSONObject json = new JSONObject(jsonUsuario);

            // Cria objeto Usuario
            Usuario usuario = new Usuario();
            usuario.setId(json.getInt("id"));
            usuario.setNome(json.getString("nome"));
            usuario.setEmail(json.getString("email"));
            usuario.setDataCriacao(System.currentTimeMillis());
            usuario.setSyncStatus(1); // SYNCED

            // 5. CHAMA UsuarioDao para salvar localmente
            UsuarioDao usuarioDao = AppDatabase.getInstance(context)
                                               .usuarioDao();
            usuarioDao.inserirOuAtualizar(usuario);

            // 6. Armazena usuário atual em memória
            currentUser = usuario;
            return true;
        }
        return false;
    } catch (Exception e) {
        Log.e("AuthManager", "Erro no login: " + e.getMessage());
        return false;
    }
}
```

O que acontece: - Hash SHA-256 da senha via `SecurityUtil.hashPassword()` - Formata comando do protocolo: "LOGIN|email|hash" - Envia ao servidor via `ServerClient` - Parse da resposta JSON - Salva usuário localmente no Room (SQLite)

PASSO 3: Comunicação com Servidor (ServerClient.java)

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

```
// ServerClient.java - Linha 50-110: Envio de comando
public String sendCommand(String comando, String parametros) {
    Socket socket = null;
    BufferedReader reader = null;
    PrintWriter writer = null;

    try {
        // 1. Conecta ao servidor via TCP/IP
        socket = new Socket(SERVER_HOST, SERVER_PORT); // porta 12345
        socket.setSoTimeout(TIMEOUT_MS); // 30 segundos

        // 2. Configura streams de entrada/saída
        OutputStream out = socket.getOutputStream();
        writer = new PrintWriter(out, true);

        InputStream in = socket.getInputStream();
        reader = new BufferedReader(new InputStreamReader(in));

        // 3. Monta mensagem completa
        String mensagem = comando + "|" + parametros;
        Log.d("ServerClient", "Enviando: " + mensagem);

        // 4. ENVIA ao servidor
        writer.println(mensagem);
        writer.flush();

        // 5. AGUARDA resposta
        String resposta = reader.readLine();
        Log.d("ServerClient", "Recebido: " + resposta);

        return resposta;

    } catch (SocketTimeoutException e) {
        Log.e("ServerClient", "Timeout na conexão");
        return "ERROR|Timeout";
    } catch (IOException e) {
        Log.e("ServerClient", "Erro de rede: " + e.getMessage());
        return "ERROR|Conexão falhou";
    } finally {
        // 6. Fecha recursos
        try {
            if (reader != null) reader.close();
            if (writer != null) writer.close();
            if (socket != null) socket.close();
        } catch (IOException e) {
            Log.e("ServerClient", "Erro ao fechar socket");
        }
    }
}
```

O que acontece: - Abre socket TCP/IP para o servidor (porta 12345) - Envia comando formatado: "LOGIN|email@exemplo.com|hash_sha256" - Aguarda resposta com timeout de 30 segundos - Retorna resposta recebida - Fecha conexão

PASSO 4: Servidor Recebe e Processa (ClientHandler.java)

Arquivo: `DESKTOP VERSION/ServidorFinanza/src/server/ClientHandler.java`

```
// ClientHandler.java - Linha 40-80: Thread que processa cliente
@Override
public void run() {
    try {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream())
        );
        PrintWriter writer = new PrintWriter(
            clientSocket.getOutputStream(), true
        );

        // Loop para processar comandos do cliente
        String linha;
        while ((linha = reader.readLine()) != null) {
            System.out.println("[" + new Date() + "] Recebido: " + linha);

            // PROCESSA comando
            String resposta = processarComando(linha);

            System.out.println("[" + new Date() + "] Enviando: " + resposta);

            // ENVIA resposta
            writer.println(resposta);
            writer.flush();
        }
    } catch (IOException e) {
        System.err.println("Erro ao processar cliente: " + e.getMessage());
    }
}

// ClientHandler.java - Linha 90-150: Processa comando específico
private String processarComando(String comando) {
    // Divide comando em partes
    String[] partes = comando.split("\\|");
    String cmd = partes[0];

    try {
        switch (cmd) {
            case "LOGIN":
                return processarLogin(partes);
            case "REGISTER":
                return processarRegistro(partes);
            // ... outros comandos
            default:
                return "ERROR|Comando desconhecido";
        }
    } catch (Exception e) {
        return "ERROR|" + e.getMessage();
    }
}

// ClientHandler.java - Linha 200-250: Processa login específico
private String processarLogin(String[] partes) {
    if (partes.length < 3) {
        return "ERROR|INVALID_DATA|Dados incompletos";
    }
```

```
    String email = partes[1];
    String senhaHash = partes[2];

    // CHAMA UsuarioDAO para autenticar
    UsuarioDAO usuarioDAO = new UsuarioDAO();
    Usuario usuario = usuarioDAO.autenticar(email, senhaHash);

    if (usuario != null) {
        // Monta resposta JSON
        StringBuilder json = new StringBuilder();
        json.append("{");
        json.append("\"id\":").append(usuario.getId()).append(",");
        json.append("\"nome\":\"").append(usuario.getNome()).append("\",");
        json.append("\"email\":\"").append(usuario.getEmail()).append("\"");
        json.append("}");

        return "OK|" + json.toString();
    } else {
        return "ERROR|INVALID_CREDENTIALS|Email ou senha incorretos";
    }
}
```

O que acontece: - Thread dedicada processa requisições do cliente - Lê comando via BufferedReader - Divide comando em partes pelo separador "|" - Identifica comando "LOGIN" - Extrai email e senha hash - Chama `UsuarioDAO.autenticar()` - Monta resposta JSON - Envia resposta de volta ao cliente

---

PASSO 5: Consulta ao Banco MySQL (UsuarioDAO.java)

Arquivo: `DESKTOP VERSION/ServidorFinanza/src/dao/UsuarioDAO.java`

```
// UsuarioDAO.java - Linha 80-130: Autentica usuário
public Usuario autenticar(String email, String senhaHash) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        // 1. OBTÉM conexão do pool
        conn = DatabaseUtil.getConnection();

        // 2. Prepara query SQL
        String sql = "SELECT id, nome, email, senha_hash, tipo_usuario " +
                     "FROM usuario " +
                     "WHERE email = ? AND senha_hash = ?";

        stmt = conn.prepareStatement(sql);
        stmt.setString(1, email);
        stmt.setString(2, senhaHash);

        // 3. EXECUTA query
        rs = stmt.executeQuery();

        // 4. Processa resultado
        if (rs.next()) {
            Usuario usuario = new Usuario();
            usuario.setId(rs.getInt("id"));
            usuario.setNome(rs.getString("nome"));
            usuario.setEmail(rs.getString("email"));
            usuario.setSenhaHash(rs.getString("senha_hash"));
            usuario.setTipoUsuario(rs.getString("tipo_usuario"));

            System.out.println("Usuário autenticado: " + usuario.getEmail());
            return usuario;
        } else {
            System.out.println("Credenciais inválidas para: " + email);
            return null;
        }

    } catch (SQLException e) {
        System.err.println("Erro ao autenticar: " + e.getMessage());
        return null;
    } finally {
        // 5. Fecha recursos
        try {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
            if (conn != null) DatabaseUtil.closeConnection(conn);
        } catch (SQLException e) {
            System.err.println("Erro ao fechar recursos");
        }
    }
}
```

SQL Executado:

```
SELECT id, nome, email, senha_hash, tipo_usuario
FROM usuario
WHERE email = 'usuario@exemplo.com'
  AND senha_hash = 'hash_sha256_aqui';
```

O que acontece: - Obtém conexão do pool de conexões - Prepara query SQL com parâmetros (evita SQL injection) - Executa query no MySQL - Se encontrar registro, cria objeto Usuario - Se não encontrar, retorna null - Fecha recursos (ResultSet, PreparedStatement, Connection)

PASSO 6: Resposta Retorna ao Mobile

Fluxo reverso:

```
UsuarioDAO.autenticar() retorna Usuario
  ↓
ClientHandler.processarLogin() recebe Usuario
  ↓
ClientHandler monta JSON: "OK|{id:1,nome:'João',email:'...'}"
  ↓
ClientHandler.writer.println() envia via socket
  ↓
ServerClient.reader.readLine() recebe no mobile
  ↓
ServerClient.sendCommand() retorna String resposta
  ↓
AuthManager.login() recebe resposta
  ↓
AuthManager parse JSON e cria Usuario
  ↓
AuthManager chama UsuarioDao.inserirOuAtualizar()
  ↓
Room INSERT INTO usuario (SQLite local)
  ↓
AuthManager retorna true
  ↓
LoginActivity recebe sucesso
  ↓
LoginActivity.irParaMenu() é chamado
```

PASSO 7: Salvamento Local no Room (UsuarioDao.java - Mobile)

Arquivo: `app/src/main/java/com/example/finanza/db/UsuarioDao.java`

```
// UsuarioDao.java - Linha 50-75: Insere ou atualiza
@Insert(onConflict = OnConflictStrategy.REPLACE)
long inserir(Usuario usuario);

@Transaction
public void inserirOuAtualizar(Usuario usuario) {
    // Verifica se já existe por UUID
    Usuario existente = buscarPorUuid(usuario.getUuid());

    if (existente != null) {
        // Atualiza ID local com o existente
        usuario.setId(existente.getId());

        // Compara timestamps para decidir qual manter
        if (usuario.getLastModified() > existente.getLastModified()) {
            atualizar(usuario);
        }
        // Se timestamp do existente é mais recente, ignora
    } else {
        // Insere novo registro
        inserir(usuario);
    }
}
```

SQL Gerado pelo Room:

```
INSERT OR REPLACE INTO usuario
(id, uuid, nome, email, senhaHash, dataCriacao,
 lastModified, syncStatus, lastSyncTime, serverHash)
VALUES
(null, 'uuid-gerado', 'João Silva', 'joao@exemplo.com',
 'hash', 1234567890, 1234567890, 1, 1234567890, null);
```

O que acontece: - Room verifica se já existe usuário com mesmo UUID - Se existir e timestamp for mais novo, atualiza - Se não existir, insere novo registro - Marca como sincronizado (syncStatus = 1) - Dados ficam disponíveis offline

---

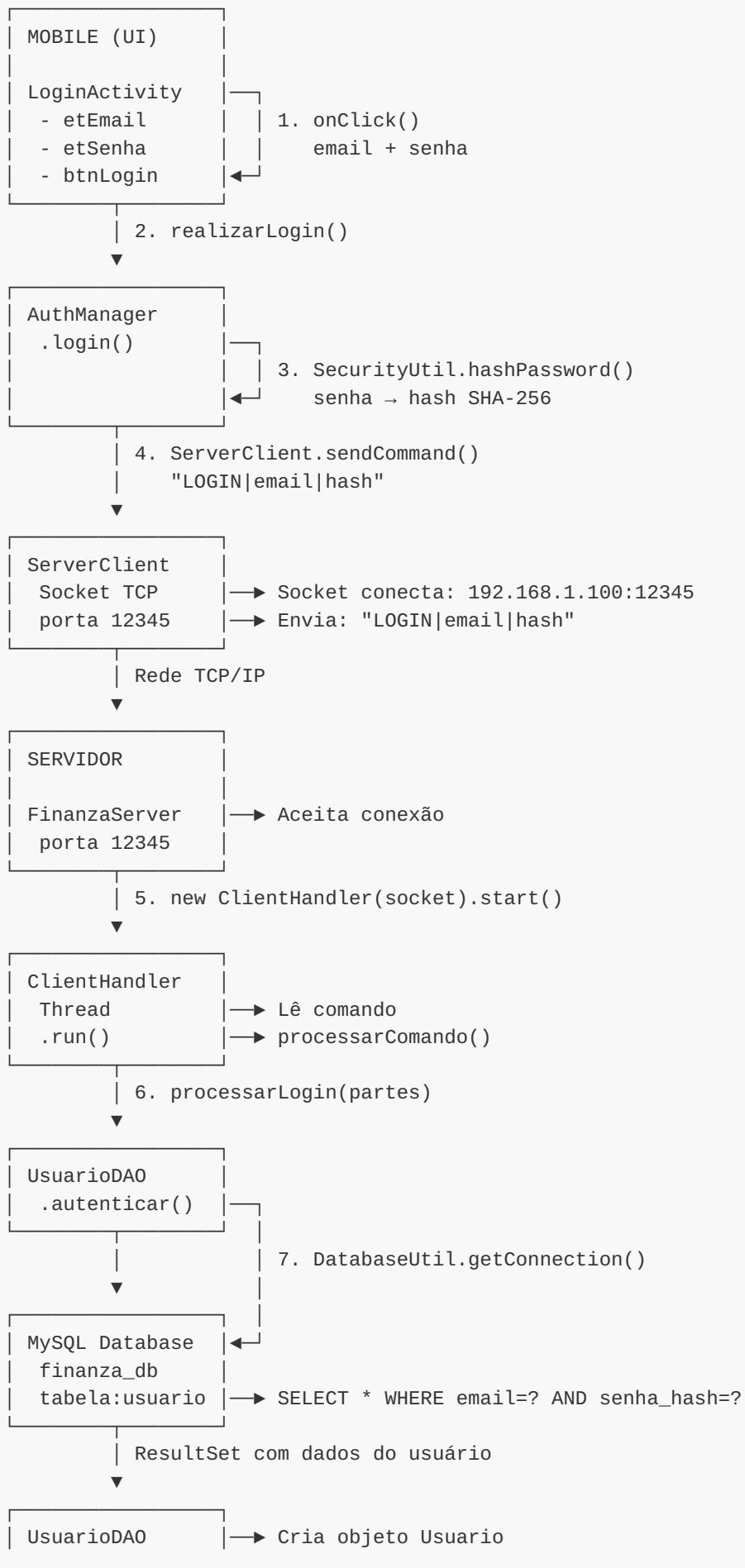
PASSO 8: Redirecionamento para Menu (LoginActivity.java)

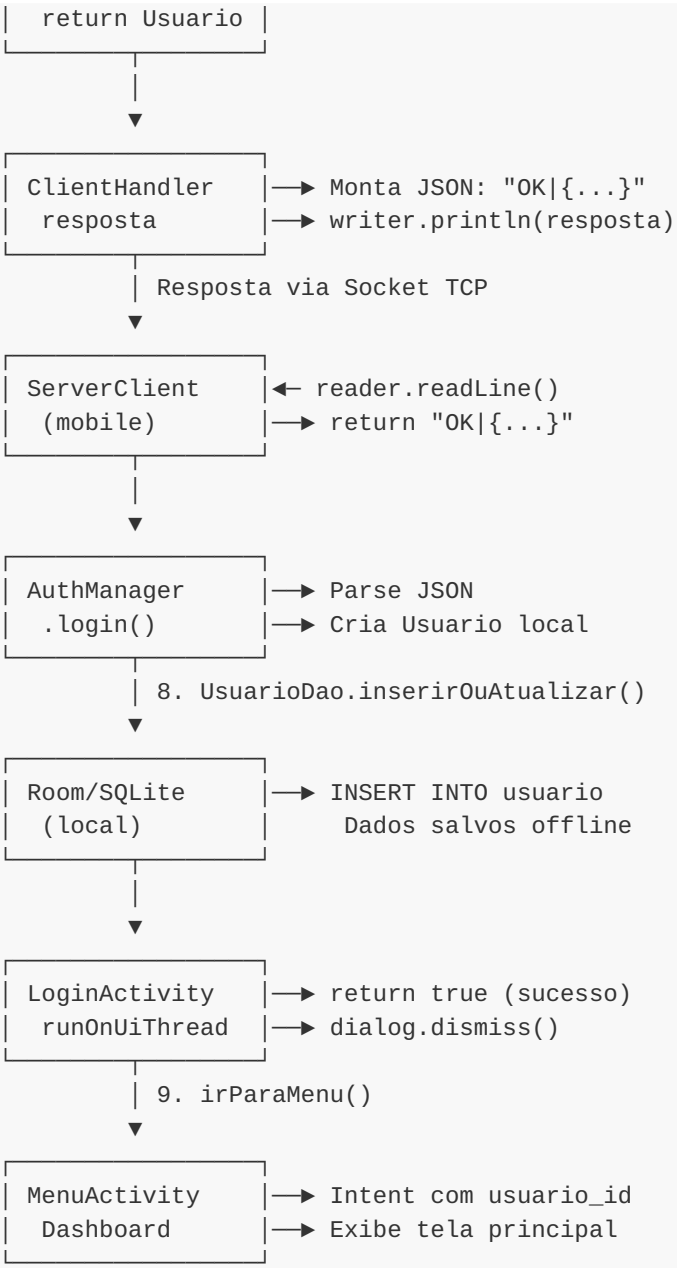
Arquivo: `app/src/main/java/com/example/finanza/ui/LoginActivity.java`

```
// LoginActivity.java - Linha 150-160
private void irParaMenu() {
    // Recupera usuário salvo no banco local
    UsuarioDao usuarioDao = AppDatabase.getInstance(this).usuarioDao();
    Usuario usuario = usuarioDao.buscarPorEmail(email);

    // Cria Intent para MenuActivity
    Intent intent = new Intent(LoginActivity.this, MenuActivity.class);
    intent.putExtra("usuario_id", usuario.getId());
    intent.putExtra("usuario_nome", usuario.getNome());

    // Inicia nova Activity
    startActivity(intent);

    // Remove LoginActivity da pilha (não permite voltar)
    finish();
}
```

O que acontece: - Busca usuário completo do banco local - Cria Intent para MenuActivity - Passa ID e nome do usuário como extras - Inicia MenuActivity - Finaliza LoginActivity (não pode voltar com botão Back)

Resumo Visual do Fluxo Login Completo





Tempo Total Estimado: 1-3 segundos **Arquivos Envolvidos:** 8 arquivos Java **Queries SQL:** 2 (1 SELECT no servidor, 1 INSERT no mobile) **Protocolos:** TCP/IP, HTTP-like text protocol **Threads:** 3 (UI thread, network thread, server thread)

Exemplo 2: Criar Lançamento Completo (Mobile → SQLite → Servidor → MySQL)

Contexto: Usuário registra uma nova despesa de R$ 50,00 em "Alimentação" da conta "Nubank".

FASE 1: Captura de Dados na Interface

Arquivo: `app/src/main/java/com/example/finanza/ui/MovementsActivity.java` **Layout Dialog:** `app/src/main/res/layout/dialog_add_transaction_movements.xml`

```
// MovementsActivity.java - Linha 120-130: Usuário clica FAB
fabAddTransaction.setOnClickListener(v -> {
    exibirDialogNovoLancamento();
});

// Linha 150-250: Dialog de adicionar lançamento
private void exibirDialogNovoLancamento() {
    // Infla layout do dialog
    View dialogView = getLayoutInflater().inflate(
        R.layout.dialog_add_transaction_movements, null
    );

    // Obtém referências dos componentes
    EditText etDescricao = dialogView.findViewById(R.id.etDescricao);
    EditText etValor = dialogView.findViewById(R.id.etValor);
    DatePicker datePicker = dialogView.findViewById(R.id.datePicker);
    RadioGroup rgTipo = dialogView.findViewById(R.id.rgTipo);
    Spinner spinnerConta = dialogView.findViewById(R.id.spinnerConta);
    Spinner spinnerCategoria = dialogView.findViewById(R.id.spinnerCategoria);

    // CARREGA lista de contas do banco
    carregarContas(spinnerConta);

    // Listener para trocar categorias quando muda tipo
    rgTipo.setOnCheckedChangeListener((group, checkedId) -> {
        if (checkedId == R.id.rbReceita) {
            carregarCategorias(spinnerCategoria, "receita");
        } else {
            carregarCategorias(spinnerCategoria, "despesa");
        }
    });

    // Carrega categorias de despesa por padrão
    carregarCategorias(spinnerCategoria, "despesa");

    // Cria e exibe dialog
    AlertDialog dialog = new AlertDialog.Builder(this)
        .setTitle("Nova Transação")
        .setView(dialogView)
        .setPositiveButton("Salvar", null) // Listener definido depois
        .setNegativeButton("Cancelar", null)
        .create();

    dialog.show();

    // Override do botão Salvar para não fechar automaticamente
    dialog.getButton(AlertDialog.BUTTON_POSITIVE)
          .setOnClickListener(v -> {
              salvarNovoLancamento(dialogView, dialog);
          });
}
```

FASE 2: Carregamento de Contas e Categorias

```
// MovementsActivity.java - Linha 260-290: Carrega contas
private void carregarContas(Spinner spinner) {
    new Thread(() -> {
        // BUSCA contas do banco local
        List<Conta> contas = AppDatabase.getInstance(this)
                                        .contaDao()
                                        .listarPorUsuario(usuarioId);

        runOnUiThread(() -> {
            // Cria adapter para spinner
            ArrayAdapter<Conta> adapter = new ArrayAdapter<>(
                this,
                android.R.layout.simple_spinner_item,
                contas
            );
            adapter.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item
            );
            spinner.setAdapter(adapter);
        });
    }).start();
}

// Linha 300-330: Carrega categorias filtradas por tipo
private void carregarCategorias(Spinner spinner, String tipo) {
    new Thread(() -> {
        // BUSCA categorias do tipo específico
        List<Categoria> categorias = AppDatabase.getInstance(this)
                                                .categoriaDao()
                                                .listarPorUsuarioETipo(usuarioId, tipo);

        runOnUiThread(() -> {
            ArrayAdapter<Categoria> adapter = new ArrayAdapter<>(
                this,
                android.R.layout.simple_spinner_item,
                categorias
            );
            adapter.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item
            );
            spinner.setAdapter(adapter);
        });
    }).start();
}
```

Queries Room Executadas:

```
-- Para carregar contas:
SELECT * FROM conta
WHERE usuarioId = 1
  AND syncStatus != 4
ORDER BY dataCriacao DESC;

-- Para carregar categorias de despesa:
SELECT * FROM categoria
WHERE usuarioId = 1
  AND tipo = 'despesa'
ORDER BY nome ASC;
```

FASE 3: Validação e Criação do Objeto

```
// MovementsActivity.java - Linha 350-450: Salva novo lançamento
private void salvarNovoLancamento(View dialogView, AlertDialog dialog) {
    // 1. EXTRAI dados do dialog
    EditText etDescricao = dialogView.findViewById(R.id.etDescricao);
    EditText etValor = dialogView.findViewById(R.id.etValor);
    DatePicker datePicker = dialogView.findViewById(R.id.datePicker);
    RadioGroup rgTipo = dialogView.findViewById(R.id.rgTipo);
    Spinner spinnerConta = dialogView.findViewById(R.id.spinnerConta);
    Spinner spinnerCategoria = dialogView.findViewById(R.id.spinnerCategoria);

    String descricao = etDescricao.getText().toString().trim();
    String valorStr = etValor.getText().toString().trim();

    // 2. VALIDA campos
    if (descricao.isEmpty()) {
        etDescricao.setError("Descrição obrigatória");
        return;
    }

    if (valorStr.isEmpty()) {
        etValor.setError("Valor obrigatório");
        return;
    }

    double valor;
    try {
        valor = Double.parseDouble(valorStr);
    } catch (NumberFormatException e) {
        etValor.setError("Valor inválido");
        return;
    }

    if (valor <= 0) {
        etValor.setError("Valor deve ser maior que zero");
        return;
    }

    Conta contaSelecionada = (Conta) spinnerConta.getSelectedItem();
    Categoria categoriaSelecionada = (Categoria) spinnerCategoria.getSelectedItem();

    if (contaSelecionada == null || categoriaSelecionada == null) {
        Toast.makeText(this, "Selecione conta e categoria",
                       Toast.LENGTH_SHORT).show();
        return;
    }

    // 3. DETERMINA tipo
    String tipo = (rgTipo.getCheckedRadioButtonId() == R.id.rbReceita) ?
                  "receita" : "despesa";

    // 4. OBTÉM data selecionada
    Calendar calendar = Calendar.getInstance();
    calendar.set(datePicker.getYear(),
                 datePicker.getMonth(),
                 datePicker.getDayOfMonth(),
                 0, 0, 0);
```

```
    long dataTransacao = calendar.getTimeInMillis();

    // 5. CRIA objeto Lancamento
    Lancamento lancamento = new Lancamento();
    lancamento.setUuid(UUID.randomUUID().toString());
    lancamento.setDescricao(descricao);
    lancamento.setValor(valor);
    lancamento.setData(dataTransacao);
    lancamento.setTipo(tipo);
    lancamento.setContaId(contaSelecionada.getId());
    lancamento.setCategoriaId(categoriaSelecionada.getId());
    lancamento.setUsuarioId(usuarioId);
    lancamento.setDataCriacao(System.currentTimeMillis());
    lancamento.setLastModified(System.currentTimeMillis());
    lancamento.setSyncStatus(2); // NEEDS_SYNC
    lancamento.setIsDeleted(0); // Ativo

    // 6. CHAMA validação de integridade
    validarEInserir(lancamento, dialog);
}
```

FASE 4: Validação de Integridade

Arquivo: `app/src/main/java/com/example/finanza/util/DataIntegrityValidator.java`

```
// DataIntegrityValidator.java - Linha 50-120
public static ValidationResult validarLancamento(Lancamento lancamento,
                                                 Context context) {
    AppDatabase db = AppDatabase.getInstance(context);

    // 1. VALIDA valor positivo
    if (lancamento.getValor() <= 0) {
        return ValidationResult.error("Valor deve ser positivo");
    }

    // 2. VALIDA conta existe
    Conta conta = db.contaDao().buscarPorId(lancamento.getContaId());
    if (conta == null) {
        return ValidationResult.error("Conta não encontrada");
    }

    // 3. VALIDA categoria existe
    Categoria categoria = db.categoriaDao()
                            .buscarPorId(lancamento.getCategoriaId());
    if (categoria == null) {
        return ValidationResult.error("Categoria não encontrada");
    }

    // 4. VALIDA tipo da categoria corresponde ao lançamento
    if (!categoria.getTipo().equals(lancamento.getTipo())) {
        return ValidationResult.error(
            "Tipo de categoria não corresponde ao lançamento"
        );
    }

    // 5. VERIFICA duplicatas recentes (últimos 5 minutos)
    long cincoMinutosAtras = System.currentTimeMillis() - (5 * 60 * 1000);
    List<Lancamento> similares = db.lancamentoDao()
                                   .buscarSimilares(
                                       lancamento.getValor(),
                                       lancamento.getData(),
                                       5 * 60 * 1000, // timeWindow
                                       lancamento.getContaId(),
                                       lancamento.getUsuarioId()
                                   );

    if (!similares.isEmpty()) {
        return ValidationResult.warning(
            "Já existe um lançamento similar nos últimos 5 minutos"
        );
    }

    // 6. VALIDA data não é muito futura (mais de 1 ano)
    long umAnoFuturo = System.currentTimeMillis() +
                       (365L * 24 * 60 * 60 * 1000);
    if (lancamento.getData() > umAnoFuturo) {
        return ValidationResult.warning("Data muito distante no futuro");
    }

    return ValidationResult.ok();
}
```

Query de busca de duplicatas:

```
SELECT * FROM lancamento
WHERE valor = 50.00
  AND ABS(data - 1234567890) < 300000  -- 5 minutos
  AND contaId = 3
  AND usuarioId = 1
  AND isDeleted = 0
  AND uuid != 'uuid-do-novo';
```

FASE 5: Inserção no Banco Local (Room)

```
// MovementsActivity.java - Linha 470-550
private void validarEInserir(Lancamento lancamento, AlertDialog dialog) {
    // Exibe progress
    ProgressDialog progress = ProgressDialog.show(this,
                                                  "Salvando",
                                                  "Aguarde...");

    new Thread(() -> {
        try {
            // 1. VALIDA integridade
            ValidationResult validacao = DataIntegrityValidator
                                        .validarLancamento(lancamento, this);

            if (!validacao.isValid()) {
                runOnUiThread(() -> {
                    progress.dismiss();
                    Toast.makeText(this, validacao.getMessage(),
                                   Toast.LENGTH_LONG).show();
                });
                return;
            }

            // 2. OBTÉM DAOs
            LancamentoDao lancamentoDao = AppDatabase.getInstance(this)
                                                     .lancamentoDao();
            ContaDao contaDao = AppDatabase.getInstance(this)
                                           .contaDao();

            // 3. INSERE lançamento
            long idInserido = lancamentoDao.inserirSeguro(lancamento);
            lancamento.setId((int) idInserido);

            // 4. ATUALIZA saldo da conta
            Conta conta = contaDao.buscarPorId(lancamento.getContaId());
            double novoSaldo = conta.getSaldoAtual();

            if (lancamento.getTipo().equals("receita")) {
                novoSaldo += lancamento.getValor();
            } else { // despesa
                novoSaldo -= lancamento.getValor();
            }

            conta.setSaldoAtual(novoSaldo);
            conta.setLastModified(System.currentTimeMillis());
            conta.setSyncStatus(2); // Marca para sincronização
            contaDao.atualizar(conta);

            // 5. SINCRONIZA com servidor (em background)
            sincronizarLancamento(lancamento);
            sincronizarConta(conta);

            // 6. ATUALIZA UI
            runOnUiThread(() -> {
                progress.dismiss();
                dialog.dismiss();
                Toast.makeText(this, "Lançamento salvo com sucesso!",
```

```
                                Toast.LENGTH_SHORT).show();
                    recarregarLancamentos();
                });

            } catch (Exception e) {
                runOnUiThread(() -> {
                    progress.dismiss();
                    Toast.makeText(this, "Erro ao salvar: " + e.getMessage(),
                                Toast.LENGTH_LONG).show();
                });
                Log.e("MovementsActivity", "Erro ao inserir", e);
            }
        }).start();
    }
```

SQL Executado pelo Room:

```
-- Inserção do lançamento
INSERT INTO lancamento (
    uuid, valor, data, descricao, tipo,
    contaId, categoriaId, usuarioId,
    dataCriacao, lastModified, syncStatus, isDeleted
) VALUES (
    'uuid-gerado', 50.00, 1234567890, 'Almoço', 'despesa',
    3, 5, 1,
    1234567890, 1234567890, 2, 0
);

-- Atualização do saldo da conta
UPDATE conta
SET saldoAtual = 450.00,  -- era 500, diminuiu 50
    lastModified = 1234567890,
    syncStatus = 2
WHERE id = 3;
```

FASE 6: Sincronização com Servidor

Arquivo: `app/src/main/java/com/example/finanza/network/EnhancedSyncService.java`

```
// EnhancedSyncService.java - Linha 200-300
public void sincronizarLancamento(Lancamento lancamento) {
    new Thread(() -> {
        try {
            // 1. PREPARA dados para envio
            String params = formatarLancamentoParaSync(lancamento);

            // 2. MONTA comando
            String comando = Protocol.CMD_ADD_MOVIMENTACAO_ENHANCED;

            // 3. ENVIA ao servidor
            String resposta = ServerClient.getInstance()
                                          .sendCommand(comando, params);

            // 4. PROCESSA resposta
            if (resposta != null && resposta.startsWith("OK|")) {
                // Extrai ID do servidor
                String[] partes = resposta.split("\\|");
                int serverId = Integer.parseInt(partes[1]);

                // 5. ATUALIZA metadados locais
                LancamentoDao dao = AppDatabase.getInstance(context)
                                               .lancamentoDao();
                dao.marcarComoSincronizado(lancamento.getId(),
                                           System.currentTimeMillis());
                dao.atualizarMetadataSync(
                    lancamento.getUuid(),
                    1, // SYNCED
                    System.currentTimeMillis(),
                    calcularHash(lancamento)
                );

                Log.d("Sync", "Lançamento sincronizado: " + serverId);
            } else {
                Log.e("Sync", "Erro ao sincronizar: " + resposta);
            }
        } catch (Exception e) {
            Log.e("Sync", "Exceção na sincronização", e);
        }
    }).start();
}

// Linha 350-380: Formata lançamento para envio
private String formatarLancamentoParaSync(Lancamento l) {
    StringBuilder sb = new StringBuilder();
    sb.append(l.getUuid()).append("|");
    sb.append(l.getValor()).append("|");
    sb.append(l.getData()).append("|");
    sb.append(l.getDescricao()).append("|");
    sb.append(l.getTipo()).append("|");
    sb.append(l.getContaId()).append("|");
    sb.append(l.getCategoriaId()).append("|");
    sb.append(l.getUsuarioId()).append("|");
    sb.append(l.getLastModified()).append("|");
    sb.append(calcularHash(l));
```

```
    return sb.toString();
}
```

Mensagem enviada via TCP:

```
ADD_MOVIMENTACAO_ENHANCED|uuid-123|50.00|1234567890|Almoço|despesa|3|5|1|1234567890|hash-md5
```

FASE 7: Processamento no Servidor

Arquivo: `DESKTOP VERSION/ServidorFinanza/src/server/ClientHandler.java`

```
// ClientHandler.java - Linha 400-450
private String processarAddMovimentacaoEnhanced(String[] partes) {
    try {
        // 1. EXTRAI parâmetros
        String uuid = partes[1];
        double valor = Double.parseDouble(partes[2]);
        long data = Long.parseLong(partes[3]);
        String descricao = partes[4];
        String tipo = partes[5];
        int contaId = Integer.parseInt(partes[6]);
        int categoriaId = Integer.parseInt(partes[7]);
        int usuarioId = Integer.parseInt(partes[8]);
        long lastModified = Long.parseLong(partes[9]);
        String hash = partes[10];

        // 2. CRIA objeto Movimentacao
        Movimentacao mov = new Movimentacao();
        mov.setUuid(uuid);
        mov.setValor(valor);
        mov.setData(new Date(data));
        mov.setDescricao(descricao);
        mov.setTipo(tipo);
        mov.setIdConta(contaId);
        mov.setIdCategoria(categoriaId);
        mov.setIdUsuario(usuarioId);

        // 3. VERIFICA se já existe (por UUID)
        MovimentacaoDAO dao = new MovimentacaoDAO();
        Movimentacao existente = dao.buscarPorUuid(uuid);

        if (existente != null) {
            // Verifica conflito de timestamp
            if (existente.getDataAtualizacao().getTime() > lastModified) {
                return "CONFLICT|Versão do servidor é mais recente";
            }
            // Atualiza existente
            mov.setId(existente.getId());
            dao.atualizar(mov);
            return "OK|" + mov.getId();
        } else {
            // 4. INSERE nova movimentação
            int id = dao.criar(mov);
            mov.setId(id);

            // 5. ATUALIZA saldo da conta no servidor
            ContaDAO contaDao = new ContaDAO();
            Conta conta = contaDao.buscarPorId(contaId);
            if (conta != null) {
                double novoSaldo = conta.getSaldo();
                if (tipo.equals("receita")) {
                    novoSaldo += valor;
                } else {
                    novoSaldo -= valor;
                }
                conta.setSaldo(novoSaldo);
                contaDao.atualizar(conta);
```

```
            }

            return "OK|" + id;
        }
    } catch (Exception e) {
        System.err.println("Erro ao adicionar movimentação: " + e.getMessage());
        return "ERROR|" + e.getMessage();
    }
}
```

FASE 8: Inserção no MySQL

Arquivo: `DESKTOP VERSION/ServidorFinanza/src/dao/MovimentacaoDAO.java`

```
// MovimentacaoDAO.java - Linha 100-160
public int criar(Movimentacao mov) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        conn = DatabaseUtil.getConnection();

        String sql = "INSERT INTO movimentacao " +
                     "(valor, data, descricao, tipo, id_conta, " +
                     "id_categoria, id_usuario, data_criacao, data_atualizacao) " +
                     "VALUES (?, ?, ?, ?, ?, ?, ?, NOW(), NOW())";

        stmt = conn.prepareStatement(sql,
                                     Statement.RETURN_GENERATED_KEYS);

        stmt.setDouble(1, mov.getValor());
        stmt.setDate(2, new java.sql.Date(mov.getData().getTime()));
        stmt.setString(3, mov.getDescricao());
        stmt.setString(4, mov.getTipo());
        stmt.setInt(5, mov.getIdConta());
        stmt.setInt(6, mov.getIdCategoria());
        stmt.setInt(7, mov.getIdUsuario());

        int affectedRows = stmt.executeUpdate();

        if (affectedRows == 0) {
            throw new SQLException("Falha ao criar movimentação");
        }

        // Obtém ID gerado
        rs = stmt.getGeneratedKeys();
        if (rs.next()) {
            int id = rs.getInt(1);
            System.out.println("Movimentação criada com ID: " + id);
            return id;
        } else {
            throw new SQLException("Falha ao obter ID");
        }

    } catch (SQLException e) {
        System.err.println("Erro SQL ao criar movimentação: " + e.getMessage());
        throw new RuntimeException(e);
    } finally {
        try {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
            if (conn != null) DatabaseUtil.closeConnection(conn);
        } catch (SQLException e) {
            System.err.println("Erro ao fechar recursos");
        }
    }
}
```

SQL Executado no MySQL:

```
INSERT INTO movimentacao (
    valor, data, descricao, tipo,
    id_conta, id_categoria, id_usuario,
    data_criacao, data_atualizacao
) VALUES (
    50.00, '2024-01-15', 'Almoço', 'despesa',
    3, 5, 1,
    NOW(), NOW()
);

-- Atualização do saldo no servidor
UPDATE conta
SET saldo_inicial = 450.00
WHERE id = 3;
```

FASE 9: Confirmação e Atualização da UI

Resposta retorna ao mobile:

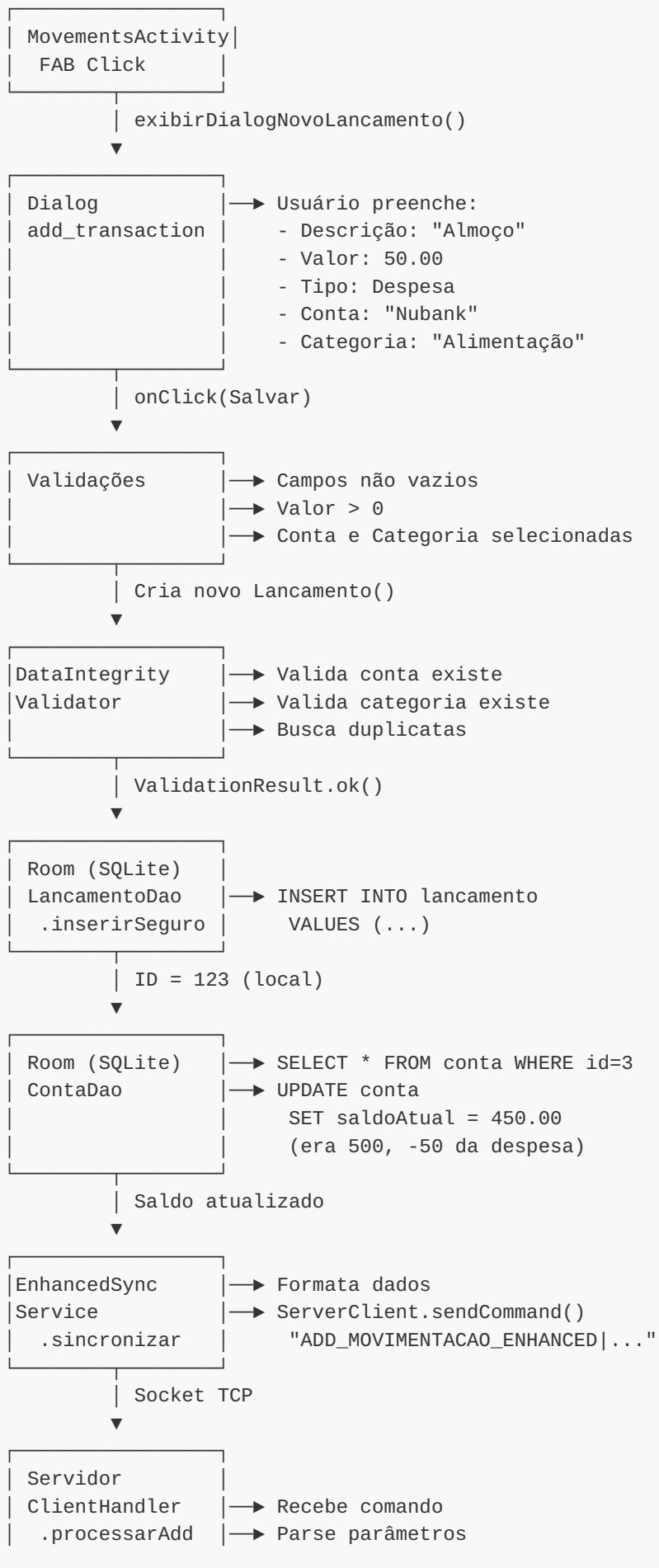
```
OK|542
```

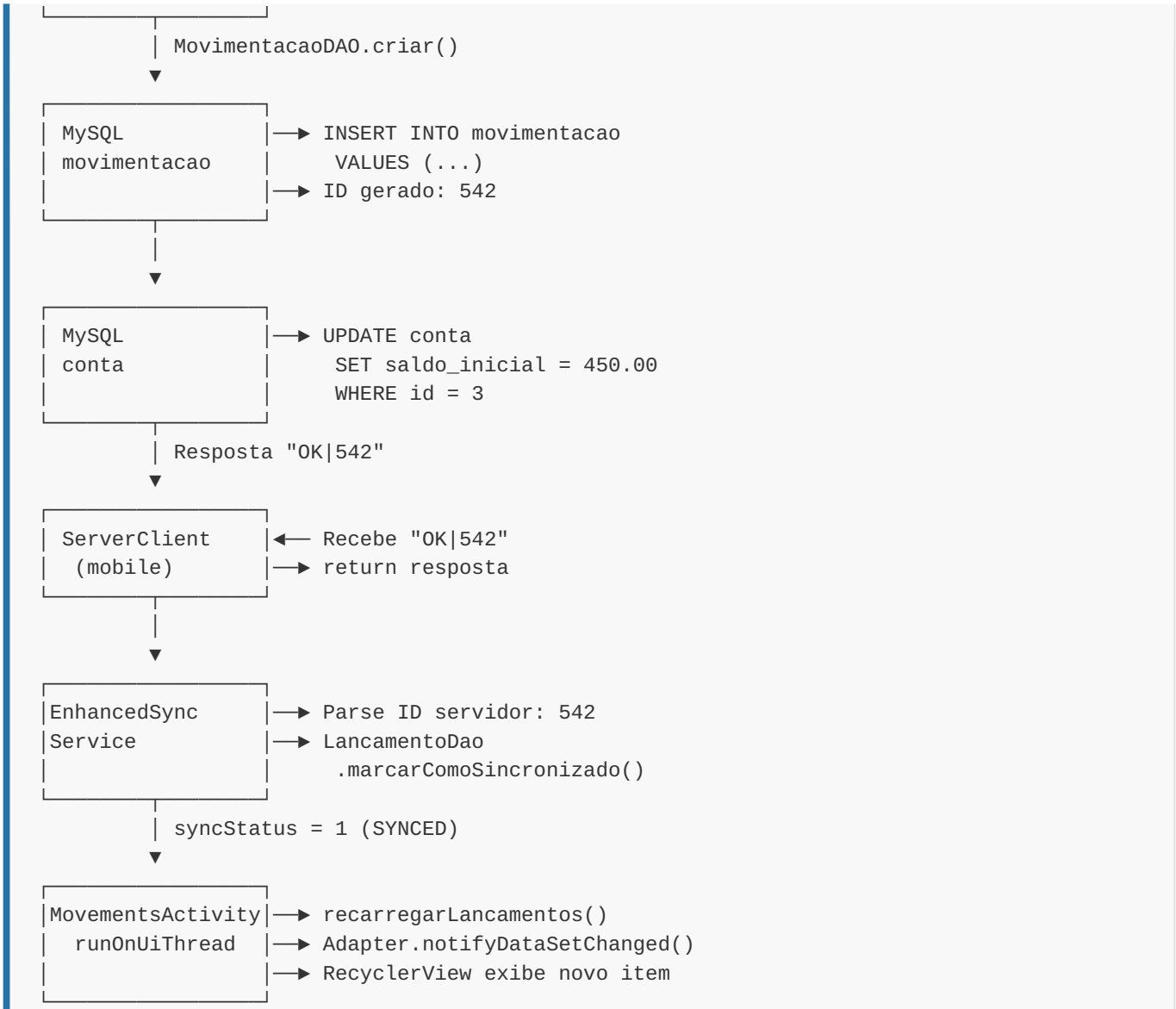
(542 é o ID gerado no MySQL)

No mobile, atualização final:

```
// MovementsActivity.java - continua...
runOnUiThread(() -> {
    // Lista é recarregada
    List<Lancamento> lancamentos = lancamentoDao
                                   .listarAtivosPorUsuario(usuarioId);
    adapter.setLancamentos(lancamentos);
    adapter.notifyDataSetChanged();

    // RecyclerView rola para o topo (mais recente)
    recyclerView.smoothScrollToPosition(0);

    // Badge ou resumo é atualizado
    atualizarResumoFinanceiro();
});
```

Resumo Visual do Fluxo Criar Lançamento





Tempo Total: 0.5-2 segundos **Arquivos Modificados:** 4 (2 tabelas SQLite, 2 tabelas MySQL) **Threads:** 4 (UI, validação, inserção local, sincronização) **Queries SQL:** 6 (2 SELECTs validação, 2 INSERTs, 2 UPDATEs) **Dados Sincronizados:** Lançamento + Conta

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

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:
    4. Envia ao servidor via `ServerClient`
    5. Aguarda confirmação
    6. Marca como sincronizado ou trata erro
7. Busca atualizações do servidor
8. Aplica atualizações localmente via DAOs
9. `ConflictResolutionManager` resolve conflitos se houver
10. Notifica activities sobre dados atualizados

🎨 CAMADA DE INTERFACE

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

Desktop - Views Swing

- **LoginView.java** → JFrame de login
- **AdminDashboardView.java** → JFrame principal com JTable de usuários
- **EditarUsuarioDialog.java** → JDialog modal para edição

💾 BANCO DE DADOS

Mobile - Room (SQLite)

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`

Tabelas e Campos Completos

Tabela: usuario

```
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 millisegundos
    lastModified INTEGER NOT NULL,           -- Timestamp última modificação
    syncStatus INTEGER NOT NULL DEFAULT 0,   -- 0=LOCAL, 1=SYNCED, 2=NEEDS_SYNC, 3=CONFLICT
    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);
```

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

Tabela: conta

```
CREATE TABLE conta (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    uuid TEXT NOT NULL UNIQUE,
    nome TEXT NOT NULL,
    tipo TEXT NOT NULL,                      -- 'corrente', 'poupanca', 'cartao', 'investimento', 'd
    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);
```

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

Tabela: categoria

```
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);
```

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

Tabela: lancamento

```
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);
```

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

Servidor - MySQL

Configuração

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

Tabelas e Campos Completos

Tabela: usuario

```
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;
```

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`

Tabela: conta

```
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;
```

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

Tabela: categoria

```
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;
```

Tabela: movimentacao

```
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;
```

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

Script de Migração

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 DEFAULT 'usuario';
UPDATE usuario SET tipo_usuario = 'admin' WHERE email = 'admin@finanza.com';
```

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

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
```

📡 COMUNICAÇÃO

Protocolo TCP/IP

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

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)

🔒 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

🚀 EXECUÇÃO

Mobile

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+)

Passos de Instalação

1. **Clonar o Repositório**

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

1. **Abrir no Android Studio**
    2. File > Open > Selecionar pasta do projeto
    3. Aguardar sincronização do Gradle
    4. Resolver dependências automaticamente
5. **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 do servidor
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)

1. **Compilar o Projeto**
   2. Build > Make Project (Ctrl+F9)
   3. Verificar erros no Build Output
   4. Garantir que todas as dependências foram baixadas
5. **Executar no Dispositivo/Emulador**
   6. Run > Run 'app' (Shift+F10)
   7. Selecionar dispositivo/emulador
   8. Aguardar instalação do APK
9. **Gerar APK para Distribuição**

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

Desktop (Servidor)

Requisitos

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

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
```

1. **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"
```

1.

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";
```

1.

Compilar e Executar o Servidor

Via NetBeans:

2. File > Open Project > Selecionar ServidorFinanza
3. Clean and Build (Shift+F11)
4. Run Project (F6)

Via Linha de Comando:

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

1.

Verificar Servidor em Execução

O servidor estará escutando na porta 12345:

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

Desktop (Cliente Admin)

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;
```

1.
Compilar e Executar

Via NetBeans:

2. File > Open Project > Selecionar ClienteFinanza
3. Clean and Build
4. Run Project

Via Linha de Comando:

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

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

🔧 CONFIGURAÇÃO E INSTALAÇÃO

Configuração de Rede

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
```

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
```

Variáveis de Ambiente

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)
```

Logs e Depuração

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
```

Servidor (server.log)

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

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

⚠️ TROUBLESHOOTING

Problemas Comuns e Soluções

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");
```

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)
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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;
```

Logs Úteis para Debug

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");
```

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());
```

Ferramentas de Diagnóstico

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
```

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;
```

📞 SUPORTE E CONTRIBUIÇÃO

Contato

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

Contribuindo

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

Licença

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