UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus da Costa Brito

Desenvolvendo um sistema de controle financeiro pessoal

Uberlândia, Brasil

2026

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus da Costa Brito

Desenvolvendo um sistema de controle financeiro pessoal

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Paulo Rodolfo da Silva Leite Coelho

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2026

Matheus da Costa Brito

Desenvolvendo um sistema de controle financeiro pessoal

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 27 de fevereiro de 2026:

Paulo Rodolfo da Silva Leite Coelho
Orientador

Professor

Professor

Uberlândia, Brasil
2026

Resumo

Este trabalho apresenta o desenvolvimento de um Sistema de Controle Financeiro Pessoal, com o objetivo de auxiliar usuários no registro e acompanhamento de suas receitas e despesas de forma simples, intuitiva e acessível. Diante do elevado índice de endividamento das famílias brasileiras e da dificuldade de adoção de ferramentas complexas ou pagas, o sistema proposto busca oferecer uma solução web focada na simplicidade de uso, permitindo o cadastro de entradas e saídas financeiras, categorização das movimentações e visualização do balanço mensal por meio de *dashboards* informativos.

A metodologia adotada envolve a definição de requisitos funcionais e não funcionais, modelagem de dados, projeto da arquitetura do sistema e implementação utilizando tecnologias modernas. O *frontend* foi desenvolvido com o *framework* Angular, proporcionando uma interface responsiva e interativa, enquanto o *backend* foi implementado em Python com o *framework* FastAPI, seguindo o padrão de arquitetura RESTful. Para persistência dos dados, utilizou-se o banco de dados SQLite, aliado ao ORM (Object Relational Mapper) SQLAlchemy, garantindo simplicidade, portabilidade e integridade das informações.

Como resultado, foi obtido um sistema funcional que atende aos requisitos propostos, permitindo a organização financeira pessoal sem a necessidade de conhecimentos técnicos avançados ou integrações bancárias. O sistema demonstrou ser uma alternativa viável para usuários que buscam uma ferramenta prática para controle financeiro, além de servir como base para futuras expansões, como geração de relatórios avançados e novas funcionalidades de análise financeira.

Palavras-chave: Controle financeiro pessoal, Sistema web, Gestão financeira, Angular, FastAPI.

Lista de ilustrações

Lista de abreviaturas e siglas

API	Application Programming Interface
CSS	Cascading Style Sheets
CNC	Confederação Nacional do Comércio de Bens, Serviços e Turismo
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IBGE	Instituto Brasileiro de Geografia e Estatística
JWT	JSON Web Token
MVC	Model-View-Controller
ORM	Object-Relational Mapping
RF	Requisito Funcional
RNF	Requisito Não Funcional
REST	Representational State Transfer
RxJS	Reactive Extensions for JavaScript
SCFP	Sistema de Controle Financeiro Pessoal
SGBD	Sistema Gerenciador de Banco de Dados
SQL	Structured Query Language
SQLite	Structured Query Language Lite
UI	User Interface
UFU	Universidade Federal de Uberlândia

Sumário

1 Introdução

A organização financeira pessoal é uma prática essencial para manter o equilíbrio econômico em um cenário de incertezas econômicas, inflação e facilidade de acesso ao crédito. Segundo o Instituto Brasileiro de Geografia e Estatística (IBGE, 2025), a média salarial no Brasil em 2023 foi de R$ 2.971,00, considerando rendimento médio habitual de todos os trabalhadores ocupados. Apesar da importância de manter o controle sobre as finanças, muitas pessoas ainda realizam esse acompanhamento de maneira informal, como anotações em cadernos e planilhas manuais, isso quando realizam esse controle. Essa falta de organização pode acarretar dificuldades como inadimplência, endividamento e falta de planejamento para objetivos futuros. De acordo com a Confederação Nacional do Comércio de Bens, Serviços e Turismo (CNC, 2024), em março de 2024 o percentual de famílias endividadas no Brasil chegou a 78,1%, ou seja, praticamente 8 em cada 10 famílias têm algum tipo de dívida (em cartão de crédito, cheque especial, carnês etc). Além disso, segundo a mesma pesquisa, cerca de 28,6% das famílias declararam que não conseguiriam pagar suas contas em atraso, caracterizando inadimplência.

Com a popularização das tecnologias web e o aumento da conectividade, surgiram diversas soluções para auxiliar na gestão financeira pessoal. No entanto, muitas dessas ferramentas apresentam interfaces complexas ou requerem conhecimentos técnicos mais avançados, tornando seu uso desafiador para usuários comuns que buscam apenas funcionalidades básicas como o registro de receitas e despesas e a visualização do saldo mensal. Uma pesquisa do SPCBrasil/CNDL (2024) mostrou que 35% dos brasileiros inadimplentes não controlam seu orçamento mensal, ou fazem apenas um controle mental, sem nenhuma anotação, aumentando as chances de perder o controle das finanças. Essa mesma pesquisa apontou que as principais razões para aqueles que não controlam o orçamento são por falta de disciplina para controlar os gastos, falta de tempo e não saber como fazer.

Embora existam diversas ferramentas no mercado, muitas são complexas ou pagas, criando barreiras para usuários que buscam soluções simples. Há uma carência de soluções tecnológicas simples e intuitivas voltadas para o controle financeiro pessoal. A maioria dos aplicativos disponíveis no mercado oferece funcionalidades excessivas ou complexas, que acabam desestimulando seu uso contínuo. Assim, surge a necessidade de um sistema web com foco na simplicidade e eficiência, voltado a usuários que desejam apenas registrar suas movimentações financeiras e visualizar de maneira clara seu balanço mensal.

Este trabalho propõe o desenvolvimento de um sistema web para controle financeiro pessoal, que permita o cadastro de entradas e saídas de recursos financeiros, bem como a visualização do balanço mensal de forma clara, objetiva e responsiva. O projeto

visa oferecer uma alternativa prática e acessível para usuários que necessitam de uma ferramenta simples para gerenciar suas finanças. Surge, assim, a necessidade de uma aplicação acessível e intuitiva, que auxilie o usuário na organização financeira sem depender de integrações bancárias. Ao focar em uma interface intuitiva e em funcionalidades básicas, o sistema pretende ser uma solução de entrada para o público leigo, incentivando hábitos de organização financeira e evitando o uso de sistemas mais robustos e muitas vezes desnecessários para a maioria dos usuários.

Para que o objetivo do trabalho fosse atingido, foram escolhidas ferramentas que vão garantir a qualidade e eficiência no seu desenvolvimento. Usando de uma arquitetura moderna baseada em Application Programming Interfaces (APIs) RESTful, o *framework* Javascript Angular (2025) foi escolhido para desenvolvimento *frontend* pois além de ser uma ferramenta moderna permite a expansão da aplicação de acordo com seu crescimento. A escolha para o *backend* foi do *framework* FastAPI (2025) juntamente com Python pois além da facilidade na implementação e documentação, possui um bom desempenho e suporte a validação de dados. Por último um sistema de banco de dados leve e embarcado, o Structured Query Language Lite SQLite (2025), que vai permitir armazenar os dados obtidos na aplicação sem necessidade de um servidor especifico de banco de dados, e servindo como base para expansões futuras ou customizações conforme as necessidades do usuário, juntamente com Docker (2025), uma plataforma de virtualização leve baseada em contêineres, utilizada para empacotar a aplicação e suas dependências.

2 Requisitos do Sistema

2.1 Descrição Geral

O sistema proposto tem como objetivo auxiliar usuários comuns na gestão de suas finanças pessoais, permitindo o registro de receitas, despesas e o acompanhamento do saldo mensal. Os requisitos foram definidos com base na simplicidade, acessibilidade e facilidade de uso.

O sistema deve permitir cadastrar um usuário, esse usuário poderá efetuar o *login* no sistema e a partir desse ponto ele poderá cadastrar as categorias de receitas e despesas de acordo com seu perfil. Em seguida o sistema permitirá o cadastro e visualização de receitas e/ou despesas, podendo ate mesmo filtra-las. Após esse cadastro o usuário poderá visualizar o balanço de suas despesas por meio de *dashboards*, sendo esses *dashboards* separados em entradas e saídas de acordo com filtro escolhido e entrada e saída de acordo com as categorias criadas.

Esse sistema foi criado principalmente para pessoas que possuem dificuldades na hora de fazer o controle de seus gastos e não conseguem gerencia-los por meio de planilhas ou anotações no caderno e não possuem condições de pagar por um serviço que seria mais complexo e demandaria um tempo a mais para aprender a utilizar.

2.2 Requisitos Funcionais

ID	Requisito Funcional	Descrição
RF01	Cadastro de usuário	Permitir que novos usuários se cadastrem com nome, e-mail, senha e telefone (opcional).
RF02	*Login* de usuário	Permitir que usuários façam *login* com e-mail e senha para acessar seus dados pessoais.
RF03	*Logout* de usuário	Encerrar a sessão do usuário de forma segura.
RF04	Cadastro/Edição de categorias	Permitir que o usuário crie e/ou edite categorias para classificar movimentações (ex: “Alimentação”, “Salário”, “Transporte”).
RF05	Exclusão de categorias	Permitir que o usuário exclua categorias criada para classificar movimentações.
RF06	Visualização de categorias	Exibir todas as categorias do usuário.
RF07	Cadastro/Edição de movimentações financeiras	Permitir registrar e/ou editar movimentações (entradas ou saídas), com data, valor, descrição e categoria associada.
RF08	Exclusão de movimentações	Permitir que o usuário exclua suas movimentações registradas.
RF09	Visualização das movimentações	Exibir todas as movimentações do usuário, com possibilidade de filtrar por mês, tipo (entrada/saída) e categoria.
RF10	Cálculo do saldo mensal	Calcular automaticamente o saldo com base nas entradas e saídas registradas no mês atual.
RF11	Exibição de resumo financeiro	Exibir um resumo com total de entradas, total de saídas e saldo em formato textual e/ou gráfico.
RF12	Exibição resumo financeiro por categoria	Exibir um resumo gráfico de saídas e entradas por categoria.
RF13	Separação por usuário	Garantir que cada usuário só possa visualizar, editar e excluir seus próprios dados e movimentações.

2.3 Requisitos Não Funcionais

ID	Requisito Não Funcional	Descrição
RNF01	Interface responsiva	O sistema deve ser totalmente funcional em diferentes tamanhos de tela.
RNF02	*Backend* leve e performático	O servidor deve utilizar FastAPI e responder rapidamente às requisições do *frontend*.
RNF03	Banco de dados leve e local	Utilizar SQLite como base de dados relacional simples e portátil.
RNF04	API RESTful	A comunicação entre *frontend* e *backend* deve seguir os padrões REST (Representational State Transfer), com *endpoints* organizados e bem definidos.
RNF05	Autenticação segura	Utilizar *hash* de senha (como bcrypt) e *tokens* JSON Web Token (JWT) para autenticação e controle de sessões.
RNF06	Validação de dados	Todos os campos devem ser validados no *frontend* e *backend* (ex: campos obrigatórios, formato de e-mail, valores numéricos positivos).
RNF07	Usabilidade	A interface deve ser simples, com fluxos de uso fáceis de entender, ícones claros e feedback visual (ex: mensagens de sucesso/erro).
RNF08	Estrutura escalável	O sistema deve estar preparado para futuras expansões, como gráficos avançados, exportação de dados, multi-idioma, etc.
RNF09	Boas práticas de código	O projeto deve ser organizado, com padrões de projeto, estrutura modular e separação de responsabilidades.
RNF10	Documentação básica	O sistema deve possuir documentação mínima com instruções para instalação e uso por desenvolvedores.

2.4 Protótipos das interfaces

2.4.1 Protótipo Interface - RF02 - *Login* de usuário

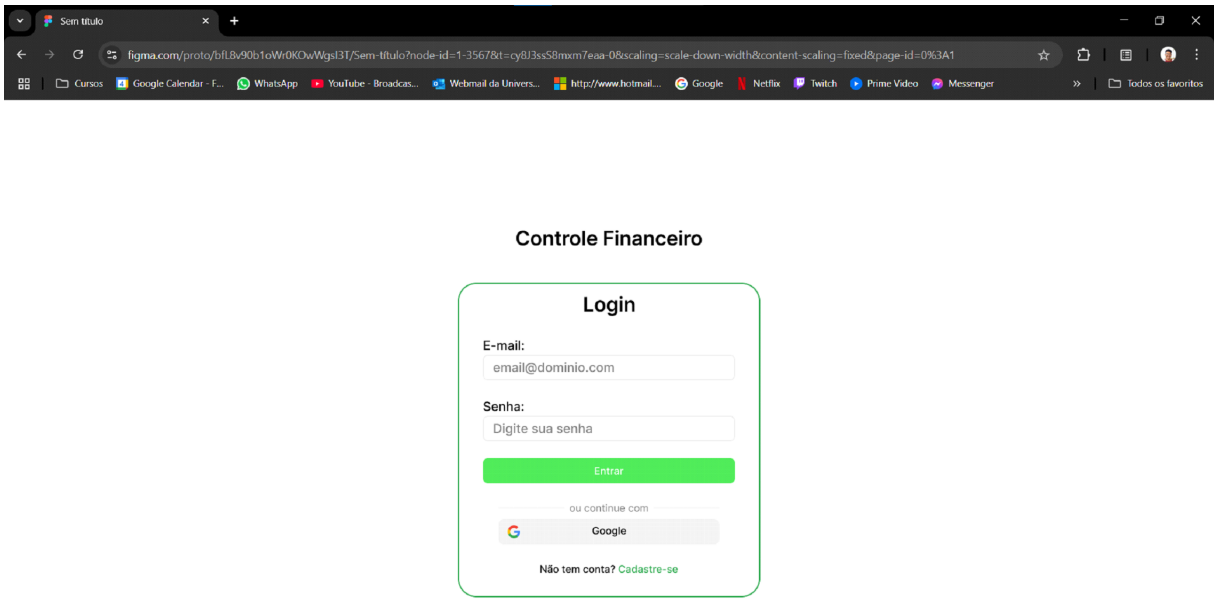

Figura 1 – Exemplo protótipo interface RF02 - Login de usuário.

2.4.2 Protótipo Interface - RF11 e RF12 - Exibição de resumo financeiro e Exibição resumo financeiro por categoria

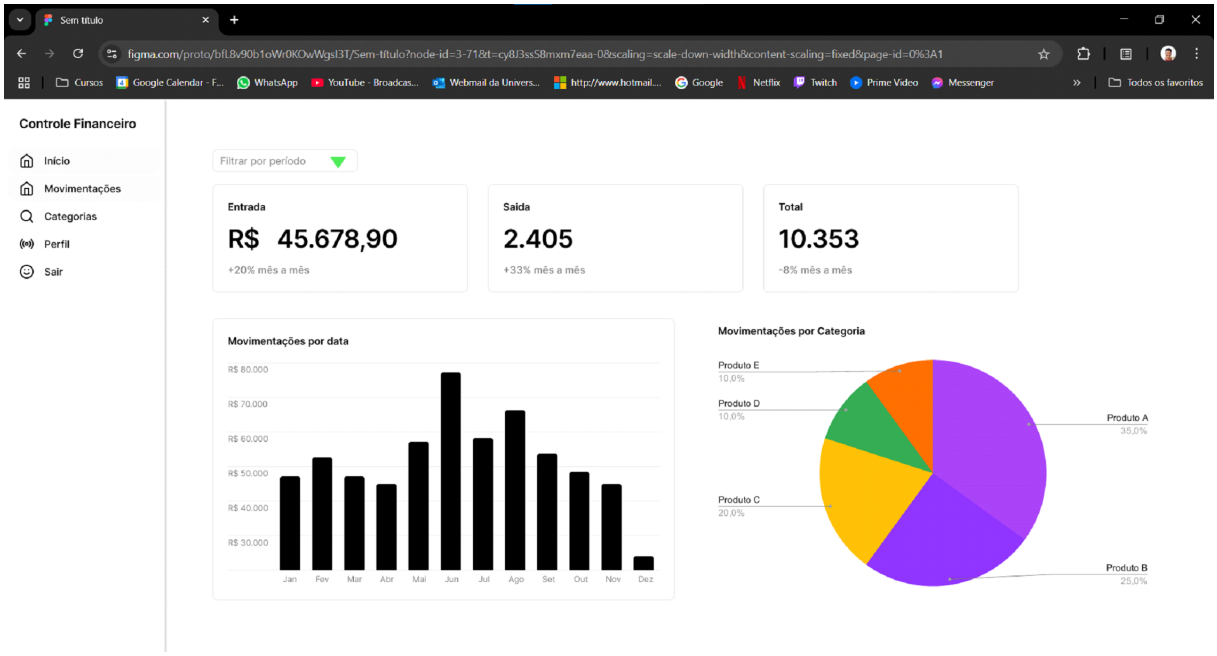

Figura 2 – Exemplo protótipo interface RF11 e RF12 - Exibição de resumo financeiro e exibição resumo financeiro por categoria.

2.4.3 Protótipo Interface - RF09 - Visualização das movimentações

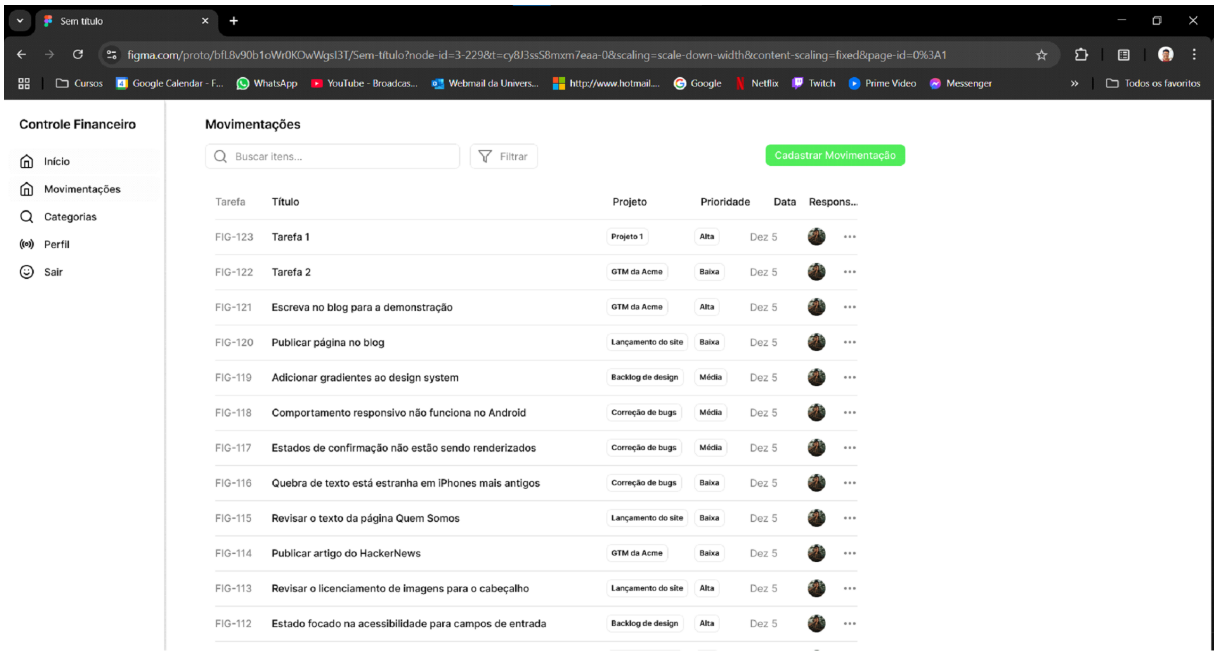

Figura 3 – Exemplo protótipo interface RF09 - Visualização das movimentações.

- Tela de cadastro de usuário será semelhante à de *Login*
- Tela de categorias será semelhante à de movimentações

3 Análise e Projeto do Sistema

3.1 Modelo de Domínio

O modelo de domínio do sistema de controle financeiro pessoal (SCFP) foi elaborado com o objetivo de representar as principais entidades que compõem a aplicação e suas relações. Entre as entidades identificadas estão Usuário, Categoria e Movimentação.

A entidade Usuário representa os dados cadastrais do indivíduo que utiliza o sistema, sendo composta por atributos como nome, e-mail, senha e telefone (opcional). Cada usuário é responsável por gerenciar suas próprias categorias e movimentações financeiras.

A entidade Categoria representa a classificação de uma movimentação e possui nome e descrição (opcional). Cada categoria é associada a um usuário específico, garantindo que as categorias sejam personalizadas conforme o perfil e necessidade de cada pessoa.

A entidade Movimentação é responsável por armazenar as informações de cada registro financeiro, contendo atributos como valor, descrição (opcional), data, tipo (entrada ou saída) e a categoria à qual está vinculada. Cada movimentação pertence a um único usuário e a uma categoria específica.

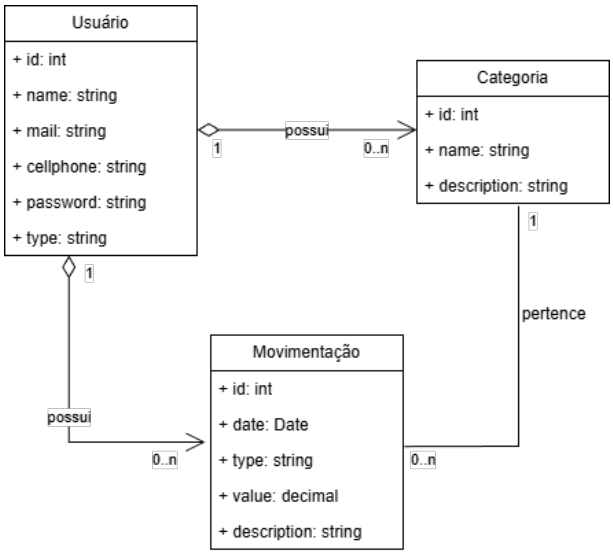

Figura 4 – Modelo Conceitual do Sistema.

Essa estrutura garante o isolamento de dados entre usuários, mantendo a integridade e segurança das informações cadastradas no sistema.

3.2 Diagramas de Sequencia do Sistema

Os diagramas de sequência a seguir foram elaborados para representar o fluxo de mensagens trocadas entre os objetos do sistema durante a execução das principais funcionalidades.

3.2.1 Diagramas de Sequência - RF02 - *Login* de usuário

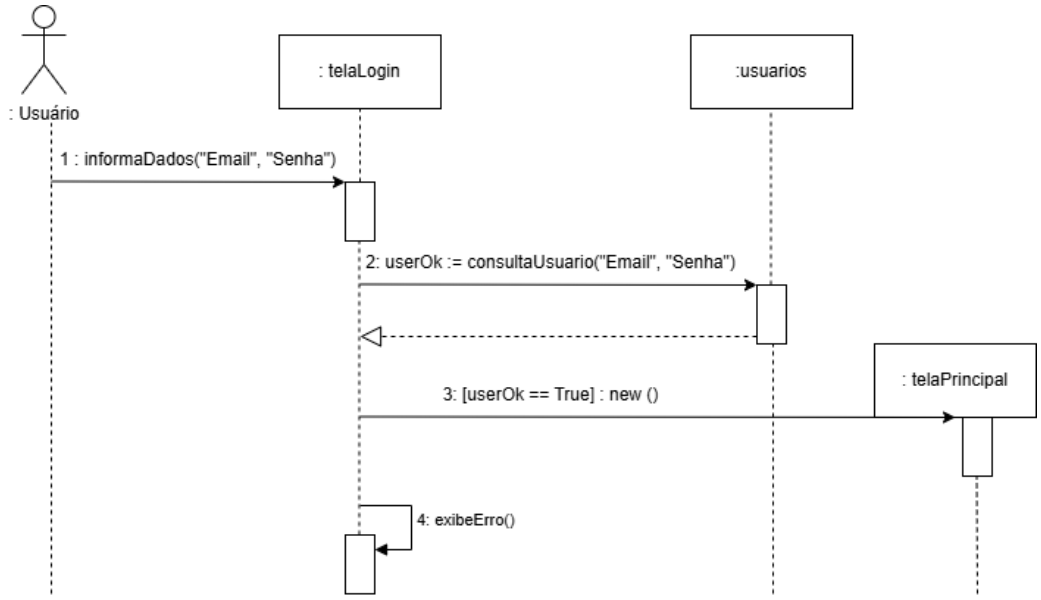

Figura 5 – Diagrama de Sequência - RF02.

3.2.2 Diagramas de Sequência - RF04 - Cadastro de categorias

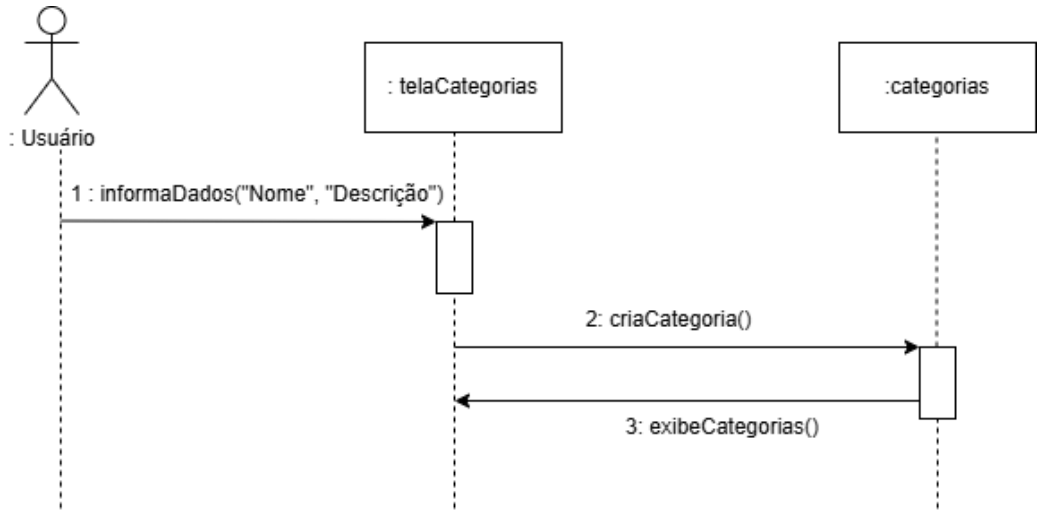

Figura 6 – Diagrama de Sequência - RF04.

3.2.3 Diagramas de Sequência - RF07 - Cadastro de movimentações financeiras

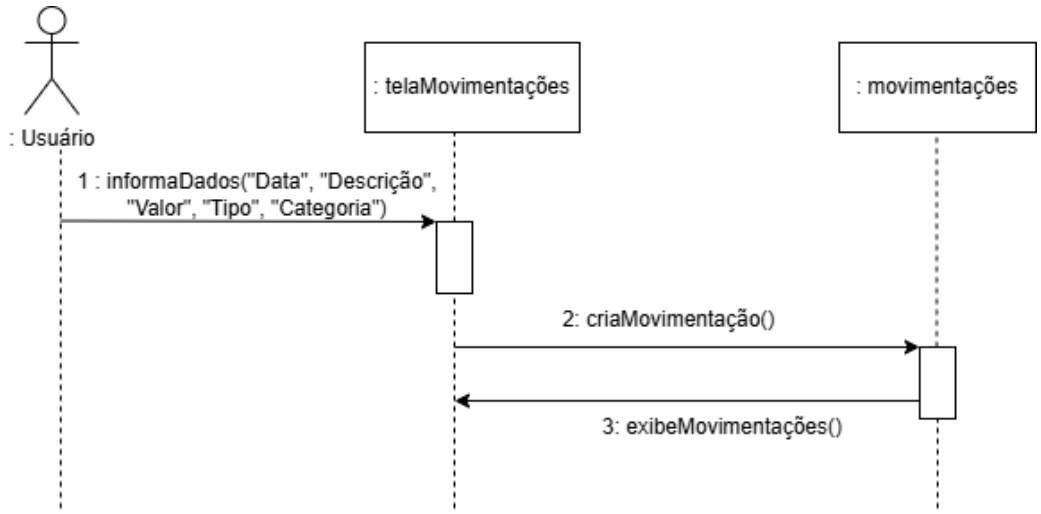

Figura 7 – Diagrama de Sequência - RF07.

3.2.4 Diagramas de Sequência - RF11 - Exibição de resumo financeiro

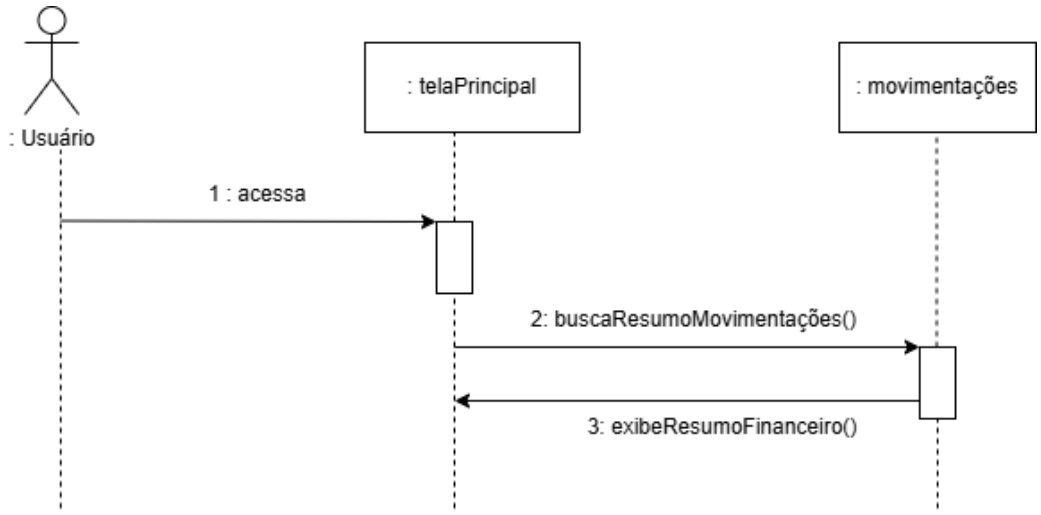

Figura 8 – Diagrama de Sequência - RF11.

3.3 Modelagem de Dados do Sistema

A modelagem de dados do Sistema de Controle Financeiro Pessoal foi elaborada com o objetivo de representar, de forma estruturada, os objetos de dados necessários para o correto funcionamento da aplicação. Esta modelagem segue os requisitos funcionais apresentados anteriormente, contemplando aspectos como cadastro de usuários, gerenciamento de categorias e registro de movimentações financeiras.

O banco de dados foi projetado utilizando um modelo relacional, garantindo a integridade das informações, a consistência dos relacionamentos entre as entidades e a separação dos dados por usuário, atendendo ao requisito RF13 de isolamento de informações. O projeto utiliza o SQLite como Sistema Gerenciador de Banco de Dados (SGBD), devido à sua simplicidade, portabilidade e independência de um servidor dedicado, conforme previsto nos requisitos não funcionais do sistema .

Os nomes das entidades no banco de dados e no código-fonte foram definidos em inglês (User, Category e Transaction), enquanto na documentação textual são apresentados em português (Usuário, Categoria e Movimentação). Essa abordagem foi adotada por ser uma prática comum no mercado de desenvolvimento de software, onde nomes de tabelas, classes e APIs são padronizados em inglês para facilitar manutenção, colaboração internacional e reutilização de código.

3.3.1 Modelo Lógico de Dados do Sistema

O modelo lógico adota o paradigma entidade-relacionamento, definindo as entidades fundamentais e seus relacionamentos. Cada entidade foi normalizada a fim de eliminar redundâncias e garantir coerência nos dados. As entidades identificadas foram:

- Usuário: Armazena dados de identificação e autenticação do sistema.
- Categoria: Classifica as movimentações financeiras como entrada ou saída.
- Movimentação: Registra informações de receitas e despesas, vinculadas a um usuário e uma categoria específica

O modelo contempla os relacionamentos:

- Usuário (1) — (N) Categoria: um usuário pode criar várias categorias próprias;
- Usuário (1) — (N) Movimentação: um usuário pode registrar várias movimentações;
- Categoria (1) — (N) Movimentação: uma categoria pode estar associada a diversas movimentações.

Dessa forma conseguimos garantir que nenhum usuário tenha acesso às informações de outro usuário, atendendo aos requisitos RF13 e RNF05 do sistema.

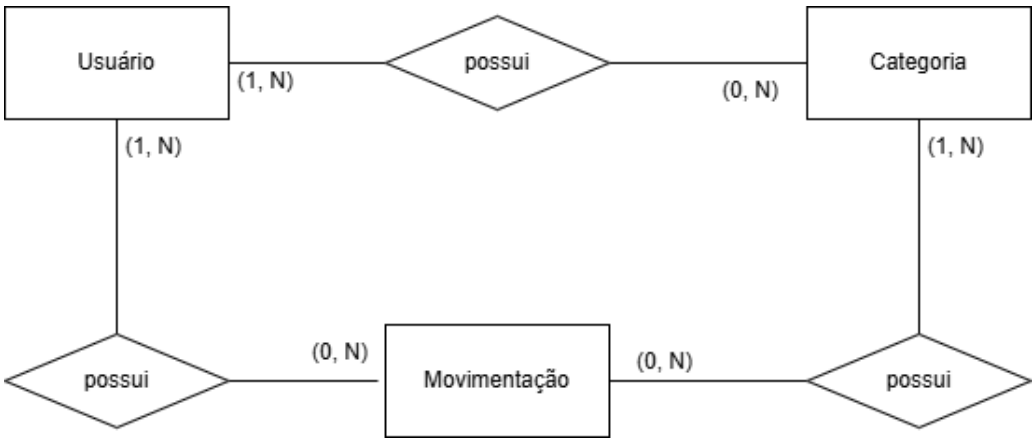

Figura 9 – Diagrama Entidade Relacionamento.

3.3.2 Modelo Físico de Dados do Sistema

O modelo físico traduz o modelo lógico em scripts SQL (Structured Query Language) executáveis no banco de dados SQLite, definindo tabelas, atributos, chaves primárias, chaves estrangeiras e restrições. A seguir são apresentados os scripts de criação das tabelas:

Tabela de Usuários

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    mail TEXT NOT NULL UNIQUE,
    cellphone TEXT,
    password TEXT NOT NULL
);
```

Tabela de Categorias

```
CREATE TABLE categories (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    user_id INTEGER NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

Tabela de Movimentações

```
CREATE TABLE transactions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    date DATE NOT NULL,
```

```
4       type TEXT NOT NULL,
5       value DECIMAL NOT NULL,
6       description TEXT,
7       category_id INTEGER NOT NULL,
8       user_id INTEGER NOT NULL,
9       FOREIGN KEY (category_id) REFERENCES categories(id),
10      FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
11  );
```

O campo value foi definido como DECIMAL em vez de FLOAT ou REAL, pois valores monetários exigem precisão decimal exata. Tipos de ponto flutuante podem gerar erros de arredondamento devido à representação binária dos números, o que pode causar inconsistências em cálculos financeiros. O tipo DECIMAL garante precisão nos valores armazenados, sendo o mais indicado para sistemas financeiros e aplicações que manipulam valores monetários.

3.4 Diagrama de Classe do Sistema

O diagrama de classes do Sistema de Controle Financeiro Pessoal apresenta as classes principais: Usuario, Categoria e Movimentacao. As classes contém atributos e operações essenciais, e os relacionamentos refletem as cardinalidades do modelo lógico:

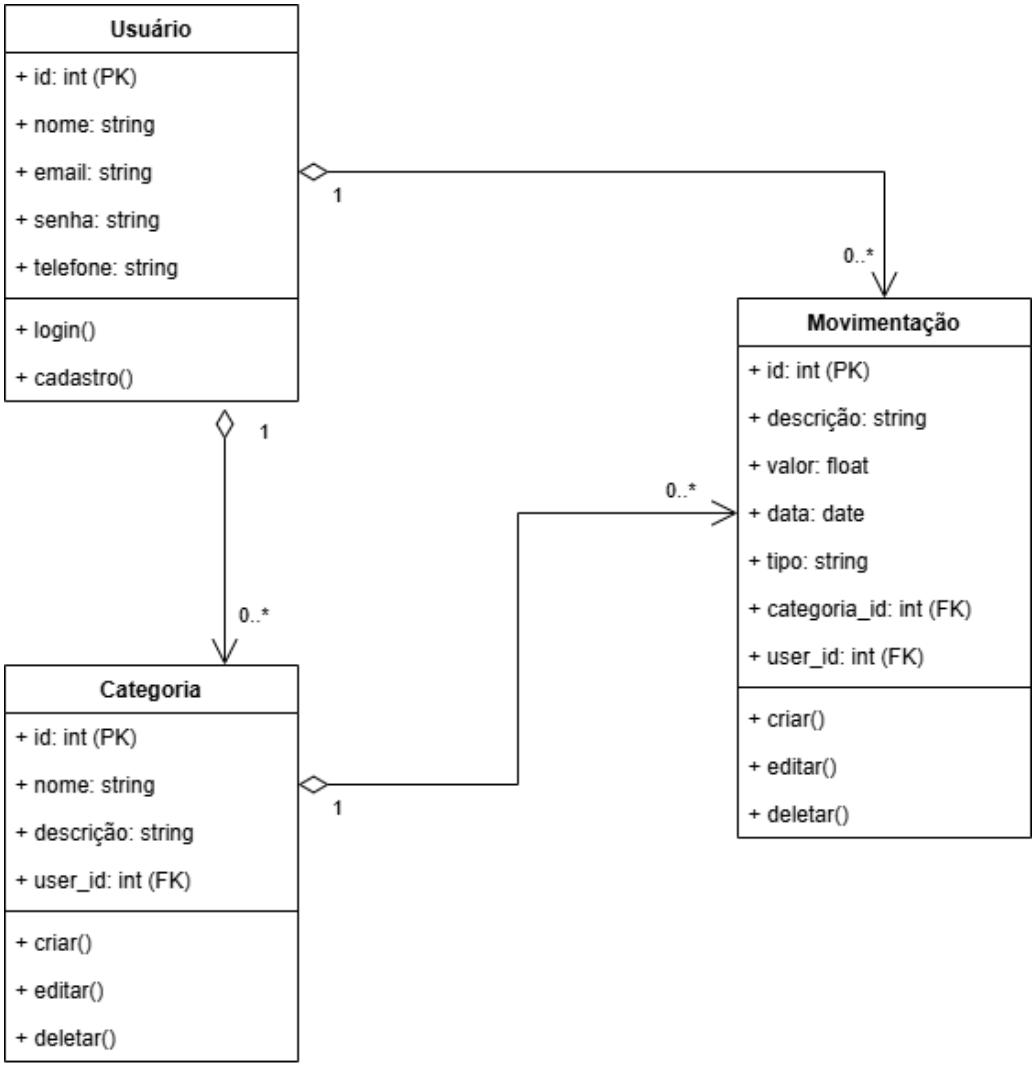

Figura 10 – Diagrama de Classes do SCFP.

4 Desenvolvimento do Sistema

4.1 Processo de Desenvolvimento

O processo de desenvolvimento adotado para o Sistema de Controle Financeiro seguiu uma abordagem incremental e iterativa, com base no modelo ágil SCRUM (2025). Essa metodologia foi escolhida por sua flexibilidade e pela capacidade de adaptação às mudanças de requisitos, características fundamentais para projetos acadêmicos e de curta duração.

O ciclo de desenvolvimento foi dividido em *sprints* que poderiam durar de uma semana à 15 dias, nas quais eram definidas as tarefas prioritárias, revisadas as funcionalidades implementadas e planejadas as próximas iterações. O controle das atividades foi realizado por meio da ferramenta Trello (2025), permitindo o acompanhamento do progresso e a organização do *backlog* do produto.

As etapas principais ficaram da seguinte forma:

- Semana 1: Planejamento do sistema e pré-projeto - levantamento de requisitos, definição de escopo e modelagem inicial.
- Semana 2 e 3: Arquitetura do projeto + *Backend* - estruturação das rotas, criação dos modelos de dados, funcionalidades básicas e testes locais
- Semana 4 e 5: *Frontend* - criação da base do projeto, implementando as telas e iniciando a comunicação com a API
- Semana 6 e 7: Finalização do sistema e testes - integração completa das funcionalidades do *backend* com *frontend*, testes com casos de uso reais e refatorações e correções de acordo com a experiência do usuário
- Semana 8 em diante: Entrega final e documentação

4.2 Arquitetura do Sistema

A arquitetura do sistema de apoio financeiro foi concebida com o objetivo de garantir modularidade, escalabilidade e facilidade de manutenção. A aplicação adota uma arquitetura cliente-servidor, na qual o *frontend* e o *backend* comunicam-se por meio de uma API RESTful, permitindo a separação clara entre as responsabilidades de apresentação, lógica de negócio e persistência de dados.

A arquitetura do sistema de controle financeiro é composto por 3 camadas, sendo elas:

- Camada de Apresentação (*Frontend*) - responsável pela interface com o usuário, desenvolvida em Angular e estilizada com CSS (2025). Essa camada realiza a comunicação com o *backend* através de requisições Hypertext Transfer Protocol (HTTP) utilizando o HttpClient do Angular e operações reativas com Reactive Extensions for JavaScript RxJS (2025). O *frontend* é responsável por apresentar os dados financeiros de forma visual e interativa, incluindo gráficos e tabelas informativas gerados com Chart.js (2025).

- Camada de Negócio (*Backend*) - implementada em Python utilizando o *framework* FastAPI, essa camada concentra as regras de negócio e o gerenciamento das requisições realizadas pelo cliente. O FastAPI organiza o código em rotas (*endpoints*), que são responsáveis por receber solicitações do *frontend*, processá-las e retornar respostas no formato JSON. Essa camada também faz uso do Pydantic para validação e serialização dos dados e do Python-Jose para autenticação via *tokens* JWT, garantindo a segurança e o controle de acesso às funcionalidades do sistema.

  Além disso, o *backend* foi desenvolvido seguindo o conceito de serviços *stateless*, no qual o servidor não mantém estado de sessão entre requisições. Dessa forma, todas as informações necessárias para autenticação e processamento são enviadas a cada requisição por meio de *tokens* JWT. Essa abordagem permite que o sistema seja escalável horizontalmente, possibilitando a execução de múltiplas instâncias do serviço simultaneamente, característica comum em arquiteturas de microsserviços. Com isso, a aplicação pode aumentar sua capacidade de atendimento apenas adicionando novas instâncias do serviço, sem necessidade de alterações estruturais, garantindo elasticidade, escalabilidade e maior disponibilidade do sistema.

- Camada de Dados (*Database*) - a camada de dados é composta pelo SQLite, que funciona como sistema de gerenciamento de banco de dados SQL, acessado por meio do SQLAlchemy (2025), que realiza o mapeamento objeto-relacional (ORM). Essa camada é responsável pelo armazenamento das informações de usuários, registros financeiros e demais entidades do sistema. O uso do ORM simplifica o gerenciamento das transações e consultas, reduzindo a necessidade de manipulação direta de comandos SQL e minimizando erros de persistência.

A comunicação entre a camada de apresentação e a camada de negócio é realizada por meio de requisições HTTP no padrão RESTful, utilizando os métodos GET, POST, PUT e DELETE, conforme a operação a ser executada. Os dados trafegam em formato JSON, o que garante interoperabilidade e facilidade de integração entre diferentes tecnologias.

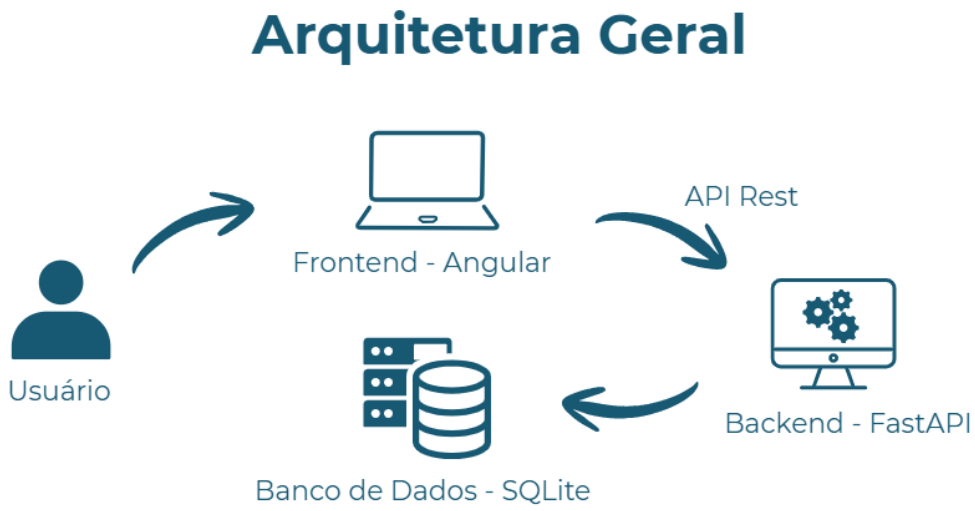

Figura 11 – Arquitetura Geral da aplicação. Fonte: Elaborado pelo autor (2025)

O fluxo básico de comunicação ocorre da seguinte forma:

1. O usuário interage com a interface Angular, acionando uma ação (como adicionar uma entrada financeira).
2. O *frontend* envia uma requisição HTTP ao *backend* FastAPI.
3. O FastAPI processa a requisição, valida os dados com Pydantic e interage com o banco de dados via SQLAlchemy.
4. Após o processamento, o *backend* retorna uma resposta JSON ao *frontend*.
5. O Angular atualiza dinamicamente a interface, refletindo as alterações no sistema.

Esse modelo de comunicação promove a separação de responsabilidades e facilita o desenvolvimento de futuras integrações, como a criação de aplicativos móveis ou *dashboards* administrativos.

Essa abordagem facilita a manutenção e a evolução do sistema, uma vez que as camadas são independentes, possibilitando a substituição ou atualização de uma parte da aplicação sem comprometer as demais. Além disso, o uso de tecnologias modernas em ambas as pontas — Angular no *frontend* e FastAPI no *backend* — proporciona um fluxo de dados eficiente e uma experiência de usuário fluida.

A arquitetura escolhida oferece diversos benefícios ao sistema, pois sua modularidade permite que cada camada seja modificada ou substituída sem impactar o restante

da aplicação, garantindo maior flexibilidade ao projeto. Além disso, sua escalabilidade possibilita a adição de novas funcionalidades sem comprometer o desempenho geral. A organização por componentes no *frontend* e por módulos no *backend* favorece a reutilização de código, contribuindo para um desenvolvimento mais eficiente. O isolamento claro das responsabilidades também facilita a manutenção, tornando mais simples a correção de erros e a implementação de melhorias. Por fim, a utilização de tecnologias amplamente suportadas, como Angular, FastAPI e SQLite, assegura a compatibilidade futura do sistema e fortalece sua longevidade, permitindo que a aplicação evolua conforme novas necessidades surgirem.

4.3 Padrões de Projetos Utilizados

O principal padrão utilizado no projeto foi o MVC (2025) (Model-View-Controller), visando a organização, reutilização de código e manutenções futuras no sistema. O MVC é um padrão de arquitetura de software que divide uma aplicação em três camadas interconectadas chamadas Model (modelo), View (visão) e Controller (controlador), com o objetivo de separar a lógica de negócios, a interface do usuário e o controle das interações, tornando o código mais organizado, manutenível e reutilizável.

A camada de modelo contém a lógica de negócios e os dados da aplicação, como a manipulação do banco de dados. Ela não se comunica diretamente com a View, mas envia os dados para a camada de controle. A camada de view é a parte da interface com o usuário (UI). Ela exibe os dados do modelo, mas não contém lógica de negócios. A View se atualiza conforme os dados do modelo mudam. Já a camada de controle atua como um intermediário. Ela recebe a entrada do usuário através da View, processa essa requisição e a encaminha para o Model para realizar as ações necessárias (como consultar o banco de dados). Em seguida, o Controller pega os dados do modelo e os envia de volta para a View ser exibida.

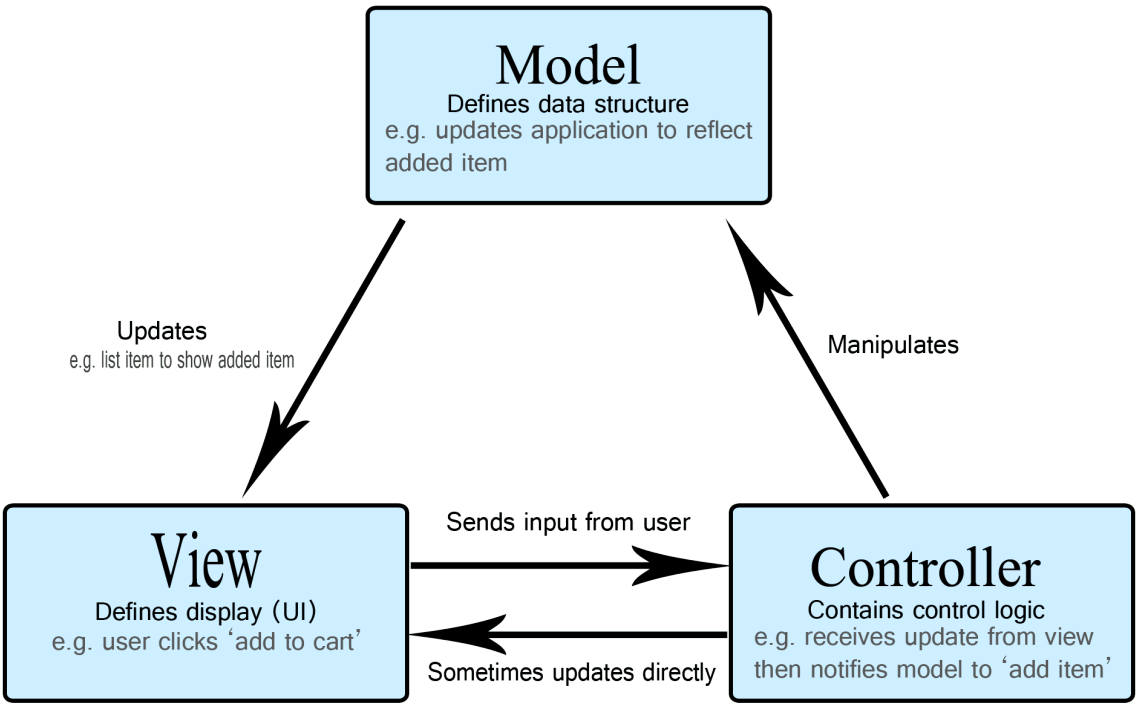

Figura 12 – Exemplo de um MVC. Fonte: MDN (2025)

Dessa forma o código do sistema tem uma melhor manutenibilidade, pois a separação das responsabilidades torna mais fácil encontrar e corrigir problemas. O código fica mais reutilizável, podendo usar um model para diversas views por exemplo, além disso cada camada pode ser testada de forma isolada.

Ainda foram utilizados mais dois padrões, um criacional e outro comportamental, sendo eles respectivamente o Singleton (2025) e o Observer (2025). O Singleton garante que classes tenham apenas uma instância, enquanto provê um ponto de acesso global para essa instância, e foi aplicado para a configuração única da conexão com o banco de dados. Enquanto o Observer permite a definição de um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando e foi aplicado para notificar alterações nos dados de transações, atualizando a interface em tempo real.

4.4 Tecnologias Utilizadas no Desenvolvimento

4.4.1 Linguagens de Desenvolvimento

O desenvolvimento do SCFP foi realizado usando duas linguagens de desenvolvimento diferentes, uma delas utilizada no desenvolvimento do *backend* da aplicação e outra no desenvolvimento do *frontend*, sendo elas, respectivamente, o Python (2025) e o Typescript (2025).

O Python é uma linguagem de programação de alto nível, interpretada de *script*, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte, e foi utilizado no para criação das rotas, controle de requisição e todo o tratamento da lógica do sistema e a interação com o banco de dados. O Typescript também é uma linguagem de programação de alto nível que adiciona tipagem estática com anotações de tipo opcionais ao Javascript (2025) e foi a linguagem principal utilizada no desenvolvimento do *frontend* do projeto.

Também foram utilizadas as linguagens HyperText Markup Language (HTML) e Cascading Style Sheets (CSS), sendo a primeira uma linguagem de marcação utilizada para exibir documentos em paginas web e a segunda sendo uma linguagem de estilo utilizada para especificar a apresentação e o estilo de um documento escrito em uma linguagem de marcação.

4.4.2 Frameworks

O *framework* escolhido para o desenvolvimento do *frontend* do sistema foi o Angular, mais especificamente na versão 19. A opção por essa tecnologia se deu pela sua robustez, escalabilidade e ampla adoção no mercado. Desenvolvido e mantido pela Google, o Angular oferece uma arquitetura baseada em componentes e o uso do TypeScript como linguagem principal, o que contribui para um código mais organizado, seguro e de fácil manutenção.

A integração nativa entre TypeScript e Angular permitiu o uso de interfaces, classes e tipos personalizados, tornando o código mais legível e de fácil manutenção. Além disso, a verificação de tipos em tempo de compilação auxiliou na detecção precoce de inconsistências, contribuindo para uma maior qualidade do software desenvolvido.

Além disso, o Angular fornece ferramentas nativas para a criação de formulários reativos, tratamento de rotas, injeção de dependência e comunicação com serviços HTTP, o que simplifica o desenvolvimento de aplicações complexas de maneira estruturada.

Para a estilização da interface, foi utilizado o Tailwind CSS, um *framework* utilitário que permite criar layouts personalizados sem a necessidade de escrever folhas de estilo extensas. Sua abordagem baseada em classes utilitárias proporcionou agilidade no desenvolvimento da interface e garantiu uma aparência moderna e responsiva ao sistema.

O Tailwind CSS também possibilita uma alta flexibilidade na personalização do design, permitindo definir cores, espaçamentos, tipografias e temas com facilidade. Essa característica foi essencial para manter a coerência visual entre os diferentes componentes da aplicação, além de facilitar futuras manutenções e ajustes de estilo.

O *backend* do sistema foi desenvolvido utilizando o *framework* FastAPI para a construção da API responsável pela comunicação entre o *frontend* e o banco de dados. O FastAPI é um *framework* moderno e de alto desempenho, voltado para a criação de APIs RESTful, com base em tipagem estática e validação automática de dados por meio do Pydantic.

A escolha do FastAPI foi motivada por diversos fatores, entre eles a clareza e expressividade da linguagem, o suporte a programação assíncrona, e a extensa gama de bibliotecas disponíveis. O FastAPI também facilita a documentação automática da API, gerando *endpoints* interativos com base no padrão OpenAPI (Swagger), o que contribui para a transparência e facilidade de integração com outras aplicações.

A arquitetura do *backend* foi estruturada em camadas, com separação entre regras de negócio, persistência e rotas, permitindo uma organização clara e escalável. Essa abordagem favoreceu a manutenção e futura expansão do sistema, além de possibilitar uma comunicação eficiente com o *frontend* Angular por meio de requisições HTTP padronizadas.

4.4.3 Banco de Dados

O SQLite foi escolhido como sistema gerenciador de banco de dados devido à sua simplicidade, leveza e integração direta com o ambiente Python. Por ser baseado em um único arquivo local, o SQLite não requer configuração de servidor, o que reduz a complexidade de implantação e torna-o ideal para aplicações pessoais ou de pequeno porte, como o sistema de apoio financeiro desenvolvido neste trabalho.

Além disso, o SQLite é totalmente compatível com o SQLAlchemy, biblioteca utilizada para o mapeamento objeto-relacional (ORM), o que facilita a manipulação e consulta dos dados de forma estruturada e segura.

4.4.4 Bibliotecas

Além das tecnologias principais, o sistema fez uso de diversas bibliotecas que desempenharam papéis fundamentais no seu funcionamento, tanto no *frontend* quanto no *backend*. A seguir, são descritas as principais:

- RxJS: Biblioteca reativa utilizada no Angular para o tratamento de fluxos assíncronos de dados. Por meio do paradigma de programação reativa, o RxJS permitiu

lidar com eventos e requisições de forma eficiente, facilitando o controle de estados e a manipulação de dados em tempo real.

- Chart.js: Empregada para a geração de gráficos no *frontend*, a biblioteca Chart.js foi utilizada para representar visualmente as informações financeiras do usuário, como entradas, saídas e balanço mensal. Sua integração com o Angular proporcionou uma visualização dinâmica e intuitiva dos dados.

- Pydantic: No *backend*, o Pydantic foi utilizado para validação e serialização de dados. Integrado ao FastAPI, ele garante que os dados enviados e recebidos pela API estejam no formato correto, evitando erros e assegurando a consistência das informações trocadas entre cliente e servidor.

- SQLAlchemy: Responsável pela camada de persistência, o SQLAlchemy é um ORM que facilita o mapeamento entre classes Python e tabelas do banco de dados. Seu uso permitiu maior abstração das operações SQL e reduziu a probabilidade de erros na manipulação direta de consultas.

- Python-Jose: Para a autenticação e segurança, foi empregada a biblioteca Python-Jose, que fornece suporte para a geração e validação de *tokens* JWT (JSON Web Tokens). Essa implementação garantiu o controle de acesso às rotas da API e protegeu os dados dos usuários contra acessos não autorizados.

O uso combinado dessas bibliotecas proporcionou ao sistema um equilíbrio entre desempenho, segurança e usabilidade, tornando o desenvolvimento mais ágil e o produto final mais robusto.

4.4.5 Ferramentas Auxiliares

Durante o desenvolvimento do sistema, foram utilizadas as ferramentas Git e GitLab para o controle de versão do código-fonte. O Git permitiu o gerenciamento de diferentes versões do projeto, possibilitando a colaboração e a reversão de alterações quando necessário. Já o GitLab foi empregado como repositório remoto, garantindo o armazenamento seguro do código e a integração com outras ferramentas de desenvolvimento.

O uso dessas ferramentas foi essencial para manter a rastreabilidade das modificações, promover boas práticas de versionamento e facilitar a continuidade do desenvolvimento de forma organizada e colaborativa.

O Docker foi outra ferramenta utilizada no projeto para conteinerização da aplicação, garantindo um ambiente de execução padronizado e independente do sistema operacional utilizado pelo desenvolvedor. Por meio da criação de contêineres, foi possível

isolar o *backend*, o *frontend* e o banco de dados, facilitando o processo de configuração e implantação do sistema.

Essa abordagem também simplificou o compartilhamento do projeto entre diferentes ambientes, evitando problemas de compatibilidade e tornando o processo de *deploy* mais ágil e confiável.

Como principal ambiente de desenvolvimento, foi escolhido o Visual Studio Code. Essa escolha se deu por sua leveza, interface intuitiva e ampla gama de extensões que facilitam o trabalho com linguagens como TypeScript e Python.

4.5 Instalação e Configurações

Este capítulo descreve os procedimentos necessários para a instalação, configuração e execução do sistema de controle financeiro pessoal. São apresentados os pré-requisitos de software, os passos para instalação manual do *frontend* e *backend*, uma alternativa de instalação utilizando Docker, bem como os comandos necessários para a execução do sistema em ambiente de desenvolvimento. Essas informações permitem a reprodução do ambiente e facilitam a implantação da aplicação.

4.5.1 Pré-requisitos do Sistema

Para a correta instalação e execução do sistema, é necessário que o ambiente atenda aos seguintes pré-requisitos mínimos:

- Hardware: Independente; requer apenas um sistema operacional capaz de executar um navegador web atualizado.
- Sistema operacional Windows, Linux ou macOS;
- Python versão 3.10 ou superior;
- Node.js versão 18 ou superior;
- Gerenciador de pacotes npm (incluso no Node.js);
- Docker e Docker Compose (opcional, para instalação via contêiner);
- Banco de dados relacional compatível com SQLAlchemy;
- Git para clonagem do repositório;
- Navegador web atualizado.

4.5.2 Instalação Manual do Sistema

A instalação manual do sistema é indicada para ambientes de desenvolvimento e testes, permitindo maior controle sobre as dependências utilizadas

4.5.2.1 Instalação e Execução do Backend

Inicialmente, deve-se clonar o repositório do projeto via Git usando **git clone <url-do-repositorio>**, e acessar o diretório do *backend*. Em seguida, recomenda-se a criação de um ambiente virtual Python para isolamento das dependências.

Comandos para criação do ambiente Pyhton e instalação das dependências

```
# 1. Cria o ambiente virtual
python -m venv venv

# 2. Ative conforme seu sistema operacional:
source venv/bin/activate # Linux/macOS
# venv\Scripts\activate   # Windows (use apenas um)

# 3. Instala as dependencias
pip install -r requirements.txt
```

Após a instalação das dependências, as variáveis de ambiente devem ser configuradas e as migrações do banco de dados executadas. Por fim, o servidor da API é iniciado.

Comandos para migração do banco de dados e iniciar backend aplicação

```
alembic upgrade head
python -m uvicorn src.server:app --host 0.0.0.0 --port 8000
   --reload
```

Com isso, a API *backend* passa a estar disponível para consumo pelo *frontend*.

4.5.2.2 Instalação e Execução do Frontend

Para o *frontend*, após clonar o repositório do projeto usando **git clone <url-do-repositorio>** e acessar o diretório correspondente, as dependências devem ser instaladas utilizando o npm, conforme definido no arquivo package.json.

Comandos instalar dependências e iniciar frontend aplicação

```
npm install
npm start
```

Após a execução desses comandos, a aplicação *frontend* pode ser acessada por meio do navegador web em http://localhost:4200, enquanto toda documentação do *backend* estará disponível em http://localhost:8000/docs.

4.5.3 Instalação via Docker

Como alternativa à instalação manual, o sistema pode ser executado utilizando contêineres Docker. Essa abordagem padroniza o ambiente de execução, reduz problemas de compatibilidade e facilita a implantação em diferentes máquinas. A instalação via Docker requer apenas que o Docker e o Docker Compose estejam previamente instalados no sistema operacional, que pode ser verificado usando os comandos docker –version e docker-compose –version. Os principais comandos utilizados nesse processo são:

Comandos para construir e subir os contêineres da aplicação

```
1    git clone <url-do-repositorio>
2    docker compose build
3    docker compose up
```

Esses comandos são responsáveis por construir as imagens dos serviços, configurar os contêineres do *backend*, *frontend* e banco de dados, e iniciar a aplicação de forma integrada. Após seguir esses passos a aplicação disponibilizara o *frontend* em http://localhost:8080 e toda a documentação das APIs do *backend* em http://localhost:8000/docs.

4.5.4 Variáveis de Ambiente e Arquivos de Configuração

Para garantir a segurança e a flexibilidade da aplicação, o sistema utiliza variáveis de ambiente para o gerenciamento de informações sensíveis no *backend*. A aplicação já vem configurada com valores padrão, que atendem ao funcionamento adequado do sistema sem necessidade de alterações manuais. As principais variáveis de ambiente são:

- DATABASE_URL – string de conexão com o banco de dados. Por padrão, utiliza sqlite:///./finn_control_app.db;
- SECRET_KEY – chave de segurança utilizada internamente pela aplicação. Essa chave não deve ser informada, alterada ou exposta pelo usuário, pois é essencial para a integridade e segurança do sistema;
- EXPIRES_IN_MIN – tempo de expiração do *token* de autenticação, definido por padrão como 30 minutos.

Essas variáveis são definidas em arquivos específicos, como .env, que não devem ser versionados em repositórios públicos. Além disso o *backend* utiliza SQLite como banco de

dados por padrão. O arquivo do banco será criado automaticamente na pasta controle-financeiro-backend/finn_control_app.db.

O usuário pode alterar apenas a variável EXPIRES_IN_MIN, caso deseje aumentar ou reduzir o tempo de duração da sessão do usuário. As demais variáveis (DATABASE_URL e SECRET_KEY) não devem ser modificadas, pois alterações indevidas podem comprometer a segurança ou o funcionamento da aplicação. Essa regra é válida tanto para instalações manuais quanto para ambientes que utilizam Docker, não havendo diferenças no comportamento do sistema entre esses cenários.

No *frontend*, durante o ambiente de desenvolvimento, a comunicação com o *backend* é realizada por meio de um proxy configurado no arquivo proxy.conf.json, apontando para http://localhost:8000. Em ambiente de produção, utilizando Docker, a API é acessada por meio do *endpoint* /api, definido no arquivo src/environments/environment.ts.

O arquivo docker-compose.yml já está previamente configurado, expondo o *backend* internamente na porta 8000 e o *frontend* na porta 8080 para acesso externo. O banco de dados SQLite é persistido no *host*, garantindo a preservação dos dados mesmo após a reinicialização dos contêineres.

4.6 Utilização do Sistema

O sistema foi desenvolvido de forma a ser intuitivo e acessível a qualquer tipo de usuário. O objetivo deste capítulo é demonstrar como o sistema pode ser utilizado no dia a dia para o registro, acompanhamento e análise das finanças pessoais.

4.6.1 Acesso ao Sistema

O acesso ao sistema é realizado por meio de uma interface web, utilizando um navegador compatível. Inicialmente, o usuário deve realizar seu cadastro informando dados básicos, como nome, e-mail e senha. Após o cadastro, o acesso ao sistema é realizado por meio de autenticação, com o preenchimento dos campos na tela de *login*, conforme indicado na Figura 12, garantindo que apenas usuários autorizados tenham acesso às suas informações financeiras.

Figura 13 – Tela de Login. Fonte: Elaborado pelo autor (2025)

4.6.2 Página Inicial - Dashboard

Após o *login*, o usuário é direcionado para o painel principal (*dashboard*), conforme ilustrado na Figura 13, no qual são apresentadas informações consolidadas sobre sua situação financeira atual, tais como:

- Saldo total;
- Total de entradas registradas;
- Total de saídas registradas;
- Gráficos demonstrativos de receitas e despesas.

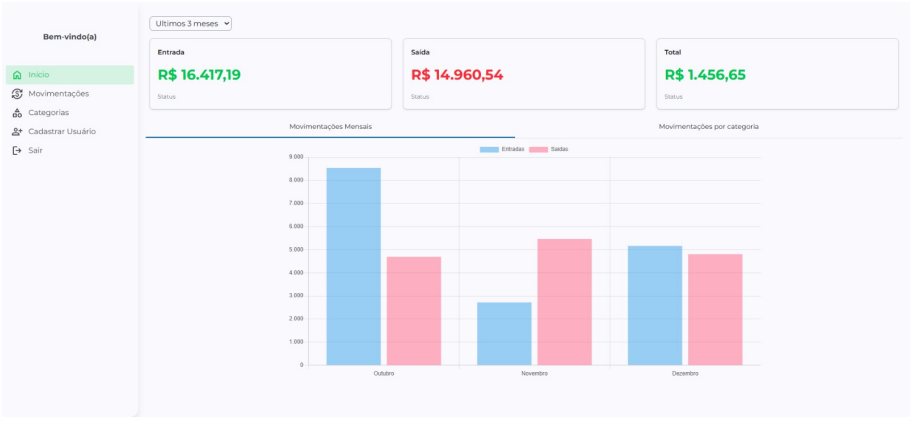

Figura 14 – Tela Inicial do SCFP. Fonte: Elaborado pelo autor (2025)

Essas informações permitem uma visão rápida e clara do estado financeiro do usuário e auxiliam na análise de seu comportamento financeiro. Por meio dessas visualizações, é possível identificar padrões de consumo, as principais categorias de gastos e a evolução do saldo ao longo do tempo, conforme apresentado na Figura 14.

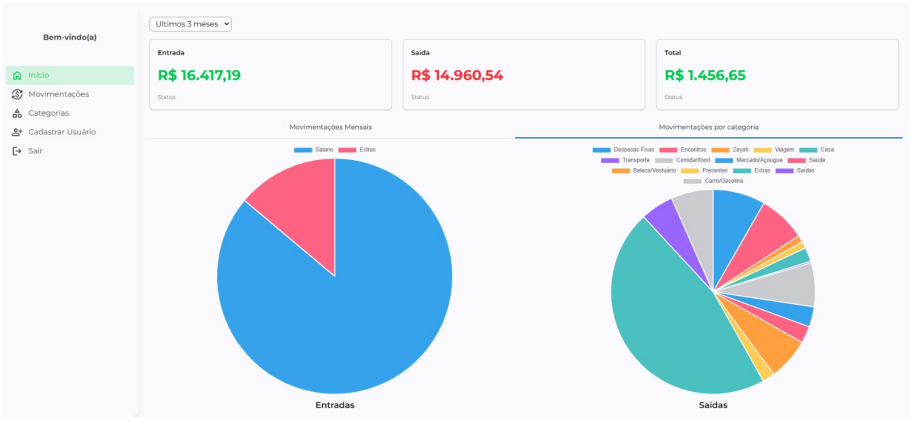

Figura 15 – Tela Inicial com gráficos de gastos por categoria. Fonte: Elaborado pelo autor (2025)

4.6.3 Cadastro e Gerenciamento de Categorias

O sistema permite o cadastro de categorias financeiras, conforme ilustrado na Figura 15, possibilitando a organização das movimentações. O usuário pode criar, editar e remover categorias, classificando-as de acordo com suas necessidades, como alimentação, moradia, transporte, lazer, entre outras.

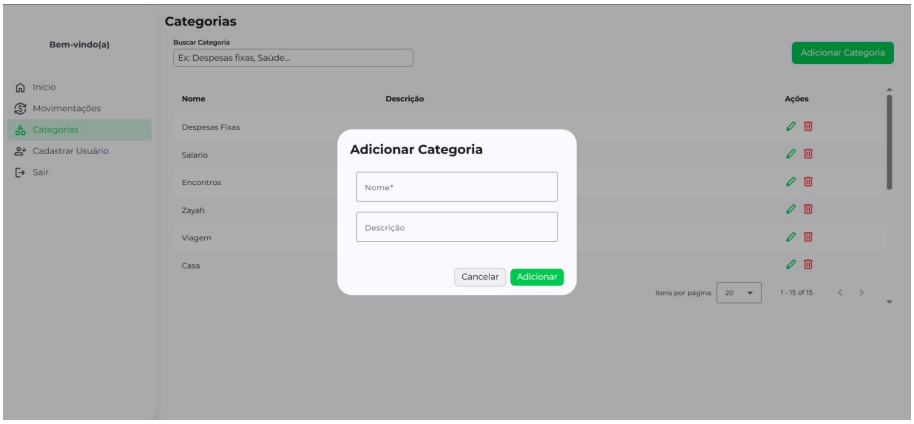

Figura 16 – Tela Categoria - Exemplo de cadastro categoria. Fonte: Elaborado pelo autor (2025)

4.6.4 Cadastro e Gerenciamento de Movimentações Financeiras

Uma das principais funcionalidades do sistema é o registro de movimentações financeiras, tanto de entradas quanto de saídas, conforme pode ser observado na Figura 16. Para cada movimentação, o usuário informa dados como valor, data, categoria e descrição. Essas informações são armazenadas e utilizadas para o cálculo automático do saldo e para a geração de relatórios.

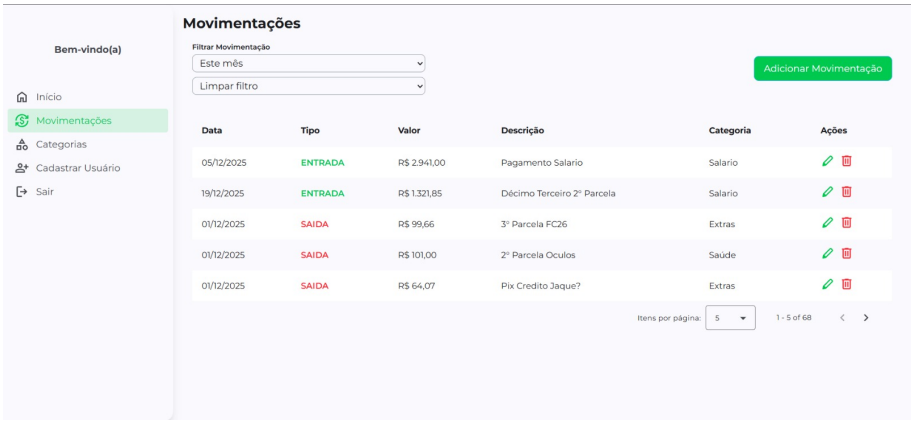

Figura 17 – Tela Movimentações. Fonte: Elaborado pelo autor (2025)

4.7 Testes do Sistema

Este capítulo apresenta de forma detalhada os testes realizados no sistema de controle financeiro pessoal, com o objetivo de validar suas funcionalidades, garantir a confiabilidade das operações e verificar a integração entre os módulos *frontend* e *backend*. A realização dos testes foi fundamental para assegurar que o sistema atende aos requisitos funcionais e não funcionais definidos para o projeto.

4.7.1 Estratégia de Testes

A estratégia de testes adotada foi baseada em uma abordagem incremental, acompanhando o avanço do desenvolvimento do sistema. Inicialmente, foram realizados testes unitários e manuais em funcionalidades isoladas e, posteriormente, testes de integração envolvendo múltiplos módulos.

Os testes priorizaram os fluxos principais do sistema, especialmente aqueles relacionados ao cadastro de usuários, autenticação, registro de movimentações financeiras e geração de informações consolidadas no *dashboard*. Essa abordagem permitiu a identificação precoce de falhas e contribuiu para a estabilidade da aplicação.

4.7.2 Testes no Backend

Os testes no *backend* tiveram como foco a validação das regras de negócio e das rotas disponibilizadas pela API REST. Foram verificados tanto cenários de uso esperado quanto situações de erro, assegurando que o sistema respondesse de forma adequada. Entre os principais testes realizados no *backend*, destacam-se:

- Cadastro de usuários: validação da criação de novos usuários, verificação de dados obrigatórios e prevenção de registros duplicados;

- Autenticação: testes de *login* com credenciais válidas e inválidas, garantindo o correto funcionamento dos mecanismos de segurança;
- Gerenciamento de categorias: testes de criação, edição, listagem e exclusão de categorias financeiras;
- Registro de movimentações: validação do cadastro de entradas e saídas, incluindo valores, datas, categorias associadas e descrições;
- Cálculo de saldo: verificação da atualização correta do saldo do usuário após cada movimentação registrada.

Esses testes asseguraram que a API mantivesse a integridade dos dados e respeitasse as regras definidas para o sistema.

4.7.3 Testes no Frontend

Os testes no *frontend* concentraram-se na verificação da interface do usuário e da experiência de navegação. Foram realizados testes manuais simulando o uso real do sistema, avaliando a interação do usuário com os componentes da aplicação. Os principais aspectos avaliados nos testes de *frontend* incluem:

- Funcionamento correto dos formulários de cadastro e *login*;
- Navegação entre as diferentes telas do sistema;
- Validação visual e funcional dos componentes da interface;
- Exibição correta dos dados recebidos da API;
- Atualização dinâmica das informações no *dashboard*;
- Comportamento responsivo da aplicação em diferentes tamanhos de tela.

Esses testes contribuíram para garantir uma interface intuitiva, consistente e adequada ao uso cotidiano do sistema.

4.7.4 Testes de Integração

Os testes de integração tiveram como objetivo validar a comunicação entre *frontend* e *backend*, assegurando que as requisições e respostas ocorressem de forma correta e consistente. Foram testados os fluxos completos do sistema, desde a ação do usuário na interface até o processamento e persistência dos dados no *backend*. Entre os fluxos avaliados nos testes de integração, destacam-se:

- Autenticação do usuário e manutenção da sessão;
- Envio e recebimento de dados financeiros;
- Sincronização das informações exibidas no *frontend* com os dados persistidos no banco de dados;
- Tratamento de erros e mensagens de retorno da API.

4.7.5 Considerações sobre os Testes Realizados

De modo geral, os testes realizados demonstraram que o sistema apresenta comportamento estável e atende aos requisitos definidos. A abordagem adotada permitiu identificar e corrigir inconsistências ao longo do desenvolvimento, resultando em uma aplicação funcional, confiável e pronta para uso em ambiente real.

4.8 Repositório de Código

O repositório do projeto foi hospedado no GitLab, garantindo versionamento, controle de histórico e rastreabilidade das alterações realizadas durante o desenvolvimento. A estrutura do repositório foi organizada da seguinte forma:

- *frontend/* – contém o código-fonte Angular; disponível em https://gitlab.com/controle-financeiro-app/controle-financeiro-frontend;
- *backend/* – contém o código-fonte Python + FastAPI; disponível em https://gitlab.com/controle-financeiro-app/controle-financeiro-backend;

O uso do GitLab possibilitou o acompanhamento contínuo das versões, a criação de branches para novas funcionalidades e a integração facilitada entre *backend* e *frontend*.

4.9 Resultados Alcançados com o Sistema

Este capítulo apresenta os principais resultados alcançados com o desenvolvimento e utilização do sistema de controle financeiro pessoal, avaliando seu desempenho em relação aos objetivos propostos neste Trabalho de Conclusão de Curso. O sistema desenvolvido atendeu aos objetivos definidos inicialmente, oferecendo uma solução funcional para o registro e acompanhamento de finanças pessoais. As funcionalidades implementadas permitem ao usuário controlar suas receitas e despesas de forma organizada e intuitiva. Entre os principais benefícios proporcionados pelo sistema, destacam-se:

- Centralização das informações financeiras;

- Facilidade no registro e acompanhamento de gastos;
- Visualização clara do saldo e das movimentações;
- Apoio à tomada de decisões financeiras mais conscientes.

A solução apresentou bom desempenho e estabilidade durante os testes realizados. A arquitetura adotada mostrou-se adequada, permitindo uma separação clara de responsabilidades entre *frontend* e *backend*, além de possibilitar futuras evoluções do sistema.

Apesar dos resultados positivos, algumas limitações foram identificadas, como a ausência de funcionalidades avançadas de planejamento financeiro e a dependência de acesso à internet para utilização do sistema. Essas limitações não comprometem o funcionamento básico, mas apontam oportunidades de melhoria.

De modo geral, os resultados alcançados demonstram que o sistema cumpre sua proposta de auxiliar no controle financeiro pessoal, validando as escolhas tecnológicas e metodológicas adotadas ao longo do desenvolvimento.

5 Conclusão

O desenvolvimento do sistema de controle financeiro pessoal permitiu demonstrar na prática a aplicação de conceitos fundamentais de engenharia de software, arquitetura de sistemas, programação web e boas práticas de desenvolvimento.

Durante o processo, foram aplicadas técnicas de levantamento de requisitos, modelagem de dados, padronização de código e integração entre diferentes tecnologias. O uso do Angular no *frontend* e do FastAPI no *backend* proporcionou uma estrutura modular, escalável e de fácil manutenção, enquanto o banco SQLite garantiu simplicidade e eficiência para o armazenamento local dos dados.

O sistema alcançou o objetivo principal de oferecer uma ferramenta intuitiva e acessível, voltada para o público que busca uma solução prática para gerenciar suas finanças pessoais. A interface responsiva e o foco na usabilidade permitiram criar uma experiência agradável e direta, mesmo para usuários com pouco conhecimento técnico.

5.1 Trabalhos Futuros

Como evolução do sistema, propõe-se a implementação de autenticação por meio de redes sociais, como Google e GitHub. Essa funcionalidade tem como objetivo simplificar o processo de acesso ao sistema, reduzindo a necessidade de criação manual de contas e melhorando a experiência do usuário. Além disso, a autenticação social pode aumentar a segurança e a confiabilidade do processo de *login*, utilizando provedores amplamente reconhecidos.

Outra melhoria relevante consiste na possibilidade de exportação dos relatórios financeiros gerados pelo sistema em formatos amplamente utilizados, como PDF e Excel. Essa funcionalidade permitiria ao usuário armazenar, compartilhar ou analisar seus dados financeiros fora da aplicação, facilitando o controle e a organização das informações, além de possibilitar o uso dos dados em outras ferramentas de análise.

A inclusão de notificações automáticas representa um avanço significativo para o sistema. Essa funcionalidade poderia alertar o usuário sobre despesas recorrentes, vencimentos de contas e lembretes de pagamento, contribuindo para um melhor planejamento financeiro e evitando atrasos ou esquecimentos. As notificações poderiam ser implementadas via e-mail, notificações no navegador ou dispositivos móveis.

Outra ideia de funcionalidade interessante seria a implementação de metas financeiras mensais, permitindo que o usuário defina objetivos de economia ou limites de gastos por categoria. O sistema poderia acompanhar o progresso dessas metas ao longo do pe-

ríodo, apresentando indicadores visuais e alertas, auxiliando o usuário a manter o controle e a disciplina financeira.

Visando maior escalabilidade e desempenho, sugere-se a migração do banco de dados para uma solução mais robusta, como o PostgreSQL. Essa mudança permitiria suportar múltiplos usuários simultâneos de forma mais eficiente, além de oferecer recursos avançados de segurança, integridade e desempenho, tornando o sistema mais adequado para ambientes de produção e possíveis expansões futuras.

A criação de uma versão mobile nativa integrada à aplicação web é outra evolução relevante. Uma aplicação móvel permitiria maior acessibilidade e praticidade para o usuário, possibilitando o registro de movimentações financeiras em tempo real. Essa versão poderia compartilhar a mesma API *backend*, garantindo consistência entre as plataformas.

Por fim, uma evolução de grande impacto seria a integração do sistema com APIs bancárias. Essa funcionalidade permitiria a importação automática de movimentações financeiras diretamente das instituições bancárias, reduzindo o esforço manual do usuário e aumentando a precisão dos dados. Essa integração tornaria o sistema mais completo e competitivo em relação a soluções comerciais existentes.

As melhorias propostas demonstram que o sistema desenvolvido possui grande potencial de evolução. A implementação desses trabalhos futuros poderia transformar a aplicação em uma solução mais robusta, escalável e alinhada às demandas modernas de controle financeiro, reforçando a relevância e a aplicabilidade prática da aplicação.

Referências

ANGULAR. **Angular**. 2025. Disponível em: <https://angular.dev/overview>. Acesso em: 31 dez. 2025. Disponível em: <https://angular.dev/overview>. Citado na página 9.

CHART.JS. **Chart.js**. 2025. Disponível em: <https://www.chartjs.org/docs/latest/>. Acesso em: 31 dez. 2025. Disponível em: <https://www.chartjs.org/docs/latest/>. Citado na página 24.

CNC. **Pesquisa de endividamento e inadimplência do consumidor (PEIC)**. [S.l.]: Confederação Nacional do Comércio de Bens, Serviços e Turismo (CNC), 2024. Acesso em: 02 jul 2025. Citado na página 8.

CSS, T. **Tailwind CSS**. 2025. Disponível em: <https://tailwindcss.com/>. Acesso em: 31 dez. 2025. Disponível em: <https://tailwindcss.com/>. Citado na página 24.

DOCKER. **Docker**. 2025. Disponível em: <https://docs.docker.com/>. Acesso em: 31 dez. 2025. Disponível em: <https://docs.docker.com/>. Citado na página 9.

FASTAPI. **FastAPI**. 2025. Disponível em: <https://fastapi.tiangolo.com/>. Acesso em: 31 dez. 2025. Disponível em: <https://fastapi.tiangolo.com/>. Citado na página 9.

IBGE. **Pesquisa Nacional por Amostra de Domicílios Contínua: Rendimento de todas as fontes 2024**. Rio de Janeiro: Centro de Documentação e Disseminação de Informações. Fundação Intituto Brasileiro de Geografia e Estatística, 2025. Acesso em: 02 jul 2025. Citado na página 8.

JAVASCRIPT. **Javascript**. 2025. Disponível em: <https://www.javascript.com/>. Acesso em: 31 dez. 2025. Disponível em: <https://www.javascript.com/>. Citado na página 28.

MDN. **MDN**. 2025. Disponível em: <https://developer.mozilla.org/en-US/docs/Glossary/MVC/>. Acesso em: 31 dez. 2025. Disponível em: <https://developer.mozilla.org/en-US/docs/Glossary/MVC>. Citado na página 27.

MVC. **MVC**. 2025. Disponível em: <https://en.wikipedia.org/wiki/ModelDisponível em: <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>. Citado na página 26.

OBSERVER. **Observer**. 2025. Disponível em: <https://refactoring.guru/design-patterns/observer/>. Acesso em: 31 dez. 2025. Disponível em: <https://refactoring.guru/design-patterns/observer>. Citado na página 27.

PYTHON. **Python**. 2025. Disponível em: <https://www.python.org/>. Acesso em: 3 nov. 2025. Disponível em: <https://www.python.org/>. Citado na página 28.

RXJS. **RxJS**. 2025. Disponível em: <https://rxjs.dev/>. Acesso em: 31 dez. 2025. Disponível em: <https://rxjs.dev/>. Citado na página 24.

SCRUM. **SCRUM**. 2025. Disponível em: <https://www.scrum.org/resources/what-scrum-module/>. Acesso em: 31 dez. 2025. Disponível em: <https://www.scrum.org/resources/what-scrum-module>. Citado na página 23.

SINGLETON. **Singleton**. 2025. Disponível em: <https://refactoring.guru/design-patterns/singleton/>. Acesso em: 31 dez. 2025. Disponível em: <https://refactoring.guru/design-patterns/singleton>. Citado na página 27.

SPCBRASIL/CNDL. **Inadimplentes e educação financeira 2024**. [S.l.]: SPC Brasil E CNDL, 2024. Acesso em: 02 jul 2025. Citado na página 8.

SQLALCHEMY. **SQLAlchemy**. 2025. Disponível em: <https://docs.sqlalchemy.org/en/20/>. Acesso em: 31 dez. 2025. Disponível em: <https://docs.sqlalchemy.org/en/20/>. Citado na página 24.

SQLITE. **SQLite**. 2025. Disponível em: <https://sqlite.org/>. Acesso em: 31 dez. 2025. Disponível em: <https://sqlite.org/>. Citado na página 9.

TRELLO. **Trello**. 2025. Disponível em: <https://trello.com/>. Acesso em: 31 dez. 2025. Disponível em: <https://trello.com/>. Citado na página 23.

TYPESCRIPT. **Typescript**. 2025. Disponível em: <https://www.typescriptlang.org/>. Acesso em: 3 nov. 2025. Disponível em: <https://www.typescriptlang.org/>. Citado na página 28.