Finance Buddy

Documentação do Software

Release 1.0

DARLEY LEAL DOS SANTOS
Curitiba, Paraná - 2025

Sumário

1. INTRODUÇÃO

O Finance Buddy é um aplicativo de gestão financeira desenvolvido para atender à necessidade de usuários que buscam uma forma simples e visual de acompanhar seus gastos e rendimentos. O aplicativo permite gerenciar despesas e receitas de maneira prática, oferecendo funcionalidades como o monitoramento de rendimentos e despesas, a criação de categorias personalizadas, a exibição de gráficos e relatórios mensais, além de opções de edição e exclusão de registros. Com base nas histórias de usuário, o Finance Buddy foi projetado para auxiliar os usuários a terem maior controle financeiro e tomarem decisões mais conscientes sobre seus gastos.

2. HISTÓRIA DE USUÁRIO

As histórias de usuário representam requisitos descritos do ponto de vista do usuário final, destacando suas necessidades e objetivos. A tabela a seguir apresenta as histórias identificadas para o desenvolvimento do aplicativo, detalhando as funcionalidades esperadas e alinhadas à experiência do usuário.

Tabela 1 – Histórias de Usuário do Aplicativo

ID	Descrição
US01	Como usuário, desejo que o aplicativo mostre na tela inicial os rendimentos e despesas.
US02	Como usuário, desejo poder adicionar um valor no qual posso ter como base o que posso gastar naquele mês e, desejo poder alterá-lo quando necessário.
US03	Como usuário, desejo que ao adicionar novas despesas esse valor seja decrementado conforme a soma total desses itens.
US04	Como usuário, desejo que o aplicativo mostre as despesas e rendimentos mensais, que serão a soma de todos os itens.
US05	Como usuário, desejo poder adicionar categorias personalizadas para as despesas e rendimentos, como: Faculdade, Conta de luz, Conta d'água, Internet, Alimentação, Cartão de crédito para despesas; Salário e Freelance para rendimentos.
US06	Como usuário, desejo que o aplicativo tenha uma tela de cadastro onde eu possa adicionar novas despesas, rendimentos e categorias.
US07	Como usuário desejo ter as opções de edição e remoção para os registros.
US08	Como usuário, desejo que o aplicativo mostre na tela principal os cards com cada rendimento ou despesa e suas informações de registro.
US09	Como usuário, desejo uma tela com relatórios em gráficos com as despesas e rendimentos por mês.
US10	Como usuário, desejo que a tela de relatórios liste todos os rendimentos e despesas agrupados por data.

3. REQUISITOS NÃO FUNCIONAIS

Tabela 2 – Restrições e tecnologias usadas no aplicativo

ID	Descrição
RNF01	O aplicativo deve oferecer autenticação por padrão ou leitor de digital para garantir a segurança do acesso do usuário.
RNF02	O sistema operacional alvo será Android, utilizando a linguagem Kotlin e o framework Jetpack Compose para o desenvolvimento da interface.

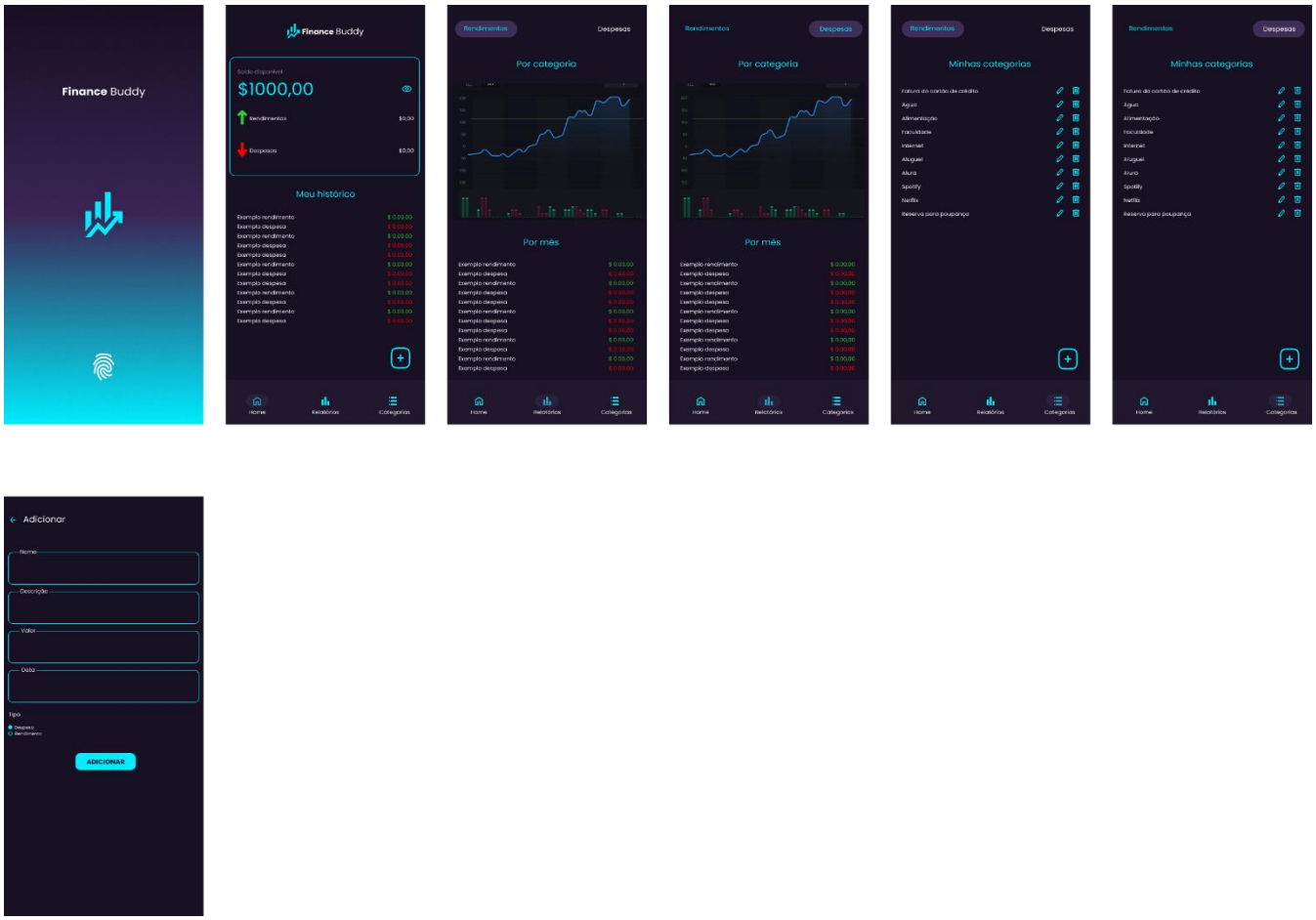
4. DESIGN E INTERFACE

O design de interface do FinanceBuddy segue as diretrizes do **Material Design 3**, proporcionando uma experiência visual intuitiva e acessível. Foram utilizadas cores que ajudam os usuários a identificar rapidamente informações importantes:

- **Tons de vermelho:** Utilizados nos ícones relacionados a despesas, sinalizando de forma clara os gastos.
- **Tons de verde:** Aplicados aos ícones de rendimentos, indicando entradas financeiras e gerando uma percepção positiva.

As mensagens apresentadas na interface são intuitivas e orientam o usuário sobre as ações que podem ser realizadas. Além disso, o aplicativo inclui imagens e ilustrações selecionadas para tornar a experiência mais amigável, alinhando funcionalidade e estética. O design foi criado utilizando o Figma, permitindo a prototipagem e a visualização antecipada das telas principais. Na sequência, será apresentada uma imagem exemplificando a interface desenvolvida.

Figura 1 – Protótipo do aplicativo, desenvolvido com o uso da ferramenta Figma.



5. ARQUITETURA

A arquitetura do Finance Buddy segue os princípios do Clean Architecture em conjunto com o padrão MVVM (Model-View-ViewModel), que separa cada camada conforme a sua responsabilidade.

5.1 Camadas do Aplicativo:

- Presentation Layer (Camada de Apresentação): É responsável pela interface com o usuário (UI) e pela interação com o ViewModel. Ela consome os dados recebidos da camada de domínio e os exibe. O MVVM é utilizado aqui, com ViewModels gerenciando o estado com as UIStates e interagindo com os UseCases.
    - App
        - A pasta contém a MainActivity, onde as ViewModels são instanciadas e passadas como parâmetro para o navigationProvider. As funções showSplashScreen e showBiometricLogin também são chamadas na activity. No setContent{}, a orientação é bloqueada para retrato com SCREEN_ORIENTATION_PORTRAIT na requestOrientation.
    - Components
        - Essa pasta é responsável por agrupar todos os componentes usados pelas telas do aplicativo:
            - **AnalyticsChartCard:** É o componente que gera o gráfico com os rendimentos e despesas agrupados por mês, as barras são criadas conforme a soma dos valores por tipo;
            - **CategoryItemCard:** É responsável por agrupar o nome da categoria, ícone de edição e de remoção na tela de Categoria e, é listado conforme o tipo, ao clicar em *"expense"* são listados todos com a categoria com esse nome e o mesmo acontece com os *"income"*;
            - **CategoryModalBottomSheet:** Este *BottomSheet* é acionado ao clicar no FAB na tela de categoria, permitindo a adição de novas categorias;
            - **CategorySection:** É responsável por listar os objetos do tipo categoria, conforme lista passado como parâmetro. Se a lista estiver vazia apresenta o composable *"ItemsNotFound"*;
            - **CustomExpandableFloatingButton:** É o FAB (*Floating Action Button*) da tela inicial que se expande ao ser clicado, exibindo opções ao usuário;
            - **CustomTextField:** Esse composable representa o *TextField*, porém com mais parâmetros e, é reutilizado nas telas onde é necessário a adição ou edição das informações;
            - **DatePickerField:** Representa o *DatePickerDialog* exibido para que o usuário insira a data;
            - **EditCategoryNameBottomSheet:** É acionado ao clicar no ícone de edição, exibindo as informações atuais do objeto em campos editáveis para alteração dos atributos;
            - **HistoryInformations:** Representa os registros na *HomeScreen*;

- **HomeScreenTopAppBar:** Representa a *TopApBar* utilizada na *HomeScreen*, contém o ícone e nome do aplicativo;
- **ItemsNotFound:** Composable reutilizado em partes do aplicativo onde a lista passada está vazia, indicando que não há informações salvas e que é necessário inserir dados;
- **OpenInFullScreenRegistrationModalSheet:** Exibe todas as informações do registro em um *BottomSheet*;
- **RadioButtomSelection:** Exibe as categorias para seleção no momento de adição de novos registros;
- **RadioButtomSingleSelectionCategory:** Mostra as categorias de seleção *"Income"* e *"Expense"* ao clicar no FAB na tela de categorias;
- **RemoveItemDialog:** É exibido quando usuário clica no ícone de lixeira, para confirmar a remoção de um item;
- **TypeOptionsTopAppBar:** Apresenta os botões *"Income"* e *"Expense"* na tela de categorias, exibindo a lista de acordo com o tipo selecionado;
- **TypeRegistration:** Exibe as categorias disponíveis para registro;
- **UpdateBalanceDialog:** Exibe um *dialog* para edição do valor disponível para gasto ao clicar no FAB da HomeScreen, na opção *"Balance"*; e
- **UpdateRegistrationModalBottomSheet:** Exibe um *bottomSheet* para edição de registros.

- Navigation
  - Essa pasta agrupa os arquivos responsáveis pela navegação entre telas do aplicativo.
    - **AppNavigation:** Responsável pelo *NavHost* com as rotas das telas;
    - **BottomNavigationRoute:** *Data class* que representa os objetos na barra de navegação na parte inferior da tela; e
    - **NavigationProvider:** Funciona como um contêiner e fornecedor de ViewModels para diferentes partes do aplicativo. Recebe várias *ViewModels* via injeção de dependência e as armazena em um mapa, onde a chave é um *enum ViewModelKey* e o valor é a *ViewModel* correspondente. O método *getViewModel* permite recuperar uma *ViewModel* específica pela chave, lançando uma exceção se a chave não for encontrada.
- Screens
  - Agrupa as telas do aplicativo, suas viewModels e UiStates, sendo responsável pela interação com o usuário.
    - **Analytics:** Nesta pasta contém os a *AnalyticsScreen* e a sua *viewModel* e, é responsável pela tela de relatórios do aplicativo;
    - **Categories:** Esta pasta contém a *CategoryScreen* e sua *viewModel*. Por se tratar de uma tela abrangente que inclui informações sobre rendimentos e despesas, ela é subdividida nas seguintes subpastas:

        - **Category_Expenses:** Contém a *viewModel* e o *composable* responsável pela apresentação e atualização do estado das categorias de despesas;
        - **Category_Incomes:** Contém a *viewModel* e o *composable* responsável pela apresentação e atualização do estado das categorias de rendimentos;
    - **Home:** Esta pasta agrupa a *HomeScreen* e sua *viewModel*, além da *HomeUiState* responsável pela alteração do estado das informações na tela conforme interação com o usuário. Nesta, contém a seguinte subpasta:
        - **Card_Information:** Pasta que agrupa o composable do *card* na *HomeScreen*, mostrando o valor disponível, somas de rendimentos e despesas. Possui *viewModel* e *UiState* próprios para atualizar informações conforme mudanças de estado.
    - **Insert:** Esta pasta agrupa a *InsertScreen*, responsável pela adição de novos registros, sua *viewModel* e *UiState;*
    - **MainScreen:** É o aplicativo que representa todas as telas e é o composable principal chamado pela *activity*;
    - **Start:** É responsável pela tela inicial do aplicativo no seu início no qual solicita ao usuário a sua impressão digital ou "padrão".
  - **Theme:** Responsável pelo tema, cores e fontes do aplicativo. Utiliza a fonte *Poppins* como padrão.
- Domain Layer (Camada de Domínio): A camada de domínio contém a lógica de negócios central do aplicativo, composta por *UseCases* que encapsulam regras de negócio específicas. Aqui, são definidos os casos de uso que atendem às necessidades dos usuários, como adicionar e editar despesas e rendimentos, as funções de conversão dos objetos, utilidades da *Activity* e *enums*.
  - Enums: Nesta pasta contém os enumeradores usados no aplicativo.
    - **Routes:** Classe *Enum* que define os nomes das rotas usadas pelo *Navigation*;
    - **Type:** Classe *Enum* que representa os tipos *Income* e *Expense*;
    - **ViewModelKey:** *Enum* que representa os nomes das *viewModels*, usado pelo *provider* para buscar a *viewModel* no mapa.
  - UserCases: Esta pasta contém as classes que implementam as regras de negócios: *BalanceUserCase*, *CategoryUserCase* e *RegistrationUserCase*, além das respectivas funções CRUD.
  - Utils: Esta pasta agrupa funcionalidades importantes do aplicativo:
    - **ActivityUtils:** Este arquivo agrupa funcionalidades utilizadas na *activity*. A função *splashScreen()* carrega o ícone e uma tela inicial enquanto o aplicativo é carregado e a tela *StartScreen* é chamada pela navegação. O método *showBiometric()* exibe as opções de autenticação ao usuário;
    - **ConvertDate:** Função que converte datas para o formato inglês com mês por extenso, dia e ano;
    - **ConvertToCurrency:** Converte os valores de moeda para um padrão;
    - **FormatMonthAndYear:** Formata a data para uma forma abreviada, de mês e ano; e

        - **ToBrazilianDateFormat:** Função de extensão do tipo primitivo *Long*, no qual converte a data para o padrão brasileiro.

- Data Layer (Camada de Dados): A camada de dados fornece e persiste os dados do aplicativo. É composta por *Repositories*, que interagem com bancos de dados locais (via *DAO* e *Room Database*) abstraindo as fontes para que a camada de domínio não precise conhecer os detalhes de armazenamento ou recuperação

    - Dao: Esta pasta contém as interfaces para manipulação de dados no banco, como *BalanceDao*, *CategoryDao* e *RegistrationDao*;

    - Database: Esta pasta contém o arquivo *AppDatabase*, uma classe abstrata que representa o banco de dados do aplicativo usando o *Room*. A anotação *@Database* define as entidades *Registration*, *Category* e *Balance* como tabelas do banco. O arquivo inclui funções *CRUD* para interagir com o *Room*;

    - Models: Esta pasta contém classes de modelo que representam tipos de objetos, sendo elas: *Balance, Category* e *Registration;*

    - Repository: Representa as classes que armazenam os valores recebidos pelas consultas das interfaces e disponibilizam para *userCases* conforme solicitados, sendo elas *BalanceRepository, CategoryRepository* e *RegistrationRepository;*

- Dependency Injection (Injeção de Dependências): A injeção de dependências é gerenciada utilizando o *Hilt* para fornecer as dependências necessárias ao longo do aplicativo. Isso permite que as instâncias de *ViewModels*, *UseCases* e *Repositories* sejam automaticamente injetadas nas classes que precisam delas, para que não haja acoplamento entre as camadas.

    - AppModule: Esse arquivo define um módulo do *Dagger Hilt* responsável por fornecer dependências relacionadas ao banco de dados do aplicativo. Ele usa anotações como *@Module*, *@InstallIn*, *@Provides* e *@Singleton* para configurar e fornecer as dependências solicitadas.

    - FinanceBuddyApplication: Este arquivo serve como ponto de entrada do aplicativo e é anotado com o *@HiltAndroidApp* para inicializar o *Dagger Hilt* durante a inicialização do app permitindo que ele gerencie as suas dependências.

6. TESTES

6.1 Testes Unitários

A classe *HomeViewModelTest* foi criada para realizar testes unitários da *HomeViewModel*, para que a lógica da *ViewModel* funcione como esperado. Os seguintes casos de teste foram implementados:

1. Verificar se o valor do campo nome é atualizado: Este caso de teste verifica se, quando a função *updateName()* é chamada com um novo nome, o campo name no *uiState* é atualizado de acordo. Isso garante que a *ViewModel* esteja atualizando corretamente o estado da interface do usuário quando o nome é alterado.

```
@Test
fun `Verify if value for name field is updated`() {
    viewModel.updateName("Teste")
    val uiState = viewModel.uiState

    assertThat(uiState.value.name == "Teste").isTrue()
}
```

2. Validar se os campos do formulário não estão vazios: Este caso de teste verifica se, quando todos os campos do formulário são atualizados com valores válidos, a função *validateFormFields()* retorna *true*, indicando que o formulário é válido. Isso garante que a *ViewModel* esteja validando corretamente os dados do formulário antes de prosseguir com qualquer operação.

```
@Test
fun `Validate forms field is not empty`() {

    viewModel.updateName("Darley")
    viewModel.updateDescription("Teste description")
    viewModel.updateValue("2025")
    viewModel.updateDate("26/01/2025")

    assertThat(viewModel.validateFormFields()).isTrue()
}
```

3. Verificar se a lista de registros está vazia: Este caso de teste verifica se, quando a função *getAllRegistrations()* é chamada, a lista registrations no *uiState* está inicialmente vazia. Isso garante que a *ViewModel* esteja inicializando corretamente o estado da interface do usuário com uma lista de registros vazia.

```
@Test
fun `Verify if registrations list is empty`() {
    viewModel.getAllRegistrations()
    val uiState = viewModel.uiState

    assertThat(uiState.value.registrations).isEmpty()
}
```

4. Código completo do caso de teste:

```
package com.darleyleal.financebuddy

import android.content.Context
import com.darleyleal.financebuddy.domain.usercase.CategoryUserCase
import com.darleyleal.financebuddy.domain.usercase.RegistrationUserCase
import com.darleyleal.financebuddy.presetation.screens.home.HomeViewModel
import com.google.common.truth.Truth.assertThat
import dagger.hilt.android.testing.HiltAndroidTest
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.test.StandardTestDispatcher
import kotlinx.coroutines.test.resetMain
import kotlinx.coroutines.test.setMain
import org.junit.After
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.Mock
import org.mockito.junit.MockitoJUnitRunner

@HiltAndroidTest
@RunWith(MockitoJUnitRunner::class)
class HomeViewModelTest {
    @Mock
    private lateinit var context: Context

    @Mock
    private lateinit var registrationUserCase: RegistrationUserCase

    @Mock
    private lateinit var categoryUserCase: CategoryUserCase

    private lateinit var viewModel: HomeViewModel

    @OptIn(ExperimentalCoroutinesApi::class)
    @Before
    fun setUp() {
        Dispatchers.setMain(StandardTestDispatcher())
        viewModel = HomeViewModel(registrationUserCase, categoryUserCase)
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @After
    fun tearDown() {
        Dispatchers.resetMain()
    }

    @Test
    fun `Verify if value for name field is updated`() {
        viewModel.updateName("Teste")
        val uiState = viewModel.uiState

        assertThat(uiState.value.name == "Teste").isTrue()
    }

    @Test
```

```
    fun `Validate forms field is not empty`() {

        viewModel.updateName("Darley")
        viewModel.updateDescription("Teste description")
        viewModel.updateValue("2025")
        viewModel.updateDate("26/01/2025")

        assertThat(viewModel.validateFormFields()).isTrue()
    }

    @Test
    fun `Verify if registrations list is empty`() {
        viewModel.getAllRegistrations()
        val uiState = viewModel.uiState

        assertThat(uiState.value.registrations).isEmpty()
    }
}
```

6.2 Testes de UI/UX

Além dos testes unitários mencionados acima, também foram realizados testes de UI/UX em dois dispositivos Android (Android 11 e Android 14) e dois emuladores (ambos Android 14). Esses testes se concentraram em verificar o comportamento dos Composable e das telas, bem como o desempenho geral do aplicativo.

- Comportamento dos Composable e Telas

Os testes de UI/UX avaliaram a interação do usuário com os elementos da interface, como botões, campos de texto e listas. O objetivo era garantir que os Composable respondessem corretamente às ações do usuário e que as telas fossem exibidas e navegadas conforme o esperado.

- Testes de Conexão

O comportamento do aplicativo em diferentes condições de conectividade também foi testado. Os testes simularam cenários em que o dispositivo estava conectado e desconectado da internet para verificar como o aplicativo se comportava em cada situação. O objetivo era garantir que o aplicativo lidasse com a falta de conexão de forma adequada, para que não travesse ou apresetasse lentidões.

7. GESTÃO DO PROJETO

Tabela 3 – Cronograma de atividades

ID	Descrição	Esforço em horas
01	Levantar Requisitos	8
02	Refinar os Requisitos	4
03	Analisar solução	8
04	Estudar sobre Design e Figma	16
05	Criar prótotipo do aplicativo	10
06	Desenvolvimento	144
07	Documentação	10
08	Testes	8
09	Estudo sobre publicação	2

10	Deploy na Google Play	**5**