UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

AMANDA SILVA GOVEIA

Plataforma para planejamento financeiro

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientadora: Profa. Dra. Renata Galante

Porto Alegre
2022

AGRADECIMENTOS

Agradeço ao Guillermo Falcão por tirar minhas dúvidas sobre desenvolvimento *React* durante a elaboração deste projeto, ao Leonardo Eich e ao Felipe Comerlato pelo companheirismo nos estudos e ao Vinicius Irale pela sugestão do tema deste trabalho e por ouvir minhas reclamações sobre a graduação durante todos estes anos.

RESUMO

Com a alta da inflação no mundo, é cada vez mais imprescindível que as pessoas organizem suas finanças de maneira eficiente. Pensando nisto, este trabalho tem por objetivo o desenvolvimento de uma aplicação *WEB* que permita gerenciar a vida financeira de uma pessoa através do controle de gastos, despesas, cartões de crédito e definição de orçamentos mensais, os quais indicam o valor máximo que se gostaria de gastar no mês, seja de uma maneira geral ou por categorias específicas. Experimentos com usuários indicam que a plataforma desenvolvida tem boas funcionalidades base e que poderia se beneficiar da adição de novas *features* para uma experiência mais completa.

Palavras-chave: Plataforma web. Gerenciamento. Finanças.

Name: Platform for financial planning

ABSTRACT

With inflation rising around the world, it is essential that people organize their finances efficiently. With this in mind, this work aims to develop a web application to help control finances. The solution created allows the management of incomes, expenses, credit cards and defining monthly budgets, which indicate the maximum amount that one would like to spend in a month, either in a general way or by specifying categories. Trials with users indicate that the platform has good base functionality and could benefit from adding new features for a more complete experience.

Keywords: Web platform. Management. Financial.

LISTA DE FIGURAS

Figura 2.1 Modelo Cliente-Servidor 14
Figura 2.2 Arquitetura Tradicional 16
Figura 2.3 *Onion Architecture* 16

Figura 3.1 Exemplo de ganhos cadastrados no *Mobills* 22
Figura 3.2 Exemplo de orçamento criado no *Mobills* 23
Figura 3.3 Exemplo de orçamento criado no *Fast Budget* 24
Figura 3.4 Exemplo de orçamento criado no *Fortuno* 25

Figura 4.1 *Income Class* 31
Figura 4.2 *IncomeCategory Class* 31
Figura 4.3 Diagrama de classes representando *ganhos* 32
Figura 4.4 *Expense Class* 33
Figura 4.5 *ExpenseCategory Class* 34
Figura 4.6 Diagrama de classes representando *despesas* 34
Figura 4.7 *Budget Class* 35
Figura 4.8 Diagrama de classes representando *orçamentos* 35
Figura 4.9 *CreditCard Class* 36
Figura 4.10 Diagrama de classes representando *cartões de crédito* 37
Figura 4.11 Adaptação da *Onion Architecture* 37
Figura 4.12 Pacotes representando a *Onion Architecture* 38
Figura 4.13 Código de *login* presente na *AuthenticationService* 39
Figura 4.14 Mapeamento da *AuthenticationException* na camada *Rest* 40
Figura 4.15 Diagrama *ER* 43
Figura 4.16 *Endpoints* do servidor 44
Figura 4.17 Exemplo de utilização de *DTO* 45
Figura 4.18 *DTO IncomeRequest* 45
Figura 4.19 Resumo do fluxo da plataforma 46
Figura 4.20 Exemplo de declaração de *useState* 60
Figura 4.21 Exemplo de uso de *useState* 60
Figura 4.22 *Hook useFetchDetails* 61
Figura 4.23 Componente *MonthPicker* 62
Figura 4.24 Uso do *MonthPicker* na tela de despesas 62
Figura 4.25 Tela de registro de usuários 63
Figura 4.26 Tela de *login* 64
Figura 4.27 Tela de ganhos 64
Figura 4.28 Seletor de data 65
Figura 4.29 Tela de ganhos quando não há itens registrados 66
Figura 4.30 Modal para cadastro de ganho 66
Figura 4.31 Modal para cadastro de ganho 67
Figura 4.32 Tela de despesas 68
Figura 4.33 Modal para o cadastro de despesa 69
Figura 4.34 Tela de cartões de crédito 70
Figura 4.35 Modal para o cadastro de cartão de crédito 70
Figura 4.36 Tela com as despesas de um cartão de crédito 71
Figura 4.37 Tela de orçamentos 72
Figura 4.38 Modal para o cadastro de orçamento 73
Figura 4.39 Tela de orçamento para categorias 74

Figura 4.40 Modal para o cadastro de orçamento para categoria 75

Figura 5.1 Gráfico da identidade de gênero dos participantes..... 78
Figura 5.2 Gráfico da faixa etária dos participantes 78
Figura 5.3 Gráfico do grau de escolaridade dos participantes 79
Figura 5.4 Gráfico sobre utilização de plataformas de controle financeiro.. 79
Figura 5.5 Gráfico da avaliação das tarefas solicitadas utilizando a plataforma 80
Figura 5.6 Gráfico da avaliação das tarefas solicitadas utilizando a plataforma 80
Figura 5.7 Sugestão de alteração no cadastro de ganhos 81
Figura 5.8 Relevância das funcionalidades disponíveis..... 82
Figura 5.9 Sugestões dos participantes..... 82

LISTA DE TABELAS

Tabela 3.1 Comparativo de funcionalidades 26

LISTA DE ABREVIATURAS E SIGLAS

JVM	Java Virtual Machine.
POM	Project Object Model.
SQL	*Structured Query Language*
API	*Application Programming Interface*
UI	*User Interface*
REST	*Representational State Transfer*
URI	*Uniform Resource Identifier*
HTTP	*Hypertext Transfer Protocol*
ER	*Entity Relationship*
DTO	*Data Transfer Object*
HTML	*HyperText Markup Language*
IOS	*iPhone Operating System*
PDF	*Portable Document Format*
CSV	*Comma-Separated Values*

SUMÁRIO

1 INTRODUÇÃO 12
2 CONCEITOS E TECNOLOGIAS 13
2.1 Modelo Cliente-Servidor 13
2.2 REST APIs 14
2.3 *Onion Architecture* 15
2.4 *Clean Code* 17
2.5 Tecnologias Utilizadas 18
2.5.1 Kotlin 18
2.5.2 Spring Framework 18
2.5.3 Maven 19
2.5.4 Spark Java 19
2.5.5 MySQL 20
2.5.6 React 20
2.5.7 Material UI 21
3 TRABALHOS RELACIONADOS 22
3.1 Mobills 22
3.2 Fast Budget 23
3.3 Fortuno 24
3.4 Análise Comparativa 25
4 MODELAGEM E IMPLEMENTAÇÃO 27
4.1 Visão Geral do Desenvolvimento 27
4.2 Requisitos 27
4.2.1 Requisitos funcionais 28
4.2.2 Requisitos não funcionais 29
4.3 *Backend* 29
4.3.1 Domínio 29
4.3.2 *Onion Architecture* 37
4.3.3 *Clean Code* 39
4.3.4 Banco de Dados 40
4.3.5 *REST APIs* 44
4.3.5.1 POST /signup 46
4.3.5.2 POST /login 47
4.3.5.3 POST /income 48
4.3.5.4 POST /expense 49
4.3.5.5 POST /budget 49
4.3.5.6 POST /budgetCategory 50
4.3.5.7 POST /creditCard 51
4.3.5.8 DELETE /income?id=<long> 52
4.3.5.9 DELETE /expense?id=<long> 52
4.3.5.10 DELETE /budget?date=<date> 53
4.3.5.11 DELETE /budgetCategory?date=<date>&category=<category> 53
4.3.5.12 DELETE /creditCard?id=<long> 53
4.3.5.13 GET /balance?date=<date> 53
4.3.5.14 GET /income?date=<date> 54
4.3.5.15 GET /incomeDetails?date=<date> 54
4.3.5.16 GET /expense?date=<date> 55
4.3.5.17 GET /expenseDetails?date=<date> 56

4.3.5.18 GET /budget?date=<date> 56
4.3.5.19 GET /budgetDetails?date=<date> 57
4.3.5.20 GET /creditCards 58
4.3.5.21 GET /creditCardDetails?id=<long>&date=<date> 58
4.4 Frontend 60
4.4.1 Registro de usuários 62
4.4.2 Login 63
4.4.3 Ganhos 64
4.4.4 Despesas 67
4.4.5 Cartões de crédito 69
4.4.6 Orçamento total 71
4.4.7 Orçamento de categorias 73
5 AVALIAÇÃO COM USUÁRIOS 76
5.1 Ambiente do experimento 76
5.2 Protocolo de testes 76
5.3 Perfil dos Usuários 77
5.4 Análise dos resultados da avaliação da plataforma 79
5.5 Considerações Gerais e Melhorias Futuras 83
6 CONCLUSÃO 84
REFERÊNCIAS 85
APÊNDICE A — FORMULÁRIO DE TESTE 87

1 INTRODUÇÃO

Com a alta da inflação no mundo (OECD, 2022) é cada vez mais importante que as pessoas gerenciem seus gastos de forma eficaz. Para tornar esta tarefa mais fácil, é possível utilizar-se de ferramentas que auxiliem no controle de finanças. Com isto em mente, este trabalho tem por objetivo a criação de uma plataforma *WEB* totalmente gratuita que auxilie no controle de gastos dos indivíduos.

A solução proposta tem como foco quatro entidades consideradas centrais à aplicações do tipo: ganhos, despesas, orçamentos e cartões de crédito. Sua implementação foi feita seguindo o modelo cliente-servidor, sendo o servidor desenvolvido em *Kotlin* e o cliente em *React*. O armazenamento de dados é realizado em um banco relacional *MySQL*.

Mais detalhes sobre a criação deste sistema serão fornecidos ao longo desta monografia, a qual está dividida em 6 capítulos. No Capítulo 2 pode-se consultar os conceitos e principais tecnologias usados na elaboração deste trabalho. O Capítulo 3 apresenta trabalhos relacionados consultados. No Capítulo 4 são dados detalhes sobre a modelagem e implementação deste projeto. O Capítulo 5 traz resultados de uma pesquisa com usuários cujo objetivo é avaliar a plataforma criada, e, por fim, no Capítulo 6 são apresentadas as conclusões deste projeto e sugestões para trabalhos futuros.

2 CONCEITOS E TECNOLOGIAS

Este capítulo apresenta os conceitos e tecnologias escolhidas para o desenvolvimento do projeto proposto.

A primeira decisão a ser tomada para a implementação desta aplicação foi a escolha da plataforma de desenvolvimento. A plataforma escolhida foi a *WEB*, pois além de ser a plataforma com a qual a autora tem mais familiaridade, também é possível atingir um público mais abrangente ao desenvolver para ela, visto que aplicações *WEB* podem ser acessadas tanto de *desktops*, quanto de dispositivos móveis.

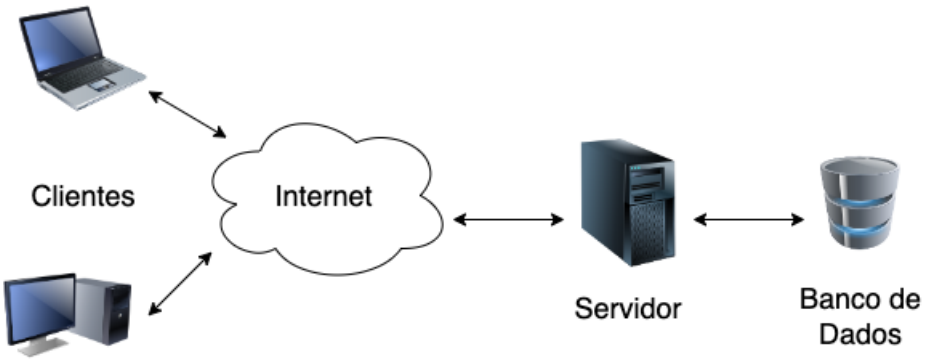
Com a plataforma escolhida, o próximo passo foi a escolha da arquitetura a ser usada. O modelo utilizado é o cliente-servidor (BERSON, 1992), sendo o servidor desenvolvido em Kotlin (KOTLIN, 2022a) e o cliente em React (REACT, 2022b). Para o armazenamento de dados foi usado o banco relacional MySQL (MYSQL, 2022).

Nas próximas seções serão apresentados detalhes das principais tecnologias escolhidas para a implementação da plataforma proposta.

2.1 Modelo Cliente-Servidor

O modelo cliente-servidor é um modelo no qual uma aplicação faz uma requisição para outra e aguarda uma resposta. A aplicação requisitante é chamada de cliente, enquanto quem provê a resposta é chamado de servidor. O cliente normalmente consiste em uma interface gráfica através da qual o usuário final pode requisitar operações. Já o servidor é a parte responsável pela realização de fato destas operações e muitas vezes também se comunica com uma base de dados. A Figura 2.1 demonstra um exemplo deste modelo.

Figura 2.1: Modelo Cliente-Servidor



Fonte: Elaborada pela autora

Conforme mencionado por Oluwatosin (OLUWATOSIN, 2014), o modelo cliente-servidor é utilizado por aplicações *WEB*. Deste modo, pareceu natural que este fosse o modelo escolhido para a implementação deste trabalho, pois, como mencionado na seção anterior, a plataforma escolhida para desenvolvimento foi a *WEB*. Além do mais, a separação em cliente e servidor fez sentido, já que, uma vez que o foco deste trabalho está na programação backend (servidor), existe a possibilidade para que outros frontends (clientes) sejam desenvolvidos futuramente e utilizem o servidor criado neste projeto sem maiores complicações.

2.2 REST APIs

Uma API (*Application Programming Interface*) é um conjunto de definições e protocolos usados para integrar aplicações de software (REDRAT, 2022). É um contrato onde se estabelece o que uma aplicação espera receber e o que deve retornar, permitindo a comunicação entre aplicações distintas.

Uma REST API (FIELDING, 2000) se comunica com requisições *HTTP* para criar (*POST*), buscar (*GET*), atualizar (*PUT*) ou deletar (*DELETE*) recursos. Ela segue os princípios de design *REST*, sendo eles:

- *Desacoplamento entre cliente e servidor*: a única informação que um cliente deve saber em relação ao servidor é a *URI* para acessar determinado recurso. Igualmente o servidor não deve fazer modificações no cliente e sim apenas retornar a informação pedida via

HTTP. Deste modo, temos a garantia de que cliente e servidor são independentes entre si.

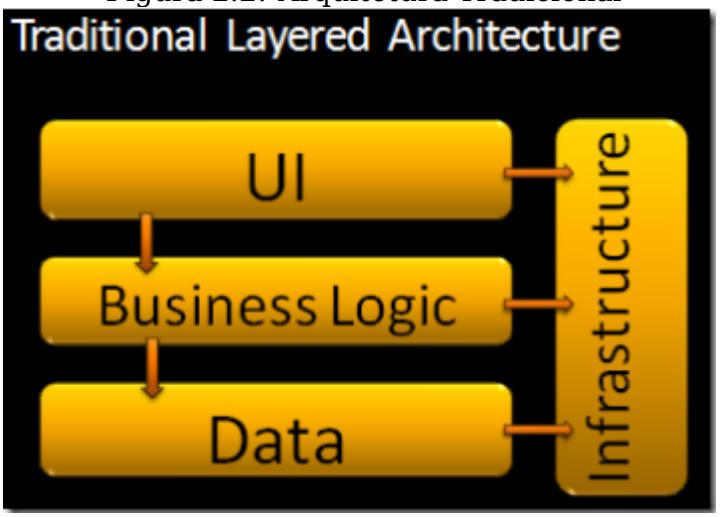
- *Stateless*: *REST APIs* devem ser *stateless*, isto é, cada requisição deve enviar toda a informação necessária para seu processamento, pois não se tem nenhum contexto relativo à elas armazenado no servidor.
- *Cache*: o servidor deve informar se um recurso pode ser armazenável em cache. Quando possível, recursos devem ser armazenados em cache para melhorar a performance do cliente e a escalabilidade do servidor.
- *Interface uniforme*: REST APIs devem possuir uma interface uniforme, sendo totalmente dissociadas dos serviços que elas proveem. O modo como a informação é transmitida não deve estar vinculada a aplicação que a requisita.
- *Sistema em camadas*: não se deve assumir que o cliente se comunica diretamente com o servidor. Podem existir N camadas entres os dois e é necessário que isso seja transparente para ambos.
- *Código executável*: normalmente *REST APIs* enviam recursos estáticos, mas também é possível que enviem código executável, como *applets* Java. Neste caso o código deve ser executado apenas sob demanda.

A comunicação entre cliente e servidor desenvolvidos nesta monografia acontece via *REST APIs*.

2.3 *Onion Architecture*

A *Onion Architecture* foi proposta por Jeffrey Palermo em 2008 (PALERMO, 2008). Seu objetivo principal é fazer a separação de responsabilidades e minimizar o acoplamento. Segundo Palermo, a Figura 2.2 representa uma arquitetura tradicional em camadas:

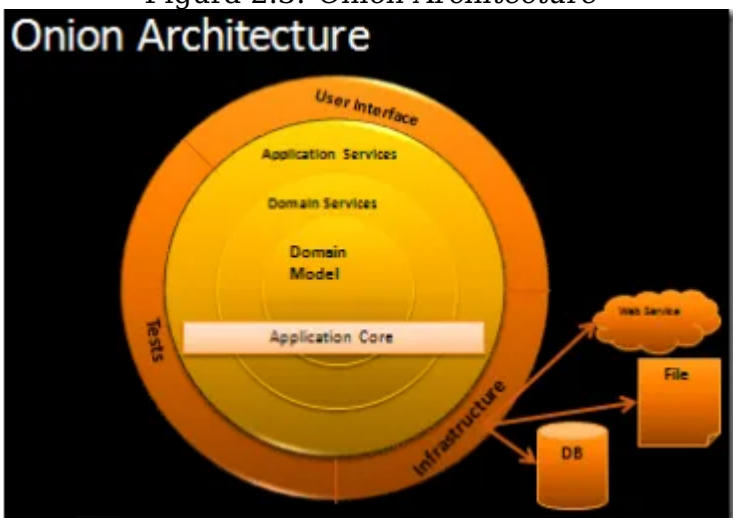
Figura 2.2: Arquitetura Tradicional



Fonte: (PALERMO, 2008)

Neste tipo de arquitetura, há um grande acoplamento. Além de cada uma das camadas normalmente depender da infraestrutura, cada camada também depende das que estão abaixo dela, o que acaba tornando difícil fazer a manutenção do sistema. Se for preciso mudar a forma como os dados são obtidos, por exemplo, seria preciso fazer alterações em todas as camadas. Por este motivo Palermo propôs a *Onion Architecture*, conforme diagrama na Figura 2.3.

Figura 2.3: *Onion Architecture*



Fonte: (PALERMO, 2008)

Nesta arquitetura, camadas mais externas podem depender das mais centrais, porém o contrário não é válido. Por exemplo, a *Domain Services* pode utilizar objetos da *Domain Model*, mas a *Domain Model* deve ser

auto-contida. As camadas mais externas são reservadas para coisas que mudam com frequência. Pensando no exemplo anterior sobre mudar a forma da obtenção de dados, enquanto na arquitetura tradicional seria preciso realizar modificações em todas as camadas, na *Onion Architecture* seria preciso alterar apenas a camada de infraestrutura.

As especificações das camadas propostas por Palermo são as seguintes:

- *Domain Model*: é o cerne da aplicação. Nesta camada se encontram os objetos centrais ao negócio;
- *Domain Services*: aqui são criadas interfaces de repositório, as quais fornecem o padrão do comportamento de acesso aos dados como busca e salvamento de objetos. Ao criar interfaces, temos uma abordagem mais fracamente acoplada, não dependendo de detalhes de implementação do agente externo de dados (bancos de dados, arquivos, etc.);
- *Application Services*: nesta camada é feita a comunicação entre a *Domain Services* e as camadas mais externas;
- *Infrastructure*: é onde acontece a comunicação com o banco de dados, arquivos ou qualquer que seja o serviço externo que esteja sendo usado;
- *User Interface* : é o local onde está o código responsável por como os usuários interagem com a aplicação;
- *Tests*: contém os testes do sistema.

O servidor criado neste projeto usa uma adaptação da *Onion Architecture*, a qual está descrita na Seção 4.3. Como a implementação das interfaces das camadas centrais reside nas bordas da aplicação, é necessário algum mecanismo que faça a injeção de código em tempo de execução. É aí que entra a injeção de dependências que, conforme mencionado na Seção 2.5, é obtido neste trabalho com o uso do *Spring Framework*.

2.4 *Clean Code*

Clean Code (MARTIN, 2008) se refere a um conjunto de boas práticas usadas para criar software fácil de ler, escrever e manter. Entre elas estão o uso de nomes significativos, o uso de métodos pequenos e com uma única responsabilidade e o uso de exceções que retratem o contexto em que elas

aconteceram. Estas práticas foram usadas no desenvolvimento deste projeto e exemplos de sua utilização podem ser conferidas na Seção 4.3.

2.5 Tecnologias Utilizadas

A seguir são descritas as principais tecnologias utilizadas neste trabalho.

2.5.1 Kotlin

Kotlin é uma linguagem de programação *open-source*, estaticamente tipada que suporta tanto orientação a objetos quanto programação funcional. É desenvolvida pela JetBrains (JETBRAINS, 2022) e roda na Java Virtual Machine (JVM), sendo totalmente interoperável com Java. Desta forma, é possível usar diversos *frameworks* existentes e que facilitam o desenvolvimento de servidores, entre eles *Spring* e *Spark*. Também é possível utilizar *Maven* para automação e gerenciamento do projeto.

Além de ser mais conciso e expressivo do que Java (KOTLIN, 2022b) , fazendo, desta maneira, com que o custo de manutenção seja reduzido, Kotlin traz outras vantagens como *null safety* (*Kotlin* busca eliminar os perigos de referências nulas, um objeto só pode ser nulo se o programador explicitar isso), parâmetros opcionais e *data classes* - classes cujo objetivo principal é armazenar dados e que tem funcionalidades padrões como *equals* e *toString* derivadas pelo compilador.

Por estes motivos, Kotlin foi a linguagem escolhida para o desenvolvimento do servidor.

2.5.2 Spring Framework

Spring Framework (SPRING, 2022) é um *framework open-source* que provê suporte de infraestrutura para o desenvolvimento de aplicativos que rodem na JVM. Ele permite ao programador abstrair certos aspectos de infraestrutura, tornando possível o foco no desenvolvimento da aplicação.

Dois dos principais conceitos do *Spring Framework* são *injeção de dependências* e *inversão de controle* e é justamente por esse motivo que ele foi utilizado na implementação deste trabalho.

A inversão de controle permite aos programadores delegar parcialmente a responsabilidade do tratamento do fluxo de execução de código para o *Spring*. Dessa maneira não precisamos nos preocupar em gerenciar certos comportamentos, como a criação de objetos na inicialização da aplicação.

Já a injeção de dependências é usada para evitar acoplamento de código e isto é feito ao prover instâncias de classes que um objeto precisa sem que ele as intancie por si mesmo. A injeção de dependências é uma forma de se obter a inversão de controle.

2.5.3 Maven

Maven (MAVEN, 2022) é uma ferramenta para automação e gerenciamento de projetos. Assim como *Spring*, ela busca facilitar a vida do desenvolvedor, permitindo que este foque na implementação do sistema em si. Ela foi escolhida como gerenciadora de construção (*build*) do servidor, pois simplifica significamente o processo de construção e manutenção de programas e também porque fornece um sistema de construção uniforme através de seu arquivo de configuração *POM* (*Project Object Model*).

Ao usar *Maven*, não é necessário se preocupar com a criação de diversos módulos, nem sua ordem de compilação. Ademais, é possível adicionar dependências (bibliotecas e *framewoks*) simplesmente configurando-as no *POM*. Essa abstração de complexidade agiliza de forma significativa o processo de gerenciamento de aplicações.

2.5.4 Spark Java

Spark Java (SPARK, 2022) é um *framework open-source* usado para criação de aplicações *WEB* em Java e Kotlin. Seu objetivo é fornecer uma alternativa para os desenvolvedores criarem aplicativos *WEB* expressivos com

o mínimo possível de código *boilerplate*.

Ele possui o componente *Spark Rest*, o qual é usado neste trabalho para a definição de *REST endpoints*. Embora também fosse possível usar *Spring* para a criação das *REST APIs*, a autora deste projeto preferiu a utilização do *Spark* por julgar ser possível criar um código mais limpo com ele, já que, ao contrário do que acontece com *Spring*, não é necessário utilizar *annotation* com o *Spark*.

2.5.5 MySQL

Para o armazenamento de dados deste projeto, foi escolhida a utilização de um banco de dados relacional, ou seja, um banco de dados que armazena dados relacionados entre si em tabelas, conforme modelo proposto por Edgar F. Codd (CODD, 1970). Este modelo propõe uma forma intuitiva de representação da relações dos dados e permite o seu gerenciamento de forma segura e consistente.

O sistema de gestão de bases de dados relacional escolhido foi o *MySQL* (MYSQL, 2022), um SGBD *open-source*. Ele fornece suporte a comandos SQL, transações e propriedades ACID (*Atomicity, Consistency, Isolation, Durability*). Além de ser o o SGBD de código aberto mais popular mundialmente, também é compatível com provedores de serviço de nuvem, como a AWS (AMAZON, 2022), o que tornaria mais fácil uma migração futura, caso se opte por isto.

2.5.6 React

React (REACT, 2022b) é uma biblioteca *open-source* de *JavaScript* feita para construir interfaces de usuário. Ela fornece grande flexibilidade ao programador, não o forçando a usar uma arquitetura específica como acontece no caso de *Angular* ou *Vue*.

React permite a criação de componentes de UI simples e reusáveis e os renderiza com eficiência, atualizando apenas os componentes necessários quando os dados forem alterados. Atualmente é a biblioteca de *JavaScript*

mais popular (NPMTRENDS, 2022), sendo possível encontrar diversos materiais para auxiliar no desenvovimento.

2.5.7 Material UI

Material UI (MATERIALUI, 2022) é uma biblioteca que permite aos programadores importar e usar diversos componentes para criação de interfaces de usuário em aplicativos *React*, economizando uma quantidade significativa de tempo de desenvolvimento. Ela torna mais fácil a criação de páginas, pois não é necessário criar todos os componentes do zero. Por este motivo, ela foi usada na implementação do cliente desta plataforma.

3 TRABALHOS RELACIONADOS

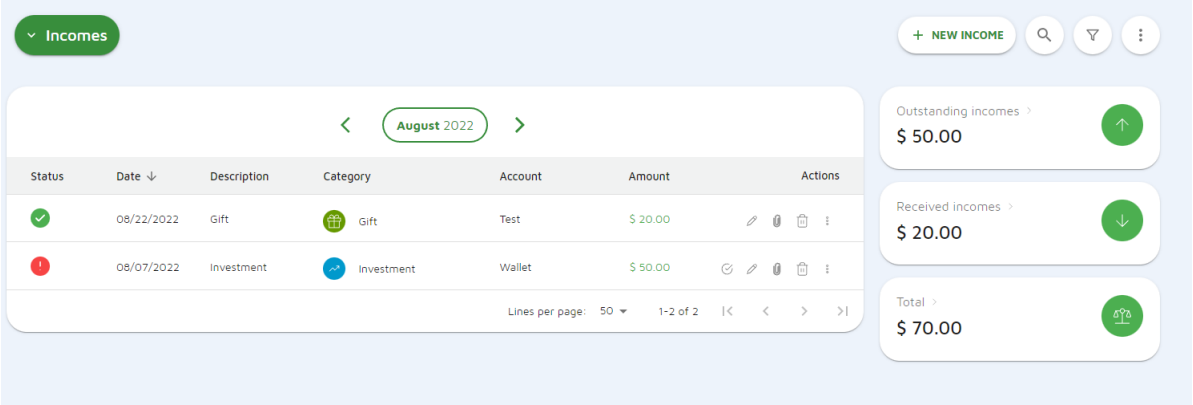
Ao procurar na *WEB* e na loja de aplicativos do *Android*, são encontradas aplicações de gerenciamento financeiro com propostas similares a deste projeto. Neste capítulo serão apresentados alguns destes trabalhos relacionados encontrados e, ao final, eles são comparados apresentando similaridades e diferenças.

3.1 Mobills

O aplicativo *Mobills* (MOBILLS, 2022) é um planejador de orçamentos e está disponível tanto para *Android* e *IOS*, quanto para *WEB*. Nele é possível gerenciar orçamentos mensais, cadastrando um valor máximo que se gostaria de gastar ao total no mês, assim como o máximo que ser quer gastar em determinada categoria. Ele também faz a gestão de ganhos e despesas, permitindo que o usuário os cadastre, edite, remova e consulte. Além disso, é possível cadastrar cartões de crédito e diferentes contas, como conta de poupança ou de investimentos, as quais são associadas aos ganhos e despesas para melhor organização.

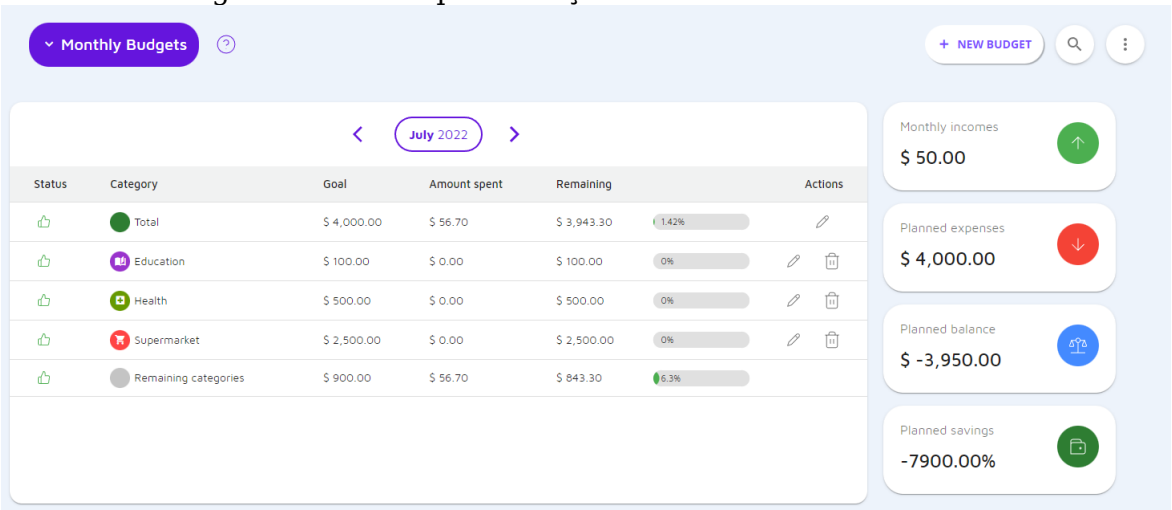
O *Mobills* possui uma versão gratuita com limitações, como permitir apenas o cadastro de um cartão de crédito e de apenas um orçamento. Para liberar mais funcionalidades, é necessário adquirir a versão paga.

Figura 3.1: Exemplo de ganhos cadastrados no *Mobills*



Fonte: (MOBILLS, 2022)

Figura 3.2: Exemplo de orçamento criado no *Mobills*



Fonte: (MOBILLS, 2022)

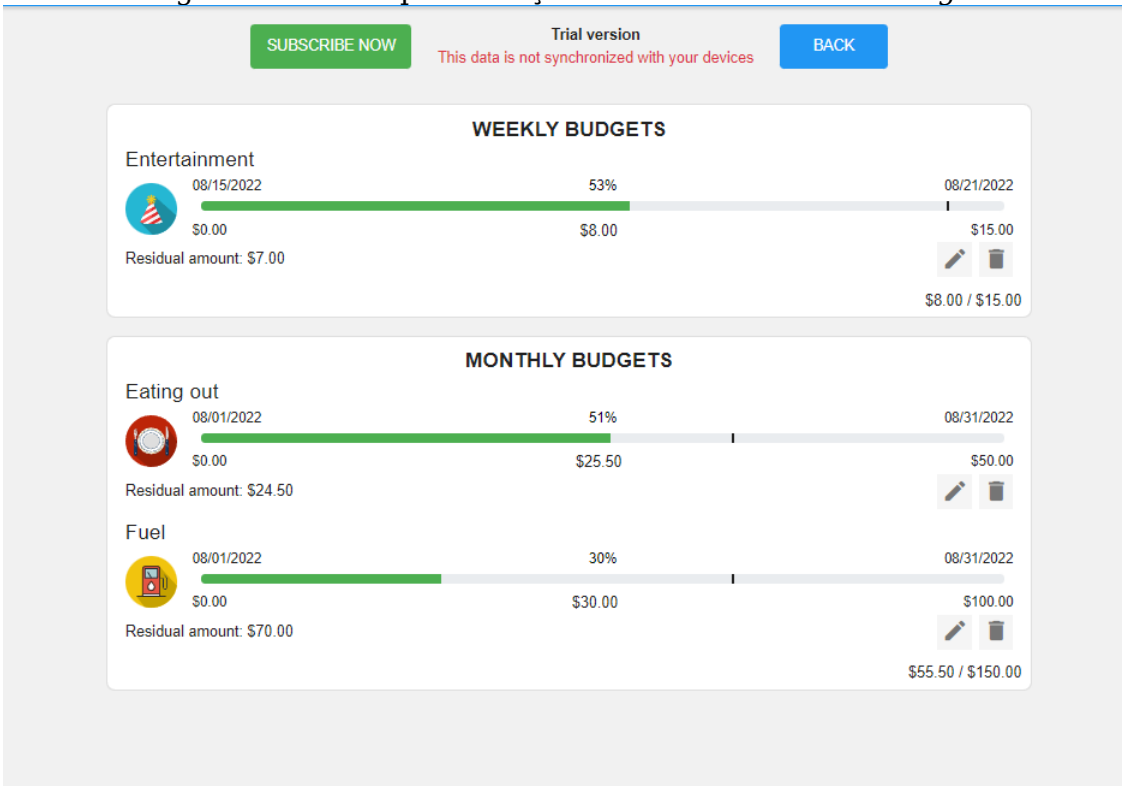
3.2 Fast Budget

O *Fast Budget* (FASTBUDGET, 2022) é outro aplicativo existente para auxiliar na gerência de finanças pessoais. Ele também está disponível para *Android*, *IOS* e *WEB* e permite o gerenciamento de ganhos, despesas, orçamento, cartões de crédito e contas, de forma similar ao *Mobills*.

Entre seus diferenciais estão o fato de ser possível criar múltiplos orçamentos para a mesma semana ou mês - sendo permitido, inclusive, que orçamentos para a mesma conta sejam cadastrados duplicadamente, algo hipoteticamente não desejável - enquanto o *Mobills* permite apenas um único orçamento mensal, independente do número de contas cadastradas. Além disso, existe uma obrigatoriedade de informar para quem uma despesa está sendo paga e de quem um ganho está sendo recebido. Ele também oferece a funcionalidade adicional de permitir a exportação de relatórios em *PDF* e *CSV* pela plataforma *WEB*.

O *Fast Budget* é pago, porém possui uma versão de testes com dados pré-cadastrados para que os usuários possam testar a plataforma antes de assiná-la.

Figura 3.3: Exemplo de orçamento criado no *Fast Budget*



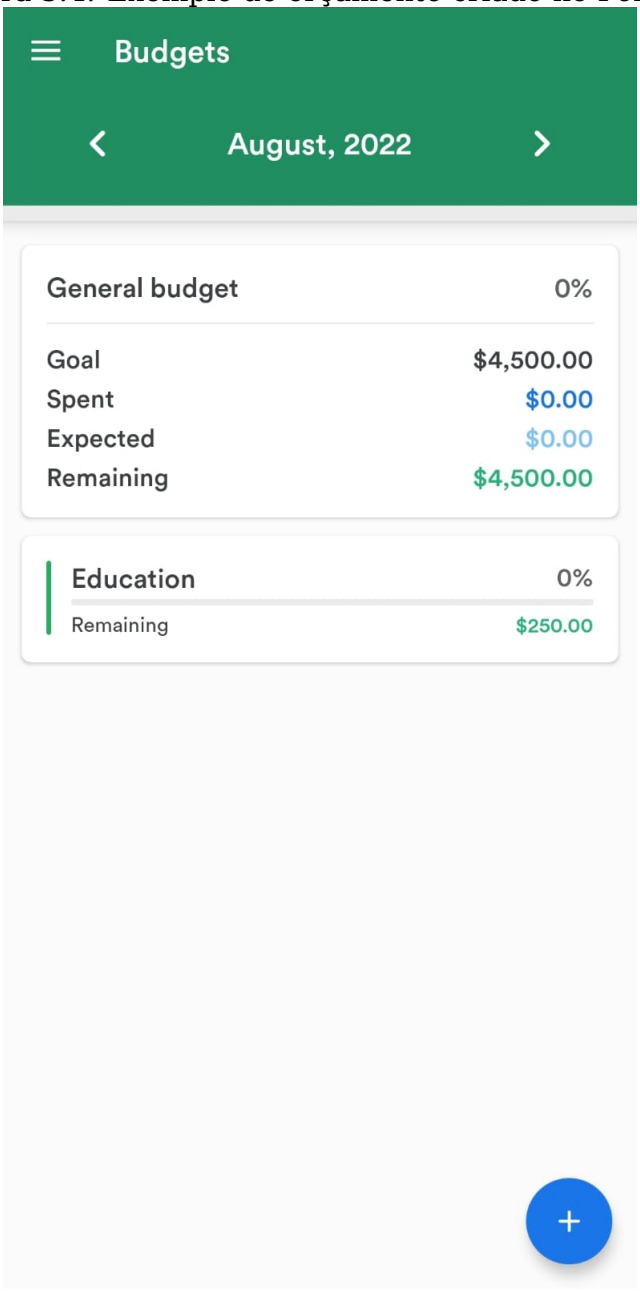
Fonte: (FASTBUDGET, 2022)

3.3 Fortuno

O *Fortuno* (FORTUNO, 2022) é um aplicativo para *Android* cujo objetivo é ajudar no controle de finanças pessoais. Tem funcionalidades similares as dos outros trabalhos apresentados neste capítulo, também oferecendo gerência de ganhos, despesas, cartões de crédito, orçamentos e contas. Seu principal diferencial é a opção de serem cadastrados alertas com relação aos orçamentos para que sejam recebidas notificações *popup* no celular.

Sua versão gratuita possui limitações, como não permitir a exportação dos dados em *PDF*, sendo necessário fazer uma assinatura da versão paga para ter acesso a todas as funcionalidades.

Figura 3.4: Exemplo de orçamento criado no *Fortuno*



Fonte: (FORTUNO, 2022)

3.4 Análise Comparativa

Após analizar as aplicações, é perceptível que existem muitas semelhanças entre elas. Todas têm como funcionalidades principais a gerência de ganhos, despesas, orçamentos, cartões de crédito e a possibilidade de se cadastrar múltiplas contas. Por este motivo, decidiu-se que a plataforma desenvolvida neste trabalho, e que será chamada de

"Presente Aplicação" no decorrer desta seção, estaria centrada em ganhos, despesas, orçamentos e cartões de crédito. Neste primeiro momento deixou-se de lado o cadastro de contas devido ao tempo disponível e escopo deste trabalho.

A Tabela 3.1 mostra um comparativo das principais funcionalidades das plataformas apresentadas nesta seção, assim como da Presente Aplicação.

Tabela 3.1: Comparativo de funcionalidades

	Mobills	**Fast Budget**	**Fortuno**	**Presente Aplicação**
Gerenciamento de ganhos	Sim	Sim	Sim	Sim
Gerenciamento de despesas	Sim	Sim	Sim	Sim
Gerenciamento de orçamentos	Sim	Sim	Sim	Sim
Gerenciamento de orçamento para contas diferentes	Não	Sim	Não	Não
Gerenciamento de orçamentos para categorias	Sim	Sim	Sim	Sim
Gerenciamento de cartões de crédito	Sim	Sim	Sim	Sim
Exportação de dados	Na versão *premium mobile*	Na versão *premium*	Na versão *premium*	Não
Notificações popup	Não	Não	Sim	Não
Limitações na versão gratuita	Sim	Sim	Sim	Não

4 MODELAGEM E IMPLEMENTAÇÃO

Neste capítulo estão descritos detalhes sobre a modelagem e a implementação deste projeto. Primeiramente, é fornecida uma visão geral do desenvolvimento, seguido por detalhes de implementação do *backend* e do *frontend*.

4.1 Visão Geral do Desenvolvimento

O desenvolvimento deste trabalho foi dividido em duas etapas principais: *backend* e *frontend*. A primeira etapa a ser implementada, e também o foco deste projeto, foi o *backend*.

O *backend* foi implementado seguindo o estilo de desenvolvimento *Clean Code* (MARTIN, 2008) e usando *Onion Architecture* (PALERMO, 2008). O primeiro passo desta etapa foi definir o domínio da plataforma e seus requisitos. Feito isto, foram definidas as *APIs* suportadas e iniciado o desenvolvimento.

Após a finalização do *backend* se deu início à implementação do cliente. Ele foi desenvolvido em *React* e com o uso da *Material UI* (MATERIALUI, 2022), uma biblioteca de componentes *React*. Nesta etapa foram criadas telas que utilizam os *endpoints* disponiblizados pelo servidor da solução.

4.2 Requisitos

Antes de ser iniciado o desenvolvimento, foram definidos o domínio e os requisitos que devem ser atendidos pela plataforma. Para isto, foram consideradas as funcionalidades presentes nos trabalhos relacionados apresentados no Capítulo 3, sendo, desta forma, decidido que o domínio da solução criada neste trabalho estaria centrada em ganhos, despesas, cartões de crédito e orçamentos.

Após a definição do domínio, foram elencados os requisitos funcionais e não-funcionais, os quais podem ser verificados nas subseções a seguir. Esses requisitos foram elencados considerando-se os pontos principais dos trabalhos

relacionados, eles estão presentes em todos os trabalhos consultados e são o *core* de aplicações do tipo. Desta maneira, eles serão incluídos também na plataforma criada neste projeto.

4.2.1 Requisitos funcionais

Requisitos funcionais são aqueles que definem o que o *software* fará, isto é, quais funcionalidades estarão disponíveis. Nesta subseção, é possível verificar os requisitos funcionais deste projeto:

- O sistema deve permitir o cadastro de usuários;
- O sistema deve permitir que usuários acessem a plataforma através de *login*;
- O sistema deve permitir que usuários logados possam cadastrar, consultar e excluir ganhos;
- O sistema deve permitir que usuários logados possam cadastrar, consultar e excluir despesas;
- O sistema deve permitir que usuários logados possam cadastrar, consultar e excluir orçamentos;
- O sistema deve permitir que usuários logados possam cadastrar, consultar e excluir orçamentos para categorias;
- O sistema deve permitir que usuários logados possam cadastrar, consultar e excluir cartões de crédito;
- O sistema não deve permitir que usuários acessem os dados de outrem;
- O sistema não deve permitir que um mesmo usuário cadastre orçamentos duplicados para o mesmo mês;
- O sistema não deve permitir que um mesmo usuário cadastre mais de um cartão de crédito com mesmo nome;
- O sistema não deve permitir que um cartão de crédito seja cadastrado com o dia de fechamento de fatura maior do que o dia de pagamento.

4.2.2 Requisitos não funcionais

Já os requisitos não funcionais definem características e limitações do sistema. Enquanto os requisitos funcionais determinam *o que* a solução faz, os requisitos não funcionais estabelecem *como* a plataforma funcionará. A seguir estão os requisitos não funcionais deste trabalho:

- Os dados devem ser salvos em um banco de dados relacional;
- O sistema será desenvolvido para rodar na *WEB*;
- O servidor da plataforma deve disponibilizar *REST APIs* para comunicação com clientes;
- As senhas dos usuários devem estar criptografadas;
- O sistema deve possuir categorias pré-cadastradas.

4.3 *Backend*

Nas próximas seções estão descritos os detalhes de implementação do servidor da plataforma.

4.3.1 Domínio

Como mencionado na seção anterior, a primeira fase do desenvolvimento desta solução foi a definição do domínio a ser trabalhado. Após consultar trabalhos relacionados, decidiu-se que neste primeiro momento a implementação deste projeto estaria centrada em **ganhos**, **despesas**, **orçamentos** e **cartões de crédito**.

Ganhos refere-se ao dinheiro recebido pelo usuário, por exemplo, salário e dividendos. As *despesas* são os gastos que uma pessoa tem, como aluguel e pagamento de contas. Já o *orçamento* permite a organização de quanto se quer gastar no período de um mês e, por fim, é possível também obter informações de *cartões de crédito*, como limite e compras realizadas.

Estas quatro entidades foram escolhidas por se tratarem do núcleo de aplicações de gerência de finanças pessoais, estando presentes em todos os

projetos relacionados verificados. Desta forma, considerou-se que desenvolver funcionalidades com base nelas criaria uma boa fundação para a primeira iteração de implementação do servidor. Cada uma dessas entidades é representada por uma classe, como podemos ver a seguir.

Os **ganhos** são representados pela classe *Income*. Ela possui quatro atributos:

- *Id*: identificador único de um ganho. É um parâmetro opcional, já que, ao fazer uma requisição para o cadastro de um ganho, ainda não haverá um *id* associado a ele. Apenas quando o registro for feito no banco de dados é que teremos um *id* associado ao ganho;
- *Value*: valor do ganho em questão. Usa-se *BigDecimal* para que se tenha uma boa precisão;
- *Category*: cada ganho possui uma categoria associada a si. Esta categoria representa o tipo de ganho, por exemplo, salário ou aluguel;
- *Date*: a data em que o ganho foi recebido;
- *Description*: de forma opcional, é possível adicionar alguma descrição ao ganho.

Nesta classe há também uma validação para garantir que o valor informado é válido, pois um ganho deve possuir um valor maior do que zero. Essa validação acontece dentro do bloco *init* do *Kotlin*, o qual é executado quando um objeto é instanciado.

Figura 4.1: *Income Class*

Fonte: Elaborada pela autora

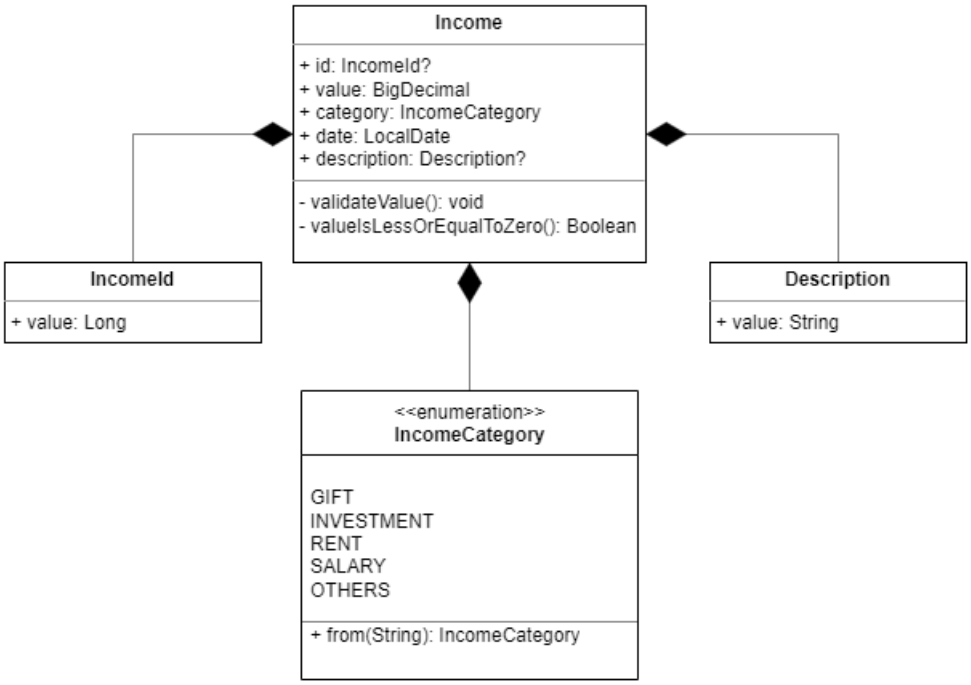
As categorias de um ganho são representadas por um *enum,* uma classe que representa um grupo de constantes.

Figura 4.2: *IncomeCategory Class*

Fonte: Elaborada pela autora

Na figura 4.3 está o diagrama de classes que representa a parte de ganhos.

Figura 4.3: Diagrama de classes representando *ganhos*



Fonte: Elaborada pela autora

Já a classe *Expense* representa as **despesas**. Seus parâmetros são:

- *Id*: identificador único de uma despesa. Da mesma forma que o *id* de um ganho, também é um parâmetro opcional;
- *Value*: valor da despesa;
- *Category*: categoria que representa o tipo de gasto, como educação e transporte;
- *Date*: a data em que a despesa foi realizada;
- *Description*: de forma opcional, é possível adicionar alguma descrição a despesa;
- *CreditCardId*: caso a despesa tenha sido feita usando um cartão de crédito, o id do cartão será associado a ela.

Aqui também existe validação para garantir que o valor informado é maior do que zero. Além disso, há um método público que informa se uma despesa foi realizada com cartão de crédito.

Figura 4.4: *Expense Class*

```
data class Expense(val id: ExpenseId? = null,
                   val value: BigDecimal,
                   val category: ExpenseCategory,
                   val date: LocalDate,
                   val description: Description? = null,
                   val creditCardId: CreditCardId? = null) {
    asgoveia
    init {
        validateValue()
    }

    asgoveia *
    private fun validateValue() {
        if(valueIsLessOrEqualToZero()) {
            throw InvalidValueException("expense")
        }
    }

    asgoveia *
    private fun valueIsLessOrEqualToZero(): Boolean {
        return value.compareTo(BigDecimal.ZERO) != 1
    }

    asgoveia
    fun isCreditCardExpense(): Boolean {
        return creditCardId != null
    }
}
```

Fonte: Elaborada pela autora

A classe *ExpenseCategory* define as categorias de despesas:

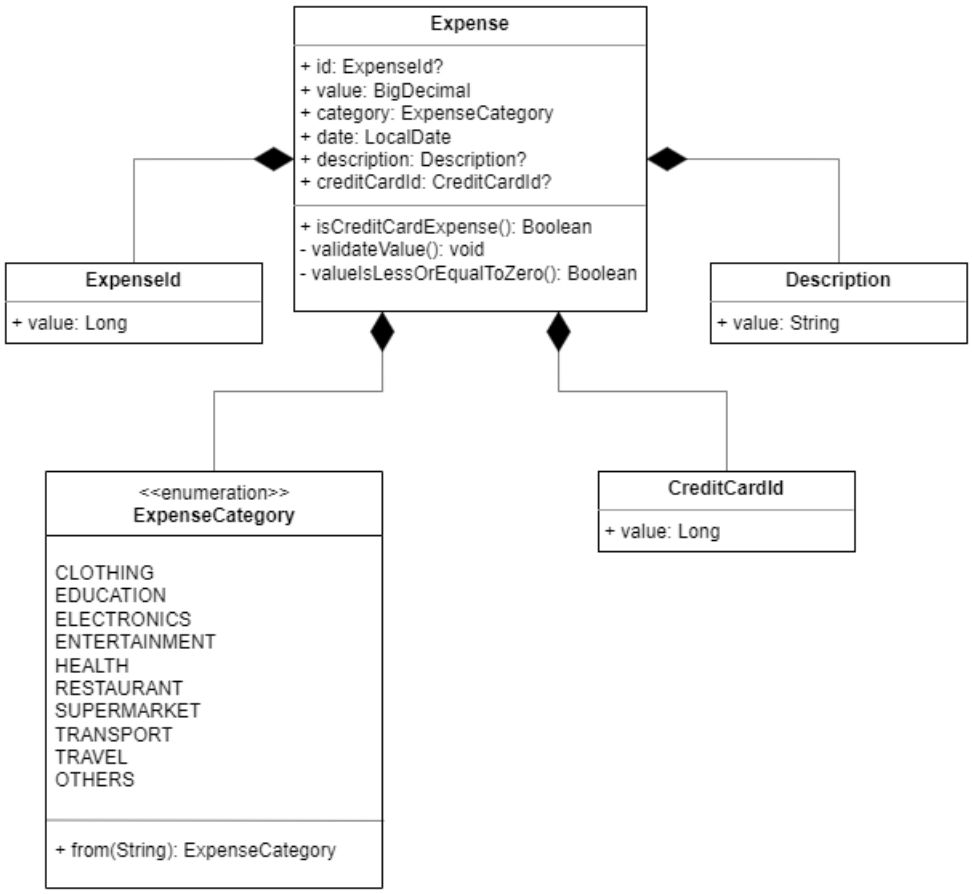
Figura 4.5: *ExpenseCategory Class*

```
enum class ExpenseCategory (val value: String) {

    CLOTHING( value: "clothing"),
    EDUCATION( value: "education"),
    ELECTRONICS( value: "electronics"),
    ENTERTAINMENT( value: "entertainment"),
    HEALTH( value: "health"),
    RESTAURANT( value: "restaurant"),
    SUPERMARKET( value: "supermarket"),
    TRANSPORT( value: "transport"),
    TRAVEL( value: "travel"),
    OTHERS( value: "others");

    asgoveia
    companion object {
        asgoveia
        fun from(value: String): ExpenseCategory {
            return Arrays.stream(values()) Stream<ExpenseCategory!>!
                .filter { category: ExpenseCategory -> category.value.equals(value, ignoreCase = true) }
                .findFirst() Optional<ExpenseCategory!>
                .orElseThrow { IllegalArgumentException(String.format("Invalid expense category %s", value)) }
        }
    }
}
```

Fonte: Elaborada pela autora

A Figura 4.6 apresenta o diagrama de classes que representa a parte de despesas.

Figura 4.6: Diagrama de classes representando *despesas*



Fonte: Elaborada pela autora

Os **orçamentos** são representados pela classe *Budget*. Ela possui quatro parâmetros:

- *Goal*: o valor máximo que se gostaria de gastar durante o mês;
- *Date*: representa o mês deste orçamento;
- *Spent*: valor já gasto do orçamento em questão;
- *Category*: campo opcional. Se não estiver preenchido, significa que o orçamento é global, isto é, o máximo que se quer gastar ao total no mês. Caso haja uma categoria, o valor máximo a ser gasto é referente somente a ela.

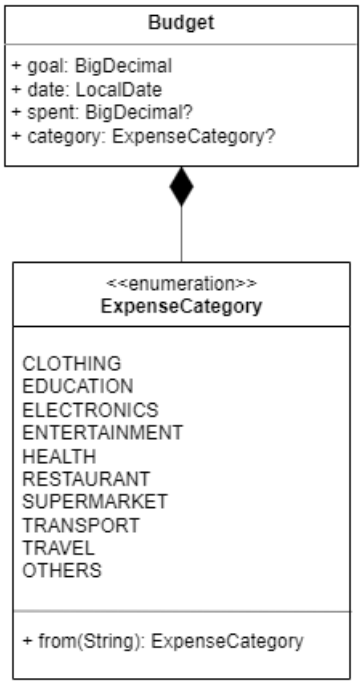
Figura 4.7: *Budget Class*

```
data class Budget(val goal: BigDecimal,
                  val date: LocalDate,
                  var spent: BigDecimal? = BigDecimal.ZERO,
                  val category: ExpenseCategory? = null)
```

Fonte: Elaborada pela autora

Na figura 4.8 está o diagrama de classes que simboliza os orçamentos.

Figura 4.8: Diagrama de classes representando *orçamentos*



Fonte: Elaborada pela autora

Por fim, a classe *CreditCard* representa os **cartões de crédito**. Seus parâmetros são:

- *Id*: identificador do cartão de crédito;
- *Name*: nome do cartão de crédito;
- *Limit*: limite do cartão;
- *ClosingDay*: dia de fechamento da fatura;
- *DueDay*: dia de pagamento da fatura.

Para a criação de um objeto *CreditCard*, é preciso que o limite informado seja maior do que zero e que o dia de fechamento da fatura aconteça antes do dia de pagamento.

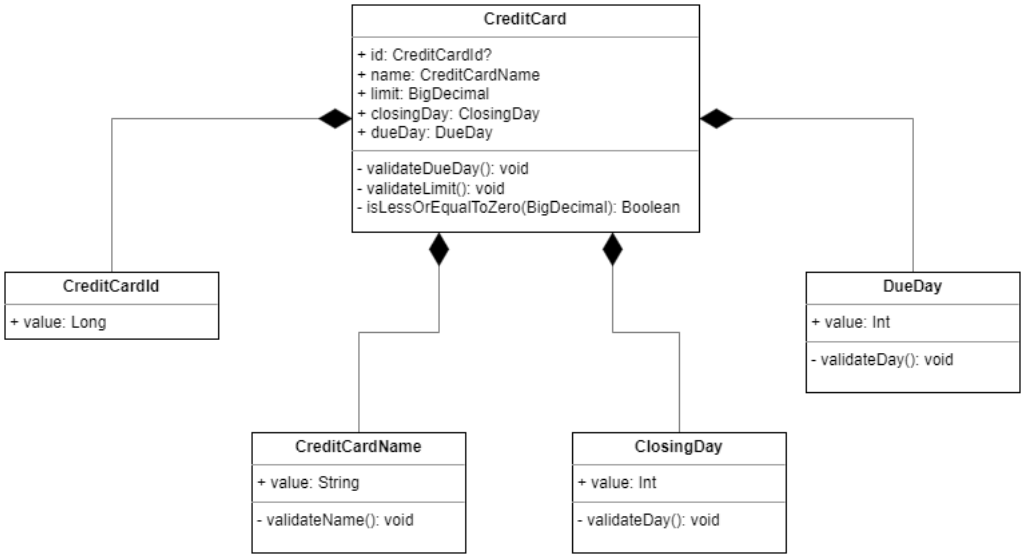
Figura 4.9: *CreditCard Class*

```
data class CreditCard(val id: CreditCardId? = null,
                      val name: CreditCardName,
                      val limit: BigDecimal,
                      val closingDay: ClosingDay,
                      val dueDay: DueDay) {
    asgoveia
    init {
        validateDueDay()
        validateLimit()
    }

    asgoveia
    private fun validateDueDay() {
        if(closingDay.value >= dueDay.value) {
            throw IllegalDateException("Due day must be greater than closing day")
        }
    }

    asgoveia
    private fun validateLimit() {
        if(isLessOrEqualToZero(limit)) {
            throw InvalidValueException("limit")
        }
    }

    asgoveia
    private fun isLessOrEqualToZero(value: BigDecimal): Boolean {
        return value.compareTo(BigDecimal.ZERO) != 1
    }
}
```

Fonte: Elaborada pela autora

O diagrama de classes dos cartões de crédito está na Figura 4.10.

Figura 4.10: Diagrama de classes representando *cartões de crédito*

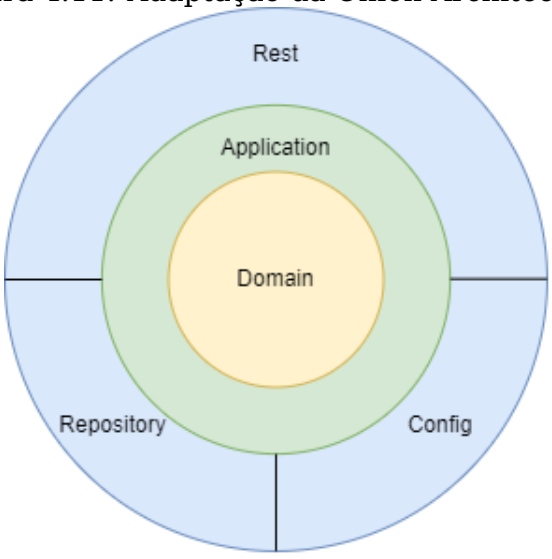


Fonte: Elaborada pela autora

4.3.2 *Onion Architecture*

A arquitetura usada para o servidor desenvolvido nesta monografia é uma adaptação da Onion Architecture (PALERMO, 2008), mencionada no Capítulo 2. As vantagens de sua utilização se mantém as mesmas: desacoplamento e separação de responsabilidades. Na Figura 4.11, um diagrama desta adaptação:

Figura 4.11: Adaptação da *Onion Architecture*



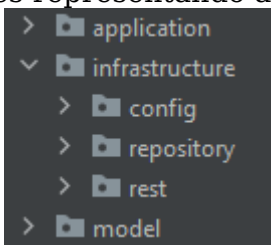
Fonte: Elaborada pela autora

A seguir estão as especificações das camadas usadas:

- *Domain*: é uma junção das camadas *Domain Model* e *Domain Services* propostas por Palermo. Aqui se encontram todas as entidades do domínio da plataforma, assim como as interfaces de repositórios. As classes *Income*, *Expense*, *Budget* e *CreditCard*, mencionadas anteriormente nesta seção, ficam aqui;
- *Application*: é equivalente à *Application Services* descrita na seção 2.3. Aqui é feita a comunicação entre as camadas externas e a *Domain*. Também é feita a orquestração da lógica de negócios, por exemplo, validação para garantir que o usuário tem acesso ao item sobre o qual está tentando realizar alguma ação;
- *Rest*: similar à *User Interface*, é aqui o ponto de entrada de requisições dos usuários na aplicação. Todas as operações que chegam ao servidor são feitas através de requisições *HTTP Rest*. Assim sendo, o nome *Rest* pareceu retratar melhor a função desta camada;
- *Repository*: é onde acontece o acesso ao banco de dados. Equivalente à *Infrastructure* na arquitetura de Palermo;
- *Config*: nesta camada ficam as configurações da aplicação. As configurações relativas à injeção de dependência do *Spring Framework* estão aqui.

Na Figura 4.12 é possível visualizar os pacotes criados para representação da *Onion Architecture*:

Figura 4.12: Pacotes representando a *Onion Architecture*



Fonte: Elaborada pela autora

O pacote *infrastructure* agrupa as camadas mais externas.

4.3.3 *Clean Code*

Durante o desenvolvimento desta plataforma, foram empregadas práticas de *Clean Code*. Na *AuthenticationService*, classe utilizada na camada *Application* e que, como o próprio nome sugere, é responsável por tratar questões de autenticação, é possível observar o uso de três destas práticas: o uso de nomes significativos, o uso de métodos pequenos e com uma única responsabilidade e o uso de exceções que retratem o contexto em que elas aconteceram.

Figura 4.13: Código de *login* presente na *AuthenticationService*

```
33    fun login(username: Username, givenPassword: String): AuthToken {
34        validateCredentials(username, givenPassword)
35        val authToken = createAuthenticationToken()
36        sessionManager.addSession(authToken, username)
37        return authToken
38    }
39
          asgoveia
40    private fun validateCredentials(username: Username, givenPassword: String) {
41        val hashedCredentials = repository.getCredentialsOf(username)
42        if(!SCryptUtil.check(givenPassword, hashedCredentials.password)) {
43            throw AuthenticationException()
44        }
45    }
46
          asgoveia
47    private fun createAuthenticationToken(): AuthToken {
48        val secureRandom = SecureRandom()
49        val base64Encoder = Base64.getUrlEncoder()
50        val randomBytes = ByteArray( size: 128)
51        secureRandom.nextBytes(randomBytes)
52        return AuthToken(base64Encoder.encodeToString(randomBytes))
53    }
```

Fonte: Elaborada pela autora

Na Figura 4.13 temos o código referente a parte de *login* presente na *AuthenticationService*. Nele é possível observar que tanto os nomes dos métodos, quantos os nomes das variáveis transmitem o seu propósito. Apenas ao ler os nomes dos métodos, sabemos o que eles fazem (*login*, validação das credenciais fornecidas pelo usuário e criação de *token* de autenticação), sem precisar entrar em detalhes da implementação para obter essa informação.

Ademais, há também preocupação com a facilidade de leitura como visto na linha 41: o repositório de autenticação tem um método chamado *getCredentialsOf*, o que torna a leitura fluída quando passamos o parâmetro *username*.

Os três métodos apresentados também são curtos e tem responsabilidades bem definidas. O método *login* é público e é quem faz a comunicação com a camada *Rest*. Ele, por sua vez, delega a parte de validação de credenciais e criação de *token* para outros métodos, ao invés de assumir múltiplas responsabilidades.

Por fim, no caso das credenciais fornecidas pelo usuário serem inválidas, temos a utilização de uma exceção de autenticação, a qual é mapeada e tratada na camada *Rest*, indicando claramente que o problema ocorreu por erro de autenticação.

Figura 4.14: Mapeamento da *AuthenticationException* na camada *Rest*

```
81      protected fun forbidden() {
82          http.exception(UserNotFoundException::class.java) { _, _, res ->
83              exception(res, HttpStatus.FORBIDDEN_403, message: "Forbidden") }
84          http.exception(AccessDeniedCustomException::class.java) { _, _, res ->
85              exception(res, HttpStatus.FORBIDDEN_403, message: "Forbidden") }
86          http.exception(AuthenticationException::class.java) { _, _, res ->
87              exception(res, HttpStatus.UNAUTHORIZED_401, message: "Unauthorized") }
88      }
```

Fonte: Elaborada pela autora

4.3.4 Banco de Dados

Conforme já mencionado na seção 2.5, foi utilizado o banco relacional *MySQL* (MYSQL, 2022) para o armazenamento de dados deste sistema. Para a criação do banco *financial_control* desta plataforma, foi executado o seguinte script *SQL* na ferramenta DBeaver (DBEAVER, 2022):

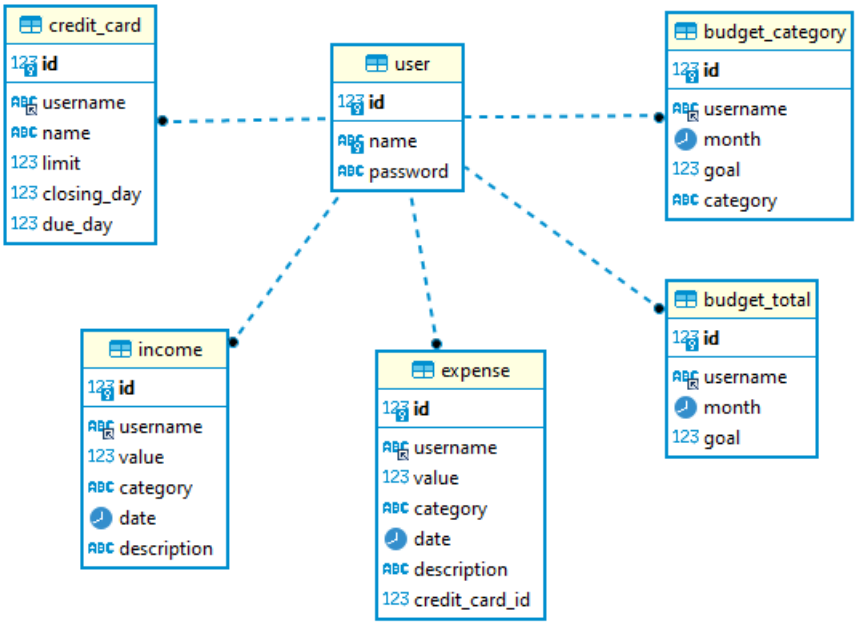
```
CREATE TABLE IF NOT EXISTS user (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(256) UNIQUE NOT NULL,
    password VARCHAR(1024) NOT NULL
```

```
);

CREATE TABLE IF NOT EXISTS income (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(256) NOT NULL,
    value DECIMAL(19,2) NOT NULL,
    category VARCHAR(256) NOT NULL,
    date DATE NOT NULL,
    description VARCHAR(1024),
    FOREIGN KEY (username) REFERENCES user(name)
);

CREATE TABLE IF NOT EXISTS expense (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(256) NOT NULL,
    value DECIMAL(19,2) NOT NULL,
    category VARCHAR(256) NOT NULL,
    date DATE NOT NULL,
    description VARCHAR(1024),
    FOREIGN KEY (username) REFERENCES user(name)
);

ALTER TABLE expense
ADD COLUMN credit_card_id BIGINT;

CREATE TABLE IF NOT EXISTS budget_total (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(256) NOT NULL,
    month DATE NOT NULL,
    goal DECIMAL(19,2) NOT NULL,
    FOREIGN KEY (username) REFERENCES user(name)
);

ALTER TABLE budget_total
```

```
ADD UNIQUE user_month_unique(username, month);

CREATE TABLE IF NOT EXISTS budget_category (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(256) NOT NULL,
    month DATE NOT NULL,
    goal DECIMAL(19,2) NOT NULL,
    category VARCHAR(256) NOT NULL,
    FOREIGN KEY (username) REFERENCES user(name)
);

ALTER TABLE financial_control.budget_category
ADD CONSTRAINT user_month_category_unique
UNIQUE KEY (username,`month`,category);

CREATE TABLE IF NOT EXISTS credit_card (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(256) NOT NULL,
    name VARCHAR(256),
    `limit` DECIMAL(19,2) NOT NULL,
    closing_day TINYINT NOT NULL,
    due_day TINYINT NOT NULL,
    FOREIGN KEY (username) REFERENCES user(name)
);

ALTER TABLE financial_control.credit_card
ADD CONSTRAINT user_card_name_unique UNIQUE KEY (username, name);
```

Após a execução deste *script*, foi obtido o diagrama *ER* da Figura 4.15:

Figura 4.15: Diagrama *ER*



Fonte: Elaborada pela autora

A tabela **user** é onde são armazenados o nome e senha dos usuários. Todas as outras tabelas tem uma chave estrangeira referenciando a coluna *name* dela. A tabela **income** guarda o valor, a categoria, a data e, opcionalmente, a descrição dos ganhos dos usuários. Similarmente, a **expense** armazena as mesmas informações para gastos, além de ter também o id do cartão de crédito quando a despesa for feita com um cartão. A tabela **budget_total** armazena o orçamento total do mês, isto é, o máximo que se quer gastar, independente de categoria. Como não faz sentido ter mais de um orçamento por mês para determinado usuário, for criada a *constraint user_month_unique* que garante esta unicidade. Temos também a **budget_category**, a qual armazena o orçamento mensal para categorias específicas. Da mesma forma, temos a *constraint user_month_category_unique* para garantir a unicidade dos orçamentos mensais para determinada categoria e usuário. Por último, temos a tabela **credit_card**, a qual guarda o limite, dia de fechamento, dia de pagamento e o nome dos cartões de crédito dos usuários. Foi criada, também, a *constraint user_card_name_unique* a qual não permite a repetição de nomes de cartões de créditos para o mesmo usuário.

4.3.5 *REST APIs*

Toda a comunicação realizada entre o servidor e os usuários se dá através da utilização de *REST APIs*. A implementação destas *APIs* é independente do cliente que a está utilizando, isto é, é possível realizar operações no *backend* através do cliente desenvolvido neste trabalho, por clientes de *API* como *Postman* (POSTMAN, 2022) ou por qualquer outro cliente que respeite os contratos das *APIs* disponíveis.

Para a criação dos *endpoints*, foi usado o *Spark Java*, já descrito na seção 2.5 desta monografia. Na Figura 4.16 estão os *endpoints* disponíveis no servidor:

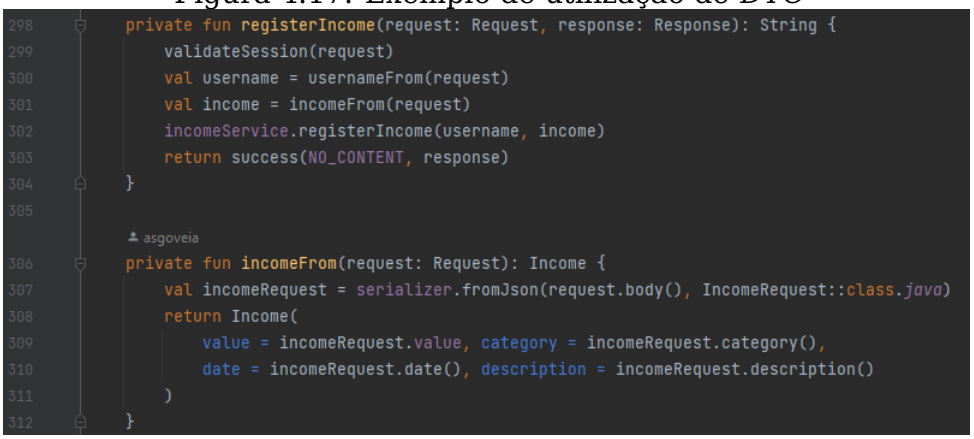
Figura 4.16: *Endpoints* do servidor

```
override fun endpoints() {
    http.post( path: "$BASE_ENDPOINT/signup") { req, res -> signUp(req, res) }
    http.post( path: "$BASE_ENDPOINT/login") { req, res -> login(req, res) }
    http.post( path: "$BASE_ENDPOINT/income") { req, res -> registerIncome(req, res) }
    http.post( path: "$BASE_ENDPOINT/expense") { req, res -> registerExpense(req, res) }
    http.post( path: "$BASE_ENDPOINT/budget") { req, res -> registerBudget(req, res) }
    http.post( path: "$BASE_ENDPOINT/budgetCategory") { req, res -> registerCategoryBudget(req, res) }
    http.post( path: "$BASE_ENDPOINT/creditCard") { req, res -> registerCreditCard(req, res) }

    http.delete( path: "$BASE_ENDPOINT/income") { req, res -> removeIncome(req, res) }
    http.delete( path: "$BASE_ENDPOINT/expense") { req, res -> removeExpense(req, res) }
    http.delete( path: "$BASE_ENDPOINT/budget") { req, res -> removeBudget(req, res) }
    http.delete( path: "$BASE_ENDPOINT/budgetCategory") { req, res -> removeCategoryBudget(req, res) }
    http.delete( path: "$BASE_ENDPOINT/creditCard") { req, res -> removeCreditCard(req, res) }

    http.get( path: "$BASE_ENDPOINT/balance") { req, res -> getMonthlyBalance(req, res) }
    http.get( path: "$BASE_ENDPOINT/income") { req, res -> getMonthlyIncome(req, res) }
    http.get( path: "$BASE_ENDPOINT/incomeDetails") { req, res -> getMonthlyIncomeDetails(req, res) }
    http.get( path: "$BASE_ENDPOINT/expense") { req, res -> getMonthlyExpense(req, res) }
    http.get( path: "$BASE_ENDPOINT/expenseDetails") { req, res -> getMonthlyExpenseDetails(req, res) }
    http.get( path: "$BASE_ENDPOINT/budget") { req, res -> getTotalBudget(req, res) }
    http.get( path: "$BASE_ENDPOINT/budgetDetails") { req, res -> getBudgetDetails(req, res) }
    http.get( path: "$BASE_ENDPOINT/creditCards") { req, res -> getCreditCards(req, res) }
    http.get( path: "$BASE_ENDPOINT/creditCardDetails") { req, res -> getCreditCardDetails(req, res) }
}
```

Fonte: Elaborada pela autora

Para evitar a exposição de estruturas internas da solução, a comunicação com o mundo externo é feita através do uso de *DTOs*, um padrão de *design* usado para a transferência de dados entre diferentes sistemas ou componentes. Foi priorizado o uso de tipos primitivos, *Strings* e *BigDecimal* na comunicação cliente-servidor, evitando que os clientes precisem lidar com a forma como o servidor estrutura seus dados nas camadas mais internas.

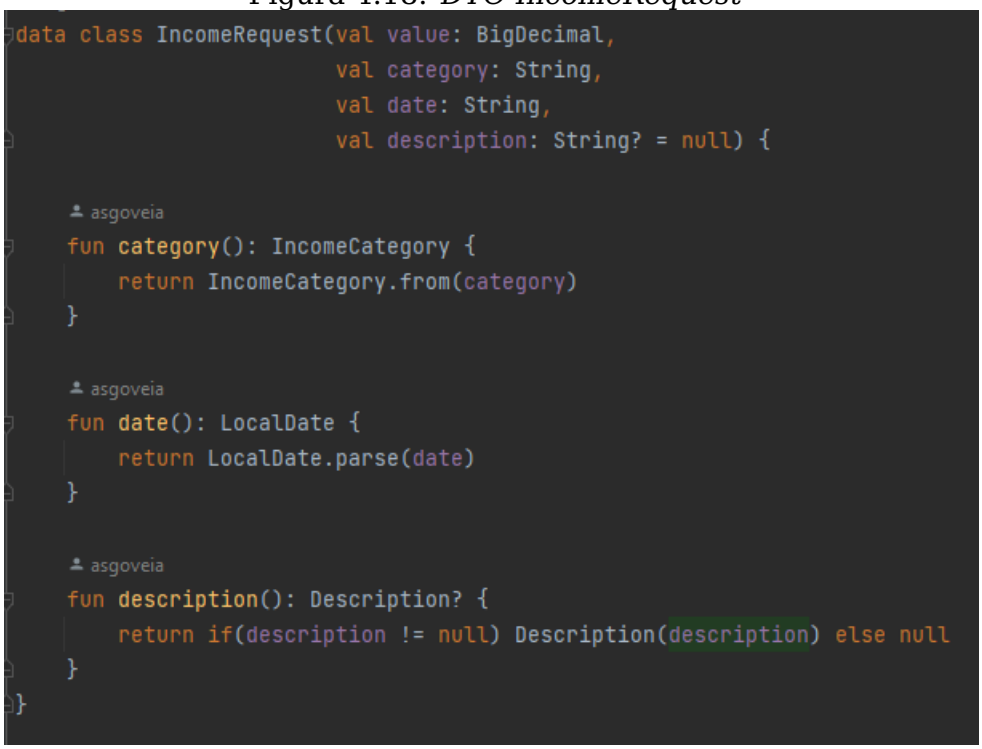
Figura 4.17: Exemplo de utilização de *DTO*



Fonte: Elaborada pela autora

Na Figura 4.17 temos um exemplo de utilização de *DTO* na camada *Rest*. Na linha 307 acontece a desserialização do corpo da requisição na classe *IncomeRequest*, o *DTO* usado na *API* de cadastro de ganhos. Depois disso é criado um objeto *Income* da *Model* da plataforma, o qual é enviado para a *Application* conforme pode ser visto na linha 302.

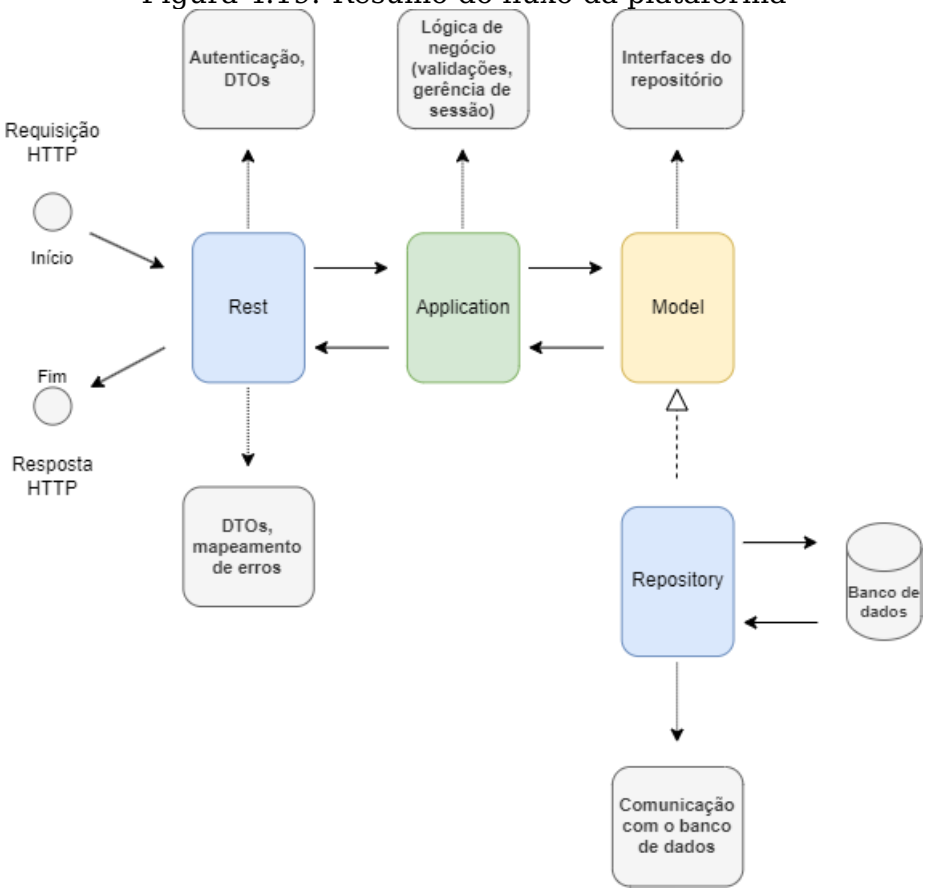
Figura 4.18: *DTO IncomeRequest*



Fonte: Elaborada pela autora

De maneira geral, o resumo do fluxo de uma requisição *HTTP* na plataforma pode ser representado pelo diagrama da Figura 4.19.

Figura 4.19: Resumo do fluxo da plataforma



Fonte: Elaborada pela autora

O primeiro contato da requisição no servidor acontece na camada *Rest*, onde é feita a transformação do *DTO* em objetos da *Model*, assim como a verificação do *token* de autenticação, quando pertinente. A seguir, os dados são passados para a *Application* onde é aplicada lógica de negócios e acessadas as interfaces de repositório. O acesso ao banco de dados é feito na camada *Repository*, a qual possui a implementação das interfaces definidas na *Model*.

Nas subseções a seguir serão abordados cada um dos *endpoints* disponíveis.

4.3.5.1 POST /signup

Endpoint para cadastro de usuários. O corpo esperado na requisição pode ser consultado na Listagem 4.1.

Listing 4.1 – Corpo *Sign Up*

```
1 {
```

```
    "username": <string>,
    "password": <string>,
    "passwordConfirmation": <string>
}
```

Para que o cadastro ocorra com sucesso, o nome de usuário e senha não podem ser vazios ou conter apenas espaços em branco. O nome de usuário não pode já estar sendo usado e a senha e sua confirmação precisam ser iguais. Em caso de sucesso, o usuário e sua senha encriptada são salvos no banco de dados.

As exceções mapeadas para este *endpoint* são as seguintes:

- *UsernameAlreadyExistsException (status code 400)*: nome de usuário já cadastrado;
- *PasswordNotEqualException (status code 400)*: a senha e sua confirmação diferem;
- *EmptyValueException (status code 400)*: o nome de usuário ou senha são vazios ou contém apenas espaços. O campo inválido é identificado na mensagem de erro retornada.

4.3.5.2 POST /login

Usado para fazer o *login* dos usuários. O formato do corpo esperado na requisição está na Listagem 4.2.

Listing 4.2 – Corpo *Login*

```
{
    "username": <string>,
    "password": <string>
}
```

Caso o nome de usuário e senha sejam válidos, é gerado um *token* de autenticação, o qual é retornado na resposta e deverá ser usado para fazer requisições a todos os *endpoints* da plataforma com exceção de *signup* e *login*:

Listing 4.3 – Corpo de resposta do *Login*

```
{
```

```
    "username": <string>,
    "token": <string>
}
```

Considerando o escopo e tempo disponível para a implementação desta solução, optou-se por se fazer o controle de sessão de autenticação em memória. Isto é possível já que há apenas uma instância da plataforma rodando em um dado momento. Para realizar este controle, está sendo usada a *CacheBuilder* (GOOGLE, 2022) da *Guava*, uma biblioteca desenvolvida pelo *Google*. Com ela é criada uma *cache* que armazena o nome de usuário e o *token* de autenticação. O *token* tem validade de 30 minutos, sendo este tempo redefinido toda vez que uma requisição é feita.

Para o *login* há apenas uma exceção mapeada:

- *AuthenticationException (status code 401)*: credenciais inválidas.

4.3.5.3 POST /income

Utilizado para cadastro de ganhos. O corpo da requisição dever ter o formato descrito na Listagem 4.4.

Listing 4.4 – Corpo do cadastro de ganhos

```
{
    "value": <big decimal>,
    "date": <string>,
    "category": <string>,
    "description": <string> (optional)
}
```

Para realizar o cadastro, é preciso que o usuário esteja logado, tenha informado uma categoria válida (categorias presentes na *IncomeCategory* apresentada anteriormente), uma data no formato yyyy-mm-dd e um valor acima de zero.

As exceções mapeadas para este *endpoint* são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *IllegalArgumentException (status code 400)*: categoria inválida;
- *DateTimeException (status code 400)*: a data informada não possui um

formato válido;

- *InvalidValueException (status code 400)*: o valor informado é menor ou igual a zero.

4.3.5.4 POST /expense

API para registro de despesas. O corpo da requisição deve estar no formato presente na Listagem 4.5.

Listing 4.5 – Corpo do cadastro de despesas

```
{
    "value": <big decimal>,
    "date": <string>,
    "category": <string>,
    "description": <string> (optional)
    "creditCardId": <long> (optional)
}
```

As validações realizadas são similares as do *endpoint* de cadastro de ganhos, sendo as exceções mapeadas as seguintes:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *IllegalArgumentException (status code 400)*: categoria inválida;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido;
- *InvalidValueException (status code 400)*: o valor informado é menor ou igual a zero;
- *AccessDeniedCustomException (status code 403)*: o cartão informado não pertence ao usuário.

4.3.5.5 POST /budget

Endpoint para cadastro de orçamentos mensais. Este orçamento é referente ao máximo que se gostaria de gastar em um mês, independente de categoria. É esperado um corpo no formato da Listagem 4.6.

Listing 4.6 – Corpo do cadastro de orçamentos

```
{
    "goal": <big decimal>
    "date": <string>
}
```

Apenas um orçamento pode ser registrado para determinado mês. A data informada na requisição deve estar no formato yyyy-mm-dd e será tratada internamente para atribuição do mês do orçamento. A seguir, as exceções mapeadas:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido (yyyy-mm-dd);
- *InvalidValueException (status code 400)*: o valor informado é menor ou igual a zero;
- *BudgetAlreadyRegisteredException (status code 400)*: já existe um orçamento cadastrado para o mês informado.

4.3.5.6 POST /budgetCategory

API responsável pelo registro de orçamentos para uma dada categoria em determinado mês. Apenas um orçamento pode ser cadastrado para uma categoria por mês. O corpo da requisição deve seguir o formato da Listagem 4.7.

Listing 4.7 – Corpo do cadastro de orçamentos de categorias

```
{
    "goal": <big decimal>,
    "date": <string>,
    "category": <string>
}
```

Assim como no *endpoint* anterior, a data informada na requisição deve estar no formato yyyy-mm-dd e será tratada internamente para atribuição do mês. O valor da categoria informada deve estar presente na *ExpenseCategory*. As exceções são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido (yyyy-mm-dd);
- *InvalidValueException (status code 400)*: o valor informado é menor ou igual a zero;
- *BudgetAlreadyRegisteredException (status code 400)*: já existe um orçamento cadastrado para o mês e categoria informados;
- *IllegalArgumentException (status code 400)*: categoria inválida.

4.3.5.7 POST /creditCard

Utilizado para a inserção de cartões de crédito. O corpo esperado é o da Listagem 4.8:

Listing 4.8 – Corpo do cadastro de cartões de crédito

```
{
    "name": <string>,
    "limit": <big decimal>,
    "dueDay": <int>,
    "closingDay": <int>
}
```

Para o correto cadastro de um cartão de crédito, o nome do cartão informado deve ser único para o usuário, o limite deve ser maior do zero, o dia de fechamento da fatura deve estar entre 1 e 28, o dia de pagamento deve estar entre 1 e 28 e ser maior do que o dia de fechamento da fatura. São estas as exceções mapeada:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *InvalidValueException (status code 400)*: o valor informado é menor ou igual a zero;
- *InvalidDueDayException (status code 400)*: o dia de pagamento não está entre os dias 1 e 28 (inclusive);
- *InvalidClosingDayException (status code 400)*: o dia de fechamento não está entre os dias 1 e 28 (inclusive);
- *IllegalDateException (status code 400)*: o dia de pagamento é menor ou

igual ao dia de fechamento;

- *DuplicatedCreditCardException (status code 400)*: o usuário já possui um cartão de crédito de mesmo nome.

4.3.5.8 DELETE /income?id=<long>

Endpoint para remoção de ganho previamente cadastrado. O identificador (id) do item a ser removido deve ser informado na *URI*. Caso o item que se esteja tentando deletar não tenha sido cadastrado pelo usuário que está fazendo a requisição de remoção ou se o identificador não existir no banco de dados, uma exceção será acionada.

As exceções mapeadas para este *endpoint* são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *AccessDeniedCustomException (status code 403)*: o usuário não tem acesso ao ganho informado;
- *IllegalArgumentException (status code 400)*: o id informado não é um número;
- *IncomeNotFoundException (status code 404)*: o item informado não existe no banco de dados.

4.3.5.9 DELETE /expense?id=<long>

Similar à *API* de exclusão de ganhos, também é possível remover despesas ao usar este *endpoint*. As exceções mapeadas para ele são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *AccessDeniedCustomException (status code 403)*: o usuário não tem acesso a despesa informada;
- *IllegalArgumentException (status code 400)*: o id informado não é um número;
- *ExpenseNotFoundException (status code 400)*: o item informado não existe no banco de dados.

4.3.5.10 DELETE /budget?date=<date>

API usada para remover o orçamento global de um mês. A data deve ser passada como parâmetro na *URI* e deve estar no formato yyyy-mm-dd, sendo tratada internamente para atribuição do mês.

As exceções dessa *API* são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.11 DELETE /budgetCategory?date=<date>&category=<category>

Endpoint para exclusão de orçamentos de uma categoria específica de um determinado mês. Espera-se que sejam informados na *URI* a data e a categoria que se quer remover.

As exceções dessa *API* são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.12 DELETE /creditCard?id=<long>

Endpoint utilizado para a exclusão de cartões de crédito. O identificador do cartão precisa ser passado como parâmetro na *URI*.

Abaixo, as exceções mapeadas:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *AccessDeniedCustomException (status code 403)*: o usuário não tem acesso ao cartão informado;
- *IllegalArgumentException (status code 400)*: o id informado não é um número.

4.3.5.13 GET /balance?date=<date>

API usada para consultar o saldo de um mês, isto é, a diferença entre gastos e despesas. Deve ser informado a data para consulta no formato yyyy-

mm-dd. O corpo da resposta tem o formato presente na Listagem 4.9.

Listing 4.9 – Corpo de resposta da consulta de saldo

```
{
    "balance": <big decimal>
}
```

As exceções mapeadas são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.14 GET /income?date=<date>

API para consulta do total de ganhos de um mês. Deve ser passada como parâmetro na *URI* a data que se quer consultar. O formato da resposta segue o padrão mostrado na Listagem 4.10.

Listing 4.10 – Corpo de resposta do total de ganhos no mês

```
{
    "income": <big decimal>
}
```

As exceções mapeadas são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.15 GET /incomeDetails?date=<date>

Endpoint para consulta dos detalhes dos ganhos de um mês. Através dele se obtém uma listagem contendo informações sobre cada um dos ganhos do mês, assim como o valor total de ganhos. Para fazer essa consulta, é necessário informar a data como parâmetro da requisição. O corpo de resposta tem o formato apresentado na Listagem 4.11.

Listing 4.11 – Corpo de resposta dos detalhes de ganhos

```
{
    "totalIncome": <big decimal>,
    "incomeDetails": [
        {
            "id": <long>,
            "value": <big decimal>,
            "category: <string>,
            "description": <string> (optional),
            "date": <string>
        }
    ]
}
```

As exceções mapeadas para este *endpoint* são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.16 GET /expense?date=<date>

API para consulta do total de despesas de um mês. Deve ser passada como parâmetro na *URI* a data que se quer consultar. Na Listagem 4.12 pode ser consultado o padrão de resposta retornada.

Listing 4.12 – Corpo de resposta do total de despesas no mês

```
{
    "expense": <big decimal>
}
```

As exceções mapeadas são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.17 GET /expenseDetails?date=<date>

Endpoint para consulta dos detalhes das despesas de um mês. Através dele se obtém uma listagem contendo informações sobre cada uma das despesas do mês, assim como o valor total de despesas. Para fazer essa consulta, é necessário informar a data como parâmetro da requisição. O corpo de resposta tem o formato mostrado na Listagem 4.13.

Listing 4.13 – Corpo de resposta dos detalhes de despesas

```
{
    "totalExpense": <big decimal>,
    "expenseDetails": [
        {
            "id": <long>,
            "value": <big decimal>,
            "category: <string>,
            "description": <string> (optional),
            "date": <string>,
            "creditCardId": <long> (optional)
        }
    ]
}
```

As exceções mapeadas para este *endpoint* são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.18 GET /budget?date=<date>

API para consulta das informações do orçamento de um mês. Através dele são obtidos o valor total do que se quer gastar no mês, o quanto já foi gasto, o quanto resta para ser gasto naquele mês, a porcentagem do orçamento já usada e a data do orçamento:

Listing 4.14 – Corpo de resposta de orçamentos

```
{
```

```
    "goal": <big decimal>,
    "spent": <big decimal>,
    "remaining": <big decimal>,
    "percentageUsed": <big decimal>,
    "date": <string>
}
```

As exceções possíveis são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.19 GET /budgetDetails?date=<date>

API para consulta dos detalhes de um orçamento, incluindo os orçamentos cadastrados especificamente para categorias. O corpo de resposta tem a formatação presente na Listagem 4.15.

Listing 4.15 – Corpo de resposta dos detalhes de orçamentos

```
{
    "total": {
        "goal": <big decimal>,
        "spent": <big decimal>,
        "remaining": <big decimal>,
        "percentageUsed": <big decimal>,
        "date": <string>
    },
    "categories": [
        {
            "category": <string>,
            "goal": <big decimal>,
            "spent": <big decimal>,
            "remaining": <big decimal>,
            "percentageUsed": <big decimal>,
            "date": <string>
```

```
        }
    ]
}
```

As exceções mapeadas são:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido.

4.3.5.20 GET /creditCards

Endpoint para consultar todos os cartões de crédito de um usuário. A resposta retornada tem a formatação da Listagem 4.16.

Listing 4.16 – Corpo de resposta de cartões de crédito

```
{
    "creditCards": [
        {
            "name": <string>,
            "limit": <big decimal>,
            "closingDay": <int>,
            "dueDay": <int>
        }
    ]
}
```

É retornada exceção de autenticação caso o usuário não esteja logado:

- *AuthenticationException (status code 401)*: credenciais inválidas.

4.3.5.21 GET /creditCardDetails?id=<long>&date=<date>

API para consulta aos detalhes de um cartão de crédito. Além das informações do cartão, retorna também todos os gastos que foram realizados nele durante o mês requisitado. É necessário passar o identificador do cartão e o mês que se quer consultar na *URI*. A resposta tem o formato da Listagem 4.17.

Listing 4.17 – Corpo de resposta dos detalhes de um cartão de crédito

```
{
    "totalExpense":<big decimal>,
    "creditCard": {
        "id": <long>,
        "name": <sting>,
        "limit": <big decimal>,
        "closingDay": <int>,
        "dueDay": <int>
    },
    "expenseDetails": [
        {
            "id": <long>,
            "value": <big decimal>,
            "category": <string>,
            "description": <string>, (optional)
            "date": <string>,
            "creditCardId": <long>
        }
    ]
}
```

As exceções mapeadas para este *endpoint* são as seguintes:

- *AuthenticationException (status code 401)*: credenciais inválidas;
- *DateTimeException (status code 400)*: a data informada não possui um formato válido;
- *AccessDeniedCustomException (status code 403)*: o usuário não tem acesso ao cartão de crédito informado;
- *CreditCardNotFoundException (status code 400)*: não há nenhum cartão de crédito com o id fornecido.

4.4 Frontend

Para este trabalho foi criado um cliente em React, o qual se comunica através de *REST APIs* com o servidor descrito na seção anterior. Do lado do cliente, as chamadas *HTTP* são feitas com o uso do *Axios* (AXIOS, 2022), uma biblioteca cujo propósito é fazer requisições para um recurso externo à aplicação.

Para auxilar nesta comunicação, foi usada uma funcionalidade disponível a partir da versão 16.8 do React: **hooks** (REACT, 2022a). *Hooks* são funções que podem ser usadas para isolar parte reutilizável da lógica de estado entre componentes. Em React, *estado* nada mais é do que o objeto onde os valores das propriedades pertecentes a um componente são armazenados. Além de guardar estado, os *hooks* também podem gerenciar efeitos colaterais. O próprio React disponibiliza alguns *hooks* padrões, como por exemplo *useState* e *useEffect*.

O *hook useState*, como o próprio nome indica, é usado para gerência de estados. Ele retorna um valor e uma função para atualizá-lo. Nas Figuras 4.20 e 4.21, um exemplo de utilização deste *hook* neste projeto:

Figura 4.20: Exemplo de declaração de *useState*

```
const [user, setUser] = useState( initialState: '');
```

Fonte: Elaborada pela autora

Figura 4.21: Exemplo de uso de *useState*

```
59          <form onSubmit={handleSubmit}>
60              <label htmlFor="username">Username</label>
61              <input type="text"
62                     id="user"
63                     ref={userRef}
64                     autoComplete="off"
65                     onChange={(e ...) => setUser(e.target.value)}
66                     value={user}
67                     required
68              />
```

Fonte: Elaborada pela autora

Na imagem 4.20 é possível observar a declaração do *useState* usado para se obter o usuário na tela de *login*. O valor inicial dele é vazio. Já na figura 4.21, há o uso deste *hook* na linha 65. Quando a pessoa que estiver

usando a plataforma preencher seu nome de usuário no formulário de *login*, o *useState* é responsável por atualizar este valor através da *setUser*.

Já o *useEffect* é responsável por gerenciar efeitos colaterais, como chamadas de *API*. Além dos disponibilizados por padrão, também é possível que o programador crie seus próprios *hooks*. A seguir, o *hook* *useFetchDetails*, o qual foi criado para realizar o *GET* dos dados das *APIs* de detalhes de ganhos (subseção 4.3.5.15), detalhes de despesas (subseção 4.3.5.17), entre outras:

Figura 4.22: *Hook useFetchDetails*

```
5   const useFetchDetails = (url, date) => {
6       const month = (date.getMonth() + 1).toString().padStart(2, "0");
7       const day = "01"
8       const year = date.getUTCFullYear();
9       const dateToSend = year + "-" + month + "-" + day
10
11      const [data, setData] = useState( initialState: "");
12      const [error, setError] = useState( initialState: "");
13
14      const headers = {
15          mode: 'no-cors',
16          withCredentials: true,
17          token: Cookies.get("token"), "Access-Control-Allow-Origin": "http://localhost:3000",
18          "Access-Control-Allow-Methods": "*", "Access-Control-Allow-Credentials": "true"
19      };
20
21      const refreshDetails = async () => {
22          try {
23              const response = await axios.get(
24                  url: url + dateToSend,
25                  headers
26              );
27              setData(response.data);
28          } catch (error) {
29              setError(error.message);
30          }
31      }
32
33      useEffect( effect: () => {
34          refreshDetails()
35      }, deps: [dateToSend]);
36
37      return { data, error, refreshDetails };
38  };
39
40  export default useFetchDetails
```

Fonte: Elaborada pela autora

Este *hook* recebe como parâmetro a *URL* a ser chamada e a data que se quer consultar. Das linhas 6 à 9 é feita a tranformação do objeto *Date* recebido para uma *string* no formato yyyy-mm-dd, o formato esperado pela *API* do *backend*. Das linhas 21 à 31 é declarada a função que faz a chamada para o servidor, usando o *Axios*. Se a requisição for um sucesso, os dados são atribuídos a variável *data*, caso contrário a mensagem de erro retornada

é atribuída a variável *error*. Por fim, da linha 33 à 35 é usado um *useEffect* com a *dateToSend* em seu *array* de dependências. Desta forma toda vez em que o valor de *dateToSend* for atualizado, uma nova chamada ao servidor será realizada para buscar os dados referentes à nova data.

Já para a renderização das telas foram usados **componentes**. Componentes são partes de códigos independentes e reutilizáveis que retornam *HTML*. Na Figura 4.23, o componente *MonthPicker*, usado em diversas páginas desta plataforma:

Figura 4.23: Componente *MonthPicker*

```
import DatePicker from "react-datepicker";
import "react-datepicker/dist/react-datepicker.css";

export const MonthPicker = (props) => {
    return (
        <DatePicker
            selected={props.date}
            onChange={(date) => props.setDate(date)}
            dateFormat="yyyy-MM"
            showMonthYearPicker
        />
    );
}
```

Fonte: Elaborada pela autora

Este componente é o responsável por renderizar a seleção do mês nas telas da plataforma. Na Figura 4.24, um pedaço de código onde o *MonthPicker* é utilizado na construção da página de despesas:

Figura 4.24: Uso do *MonthPicker* na tela de despesas

```
<div style={{backgroundColor:'#ffffff', border: '1px solid', borderRadius: '5px'}}>
    <MonthPicker date={date} setDate={setDate} />
```

Fonte: Elaborada pela autora

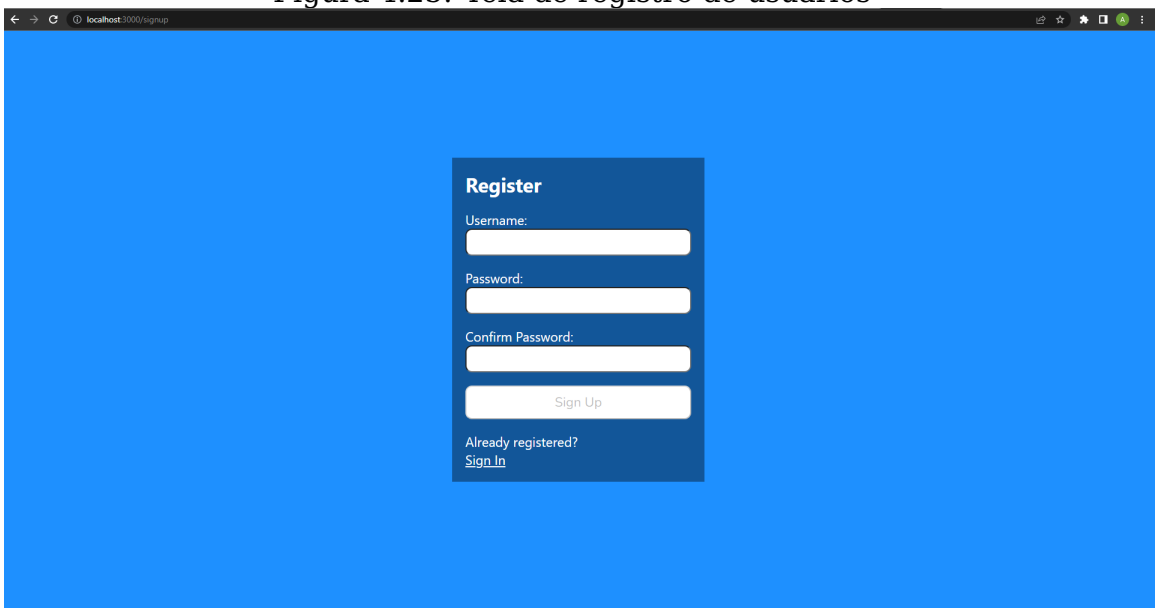
Nas próximas subseções deste capítulo serão mostradas as telas desenvolvidas para esta solução.

4.4.1 Registro de usuários

A tela de registro de usuários foi a primeira tela criada para o cliente desta plataforma. Como até então a autora não possuia experiência com

desenvolvimento React, esta e a tela de *login* foram criadas seguindo um tutorial disponibilizado por Dave Gray (GRAY, 2022) e fazendo os ajustes necessários para o correto funcionamento delas com o servidor da solução. Desta maneira foi possível que a autora entendesse alguns conceitos e funcionalidades de React para que os empregasse na construção das outras páginas da plataforma.

Figura 4.25: Tela de registro de usuários



Fonte: Elaborada pela autora

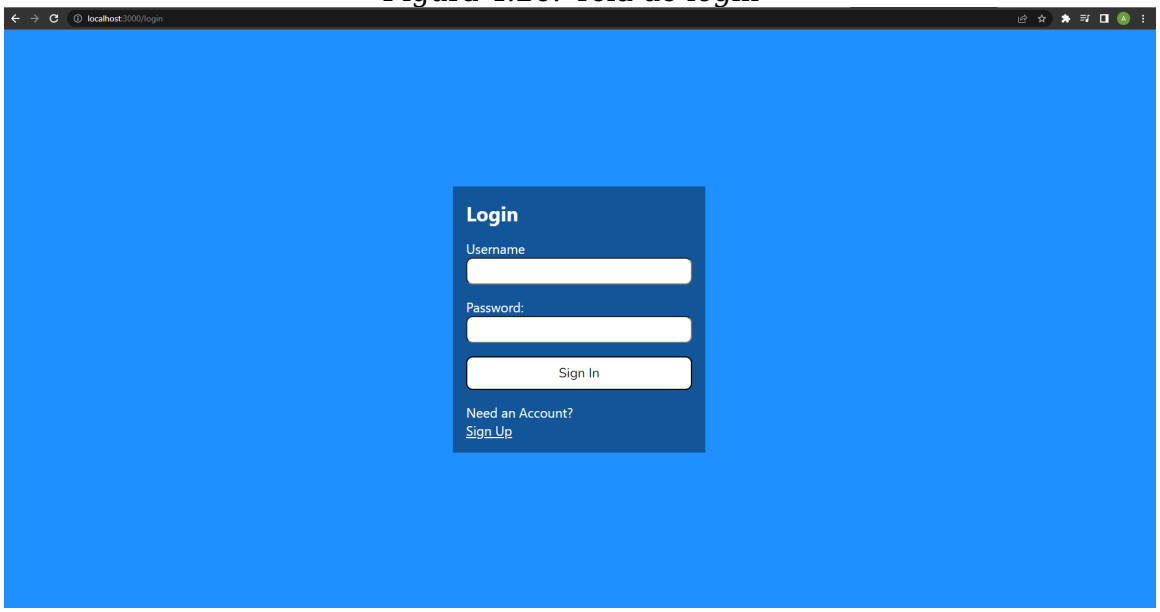
A Figura 4.25 mostra o resultado da implementação da tela de registro de usuários. Nela o usuário preenche seu nome de usuário, a senha que gostaria de usar e a confirmação desta. Ao clicar no botão *submit*, uma requisição é feita para o *endpoint* descrito na seção *4.3.5.1 POST /signup*. Em caso de sucesso no cadastro, o usuário é automaticamente redirecionado para a tela de *login*.

4.4.2 Login

A segunda página a ser implementada foi a de *login*. Como pode ser visto na Figura 4.26, nela o usuário deve preencher seu nome de usuário e senha previamente cadastrados. Ao clicar em *sign in*, uma requisição para a *API* descrita na seção *4.3.5.2 POST /login* é realizada. Em caso de sucesso, o *token* de autenticação é retornado na resposta do *backend* e é então salvo

nos *cookies* pelo cliente para ser usado nas próximas requisições.

Figura 4.26: Tela de *login*

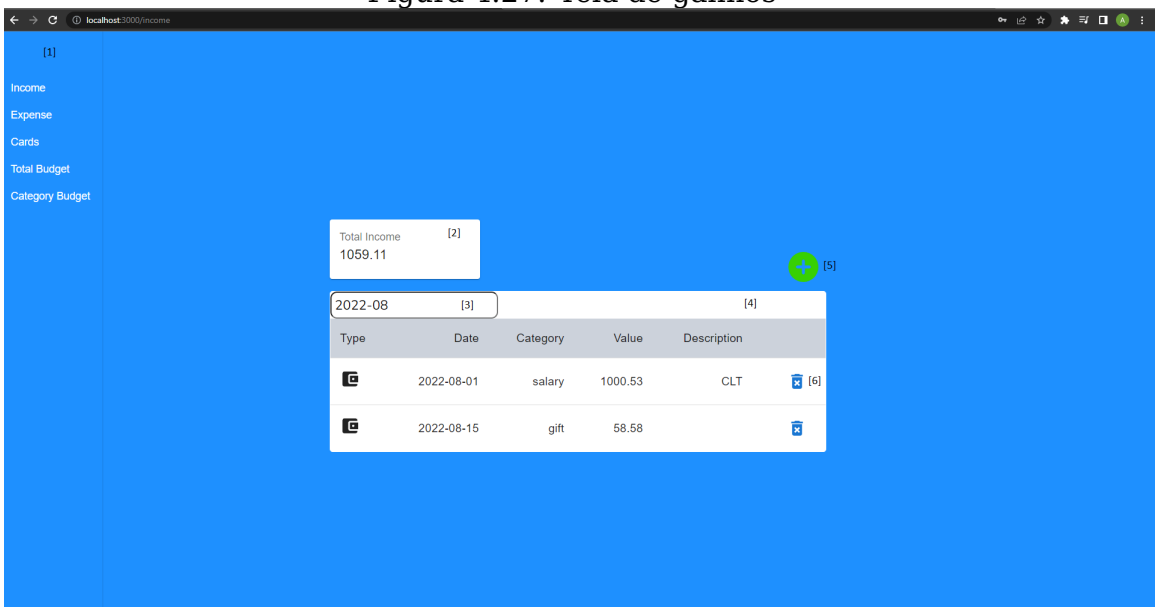


Fonte: Elaborada pela autora

4.4.3 Ganhos

Após logar na plataforma com sucesso, o usário é redirecionado para a tela de ganhos:

Figura 4.27: Tela de ganhos



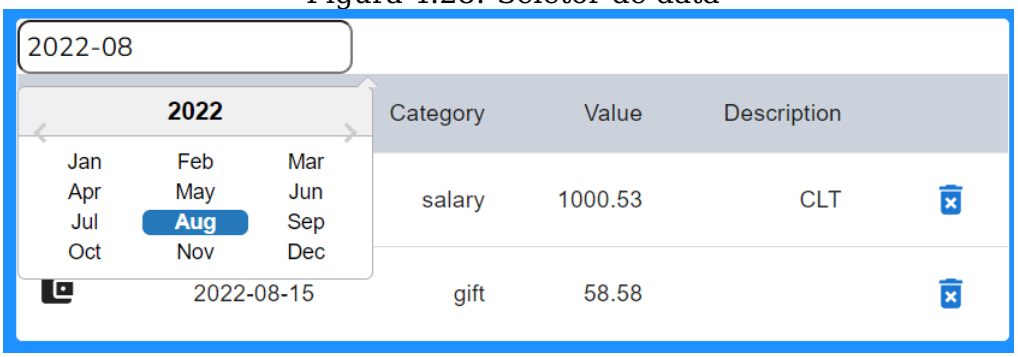
Fonte: Elaborada pela autora

A esquerda (Figura 4.27 [1]) há um menu para navegação das páginas,

o qual está presente em todas as telas com exceção das de registro de usuários e *login*.

Na página de ganhos propriamente dita, há a informação do total de ganhos (Figura 4.27 [2]) do usuário para o mês selecionado. Essa informação é obtida através de uma chamada para o *endpoint* mencionado na seção *4.3.5.14 GET /income?date=<date>*. Através do seletor de mês (Figura 4.27 [3]) é feita a troca do mês de visualização dos ganhos. Ao clicar em [3], um calendário é aberto para a seleção do mês, como pode ser visto na Figura 4.28:

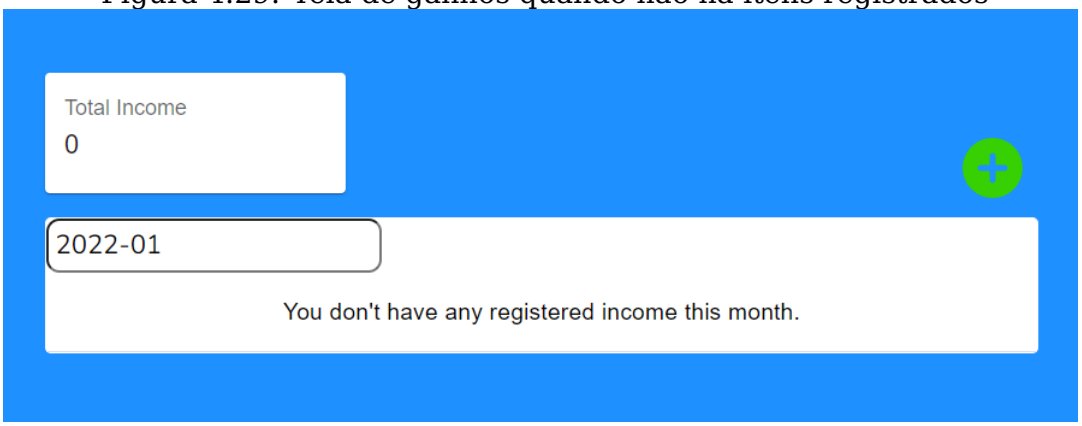
Figura 4.28: Seletor de data



Fonte: Elaborada pela autora

Já na tabela (Figura 4.27 [4]) estão presentes todos os ganhos registrado no mês selecionado. Os ganhos listados são buscados através de uma chamada para a *API* da seção *4.3.5.15 GET /incomeDetails?date=<date>*. Esta *API* também retorna o valor total de ganhos do mês, de forma que não seria preciso realizar a chamada ao *endpoint* da seção *4.3.5.14* nesta página. Entretanto, como a *dashboard* que estava planejada inicialmente para ser implementada e que usuaria o *endpoint* em questão acabou não sendo desenvolvida por questões de tempo, optou-se por realizar esta chamada a mais para que o correto funcionamento do *endpoint* pudesse ser averiguado do lado do cliente. Caso não haja nenhum ganho registrado para o mês selecionado, uma mensagem é mostrada no lugar dos itens, conforme Figura 4.29:

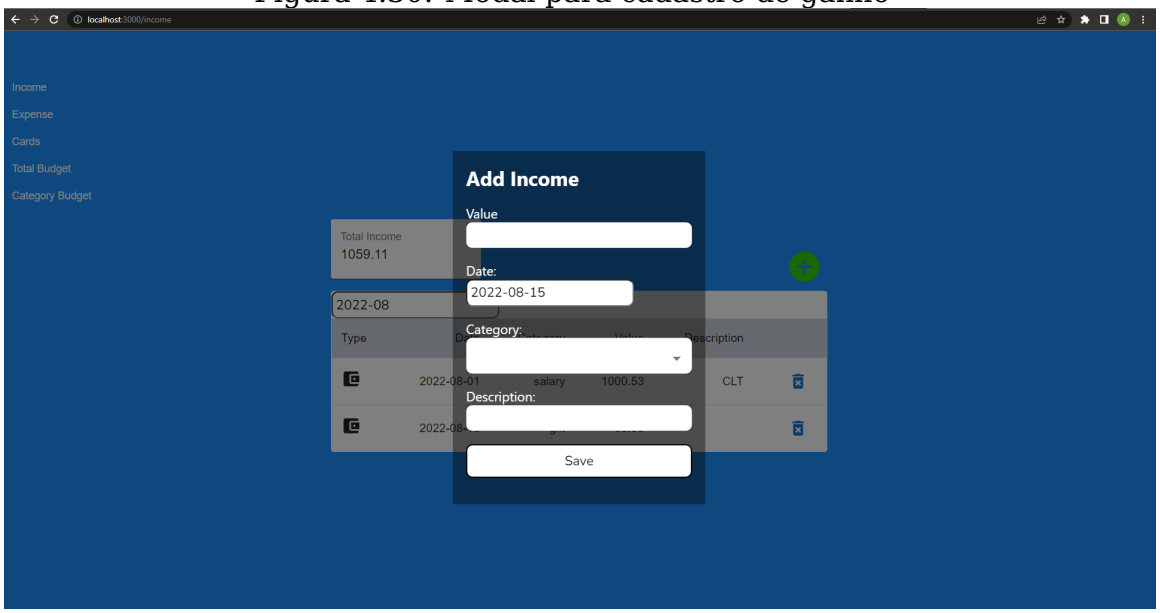
Figura 4.29: Tela de ganhos quando não há itens registrados



Fonte: Elaborada pela autora

Ao clicar no botão de adição (Figura 4.27 [5]), uma modal é aberta para que se possa fazer o registro de um novo ganho.

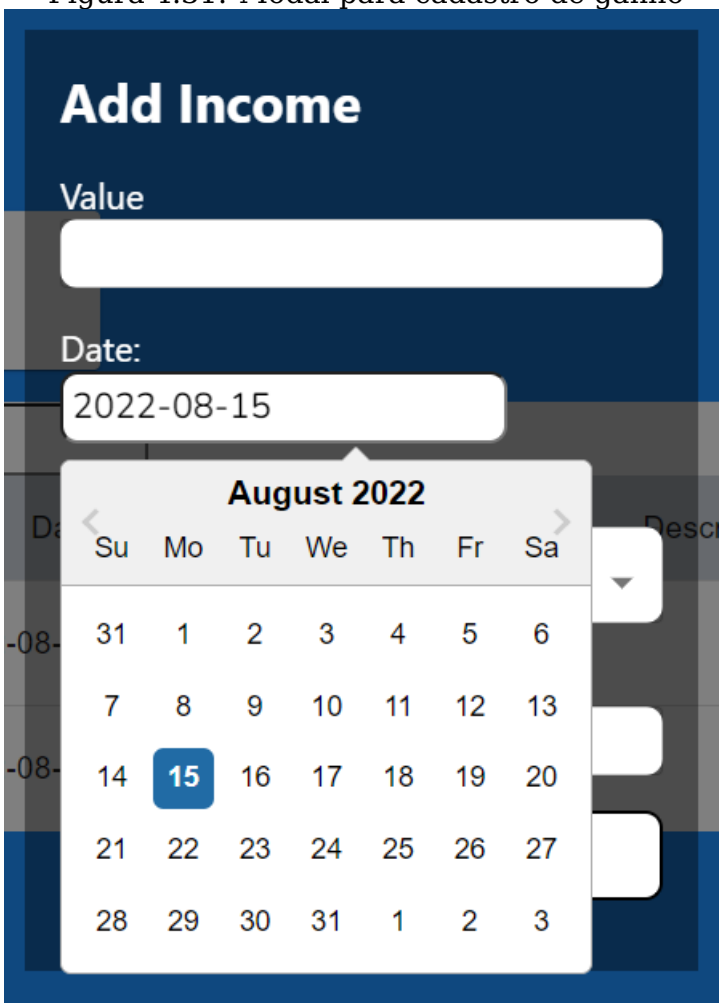
Figura 4.30: Modal para cadastro de ganho



Fonte: Elaborada pela autora

O campo *value* aceita valores numéricos. Ao clicar no campo *date*, um seletor de data é aberto para que se possa selecionar o dia em que o ganho foi recebido. Este seletor pode ser visualizado na Figura 4.31.

Figura 4.31: Modal para cadastro de ganho



Fonte: Elaborada pela autora

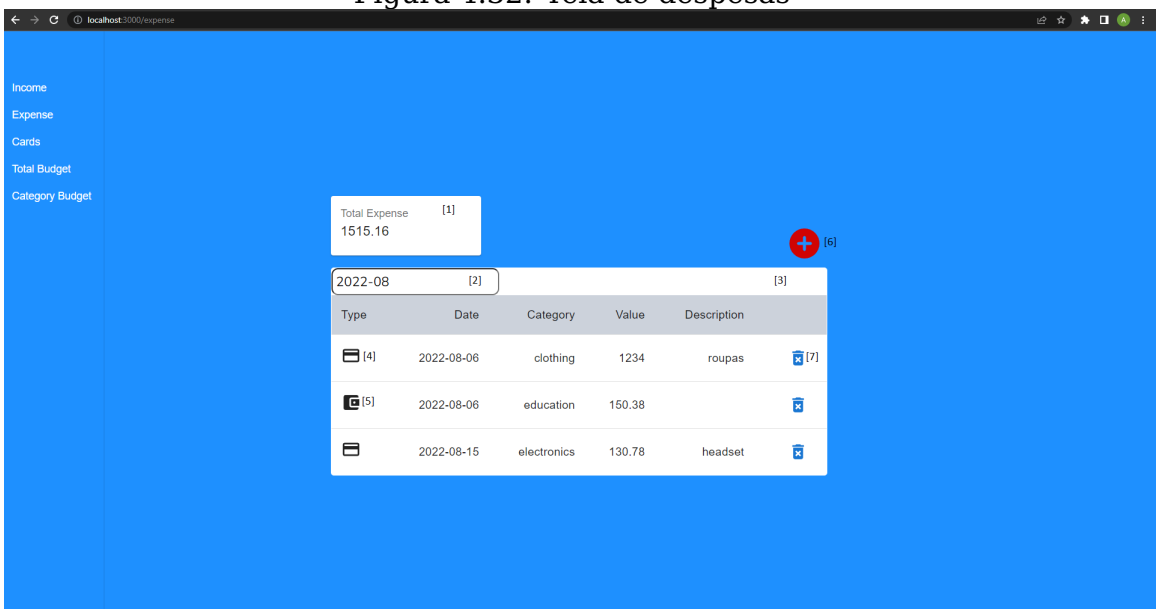
O campo *categoria* é um *dropdown* com as categorias de ganhos disponíveis no *backend* e, por fim, *description* é um campo de texto livre. Ao clicar em *save*, uma requisição para a *API* da seção *4.3.5.3 POST /income* é feita.

Por último, ao clicar no ícone de lixeira (Figura 4.27 [6]) uma requisição ao *endpoint* da seção *4.3.5.8 DELETE /income?id=<long>* é executada para excluir o item daquela linha da tabela.

4.4.4 Despesas

A tela de despesas tem uma estrutura similar à tela de ganhos:

Figura 4.32: Tela de despesas



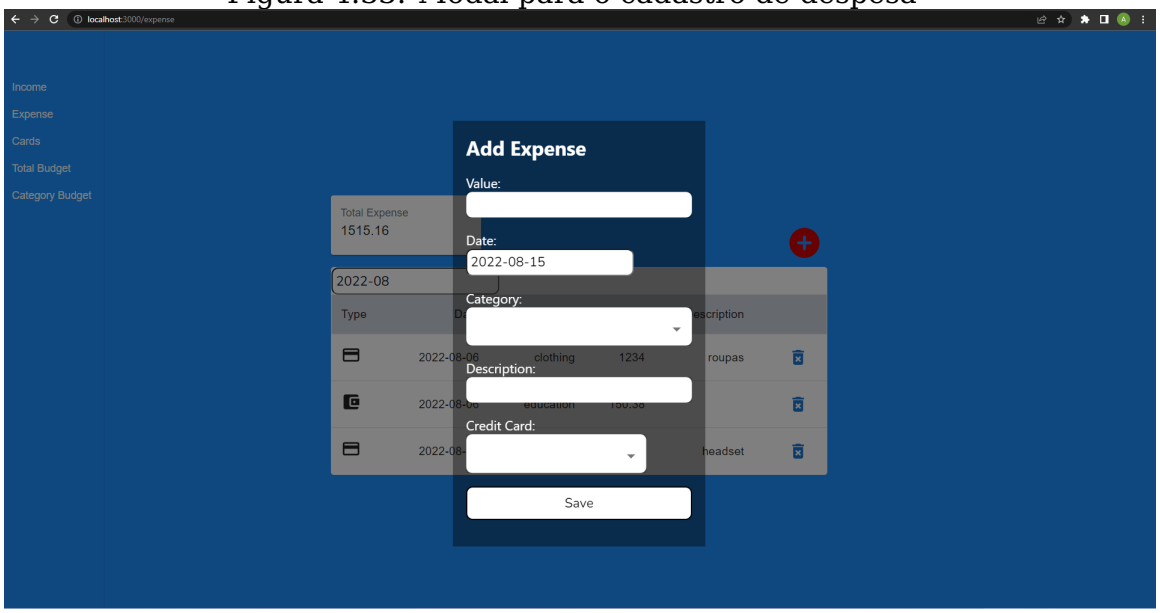
Fonte: Elaborada pela autora

É feita uma chamada à *API* descrita na seção *4.3.5.16 GET /expense?date=<date>* para se buscar o valor total de despesas do mês mostrado na Figura 4.32 [1]. O seletor de mês (Figura 4.32 [2]) é onde o mês das despesas que devem ser mostradas é escolhido, de forma parecida ao que acontece na página de ganhos.

Na tabela (Figura 4.32 [3]) são listados todas as despesas do mês selecionado. Estas informações são obtidas através de uma chamada para o *endpoint 4.3.5.17 GET /expenseDetails?date=<date>*. Os ícones da coluna *type* indicam o tipo de despesa: caso seja uma despesa de cartão de crédito, será mostrado o ícone [4] da Figura 4.32. Já se for uma despesa em dinheiro, será mostrado o ícone [5] da Figura 4.32.

Ao clicar no ícone de adição (Figura 4.32 [6]), uma modal será aberta para o cadastro de uma nova despesa:

Figura 4.33: Modal para o cadastro de despesa



Fonte: Elaborada pela autora

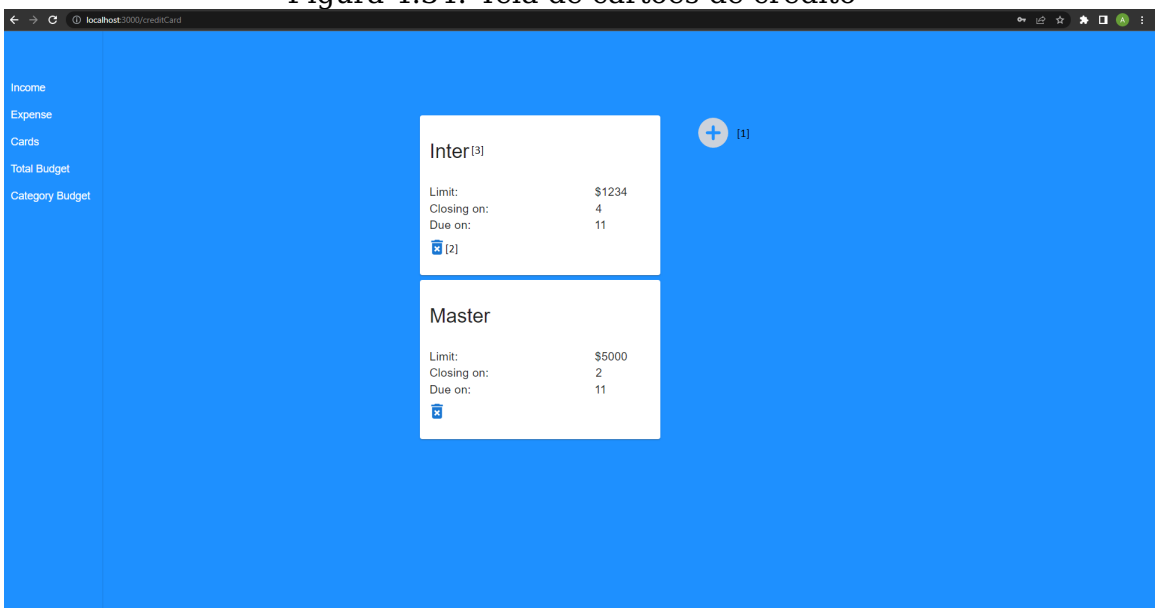
O campo *value* aceita valores numéricos, o campo *date* é um seletor de data semelhante ao da página de ganhos, o campo *category* é um *dropdown* com a lista de categorias de despesa disponíveis no servidor, o campo *description* é de texto livre e o campo *credit card* é um *dropdown* onde serão mostrados os cartões de crédito do usuário, caso ele os tenha cadastrado. Para registrar uma despesa feita em dinheiro, basta deixar o campo *credit card* vazio. Ao clicar em *save*, uma requisição a *API 4.3.5.4 POST /expense* é realizada.

Por fim, é possível excluir uma despesa ao clicar no ícone da lixeira (Figura 4.32 [7]). Ao fazer isto, uma chamada ao *endpoint 4.3.5.9 DELETE /expense?id=<long>* é executada.

4.4.5 Cartões de crédito

Ao selecionar a página *cards* no menu, o usuário é direcionado para a página de cartões de crédito, a qual mostra todos os cartões cadastrados por ele, conforme Figura 4.34.

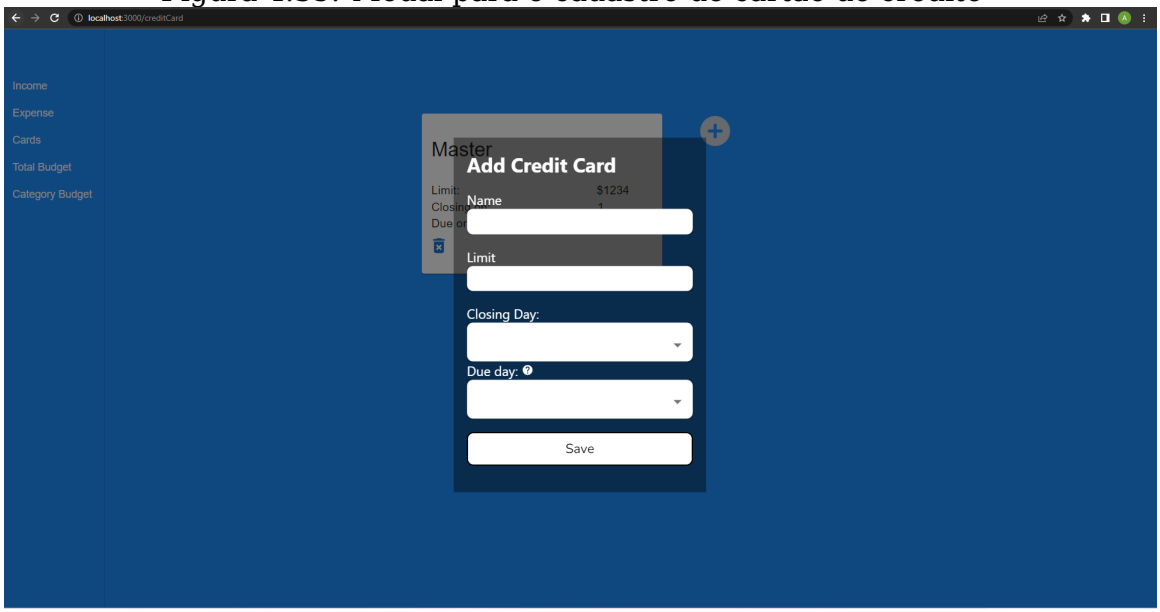
Figura 4.34: Tela de cartões de crédito



Fonte: Elaborada pela autora

Para a busca das informações dos cartões o *endpoint 4.3.5.20 GET /creditCards* é usado. Ao clicar no botão de cadastro (Figura 4.34 [1]), uma modal para adição de novo cartão é aberta.

Figura 4.35: Modal para o cadastro de cartão de crédito



Fonte: Elaborada pela autora

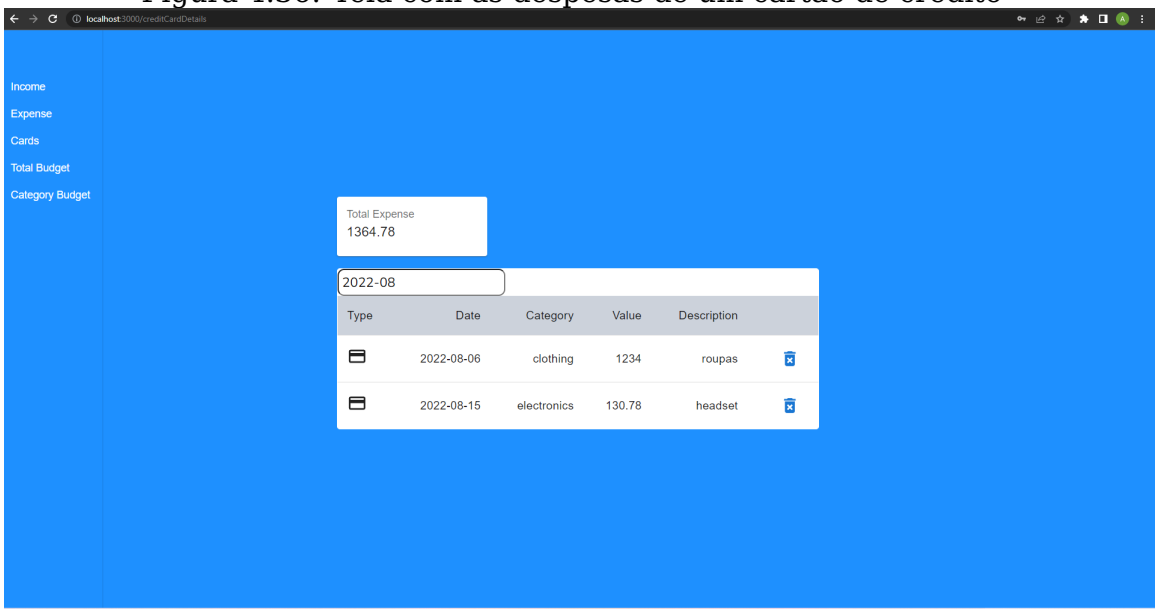
Nela o usuário deve preencher o nome do cartão no campo de texto livre *name*, o limite no campo numérico *limit* e deve selecionar o dia de fechamento e de pagamento da fatura nos *dropdowns closing day* e *due day*, respectivamente. Ao passar o mouse sobre o ícone de interrogação ao lado do

campo *due day*, uma *tooltip* com a mensagem *"due day must be greater than closing day"* é mostrada. Ao clicar em *save*, uma requisição para o *endpoint 4.3.5.7 POST /creditCard* é efetuada para que o cartão seja cadastrado.

Para remoção de itens, basta clicar no ícone da lixeira (Figura 4.34 [2]) do cartão que se deseja excluir. Desta forma, uma chamada a *API 4.3.5.12 DELETE /creditCard?id=<long>* é feita para que o cartão seja excluído.

Através desta página também é possível consultar quais despesas foram realizadas com cada cartão de crédito. Para isto, basta clicar no nome do cartão (Figura 4.34 [3]) que se deseja consultar para ser redirecionado a uma nova tela com estas informações, como pode ser visto na Figura 4.36.

Figura 4.36: Tela com as despesas de um cartão de crédito



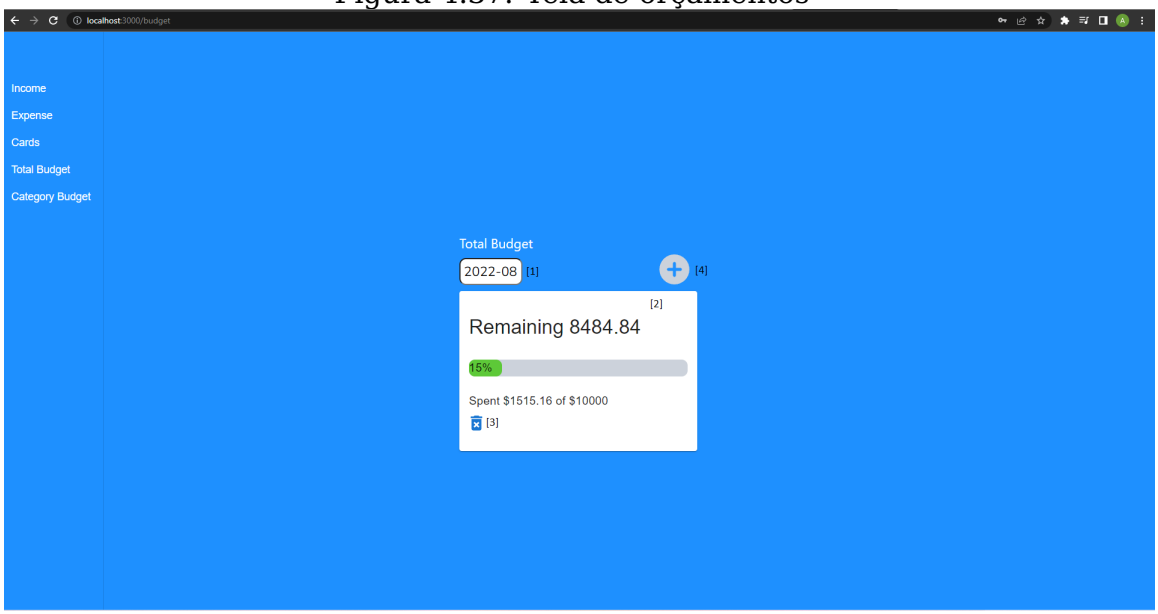
Fonte: Elaborada pela autora

Para mostrar estes dados, a *API 4.3.5.21 GET /creditCardDetails?id=<long>&date=<date>* é usada.

4.4.6 Orçamento total

Ao clicar em *total budget*, o usuário é redirecionado para a página de orçamentos mensais.

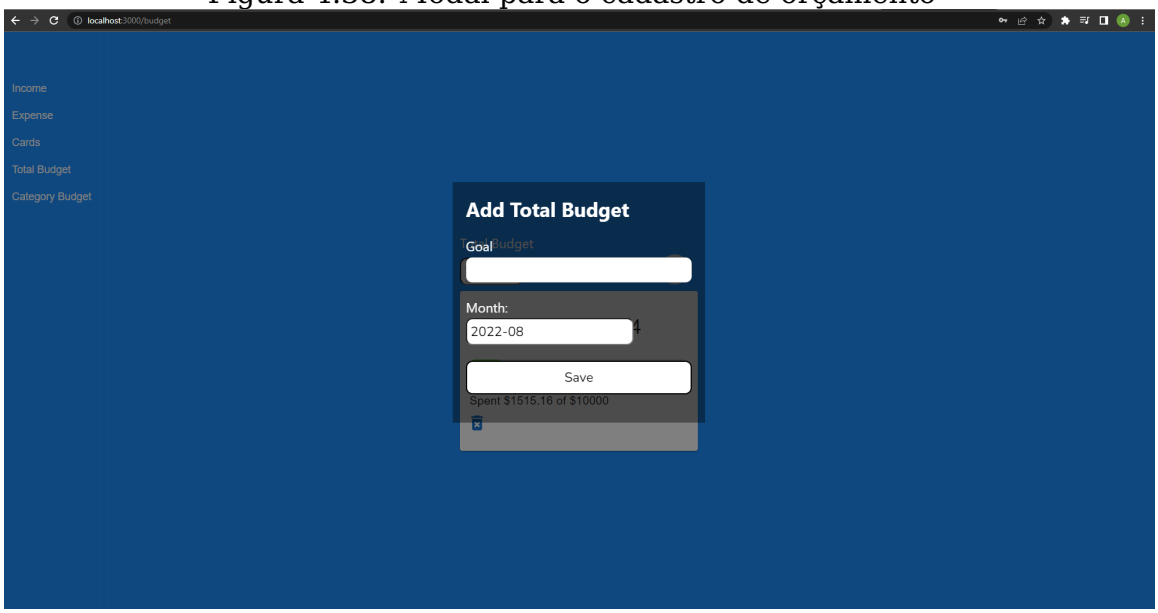
Figura 4.37: Tela de orçamentos



Fonte: Elaborada pela autora

Nesta página é possível fazer a seleção do mês para o qual se quer consultar o orçamento através do seletor de mês (Figura 4.37 [1]). Os dados do orçamento mostrados em [2] na Figura 4.37 são obtidos através da *API* descrita na seção *4.3.5.18 GET /budget?date=<date>*. Na interface se pode consultar qual o valor máximo de despesas se quer realizar no mês, o quanto já foi gasto e o quanto há sobrando, além de uma barra indicando qual a porcentagem já usada do objetivo de gastos do mês. Ao clicar no ícone da lixeira (Figura 4.37 [3]), uma chamada à *API 4.3.5.10 DELETE /budget?date=<date>* é executada para que o orçamento seja excluído. A adição de um novo orçamento pode ser efetuada ao se clicar no botão de adição (Figura 4.37 [4]). Ao fazer isto, a modal mostrada na Figura 4.38 é aberta.

Figura 4.38: Modal para o cadastro de orçamento



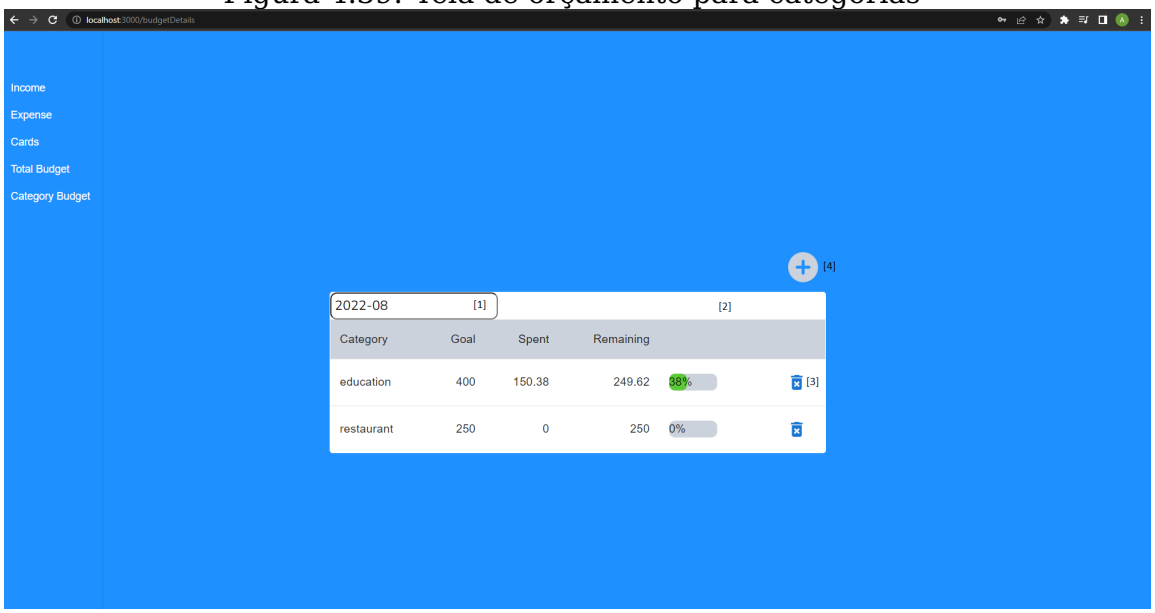
Fonte: Elaborada pela autora

Nesta modal deve ser selecionado o mês para o qual se deseja cadastrar o orçamento e, no campo númerico *goal*, deve ser inserido o valor máximo que se deseja gastar. Ao clicar em *save*, uma chamada à *API 4.3.5.5 POST /budget* é feita para que o novo orçamento seja registrado.

4.4.7 Orçamento de categorias

Por último, há a página de orçamentos específicos para categorias.

Figura 4.39: Tela de orçamento para categorias

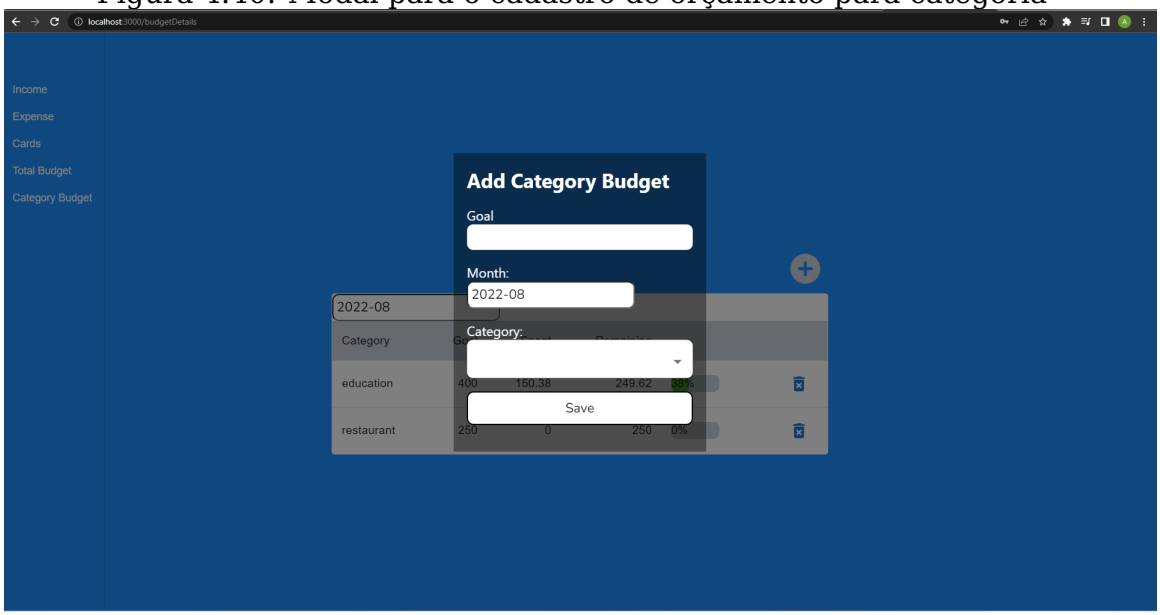


Fonte: Elaborada pela autora

Nesta tela, novamente o mês que se quer consultar é escolhido através do seletor de mês (Figura 4.39 [1]). Na tabela (Figura 4.39 [2]) são mostrados todos os orçamentos para categorias que o usuário possui, os quais são obtidos pelo *endpoint* descrito em *4.3.5.19 GET /budgetDetails?date=<date>*. Para exclusão de um orçamento de categoria, basta clicar no ícone da lixeira (Figura 4.39 [3]) para que uma chamada à *API 4.3.5.11 DELETE /budgetCategory?date=<date>&category=<category>* seja feita.

Para o cadastro de orçamento, é preciso clicar no ícone de adição (Figura 4.39 [4]) para que a modal de cadastro seja aberta.

Figura 4.40: Modal para o cadastro de orçamento para categoria



Fonte: Elaborada pela autora

Esta modal tem um campo númerico *goal* que indica o valor máximo que se quer gastar no mês, um seletor de mês e um *dropdown* com a lista de categorias de despesas presentes no servido da plataforma. Após preencher todos os campos, o usuário deve clicar em *save* para que seja executada uma requisição ao *endpoint 4.3.5.6 POST /budgetCategory* e, então, o novo orçamento seja salvo.

5 AVALIAÇÃO COM USUÁRIOS

Neste capítulo será apresentado o teste realizado com usuários para a avaliação da solução desenvolvida neste trabalho. O teste aplicado é composto por tarefas pré-definidas a serem realizadas na plataforma criada, com o objetivo de avaliar as funcionalidades disponíveis. O formulário de testes pode ser consultado no Apêncide A.

5.1 Ambiente do experimento

Os testes foram aplicados através de chamadas de vídeo entre a autora e cada um dos usuários. Estas chamadas foram realizadas na plataforma de videoconferência *Zoom* (ZOOM, 2022), a qual disponibiliza a opção de um dos participantes da sessão fornecer o controle remoto de seu computador a outrem. Assim sendo, a autora permitiu que os usuários tivessem acesso a seu computador, onde a plataforma deste projeto está instalada, para que o teste fosse realizado.

5.2 Protocolo de testes

O teste aplicado é composto por três etapas:

- **Formulário pré-teste:** perguntas de caracterização dos participantes, como idade, gênero e grau de escolaridade;
- **Utilização da plataforma:** foi fornecida a seguinte sequência de tarefas a serem realizadas dentro da plataforma desenvolvida:
    1. Realizar cadastro de usuário;
    2. Realizar login;
    3. Registrar um ganho no mês atual;
    4. Registrar um ganho em algum outro mês de sua escolha;
    5. Deletar um ganho cadastrado;
    6. Registrar dois cartões de crédito;
    7. Deletar um cartão de crédito;

8. Registrar uma despesa;
9. Registrar uma despesa de cartão de crédito;
10. Consultar as despesas de cartão de crédito através da tela "Cards";
11. Registrar um orçamento total para o mês atual;
12. Registrar um orçamento total para outro mês de sua escolha;
13. Deletar um orçamento cadastrado;
14. Registrar um orçamento para uma categoria de sua escolha;
15. Deletar o orçamento de uma categoria.

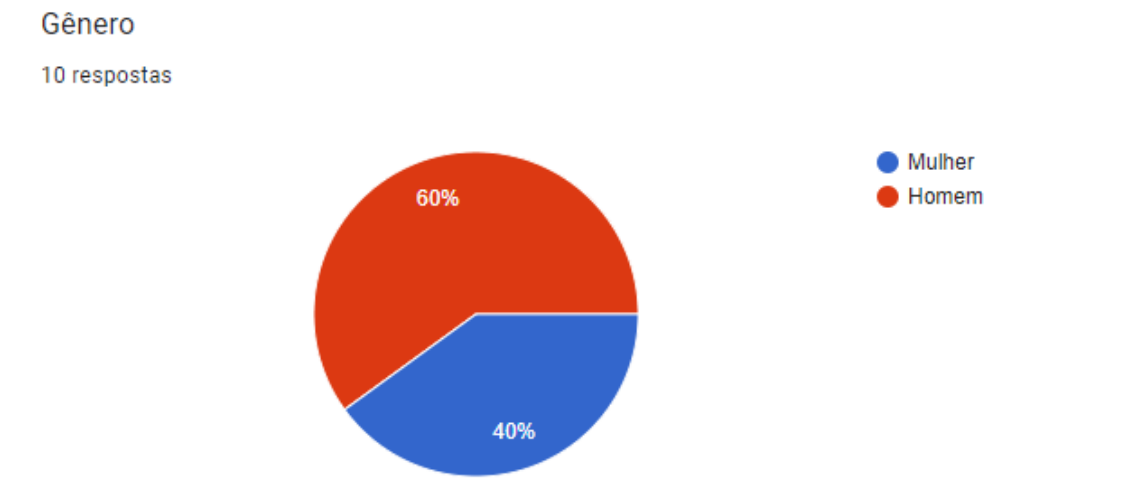
- **Formulário pós-etapa:** coleta dos dados com relação a utilização da plataforma.

A avaliação foi feita através da dificuldade encontrada para realizar as tarefas propostas, variando entre "não consegui realizar", "consegui realizar com muita dificuldade", "consegui realizar com alguma dificuldade" e "consegui realizar". Além disso, foi disponibilizado um campo livre para que os usuários pudessem deixar quaisquer sugestões que desejassem.

5.3 Perfil dos Usuários

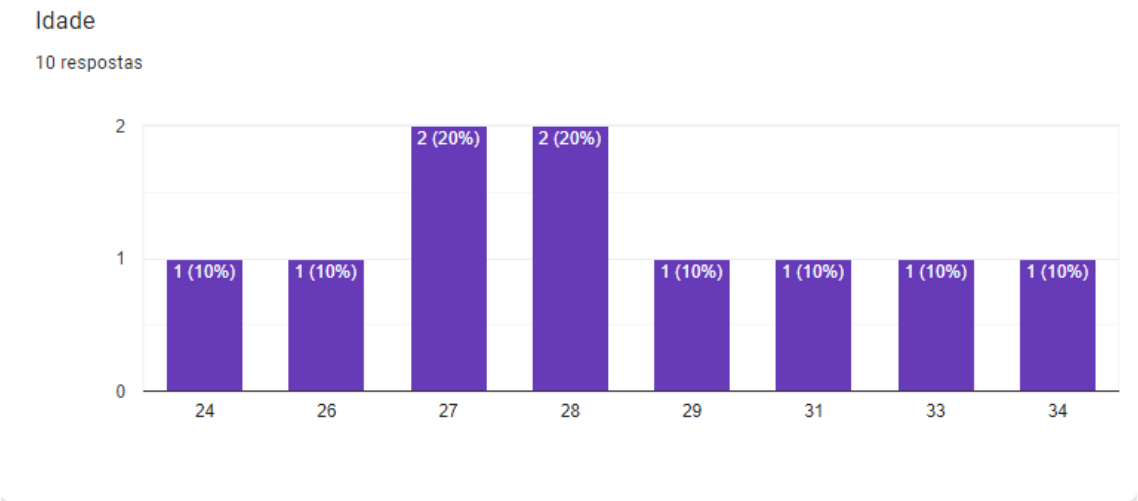
A primeira etapa da avaliação com usuários, o formulário pré-teste, tem por objetivo a caracterização do perfil dos participantes da pesquisa aplicada. Ao todo, 10 pessoas participaram da análise desta plataforma, sendo 6 homens e 4 mulheres, conforme pode ser visto na Figura 5.1. Já a faixa etária dos usuários, indicada na Figura 5.2, está entre 24 e 34 anos.

Figura 5.1: Gráfico da identidade de gênero dos participantes



Fonte: Elaborada pela autora

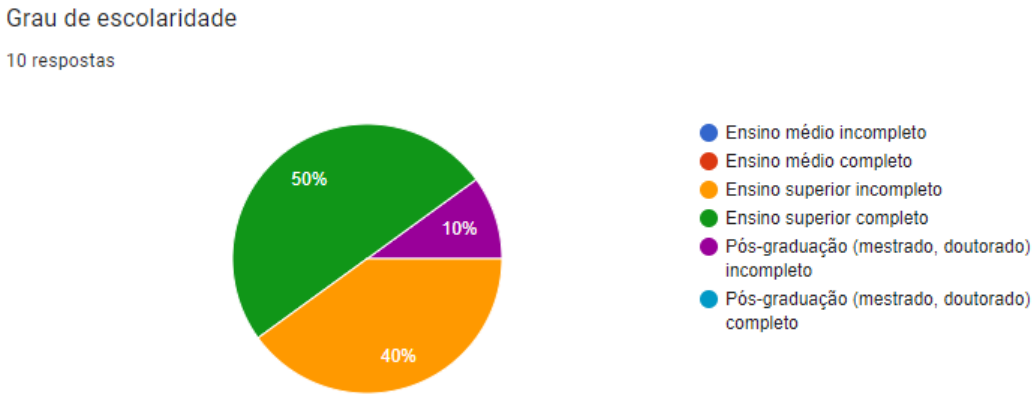
Figura 5.2: Gráfico da faixa etária dos participantes



Fonte: Elaborada pela autora

O grau de escolaridade pode ser consultado na Figura 5.3, onde é possível ver que a maioria dos participantes possui ensino superior. Destes, 5 completaram o ensino superior e 1 iniciou os estudos em uma pós-graduação. Os participantes restantes tem ensino superior incompleto.

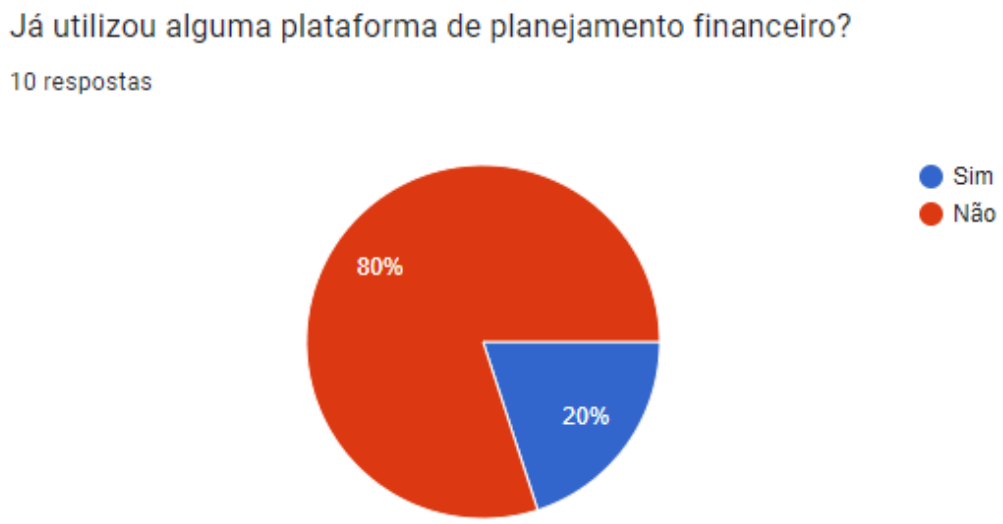
Figura 5.3: Gráfico do grau de escolaridade dos participantes



Fonte: Elaborada pela autora

Por fim, foi perguntado se os usuários já haviam utilizado alguma plataforma de controle financeiro antes. Como pode ser visto na Figura 5.4, apenas 2 dos participantes já haviam usado alguma plataforma do tipo.

Figura 5.4: Gráfico sobre utilização de plataformas de controle financeiro

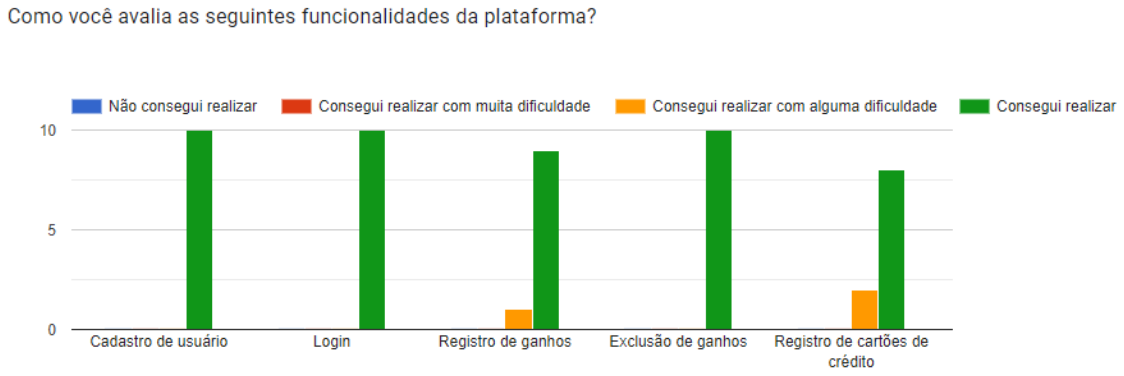


Fonte: Elaborada pela autora

5.4 Análise dos resultados da avaliação da plataforma

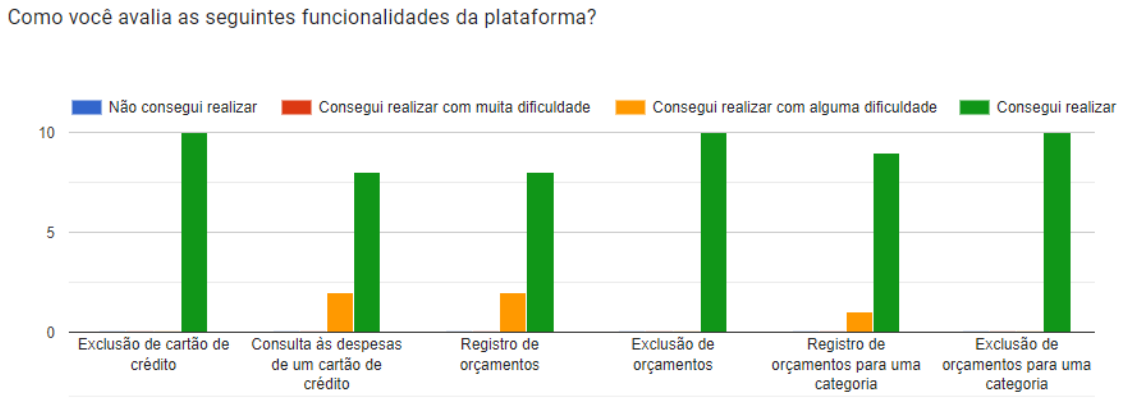
Após a finalização da sequência de tarefas propostas a serem efetuadas na plataforma criada neste trabalho, os usuários avaliaram a dificuldade que tiverem para concluí-las. A Figura 5.5 e a Figura 5.6 mostram o resultado desta avaliação.

Figura 5.5: Gráfico da avaliação das tarefas solicitadas utilizando a plataforma



Fonte: Elaborada pela autora

Figura 5.6: Gráfico da avaliação das tarefas solicitadas utilizando a plataforma



Fonte: Elaborada pela autora

De maneira geral, os participantes não tiveram grandes dificuldades para executar as tarefas propostas. Não houve nenhum participante que não conseguiu realizar alguma tarefa ou teve muita dificuldade para fazê-la e a maioria conseguiu concluí-las sem nenhuma dificuldade. Houve, entretanto, o reporte de alguma dificuldade para realização de algumas tarefas: 1 usuário teve pequenas dificuldades para o registro de ganhos, 2 para o registro de cartões de crédito, 2 para a consulta às despesas de um cartão de crédito, 2 para o registro de orçamentos e 1 para o registro de orçamentos de uma categoria.

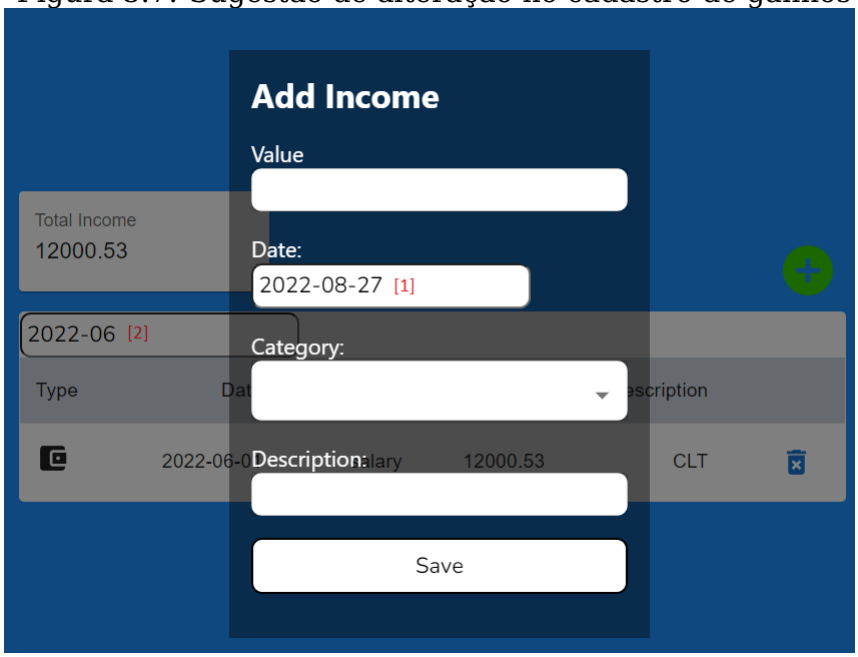
Durante as chamadas de vídeo feitas para a realização da pesquisa, alguns participantes fizeram comentários sobre o funcionamento da plataforma. O comentário predominante foi em relação ao cadastro de

cartões de crédito: três usuários expressaram o desejo de poder cadastrar cartões de crédito com o dia de fechamento da fatura maior do que o dia de pagamento - algo não permitido pela plataforma no momento em que a pesquisa foi realizada. Isto pode explicar a dificuldade informada no registro de cartões de crédito, na Figura 5.5.

Um participante também informou que gostaria que houvessem mais categorias disponíveis para a criação de orçamentos. Atualmente, é permitido que apenas um orçamento por categoria seja criado para determinado mês, sendo necessário agrupar orçamentos de categorias não presentes na plataforma na categoria *others*. Possivelmente seja por este motivo que tenha sido reportada alguma dificuldade no cadastro de orçamentos para categorias, na Figura 5.6.

Além disso, um usuário também reportou que gostaria que o mês para o cadastro de um ganho (Figura 5.7 [1]) fosse o mesmo que o mês que está sendo consultado (Figura 5.7 [2]) quando a modal de cadastro for aberta. Talvez seja por este motivo a pequena dificuldade informada no registro de ganhos, na Figura 5.5.

Figura 5.7: Sugestão de alteração no cadastro de ganhos

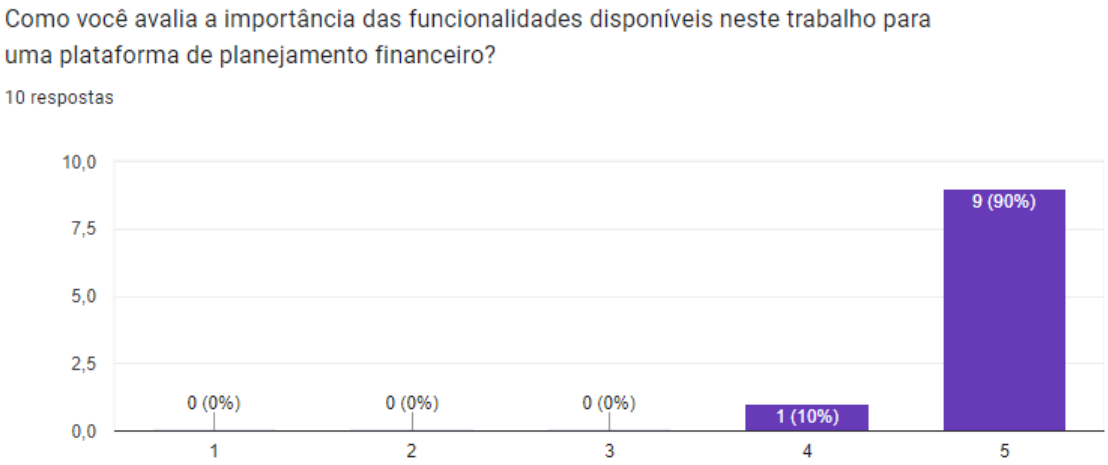


Fonte: Elaborada pela autora

Também foi perguntado aos participantes, em uma escala de 1 à 5, sendo 1 irrelevante e 5 muito relevante, o quão importantes eles consideram as funcionalidades disponíveis. Como pode ser visto na Figura 5.8, há um

consenso de que as funcionalidades presentes neste trabalho são importantes para uma plataforma de planejamento financeiro.

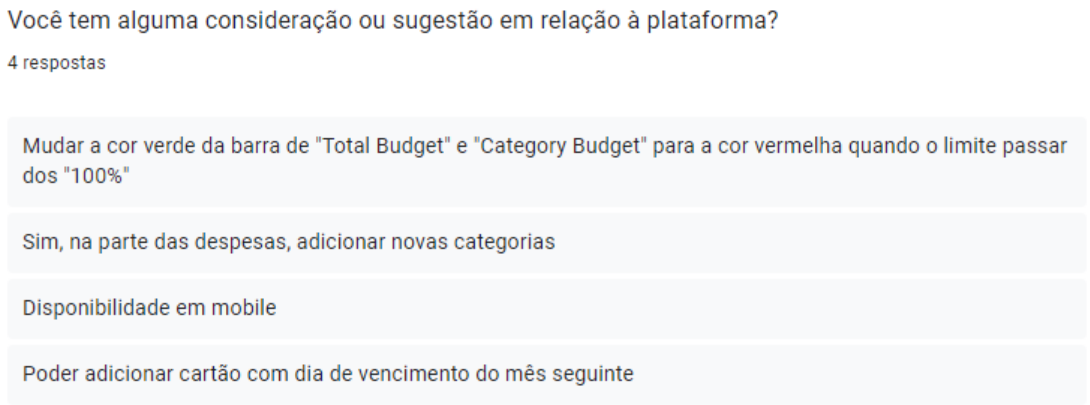
Figura 5.8: Relevância das funcionalidades disponíveis



Fonte: Elaborada pela autora

Por fim, foi disponibilizado um campo livre para que os participantes pudessem deixar sugestões ou considerações em relação à plataforma, se assim quisessem. As respostas podem ser conferidas na Figura 5.8.

Figura 5.9: Sugestões dos participantes



Fonte: Elaborada pela autora

Neste campo, novamente aparecem as sugestões de possibilidade de adicionar cartões com o dia de pagamento menor do que o dia de fechamento da fatura e o desejo por mais categorias, reforçando as teorias apresentadas anteriormente.

5.5 Considerações Gerais e Melhorias Futuras

Levando em consideração o *feedback* fornecido, a solução desenvolvida tem funcionalidades condizentes e uma boa base inicial para uma plataforma de planejamento financeiro. Ela pode, entretanto, oferecer uma experiência mais completa caso novas funcionalidades sejam adicionadas.

Considerando as sugestões passadas, dois ajustes já foram implementados na plataforma: foi removida a necessidade de um cartão de crédito ter o dia de fechamento de fatura menor do que o dia de pagamento e, ao abrir uma modal para o cadastro de um ganho, de uma despesa ou de um orçamento, a data inicial selecionada na modal de cadastro já se encontra no mês que se está consultando, facilitando a seleção.

Além destas mudanças, seria interessante permitir aos usuários que criassem as categorias que desejassem, permitindo uma forma de organização mais personalizada, e disponibilzar uma *dashboard* com informações como saldo e gráficos para a visualização de valores de ganhos e despesas por categoria. Outra adição interessante, seria permitir a adição de contas para uma melhor organização. Ademais, fornecer integração com *APIs* de instituições bancárias para buscar os dados financeiros de maneira automática facilitaria o uso da plataforma.

6 CONCLUSÃO

Este trabalho teve por objetivo o projeto e implementação de uma aplicação destinada a auxiliar no controle de finanças. Para isto, foi desenvolvida uma plataforma *WEB*, a qual foi dividida em dois componentes: servidor e cliente, sendo o primeiro implementado em *Kotlin* e o segundo em *React*.

Devido ao escopo deste projeto, neste momento foram disponibilizadas apenas funcionalidades centrais a uma plataforma do tipo, como gerência de ganhos e despesas. Como pode ser visto na pesquisa com usuários, elas são de fato consideradas importantes pelos participantes, porém para uma experiência mais completa seria interessante a adição de novas funcionalidades como permitir que usuários criassem suas próprias categorias, a disponibilização de gráficos para visualização das finanças e a integração com *APIs* de instituições bancárias para preenchimento automático das informações de finanças dos usuários.

Ademais, devido ao tempo disponível para a conclusão deste trabalho, hoje a plataforma está sendo executada apenas em ambiente local. Futuramente faria sentido migrá-la para nuvem, de forma que pudesse ser acessada de qualquer local.

REFERÊNCIAS

AMAZON. **Amazon RDS for MySQL**. 2022. Acessado em 26/06/2022. Disponível na Internet: <https://aws.amazon.com/pt/rds/mysql>.

AXIOS. **AXIOS**. 2022. Acessado em 13/08/2022. Disponível na Internet: <https://axios-http.com/docs/intro>.

BERSON, A. Client/server architecture. Em: _____. **Client/Server Architecture**. [S.l.]: Mcgraw-Hill (Tx), 1992. ISBN 978-0070050761.

CODD, E. F. **A Relational Model of Data for Large Shared Data Banks**. 1970. Acessado em 25/06/2022. Disponível na Internet: <ttps://www.seas.upenn.edu/zives/03f/cis550/codd.pdf>.

DBEAVER. **DBEAVER**. 2022. Acessado em 17/07/2022. Disponível na Internet: <https://dbeaver.io/>.

FASTBUDGET. **FASTBUDGET**. 2022. Acessado em 21/08/2022. Disponível na Internet: <https://fastbudget.app/>.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Acessado em 06/07/2022. Disponível na Internet: <https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>.

FORTUNO. **FORTUNO**. 2022. Acessado em 21/08/2022. Disponível na Internet: <https://fortuno.app/>.

GOOGLE. **GUAVA**. 2022. Acessado em 06/08/2022. Disponível na Internet: <https://guava.dev/releases/19.0/api/docs/com/google/common/cache/CacheBuilder.html>.

GRAY, D. **DAVE GRAY**. 2022. Acessado em 14/08/2022. Disponível na Internet: <https://www.youtube.com/c/DaveGrayTeachesCode>.

KOTLIN. **Kotlin Programming Language**. 2022. Acessado em 25/06/2022. Disponível na Internet: <https://kotlinlang.org/>.

KOTLIN. **What advantages does Kotlin give me over the Java programming language**. 2022. Acessado em 25/06/2022. Disponível na Internet: <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.htm>.

MARTIN, R. Clean code: A handbook of agile software craftsmanship. Em: _____. **Clean Code: A Handbook of Agile Software Craftsmanship**. [S.l.]: Prentice Hall PTR, 2008. ISBN 978-0132350884.

MATERIALUI. **Material UI Documentation**. 2022. Acessado em 13/08/2022. Disponível na Internet: <https://mui.com/>.

MAVEN. **What is Maven?** 2022. Acessado em 25/06/2022. Disponível na Internet: <ttps://maven.apache.org/what-is-maven.htm>.

MOBILLS. **MOBILLS**. 2022. Acessado em 21/08/2022. Disponível na Internet: <https://www.mobills.com.br/>.

MYSQL. **MySQL**. 2022. Acessado em 28/06/2022. Disponível na Internet: <https://www.mysql.com/>.

NPMTRENDS. **Angular vs React vs Vue**. 2022. Acessado em 25/06/2022. Disponível na Internet: <https://www.npmtrends.com/angular-vs-react-vs-vue>.

OECD. **Consumer Prices**. 2022. Acessado em 02/09/2022. Disponível na Internet: <https://data.oecd.org/price/inflation-cpi.htm>.

OLUWATOSIN, H. S. **Client-Server Model** . 2014. Acessado em 02/07/2022. Disponível na Internet: <https://www.iosrjournals.org/iosr-jce/papers/Vol16-issue1/Version-9/J016195771.pdf>.

PALERMO, J. **The Onion Architecture**. 2008. Acessado em 05/07/2022. Disponível na Internet: <https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/>.

POSTMAN. **POSTMAN**. 2022. Acessado em 06/08/2022. Disponível na Internet: <https://www.postman.com/>.

REACT. **HOOKS**. 2022. Acessado em 13/08/2022. Disponível na Internet: <https://reactjs.org/docs/hooks-intro.html>.

REACT. **React - A Javascript library for building user interfaces**. 2022. Acessado em 25/06/2022. Disponível na Internet: <https://reactjs.org/>.

REDRAT. **What is a REST API?** 2022. Acessado em 06/07/2022. Disponível na Internet: <https://www.redhat.com/en/topics/api/what-is-a-rest-api#rest>.

SPARK. **SPARK**. 2022. Acessado em 06/08/2022. Disponível na Internet: <https://sparkjava.com/>.

SPRING. **Introduction to Spring Framework**. 2022. Acessado em 26/06/2022. Disponível na Internet: <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.htm>.

ZOOM. **ZOOM**. 2022. Acessado em 20/08/2022. Disponível na Internet: <https://zoom.us/>.

APÊNDICE A — FORMULÁRIO DE TESTE

Avaliação de plataforma de planejamento financeiro

Este teste busca avaliar a experiência de usuários ao utilizar uma plataforma de planejamento financeiro desenvolvida como trabalho de conclusão do curso de Ciência da Computação da Universidade Federal do Rio Grande do Sul (UFRGS).

Este questionário faz parte do trabalho de conclusão, e será feito de maneira totalmente anônima, ou seja, suas respostas serão utilizadas somente para este fim. Se você concorda com esses termos, basta iniciar a pesquisa na próxima seção.

*Obrigatório

1. Idade *

_____

2. Gênero *

Marcar apenas uma oval.

- ◯ Mulher
- ◯ Homem
- ◯ Outro: _____

3. Grau de escolaridade *

Marcar apenas uma oval.

- ◯ Ensino médio incompleto
- ◯ Ensino médio completo
- ◯ Ensino superior incompleto
- ◯ Ensino superior completo
- ◯ Pós-graduação (mestrado, doutorado) incompleto
- ◯ Pós-graduação (mestrado, doutorado) completo

4. Já utilizou alguma plataforma de planejamento financeiro? *

   Marcar apenas uma oval.

   - ◯ Sim
   - ◯ Não

O objetivo desta etapa é avaliar as funcionalidades da plataforma. Para esta avaliação, é necessário que os seguintes objetivos sejam concluídos:

1. Realizar cadastro de usuário
2. Realizar login
3. Registrar um ganho no mês atual
4. Registrar um ganho em algum outro mês de sua escolha
5. Deletar um ganho cadastrado
6. Registrar dois cartões de crédito
7. Deletar um cartão de crédito
8. Registrar uma despesa
9. Registrar uma despesa de cartão de crédito
10. Consultar as despesas de cartão de crédito através da tela "Cards"
11. Registrar um orçamento total para o mês atual
12. Registrar um orçamento total para outro mês de sua escolha
13. Deletar um orçamento cadastrado
14. Registrar um orçamento para uma categoria de sua escolha
15. Deletar o orçamento de uma categoria

5. Como você avalia as seguintes funcionalidades da plataforma? *

Marcar apenas uma oval por linha.

	Não consegui realizar	Consegui realizar com muita dificuldade	Consegui realizar com alguma dificuldade	Consegui realizar
Cadastro de usuário				
Login				
Registro de ganhos				
Exclusão de ganhos				
Registro de cartões de crédito				
Exclusão de cartão de crédito				
Consulta às despesas de um cartão de crédito				
Registro de orçamentos				
Exclusão de orçamentos				
Registro de orçamentos para uma categoria				
Exclusão de orçamentos para uma categoria				

6. Como você avalia a importância das funcionalidades disponíveis neste trabalho * para uma plataforma de planejamento financeiro?

Marcar apenas uma oval.

	1	2	3	4	5	
Irrelevantes	◯	◯	◯	◯	◯	Muito relevantes

7. Você tem alguma consideração ou sugestão em relação à plataforma?