

UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO HOFFART VIEIRA

**APLICATIVO DE CONTROLE FINANCEIRO UTILIZANDO
RECONHECIMENTO DE IMAGEM**

BAURU
Dezembro/2020

THIAGO HOFFART VIEIRA

**APLICATIVO DE CONTROLE FINANCEIRO UTILIZANDO
RECONHECIMENTO DE IMAGEM**

Trabalho de Conclusão de Curso do Bacharelado
em Ciência da Computação da Universidade
Estadual Paulista “Júlio de Mesquita Filho”,
Faculdade de Ciências, Campus Bauru.
Orientadora: Prof. Dra. Simone das Graças
Domingues Prado

BAURU
Dezembro/2020

Thiago Hoffart Vieira APLICATIVO DE CONTROLE FINANCEIRO UTILIZANDO RECONHECIMENTO DE IMAGEM/ Thiago Hoffart Vieira. – Bauru, Dezembro/2020- 50 p. : il. (algumas color.) ; 30 cm.
Orientadora: Prof. Dra. Simone das Graças Domingues Prado
Trabalho de Conclusão de Curso – Universidade Estadual Paulista “Júlio de Mesquita Filho”
Faculdade de Ciências
Ciência da Computação, Dezembro/2020.
1. Tags 2. Para 3. A 4. Ficha 5. Catalográfica

Thiago Hoffart Vieira

APLICATIVO DE CONTROLE FINANCEIRO UTILIZANDO RECONHECIMENTO DE IMAGEM

Trabalho de Conclusão de Curso do Bacharelado em Ciência da Computação da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof. Dra. Simone das Graças Domingues Prado
Orientador
Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Prof. Dr. João Pedro Albino
Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Prof. Dr. Aparecido Nilceu Marana
Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Bauru, 16 de dezembro de 2020.

Agradecimentos

Agradeço primeiramente à minha família, que proporcionou os meios para que eu estivesse aqui hoje, sempre me apoiou nos momentos de dificuldade e me deu forças para continuar minha trajetória. Em seguida agradeço aos meus amigos, tanto os que pude conhecer durante essa faculdade quanto os de São Paulo, pois todos agregaram para que eu seja o que sou hoje, desde os incentivos para mudar de carreira, os conselhos nos momentos difíceis, até os momentos de felicidade que vivemos e espero que possamos viver por mais muitos anos juntos.

Dedico esse trabalho à minha namorada, que esteve comigo nesses últimos quatro anos, acreditou no meu potencial e esteve presente nas dificuldades, me incentivando em momentos em que eu mesmo não acreditei em mim; sou muito feliz pelo que construímos juntos e por nossas gatas e cachorra, que nos dão carinho incondicional e animam nossos dias.

Agradeço também aos funcionários e professores da UNESP Bauru, especialmente os do Departamento de Computação, que estiveram presentes durante a minha graduação e com certeza contribuíram para minha formação e aprendizado; à Jr.COM que teve papel essencial no início da minha vida profissional.

Por fim, agradeço à minha orientadora, que me deu apoio durante esse projeto, contribuindo para a conclusão da minha graduação mesmo em meio a uma pandemia, compreendendo as dificuldades desse momento tão difícil.

Resumo

O avanço da tecnologia e o desenvolvimento de *smartphones* com crescente poder de processamento permitem que hoje sejam utilizados algoritmos de *machine learning* para otimizar o cotidiano do usuário. O presente projeto consistiu na construção de um aplicativo de orçamentos financeiros através do reconhecimento de objetos pela integração com o Tensorflow. O reconhecimento de imagem associado ao gerenciamento de gastos pessoais, permite otimizar o tempo despendido pelos usuários para realização de tarefas cotidianas, como a confecção de orçamentos e listas de compras. O sistema foi desenvolvido utilizando *frameworks* modernos, como Flutter para o desenvolvimento do aplicativo e Laravel e Lighthouse para o desenvolvimento da API.

Palavras-chave: Aplicativo financeiro; Reconhecimento de imagem; Tensorflow; Flutter; Laravel; Inteligência artificial.

Abstract

The advancement of technology and the development of smartphones with increasing processing power allow today machine learning algorithms to be used to optimize the user's daily life. The present project consisted of building a financial budgeting application through object recognition through integration with Tensorflow. The image recognition associated with the management of personal expenses, allows to optimize the time spent by users to carry out daily tasks, such as making budgets and shopping lists. The system was developed using modern *frameworks*, such as Flutter for application development and Laravel and Lighthouse for API development.

Keywords: Financial application, Image recognition; Tensorflow; Flutter; Laravel; Artificial intelligence.

Lista de códigos

1	<i>flutter.yml</i> do projeto.	31
2	<i>Migration</i> de Usuario.	33
3	<i>Migration</i> de Despesa.	34
4	<i>Model</i> de Despesa.	35
5	<i>Type, queries, mutations e inputs</i> de Despesa.	36
6	Teste da <i>query getDespesa</i> .	36
7	Requisição para criação de produto.	41
8	Teste da função de normalização de objetos.	42
9	Classe base de integração com o Tensorflow	44

Listas de figuras

Figura 1 – Arquitetura <i>Model-View-Controller</i>	16
Figura 2 – Exemplo do fluxo de estado utilizando a biblioteca NgRx.	17
Figura 3 – Relação de testes de <i>software</i>	18
Figura 4 – Exemplo da arquitetura DSVC.	20
Figura 5 – Exemplo de estrutura de <i>branches</i> utilizando Gitflow.	21
Figura 6 – Arquitetura do Flutter	24
Figura 7 – Exemplo de uma <i>story</i> no Pivotal Tracker	26
Figura 8 – MER do banco de dados	30
Figura 9 – Notificação do CI no Slack.	32
Figura 10 – Fluxo do estado do aplicativo.	38
Figura 11 – Tela de introdução	39
Figura 12 – Tela de cadastro	39
Figura 13 – Tela de <i>dashboard</i>	40
Figura 14 – Tela de listagem de produto	40
Figura 15 – Tela de cadastro/edição de produto	41
Figura 16 – Aplicativo reconhecendo bananas	46
Figura 17 – Aplicativo reconhecendo laranjas	46

Listas de abreviaturas e siglas

UNESP	Universidade Estadual Paulista "Júlio de Mesquita Filho"
API	Application Programming Interface
MVC	Model-View-Controller
UI	User Interface
BLoC	Business Logic of Component
E2E	End-to-End
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	Structured Query Language
GraphQL	Graph Query Language
CI	Continuous Integration
CD	Continuous Delivery
VCS	Version Control Systems
LVCS	Local Version Control Systems
RCS	Revision Control Systems
CVCS	Centralized Version Control Systems
DVCS	Distributed Version Control Systems
PHP	HyperText Preprocessor
HTML	HyperText Markup Language
ORM	Object-relacional Mapping
ML	Machine Learning
CRUD	Create, Read, Update and Delete
MER	Modelo Entidade Relacionamento
SDK	Software Development Kit

Sumário

1	INTRODUÇÃO	12
1.1	Problema	13
1.2	Objetivos	14
1.2.1	Objetivo Geral	14
1.2.2	Objetivos Específicos	14
1.3	Justificativa	14
2	FUNDAMENTAÇÃO TEÓRICA	15
2.1	Inteligência Artificial	15
2.1.1	<i>Machine Learning</i>	15
2.2	Arquitetura de Software	15
2.2.1	Arquitetura MVC	16
2.2.2	<i>Framework</i>	16
2.2.3	<i>State Management</i>	16
2.2.4	Testes de <i>software</i>	17
2.3	Banco de dados	18
2.3.1	Linguagem de Query	18
2.4	CI/CD	18
2.5	Sistema de controle de versões	19
2.6	Gitflow	20
3	FERRAMENTAS	22
3.1	Git	22
3.2	PostgreSQL	22
3.3	Docker	22
3.4	Dart	22
3.5	PHP	23
3.6	GraphQL	23
3.7	Flutter	23
3.8	Laravel	24
3.9	Lighthouse	25
3.10	Pivotal Tracker	25
3.11	Tensorflow	26
4	DESENVOLVIMENTO	28
4.1	Pré-desenvolvimento	28

4.1.1	Levantamento dos requisitos	28
4.1.2	Modelagem do banco de dados	29
4.1.3	Geração dos projetos	30
4.1.3.1	Flutter	30
4.1.3.2	Laravel	31
4.1.4	Construção da infraestrutura	31
4.1.4.1	Docker	31
4.1.4.2	CI	31
4.2	Projeto	33
4.2.1	Desenvolvimento da API	33
4.2.1.1	<i>Migrations</i>	33
4.2.1.2	<i>Models</i>	34
4.2.1.3	GraphQL	35
4.2.1.4	Testes de integração	36
4.2.2	Desenvolvimento do aplicativo	37
4.2.2.1	Modelagem do <i>State Management</i>	37
4.2.2.2	Construção das páginas	38
4.2.2.3	Requisições à API	41
4.2.2.4	Testes unitários	42
4.2.3	Integração com o Tensorflow	43
5	CONCLUSÃO	47
6	PRÓXIMOS PASSOS	48
	REFERÊNCIAS	49

1 Introdução

A constante evolução da tecnologia proporciona à sociedade fácil acesso a dispositivos eletrônicos a qualquer momento. Os *smartphones*, com grande poder de processamento nos dias atuais, permitem executar algoritmos de *machine learning* em qualquer lugar. Por essa facilidade, o desenvolvimento de aplicativos que utilizam *machine learning*, cujo objetivo é facilitar o cotidiano do usuário ou até mesmo entreter-lo, se tornou algo mais comum e mais fácil de se alcançar. Um exemplo de aplicativo que utiliza *machine learning* é o *snapchat*, neste utilizam um algoritmo de rastreamento facial para aplicar filtros nos rostos de seus usuários enquanto eles tiram fotos.

Machine learning é programar computadores para otimizar o critério de performance utilizando exemplos de dados ou experiências passadas (ALPAYDIN, 2020), ou seja, um algoritmo de *machine learning* tem a capacidade de aprender e se aprimorar conforme a mudança dos dados e seus resultados anteriores. Essa área da inteligência artificial possui três formas de algoritmos: aprendizado supervisionado, aprendizado não supervisionado e aprendizado reforçado.

O primeiro tipo de algoritmo de *machine learning* consiste em regressão e classificação, onde a partir de um banco de dados, a máquina é capaz de reconhecer objetos através de padrões. Já o segundo, tem como objetivo encontrar padrões de dados sem *labels* predefinidos. Enquanto isso, o terceiro trabalha através de seus erros, com a máquina procurando a solução correta após cometer uma falha (FREITAS, 2019).

Para que o reconhecimento de imagem ocorra de maneira efetiva, um dos conceitos utilizados é o *computer vision*, que surgiu nos anos 60 como um ramo na inteligência artificial, onde a intenção é imitar a funcionalidade do sistema de visão humana. A partir dos anos 80, houve o aprimoramento dos modelos matemáticos e técnicas de análise de imagem (ALSING, 2018). Atualmente um dos algoritmos comumente utilizados para detecção de objetos é Rede Neural Convolucional, que é uma classe de rede neural.

As redes neurais e o *deep learning* atualmente fornecem as melhores soluções para muitos problemas no reconhecimento de imagens, reconhecimento de fala e processamento de linguagem natural. As redes neurais permitem ao computador aprender a partir de dados observacionais, enquanto o *deep learning* consiste em um conjunto poderoso de técnicas para aprender em redes neurais (MULFARI; MINNOLO; PULIAFITO, 2017). Este último permite modelos computacionais que são compostos de várias camadas de processamento para aprender representações de dados com vários níveis de abstração, além de muitos softwares de *deep learning* fornecerem suporte a *smartphones* ou *tablets*.

O uso do reconhecimento de imagem em aplicações *mobile* já é utilizado para algumas

finalidades descritas na literatura. Um dos trabalhos utiliza o reconhecimento de imagem em *smartphones* para classificar diferentes tipos de comidas tailandesas e assim informar as calorias e realizar sugestões para seus usuários; a aplicação foi desenvolvida integrando o *framework mobile* React Native com o Tensorflow Lite para o reconhecimento de imagem ([TIANKAEW; CHUNPONGTHON; METTANANT, 2018](#)). Já em outro artigo, os autores utilizaram o reconhecimento de imagem para identificar placas de trânsito em um dispositivo *mobile*, podendo ser aplicado na automatização de tarefas dentro de carros autônomos ([BENHAMIDA; VARKONYI-KOCZY; KOZLOVSZKY, 2020](#)).

O presente trabalho pretende demonstrar o funcionamento dos conceitos de reconhecimento de imagens descritos acima em uma plataforma *mobile* através de um sistema de controle financeiro, onde almeja-se o reconhecimento de objetos cadastrando estes como produtos de um orçamento do usuário. No segundo capítulo serão abordados conceitos teóricos importantes à realização do projeto; no capítulo três são apresentadas as ferramentas utilizadas para a realização do mesmo, incluindo linguagens e *frameworks*. Já no capítulo quatro são discutidos o desenvolvimento do aplicativo, da API e a integração com a biblioteca de reconhecimento de imagem.

1.1 Problema

Os avanços tecnológicos da atualidade permitem a criação de modelos inteligentes para a resolução de problemas cotidianos. Entre eles, encontram-se as cidades inteligentes, que integram o desenvolvimento urbano com a tecnologia da informação, comunicação e internet, de forma a gerenciar os recursos de uma cidade, melhorando, assim, a eficiência de serviços. Tendo em vista as dificuldades que podem ser encontradas na manutenção de uma cidade inteligente, o *deep learning* pode fornecer métodos baseados em redes neurais artificiais, que buscam soluções eficientes para o reconhecimento de objetos em um ambiente inteligente. Considerando a interação entre uma pessoa e seu ambiente, vários sistemas consistem em aplicativos móveis usando a câmera do *smartphone* e algoritmos inteligentes para entender o espaço ao redor do usuário ([MULFARI; MINNOLO; PULIAFITO, 2017](#)).

O presente trabalho consiste no desenvolvimento de um sistema de controle financeiro utilizando reconhecimento de imagem, permitindo que o usuário utilize seu *smartphone* para reconhecer objetos e anotar seu preço em ambientes comerciais e assim realizar um orçamento de seus gastos de maneira rápida e automatizada.

1.2 Objetivos

1.2.1 Objetivo Geral

Criar um aplicativo de controle financeiro utilizando reconhecimento de imagem para automatização de compra de produtos.

1.2.2 Objetivos Específicos

O presente trabalho tem como objetivos específicos:

- Construir a infraestrutura necessária para o desenvolvimento de um aplicativo para *smartphone* em *Flutter* e uma API *Laravel* utilizando *Docker*;
- Modelar banco de dados necessário para suprir os requisitos do aplicativo;
- Desenvolver API para a comunicação do aplicativo com o banco de dados;
- Elaborar aplicativo em *Flutter* e integrá-lo a biblioteca *TensorFlow*;
- Implementar modelo de treinamento dos dados no *TensorFlow* e avaliar seu desempenho.

1.3 Justificativa

O sistema em questão pode ser utilizado para otimizar o gerenciamento de gastos pessoais, integrando a tecnologia de reconhecimento de imagem em *smartphones* com preços e produtos de um comércio, diminuindo o tempo dispendido pelos usuários para a realização de tarefas cotidianas como a confecção de orçamentos e listas de compras.

Em relação a parte técnica, esta consiste em integrar modelos de *machine learning* através da biblioteca *open source* Tensorflow criada pela Google com dois sistemas: Um sistema móvel utilizando o *framework* Flutter e uma API utilizando Laravel. Gerenciar a infraestrutura da aplicação isolando o ambiente de desenvolvimento com *containers* criados utilizando Docker. Versionar o código utilizando Git e Pivotal Tracker. Dispondo-se dessa *stack*, podemos ter avanços na integração da biblioteca do Tensorflow com Flutter, tanto na forma de construção de novas bibliotecas de integração como na colaboração em bibliotecas já disponíveis. Um ponto positivo de se utilizar o Flutter, é que além de ele ser *open source*, também há a possibilidade de se desenvolver nativamente para as plataformas Android e iOS, com isso, abre-se um leque maior de possibilidades para o desenvolvimento da tecnologia.

2 Fundamentação Teórica

2.1 Inteligência Artificial

Atualmente, os computadores já possuem a capacidade de resolução de diversos problemas, entre eles aritméticos, de ordenação e de busca. Inicialmente, alguns deles eram pertencentes à área de Inteligência Artificial, entretanto foram sendo resolvidos de maneira mais compreensiva, de forma que não fazem mais parte desta ramificação da Ciência da Computação. A Inteligência Artificial surgiu com a finalidade de executar tarefas inteligentes, capacidade de humanos e outros animais, como reconhecer objetos, nossa linguagem e tomar decisões de maneira independente ([MILLIGTON; FUNGE, 2009](#)).

Como engenharia, a Inteligência Artificial diz respeito aos conceitos, teorias e prática da construção de máquinas inteligentes; já do ponto de vista científico, a Inteligência Artificial desenvolve conceitos para nos ajudar a entender como o comportamento inteligente humano e animal funciona ([GENESERETH; NILSSON, 1987](#)).

2.1.1 *Machine Learning*

Machine Learning ou aprendizado de máquinas é uma área da computação que desenvolve algoritmos que buscam melhorar seu próprio desempenho utilizando dados de exemplos ou experiências passadas. Essa inteligência é necessária em casos em que não é possível programar a resolução de um problema diretamente, mas são necessários dados como exemplos ([ALPAYDIN, 2020](#)). No reconhecimento de imagens, por exemplo, é recebida uma imagem como entrada, e como saída é esperado o reconhecimento desta, utilizando-se de uma base de dados para treinar o algoritmo e ajudá-lo a reconhecê-la.

2.2 Arquitetura de *Software*

O estudo da Arquitetura de *Software* consiste em entender como os sistemas de *software* são desenhados e construídos, por meio de um conjunto das principais decisões de *design* de um sistema. A arquitetura inclui algumas decisões estruturais, como a construção de blocos de alto nível - componentes, conectores, e configurações -, que consistem em uma evolução constante dos padrões do sistema. A formação desse modelo é essencial para a produção e evolução de produtos de qualidade ([MEDVIDOVIC; TAYLOR, 2010](#)).

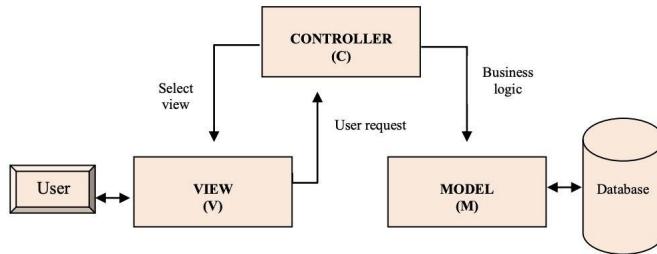
2.2.1 Arquitetura MVC

A arquitetura MVC é um padrão de arquitetura dentro da engenharia de *software*, que contém três módulos: *model*, *view* e *controller*. Os módulos isolados uns dos outros facilitam o entendimento e modificação de cada módulo separadamente, sem ter que conhecer tudo sobre os outros módulos (POP; ALTAR, 2014). A definição de cada um dos módulos é:

- *Model* - Representação do modelo de dados, contendo a lógica para a leitura, escrita e validações;
- *View* - Representação da visualização dos dados que o *Model* contém, realizando a interação com o usuário;
- *Controller* - Controla os dados no *Model* por meio das *requests* e atualiza *view* quando necessário.

O esquema da Figura 1 apresenta o funcionamento da arquitetura MVC.

Figura 1 – Arquitetura *Model-View-Controller*



Fonte: Sarker; Apu (2014).

2.2.2 Framework

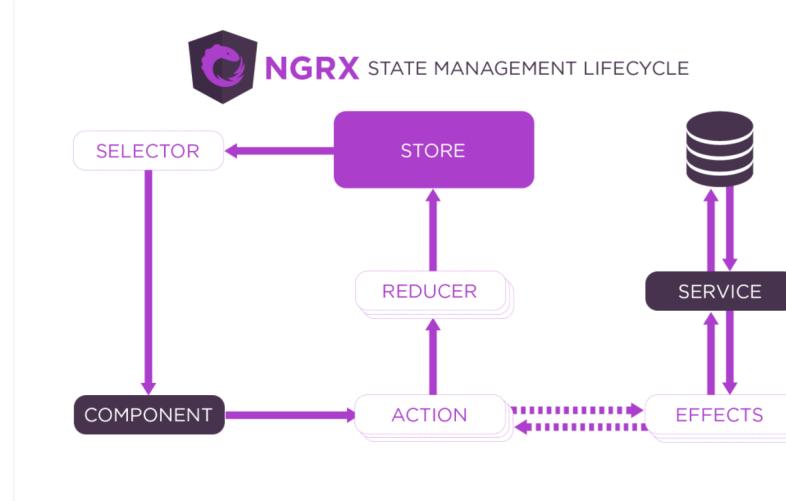
O *framework* é uma ferramenta para o desenvolvimento de *software* que reúne diversos módulos para auxiliar em sua engenharia, Provendo uma base onde os desenvolvedores conseguirão programar para uma ou mais plataformas específicas. Nessa base do *framework* haverão classes, funções, interfaces, entre outros, que se comunicarão de maneira pré-definida. Além disso, ele disponibilizará a opção de reescrever alguns métodos e subclasses diante da necessidade do desenvolvedor (PREE, 1994).

2.2.3 State Management

O *State Management* é o paradigma de como gerenciar o estado de sua aplicação, ou seja, seus dados. Através dele devemos controlar o fluxo de dados, como mostra Figura 2, buscando persisti-los e controlar a maneira de como eles vão interagir com a UI. Para auxiliar no *State Management*, existem diversas bibliotecas, como Redux, NgRx e BLoC, que permitem

lidar com o estado de maneira previsível, centralizada, flexível e fácil de "debbugar" (ROUSE, 2020).

Figura 2 – Exemplo do fluxo de estado utilizando a biblioteca NgRx.



Fonte: <https://ngrx.io/guide/store>. Acesso em: 29 de novembro de 2020.

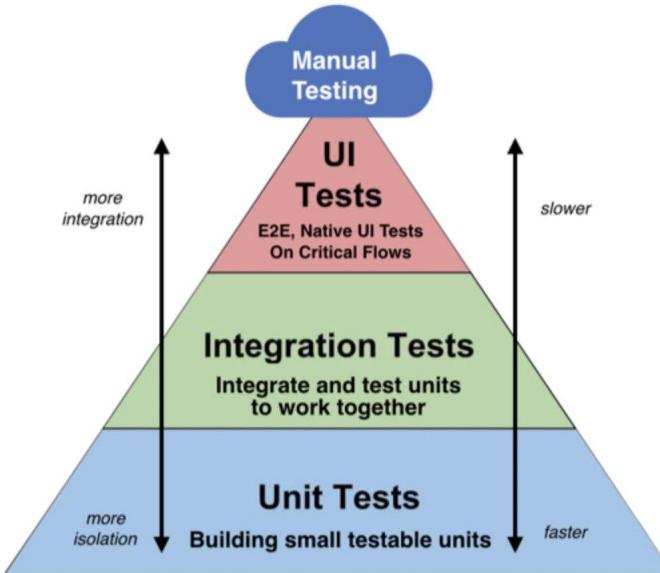
2.2.4 Testes de *software*

Conforme o desenvolvimento de *software* foi se tornando mais complexo passou-se a existir a necessidade de testá-lo e garantir seu funcionamento. Na computação moderna existem três categorias principais de testes de *software*: os testes unitários, os testes de integração e os testes End-to-End (E2E).

- Teste Unitário: como diz o próprio nome, tem como objetivo testar uma unidade do *software*; é o teste mais simples dos três, ele espera uma entrada e uma saída; é mais utilizado para garantir o funcionamento de funções e lógicas simples; geralmente, quando são necessários informações externas, são utilizados dados “mockados” simulando os parâmetros reais.
- Testes de integração: visam testar a integração entre diferentes módulos do *software*; geralmente são executados utilizando *requests* reais para garantir o funcionamento do sistema como um todo;
- Testes E2E: visam simular a interação de um usuário real com o *software*, testando todo o fluxo de interações existentes dentro dele, do começo ao fim; com isso, é possível englobar todos os módulos; é o teste mais difícil a ser executado, entretanto é o que apresenta a maior confiabilidade.

A Figura 3 apresenta uma pirâmide que sugere a proporção de testes que devem existir dentro de um sistema, como também a relação de integração e velocidade.

Figura 3 – Relação de testes de *software*



Fonte: https://miro.medium.com/max/1494/1*IqWygfNJqWQ4VCyecQ6Eg.png. Acesso em: 29 de novembro de 2020.

2.3 Banco de dados

O banco de dados é uma coletânea de dados virtual, utilizado para salvar informações. Geralmente o banco de dados é estruturado em linhas e colunas e está ligado a um Sistema de Gerenciamento de Banco de Dados (SGBD), podendo ser relacional ou não, entre eles temos PostgreSQL, Oracle Database, Firebase e MongoDB ([ORACLE, 2020](#)).

2.3.1 Linguagem de Query

Linguagem de Query é uma linguagem de programação para recuperar dados do banco enviando *queries*, por meio de *requests*. As *queries* são maneiras de solicitar as informações, entre as mais conhecidas estão SQL e GraphQL, sendo a última uma Linguagem de Query baseada em grafos ([TECHOPEDIA, 2016](#)).

2.4 CI/CD

O CI/CD é um conjunto de princípios que buscam garantir uma maior eficiência no fluxo de entrega do *software* e na integração da equipe de desenvolvimento, com o objetivo final de gerar valor da maneira mais segura possível. O CI é responsável por unificar o processo de publicação da aplicação, produzindo o *upload* constante das mudanças de código em repositórios de controlamento de versão, com a finalidade de automatizar o processo de *build* e teste da mesma. Já o CD é a continuação do CI, automatizando o processo de *deploy* da

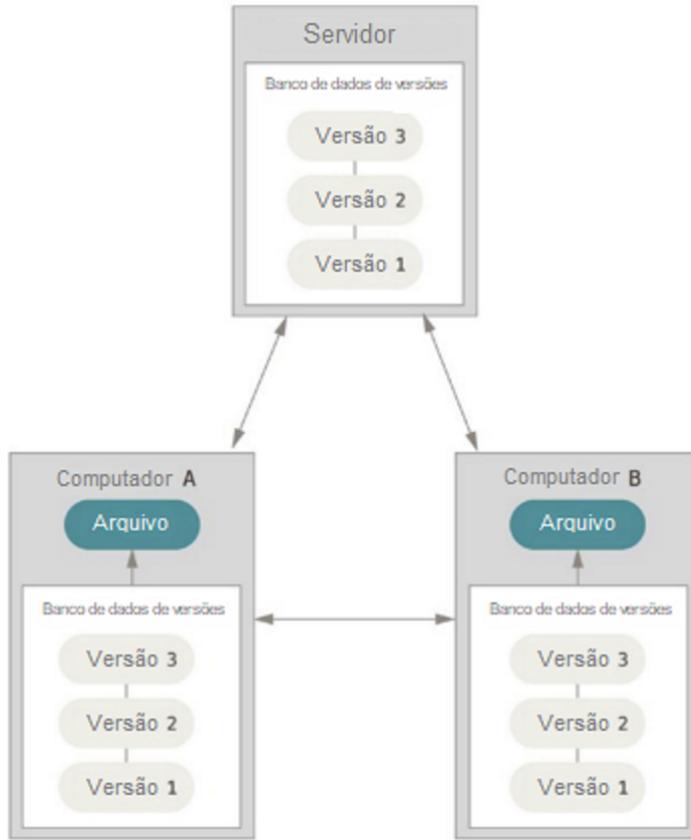
aplicação em relação as ferramentas e aos ambientes de infraestrutura escolhidos, garantindo que as mudanças feitas no código reflitam em todos os ambientes da equipe ([SACOLICK, 2020](#)).

2.5 Sistema de controle de versões

Um sistema de controle de versões (VCS) tem como objetivo salvar as mudanças feitas nos arquivos com o passar do tempo, para assim, ser possível resgatar versões anteriores. Permite realizar comparações com as mudanças que foram feitas, revertê-las e verificar quem realizou as alterações. Isso gera benefícios ao projeto, por exemplo, ao acontecer algum problema, como perda de arquivos ou geração de *bugs*, será viável recuperar o arquivo ou rastrear o *bug* e retornar o arquivo para um estado anterior. Temos três tipos principais de VCS ([CHACON; STRAUB, 2016](#)):

- *Local Version Control Systems* (LVCS): O LVCS foi uma solução desenvolvida para controlar o versionamento localmente. Possui um banco de dados para manter salvo as mudanças nos arquivos que estão versionados. Um exemplo de ferramenta de LVCS é o RCS.
- *Centralized Version Control Systems* (CVCS): Sistema de controlamento de versões que possui um único servidor contendo todos os arquivos versionados, e todas as máquinas precisarão acessar os arquivos através dele. Um dos softwares mais utilizados com essa arquitetura é o Subversion, mas essa solução ainda contém falhas graves, como um servidor único, caso ele falhe, todos os versionamentos dos arquivos serão perdidos.
- *Distributed Version Control Systems* (DVCS): Esse VCS é o mais utilizado atualmente, sendo o Git a ferramenta mais famosa que o utiliza. A principal diferença entre o DVCS e o CVCS é que as máquinas não apenas copiam os arquivos de um servidor, e sim espelham ele completamente, possuindo toda a estrutura de versionamento do repositório, como mostra a Figura 4. Isso é vantajoso, pois caso algum servidor apresente alguma falha basta apenas alguma máquina ter acessado o repositório para ter todas as informações do controlamento de versão.

Figura 4 – Exemplo da arquitetura DSVC.

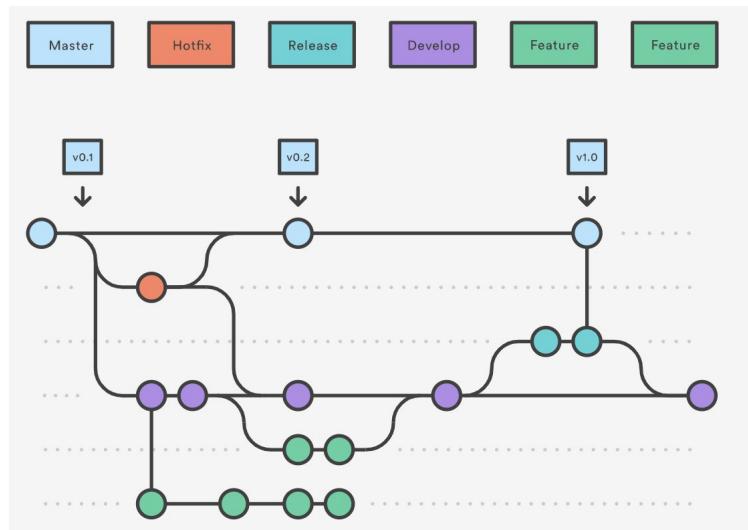


Fonte: Chacon; Straub (2016).

2.6 Gitflow

Gitflow é um fluxo de controle de versão definido por Vincent Driessen. Nele é definida uma estrutura de *branches*, onde cada *branch* tem sua funcionalidade específica (Figura 5), com o objetivo de prover uma estrutura mais definida de desenvolvimento, onde é possível rastrear as mudanças feitas no código com mais facilidade, aumentar a velocidade de desenvolvimento, facilitar a resolução de conflitos de código e ter um controle de versão mais organizado. O funcionamento do Gitflow ocorre a partir de duas *branches* principais, a *master* e a *develop*. Na *master* deve ficar o código que será publicado, e na *develop* deve ficar o código que receberá novas *features*. Cada *feature* que será desenvolvida terá sua própria *branch*, quando completado o desenvolvimento desta, a mesma será "mergeada" de volta na *develop*, e assim que a *develop* tiver *features* suficientes para uma nova publicação, ela será "mergeada" na *master*. Temos também as *branches* do tipo *release* e *hotfix*, onde a primeira tem como objetivo preparar o código para a publicação e a segunda tem como objetivo corrigir erros que foram publicados de maneira urgente ([ATLASSIAN, 2020](#)).

Figura 5 – Exemplo de estrutura de *branches* utilizando Gitflow.



Fonte: Atlassian (2020).

3 Ferramentas

3.1 Git

O Git é um DVCS que originou-se da necessidade de realizar a manutenção de grandes projetos com muitos desenvolvedores trabalhando simultaneamente. Foi criado por Linus Torvalds no ano de 2005 para auxiliar no desenvolvimento do *kernel Linux* após um conflito entre o criador e a empresa dona do DVCS proprietário BitKeeper. Construído com algumas características em foco, como um bom suporte para programação não linear e eficiência no momento de manejar altos volumes de dados, o Git se tornou a ferramenta mais popular de VCS atualmente ([CHACON; STRAUB, 2016](#)).

3.2 PostgreSQL

O PostgreSQL é um sistema *open source* para gerenciar banco de dados objeto-relacional derivado de um pacote chamado POSTGRES desenvolvido na Universidade da Califórnia. Aprimorou diversos conceitos relacionados à banco de dados, garantindo suporte completo ao SQL e contendo diversas funcionalidades modernas, como suporte à *queries* complexas, *triggers*, *data types*, entre outras ([POSTGRESQL, 2020](#)).

3.3 Docker

Docker é um *software open source* desenvolvido com o objectivo de facilitar o desenvolvimento e o *deploy* de aplicações. Tem a capacidade de criar uma plataforma unificada que possui a habilidade de compilar e executar *software* dentro de um *container* isolado.

Containers são processos que são executados isolados da máquina *host*. Dentro de um *container* há disponível um *kernel Linux*, no caso de não estar sendo executado nativamente em um sistema operacional baseado em Linux e um próprio sistema de arquivos disponibilizado por uma *Docker image*, uma vez que sendo executado em Linux, ele compartilha os recursos do *kernel* ([DOCKER, 2020](#)).

3.4 Dart

Dart é uma linguagem de programação baseada em classes, heranças únicas e orientada a objeto, possui um sistema de tipos e genéricos, que permitem uma checagem estática, ou seja, é possível verificar por erros no código antes de executá-lo. Programas desenvolvidos

em Dart são organizados em *libraries*, que são módulos encapsulados de códigos para serem utilizados pelo programa. No quesito de segurança, contém dois tipos de variáveis: pública e privada, sendo o método de categorização um *underscore* como prefixo no nome da variável. Não existe concorrência compartilhada na linguagem, ela sempre será executada em um único núcleo, o único modo de executar código simultaneamente é utilizando entidades chamadas de *isolates*, estas são medidas de concorrência, contendo seu próprio estado e comunicando-se por meio de passagem de mensagens ([DART, 2019](#)).

3.5 PHP

O PHP é uma linguagem de *script open source* muito dinâmica, que pode ser utilizada em diversas situações, como *server-side scripting*, *command line scripting* e em aplicações de *desktop*. A principal vantagem da linguagem PHP é que ela é capaz de ser embutida dentro da linguagem de marcação HTML, proporcionando uma boa experiência no desenvolvimento *web*, isso em conjunção com a capacidade de ser executada do lado do servidor, lhe permite criar conteúdos dinâmicos, trabalhar com *cookies* e gerar formulários ([PHP, 2020](#)).

3.6 GraphQL

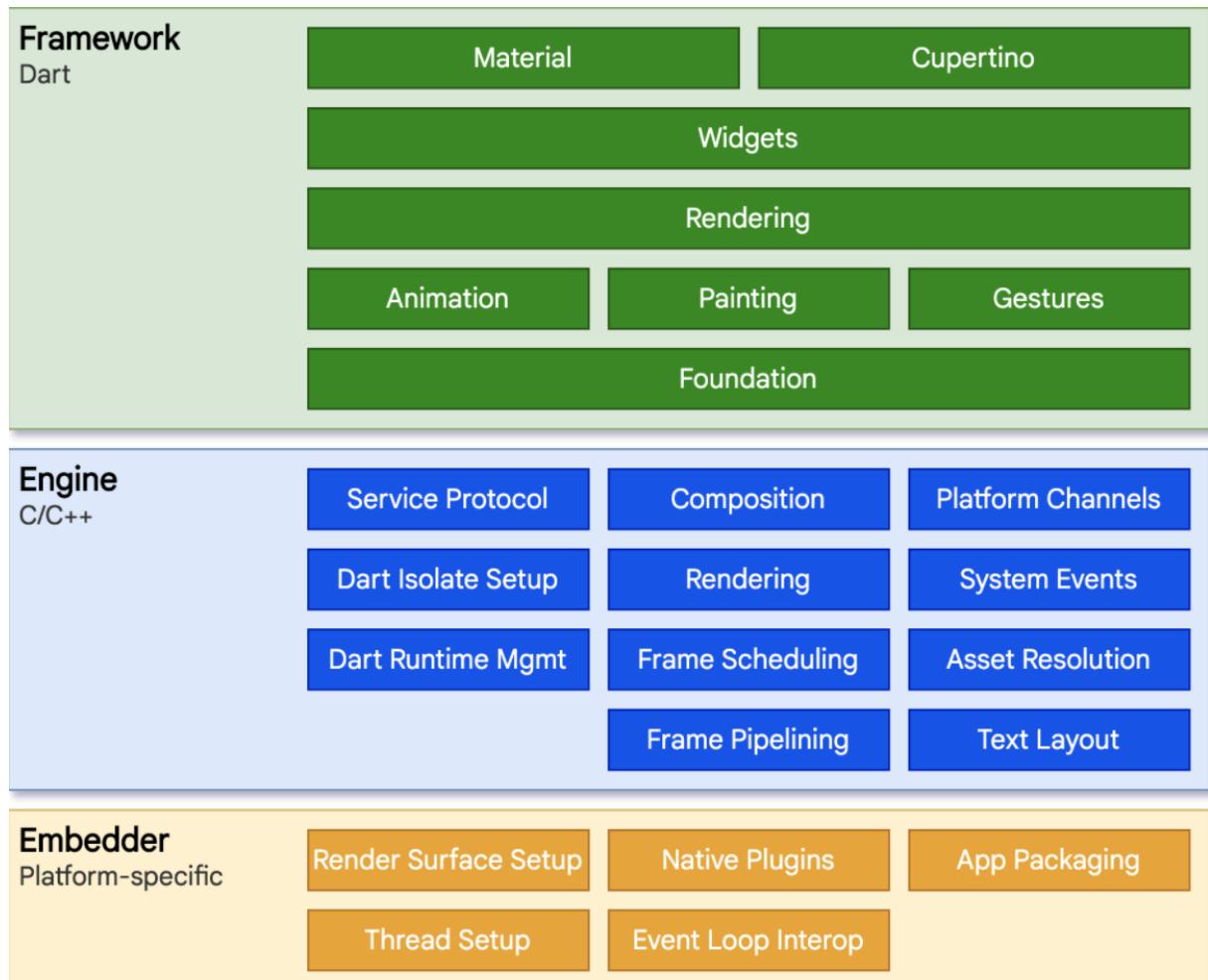
GraphQL é uma linguagem de *query open source* baseada em grafos, com a capacidade de executar *queries* utilizando um sistema de tipagem definido no *schema* de dados. Basicamente ele tem como objetivo enviar uma *string* para o servidor interpretar e retornar um *json* para o *client*. Os principais benefícios de sua utilização são: ter uma estrutura de dados definida, que facilita prever os dados que serão recebidos de uma *query*; relação de dados hierárquica, que faz com que o relacionamento entre os objetos ocorra de forma mais natural; sistema de tipos, que faz com que não seja necessário executar a *query* para saber se há erros ([KHACHATRYAN, 2018](#))

3.7 Flutter

O Flutter é uma *cross-platform toolkit* de UI *mobile*, *web* e *desktop* gratuita e de código aberto criada pela Google e lançada em dezembro de 2018. Permite que você crie aplicativos nativos com apenas uma base de código. Trabalha com o conceito de *widgets* para a construção da UI, ele determina como a *view* deve ser a partir de sua configuração e estado. Quando o estado do *widget* muda, o Flutter detecta automaticamente e realiza o *rebuild* do *widget* para a UI refletir o novo estado. Sua arquitetura é dividida em três camadas que se comunicam sem grau hierárquico, como mostra a Figura 6. A camada do *embedder* é responsável por fornecer um ponto de entrada para acessar os serviços das plataformas suportadas, atualmente ele é desenvolvido em diversas linguagens, cada uma correspondente ao sistema operacional

suportado. A *engine* tem a função de realizar a renderização dos *frames* sempre que necessário; é desenvolvido em C e C++ para ter um bom desempenho. O *framework* escrito em Dart é encarregado de compor as bibliotecas responsáveis para auxiliar o desenvolvimento ([FLUTTER, 2020](#)).

Figura 6 – Arquitetura do Flutter



Fonte: Flutter (2020).

3.8 Laravel

Laravel é um *framework* PHP de código aberto gratuito e destinado ao desenvolvimento de aplicações *web* seguindo o padrão de arquitetura MVC. Concebido utilizando diversos fundamentos, entre os principais estão ([SAXENA, 2019](#)):

- *Model*: Determina a modelagem da regra de negócio dentro da arquitetura MVC, sendo responsável por interagir com o banco de dados por meio de operações de criar, atualizar, ler e deletar.

- *View*: Tem como função apresentar os dados para o usuário separando a lógica de apresentação da de negócio. Geralmente é representada por arquivos HTML, que podem ser modularizados contendo partes de uma página da web, como arquivos contendo o *header* e *footer*.
- *Controller*: Encarregado de transitar os dados entre o *model* e a *view*.
- *Migration*: Executa o versionamento do banco de dados, com o objetivo de manter o histórico das alterações realizadas, e assim, ser possível revertê-las caso haja necessidade.
- *Eloquent ORM*: Transforma *queries* complexas do SQL em código simplificado em PHP.

3.9 Lighthouse

O Lighthouse é um *framework* cuja funcionalidade é realizar a conexão entre a Linguagem de Query GraphQL e o *framework* em PHP Laravel. Seu principal benefício é sua integração com o Eloquent ORM de forma coesa e de fácil utilização, criando *queries* automaticamente a partir dos *models* existentes ([LIGHTHOUSE, 2020](#)).

3.10 Pivotal Tracker

O Pivotal Tracker é uma ferramenta para planejamento de projetos de *software*, onde é possível visualizar as tarefas no formato de *stories* (Figura 7) e movê-las através de um *workflow* pré-definido, isso propicia uma quebra das tarefas em blocos organizados, viabilizando a geração de relatórios detalhados sobre o processo de desenvolvimento, com dados como a velocidade, as tarefas realizadas e bloqueadas, tempos dos ciclos, *burndowns*, entre outros. Dentro do Pivotal Tracker vemos quatro opções na barra lateral:

- *My Work*: É o trabalho atual do desenvolvedor, todos os *tickets* que terá que trabalhar.
- *Current/Backlog*: São todos os *tickets* que tem que ser desenvolvidos.
- *Icebox*: São os *tickets* que foram "congelados". *Tickets* "congelados" são aqueles que ainda não devem ser desenvolvidos pois existe algum impedimento, como a necessidade de ser melhor discutido.
- *Done*: Tickets finalizados.

Figura 7 – Exemplo de uma *story* no Pivotal Tracker

Mobile | Adicionar biblioteca do tensorflow

STATE: Finish Started ▾

REVIEWS: + add review

STORY TYPE: Feature

POINTS: 8 Points

REQUESTER: Thiago Hoffart

OWNERS: Thiago Hoffart +

FOLLOW THIS STORY: (1 follower)

Updated: 30 Sep 2020, 11:55pm

BLOCKERS

+ Add blocker or impediment

Fonte: Elaborado pelo autor.

3.11 Tensorflow

Tensorflow é uma plataforma *open source*, que contém um ecossistema de bibliotecas e ferramentas para a construção de aplicativos, utilizando *machine learning*. Essa plataforma possui diversas APIs com o objetivo de facilitar a construção e o treinamento de modelos de ML, e possibilitar o *debugging* e interação desses modelos. Os principais modelos utilizados

para uso em *mobile* são ([TENSORFLOW, 2020](#)):

- *Image classification*: identificação de objetos em imagem, como pessoas, animais e plantas;
- *Object detection*: detecta múltiplos objetos, utilizando *bounding box*;
- *Text classification*: categoriza textos em grupos pré-definidos. Bastante utilizado para identificar conteúdos abusivos.
- *Smart reply*: geração de sugestões de respostas em *chat* de conversas.

4 Desenvolvimento

O projeto surgiu da necessidade de otimizar orçamentos financeiros através do reconhecimento de imagens, facilitando a confecção de listas de compras do cotidiano. O objetivo foi a realização de um aplicativo que contivesse essas características. O aplicativo foi desenvolvido utilizando ferramentas como Flutter para a construção do aplicativo mobile, Laravel no desenvolvimento da API, PostgreSQL sendo a ferramenta para a conexão com o banco de dados e Tensorflow para o reconhecimento de imagem.

4.1 Pré-desenvolvimento

4.1.1 Levantamento dos requisitos

Antes de iniciar, foi necessário avaliar o escopo de projeto, e o que deveria ser desenvolvido diante da proposta realizada. Como dito anteriormente, foi feito um aplicativo financeiro com a implementação de funcionalidades de reconhecimento de imagem.

Para alcançar esse objetivo, foi importante definir todas as telas e funcionalidades que estariam dentro do aplicativo. No âmbito das telas foram definidas:

- Tela de introdução;
- Tela para realização de *login* do usuário;
- Tela para cadastro do usuário;
- Tela de *dashboard* para expor informações interessantes para o usuário;
- Tela com listagem das despesas;
- Tela com listagem dos produtos da despesa;
- Tela de criação e edição de despesas;
- Tela de criação e edição de produtos.

Após a definição de quais telas seriam construídas, foi possível definir todas as funcionalidades que seriam necessárias existir dentro do aplicativo:

- Sistema de autenticação para realização do *login*;
- *Create, Read, Update and Delete* (CRUD) de usuários;

- CRUD de despesas;
- CRUD de produtos;
- Sistema de reconhecimento de imagens;

Após a definição das telas e das funcionalidades necessárias para construí-las, o próximo passo foi modelar um banco de dados que conseguisse atender os requisitos levantados.

4.1.2 Modelagem do banco de dados

Na modelagem do banco de dados o objetivo foi definir as tabelas existentes, os campos dentro destas e a relação entre eles. Essa etapa do projeto foi muito importante, pois após a definição da modelagem é muito difícil realizar qualquer modificação na mesma, pois gera muitos efeitos colaterais, sendo necessário fazer alterações em muitos lugares.

Diante disso, a definição das tabelas e das colunas foi realizada da seguinte maneira:

- Usuario: id, nome, email, password, created_at, update_at e deleted_at;
- Despesa: id, id_usuario, nome, categoria, created_at, update_at e deleted_at;
- Produto: id, id_despesa, quantidade, valor, created_at, update_at e deleted_at;

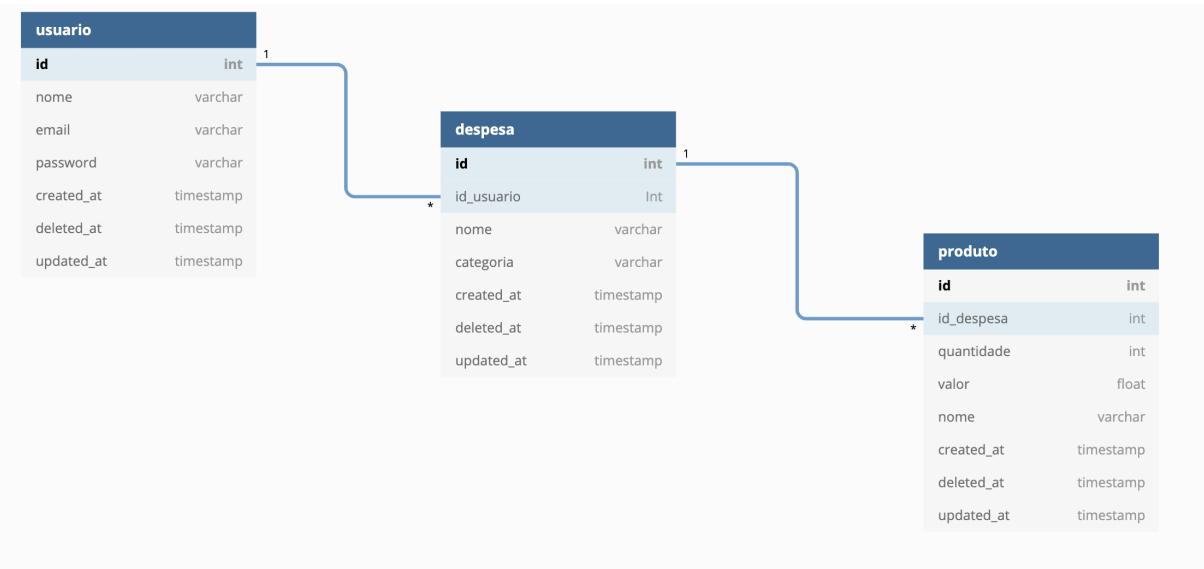
Os campos created_at, update_at e deleted_at tem como função definir quando os registros foram criados, atualizados e deletados respectivamente, por isso estão presentes em todas as tabelas. Já os campos id_usuario e id_despesa são as *foreign keys* ou *primary keys*, dependendo da tabela que tem como objetivo fazer a conexão entre duas tabelas.

A conexão entre Usuario e Despesa tem relação de 1 para N, dado que um usuário pode ter múltiplas despesas e uma despesa pode apenas ter um usuário e é representada pelo campo id_usuario dentro da tabela Despesa.

Já o campo id_despesa representa a conexão entre as tabelas despesa e produto com a relação de 1 para N, onde uma despesa pode ter vários produtos e um produto pode ter apenas uma despesa.

o Modelo Entidade Relacionamento (MER) representativo de todas as relações encontra-se na Figura 8.

Figura 8 – MER do banco de dados



Fonte: Elaborado pelo autor.

4.1.3 Geração dos projetos

Após o levantamento de requisitos e a estruturação da modelagem do banco dados, foi necessária a geração dos projetos localmente. Para isso, foi necessário a instalar o SDK do Flutter e a instalação do Composer para o Laravel.

4.1.3.1 Flutter

Após a instalação do SDK do Flutter e a configuração do mesmo, foi necessário executar o comando *flutter create tcc-2020* dentro do terminal para a criação do projeto. Em seguida, foram importadas algumas bibliotecas dentro do arquivo *pubspec.yaml*, dentre elas as principais são:

- *tflite*: biblioteca Tensorflow Lite para fazer a integração entre o Tensorflow e o Flutter;
- *pedantic*: realiza a validação do código utilizando regras descritas pela Google;
- *image_picker*: Captura imagens da câmera ou do armazenamento do celular;
- *flutter_redux*: auxilia no gerenciamento do estado da aplicação utilizando as normas do Redux;
- *charts_flutter*: biblioteca para criação de gráficos que serão utilizados na dashboard;

Em seguida iniciou-se o desenvolvimento do aplicativo, descrito posteriormente neste documento.

4.1.3.2 Laravel

Para o Laravel, o primeiro passo foi a instalação do gerenciador de pacotes Composer localmente. Foi então instalado o pacote do Laravel e gerado um novo projeto, através dos comandos *composer global require laravel/installer* e *laravel tcc-2020*. Em seguida foi instalado o Lighthouse no projeto para gerenciar a conexão entre o GraphQL e Laravel.

4.1.4 Construção da infraestrutura

A construção da infraestrutura se deu pela criação de um *container* para a aplicação do Laravel utilizando Docker, o *upload* de ambos os projetos para o Github e a implementação de um CI integrando o projeto do Flutter, Github, Pivotal Tracker e Slack, sendo o último uma plataforma de comunicação.

4.1.4.1 Docker

A implementação do Docker no projeto necessitou da instalação do mesmo e do Docker Compose. Criou-se um arquivo chamado *docker-compose.yml* na raiz do projeto em Laravel. Dentro deste arquivo temos as configurações das *Docker images* necessárias para serem executados a API e o banco de dados, entre elas encontram-se as imagens base: php-fpm; pgadmin4; e postgres.

4.1.4.2 CI

Foi estruturado um CI no projeto do Flutter, utilizando Github *actions*, sendo necessário fazer o *upload* dos projetos no Github e criar um arquivo *flutter.yml* dentro da pasta *.github/workflows*. Dentro deste arquivo, como mostra o Código 1, foi configurado o nome das *branches* em que o CI será executado, um analisador de código para se manter uma boa qualidade do mesmo e uma chave de integração com o Slack para ser notificado quando o CI foi executado com sucesso ou com falha, como na Figura 9.

Código 1 – *flutter.yml* do projeto.

```
name: Flutter CI

on:
  push:
    branches: ['feature/*']

jobs:
  build:
    runs-on: ubuntu-latest
    env:
      SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}

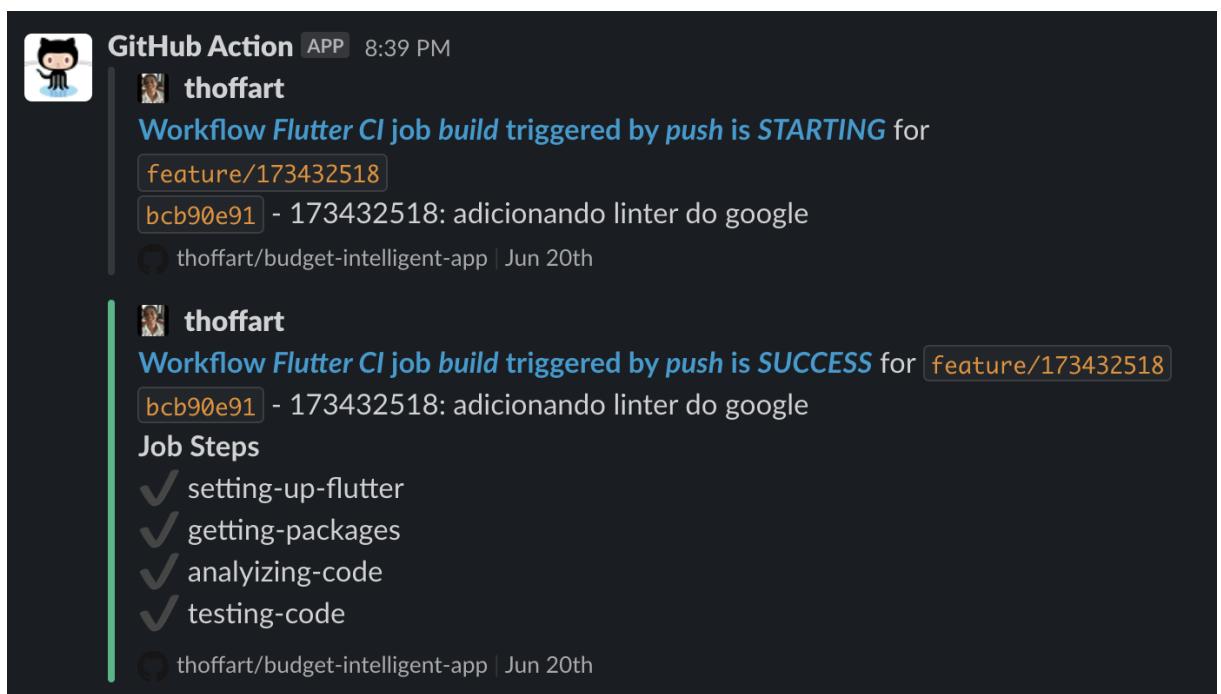
    steps:
      - uses: act10ns/slack@v1
```

```

with:
  status: starting
  channel: '#github-ci'
  if: always()
- uses: actions/checkout@v2
- name: Setup Java JDK
  uses: actions/setup-java@v1.3.0
  with:
    java-version: '12.x'
- uses: subosito/flutter-action@v1
  id: setting-up-flutter
  with:
    channel: 'stable'
- name: Getting Flutter Packages
  id: getting-packages
  run: flutter pub get
- name: Analyzing code
  id: analyzing-code
  run: flutter analyze
- name: Running tests
  id: testing-code
  run: flutter test
- uses: act10ns/slack@v1
  with:
    status: ${{ job.status }}
    steps: ${{ toJson(steps) }}
    channel: '#github-ci'
  if: always()

```

Figura 9 – Notificação do CI no Slack.



Fonte: Elaborado pelo autor.

4.2 Projeto

4.2.1 Desenvolvimento da API

4.2.1.1 *Migrations*

No desenvolvimento da API, primeiramente, foram criadas as tabelas dentro do banco de dados, por meio das *migrations*. Para gerar uma *migration* foi necessário utilizar o *artisan*, uma *interface* de linha de comando, que executa o *php artisan make:migration*. Após a execução deste, é gerado um arquivo dentro da pasta *database/migration* com o nome passado como argumento no comando. Dentro desse arquivo existe uma classe que é estendida de outra classe chamada *Migration*, que fornece diversos métodos úteis para transformar código PHP em SQL. No exemplo de Código 2 utilizamos esses métodos para criar a tabela *Usuario*. Também temos a opção de criar as tabelas utilizando diretamente código em SQL, como no exemplo de Código 3, utilizado para criar a tabela *Despesa*.

Código 2 – *Migration* de *Usuario*.

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateTableUsuario extends Migration
{
    /**
     * Run the migrations.
     */
    public function up()
    {
        Schema::create('usuario', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('email')->unique();
            $table->string('password');
            $table->string('username')->nullable();
            $table->string('nome');
            $table->string('sobrenome')->nullable();
            $table->rememberToken();

            $table->softDeletes();

            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down()
    {
        DB::Statement('DROP TABLE IF EXISTS usuario');
    }
}
```

Código 3 – *Migration* de Despesa.

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateDespesaTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        DB::Statement("CREATE TABLE despesa (
            id SERIAL PRIMARY KEY,
            id_usuario INTEGER NOT NULL REFERENCES usuario(id)
            ON UPDATE CASCADE ON DELETE RESTRICT,
            nome VARCHAR(100),
            categoria VARCHAR(100),
            created_at TIMESTAMP,
            updated_at TIMESTAMP,
            deleted_at TIMESTAMP
        );
    ");

    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('despesa');
    }
}
```

4.2.1.2 Models

Em seguida foi necessária a criação dos *models* respectivos de cada tabela. Um *model* tem como função realizar as configurações de cada tabela, como definir campos privados e públicos, chaves primárias, *timestamps* e relações entre as tabelas. A classe do *Model* segue o mesmo padrão das *migrations*: existe uma classe que estende da classe Model, com esta tendo várias funções auxiliares. O Código 4 exemplifica o *model* da tabela Despesa; dentro deste *model* há o mapeamento das relações entre as tabelas Despesa e Usuario e Despesa e Produto, assim como os campos que podem ser preenchidos, os campos protegidos e os campos que serão datas.

Código 4 – *Model* de Despesa.

```
<?php

namespace App\Models;

use BenSampo\Enum\Traits\CastsEnums;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Relations\HasMany;
use Illuminate\Database\Eloquent\SoftDeletes;

class Despesa extends Model
{
    use SoftDeletes,
        CastsEnums;

    protected $table = 'despesa';

    protected $dates = ['created_at', 'updated_at', 'deleted_at'];

    protected $guarded = [
        'id',
        'created_at',
        'updated_at',
        'deleted_at',
    ];

    protected $fillable = [
        'id_usuario',
        'nome',
        'categoria',
    ];
}

public function usuario(): BelongsTo
{
    return $this->belongsTo(Usuario::class, 'id_usuario');
}

public function produto(): HasMany
{
    return $this->hasMany(Produto::class, 'id_despesa');
}

}
```

4.2.1.3 GraphQL

Para o GraphQL é necessário definir quatro tipos de arquivos para cada tabela, sendo estes: *types*, a definição do tipo de cada campo da coluna, como *int*, *string* e *DateTime*; *queries*, ou seja, requisições para trazer dados do banco, que define todas as *queries* possíveis para a respectiva tabela, como trazer todos os dados, trazer todos os dados paginados ou trazer apenas uma linha da coluna, entre outros; *mutations* ou requisições para "mutar" os dados das tabelas, diferindo das *queries* na capacidade de alterar os dados do banco, como criar, atualizar ou deletar; e os *inputs*, que definem os dados esperados que as *queries* e as

mutations esperam receber como parâmetros. No exemplo de código 5 encontram-se os *types*, as *mutations*, as *queries* e os *inputs* da tabela Despesa.

Código 5 – *Type, queries, mutations e inputs* de Despesa.

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateTableUsuario extends Migration
{
    type Despesa {
        id: ID
        created_at: DateTime
        updated_at: DateTime
        deleted_at: DateTime
        id_usuario: Int
        usuario: Usuario @belongsTo
        produto: [ Produto ] @hasMany
        nome: String
        categoria: String
    }

    extend type Query {
        getAllDespesa(filter: DespesaFilterInput @spread):[ Despesa ]
            @all
            @middleware(checks: ["auth:api"])
    }

    extend type Mutation {
        createDespesa(input: CreateDespesaInput! @spread): Despesa
            @inject(context: "user.id" name: "id_usuario")
            @create
            @middleware(checks: ["auth:api"])
    }

    input CreateDespesaInput {
        id_usuario: Int
        nome: String @rules(apply: ["required"])
        categoria: String @rules(apply: ["required"])
    }
}
```

4.2.1.4 Testes de integração

Por fim, houve a necessidade de desenvolver alguns testes de integração para garantir o funcionamento da API. Foram desenvolvidos os testes para as *queries* e as *migrations*, onde as requisições são feitas e é verificado se o resultado é compatível com o objeto esperado. No código 6 é exemplificado um teste para a *query getDespesa*.

Código 6 – Teste da *query getDespesa*.

```
public function testGetDespesa() {
    $despesa = factory(Despesa::class)->create();
    $filter = [];
    $filter['filter'][ 'id' ] = $despesa->id;
```

```

$query = 'query GetDespesa ($filter: FindFilterInput!){
  getDespesa(filter: $filter)
    {
      id
      nome
      categoria
    }
  }';

$result = $this->graphQL($query, $filter);

$this->assertEquals(
  $despesa->id,
  $result->json('data.getDespesa.id')
);
}

```

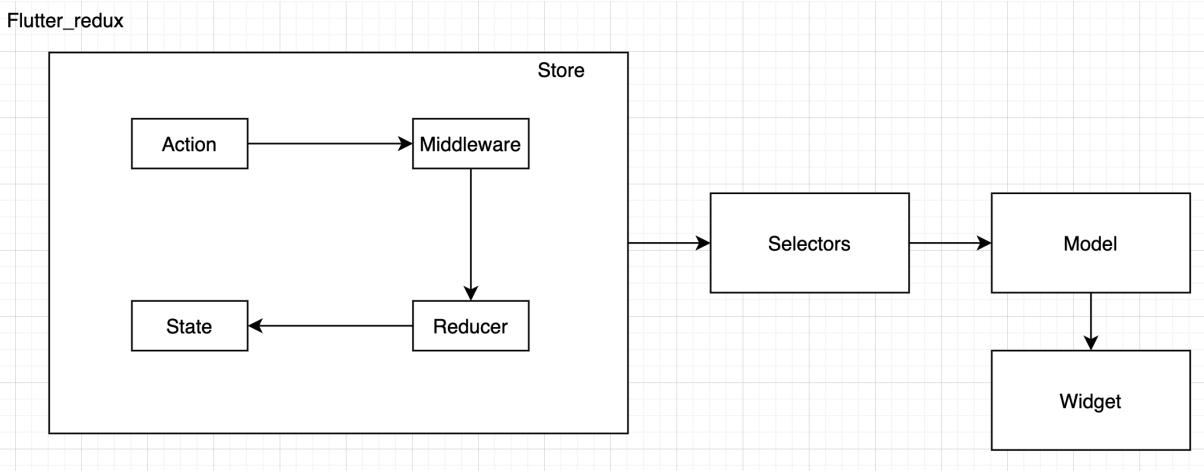
4.2.2 Desenvolvimento do aplicativo

4.2.2.1 Modelagem do *State Management*

Antes de começar o desenvolvimento das telas do aplicativo foi definida a arquitetura do estado da aplicação. Esta utilizará a biblioteca `flutter_redux` em conjunto com o estado dos *widgets* do Flutter, o primeiro sendo utilizado para o estado dos dados que precisam ser compartilhados por toda a aplicação, e o segundo para os dados específicos de uma página. A arquitetura funcionará da maneira como mostra a Figura 10: a biblioteca criou uma *store*, que contém diversos *states* modularizados com o objetivo de refletir o estado do banco, dentro desses *states* existem as variáveis, que necessitam ser acessadas a partir de diversas páginas do aplicativo. A fim de alterar os dados dos *states*, cada arquivo tem a seguinte responsabilidade:

- *Action*: ações para serem disparadas, que refletirão em mudanças nos *states*;
- *Middleware*: responsável pelos *side effects* de uma *action*, como por exemplo, requisições e disparo de outras *actions*;
- *Reducer*: altera os dados do *state*;
- *Selectors*: realiza o *Cache* de acessos à *store*, que necessitam de alguma manipulação dos dados da mesma;
- *Model*: conexão entre o *widget* do Flutter e a *store*; verifica por mudanças na mesma para avisar o *widget* da necessidade dele ser "rebuildado";
- *Widget*: contém a *view* e seu estado interno.

Figura 10 – Fluxo do estado do aplicativo.



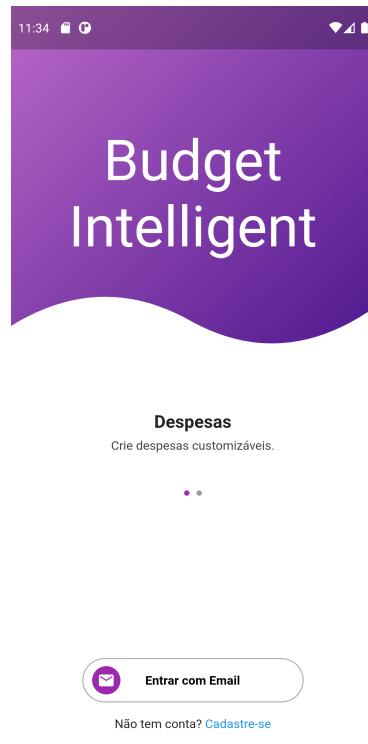
Fonte: Elaborado pelo autor.

4.2.2.2 Construção das páginas

A construção das páginas seguiram o fluxo de interação do usuário dentro do aplicativo, ou seja, primeiro foram construídas as páginas de *login* e cadastro, depois as listagens e CRUD de despesa e produto. A seguir encontram-se as telas feitas com suas respectivas características.

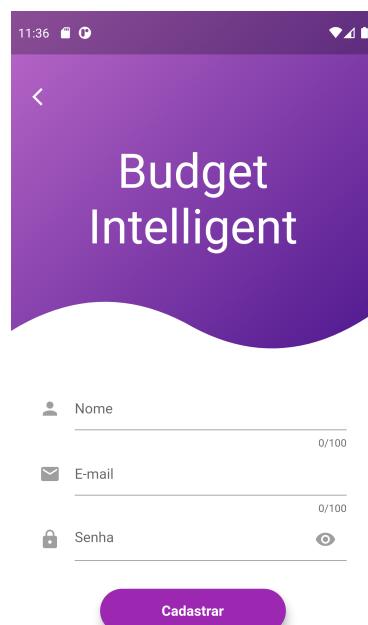
- Tela de introdução: título do aplicativo, carrossel com *features*, botão de entrar com *e-mail* e botão de cadastro, como mostra a Figura 11;
- Tela de cadastro: título do aplicativo, formulário de cadastro e botão para realizar o cadastro, apresentada na Figura 12;
- Tela de *login*: título do aplicativo, formulário de *login* e botão para realizar *login*;
- Tela de *dashboard*: *dashboard* com número total de despesas, produtos, gastos e gráfico com os gastos dos últimos sete dias, como na Figura 13;
- Tela de listagem de despesas: lista com *cards* de despesas contendo nome e categoria;
- Tela de cadastro/edição de despesa: formulário de despesa e botão para criar ou editar uma despesa;
- Tela de listagem de produto: página com os dados da despesa e listagem de *cards* de produtos clicáveis para edição, com botão de adicionar produto e um botão para editar a despesa, de acordo com a Figura 14;
- Tela de cadastro/edição de produto: Página com formulário de produto, botão para criar ou editar um produto e botões para enviar foto através da câmera ou galeria, como mostra a Figura 15;

Figura 11 – Tela de introdução



Fonte: Elaborado pelo autor.

Figura 12 – Tela de cadastro



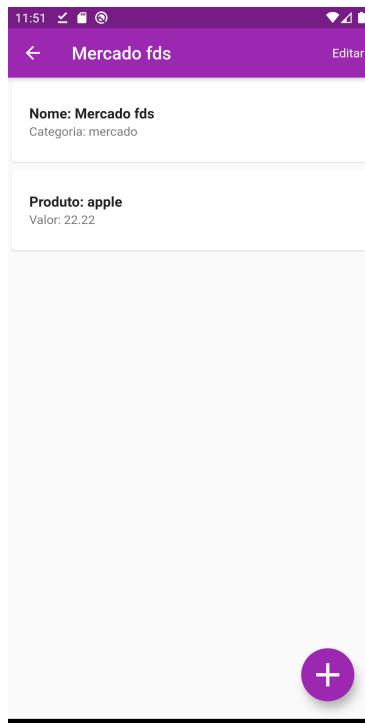
Fonte: Elaborado pelo autor.

Figura 13 – Tela de *dashboard*



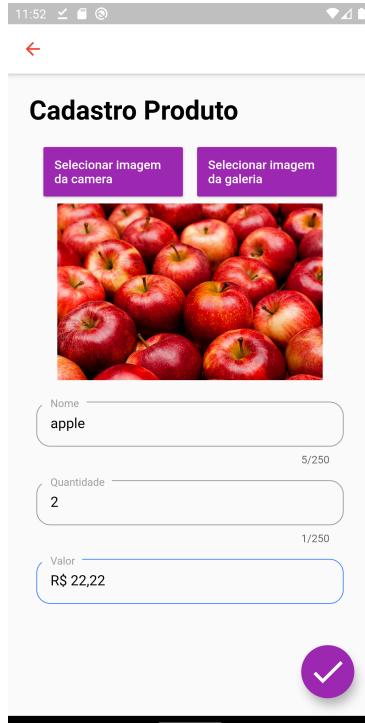
Fonte: Elaborado pelo autor.

Figura 14 – Tela de listagem de produto



Fonte: Elaborado pelo autor.

Figura 15 – Tela de cadastro/edição de produto



Fonte: Elaborado pelo autor.

4.2.2.3 Requisições à API

As requisições para a API foram feitas no *middleware* da *store*. Para isso, foi criada uma classe base, com métodos auxiliares, utilizando a biblioteca GraphQL para o Flutter. O Código 7 mostra uma requisição para a criação de produto dentro do *middleware* deste.

Código 7 – Requisição para criação de produto.

```
const String createProdutoMutation = r'''
mutation createProduto($input: CreateProdutoInput!) {
  createProduto(input: $input) {
    id
    id_despesa
    quantidade
    valor
    nome
    created_at
  }
}
''';
class ProdutoMiddleware extends MiddlewareClass {
  @override
  void call(Store store, dynamic action, NextDispatcher next) async {
    if (action is CadastroProduto) {
      final MutationOptions _createProdutoMutationOptions = MutationOptions(
        documentNode: gql(createProdutoMutation),
        variables: <String, dynamic>{
          'input': action.formValue,
        }
      );
      await next();
    }
  }
}
```

```
        }
    );
    final _createProdutoMutationResult = await BaseGraphQLClient
        .client
        .mutate(_createProdutoMutationOptions);

    if (!_createProdutoMutationResult.hasException) {
        store.dispatch(
            AddProdutoToDespesa(
                _createProdutoMutationResult.data['createProduto']['id_despesa'],
                _createProdutoMutationResult.data['createProduto']['id']
            )
        );
        store.dispatch(
            CadastroProdutoSuccess(
                _createProdutoMutationResult.data['createProduto']
            )
        );
    }
}
}
```

4.2.2.4 Testes unitários

Para se garantir o funcionamento das funções mais importantes do aplicativo, houve a necessidade de desenvolver alguns testes unitários. Dentre eles, foram desenvolvidos os testes para garantir a normalização do estado da aplicação e garantir a integração com o Tensorflow e o formulário de cadastro de produto. No código 8 é exemplificado um teste unitário que por meio de uma entrada realiza a verificação se a função de normalização do estado está correta, através da comparação do objeto retornado da função com o objeto previamente esperado.

Código 8 – Teste da função de normalização de objetos.

```
import 'dart:convert';

import 'package:dermoanamnese_app/utils/normalizer/normalizer.dart';
import 'package:test/test.dart';

void main() {
    test('Json should be normalized', () {
        final blogPosts = [
            {
                'id': 'post1',
                'author': { 'username': 'user1', 'name': 'User 1' },
                'comments': [
                    {
                        'id': 'comment1',
                        'author': { 'username': 'user2', 'name': 'User 2' },
                    },
                    {
                        'id': 'comment2',
                        'author': { 'username': 'user3', 'name': 'User 3' },
                    }
                ]
            },
        ];
    });
}
```

```

    {
      'id': 'post2',
      'author': { 'username': 'user2', 'name': 'User 2' },
      'comments': [
        {
          'id': 'comment3',
          'author': { 'username': 'user3', 'name': 'User 3' },
        },
        {
          'id': 'comment4',
          'author': { 'username': 'user1', 'name': 'User 1' },
        },
        {
          'id': 'comment5',
          'author': { 'username': 'user3', 'name': 'User 3' },
        }
      ]
    }
  ];
  final objectNormalized = Normalizer.normalizeList(
    blogPosts, 'id', [NestedObject(objectIdKey: 'id', objectKey: 'comments'),
    NestedObject(objectIdKey: 'username', objectKey: 'author')]
  );
  expect(
    jsonEncode(objectNormalized),
    '{
      "post1":{
        "id":"post1",
        "author":"user1",
        "comments":["comment1","comment2"]},
      "post2":{
        "id":"post2",
        "author":"user2",
        "comments":["comment3","comment4","comment5"]
      }
    }'
  );
});
});
}
}

```

4.2.3 Integração com o Tensorflow

Na integração com o Tensorflow foi realizada a importação da biblioteca TFLite dentro do arquivo *pubspec.yml*. Em seguida, foi criada uma classe base para fazer a conexão entre o aplicativo e a biblioteca; essa classe possui métodos que auxiliam na importação e execução de modelos, integração com formulários e acesso a imagens através do armazenamento ou câmera do dispositivo, como mostra o Código 9. Foi escolhido o modelo SSD_MobileNet que melhor se adapta ao aplicativo. Esse modelo de *object detection* permite identificar um grupo de objetos e informar suas posições a partir de uma imagem ou vídeo. Até o momento, o *plugin* do TFLite suporta apenas os modelos SSD_MobileNet e YOLO para *object detection*, foi escolhido o primeiro por sua facilidade de uso e suporte ao reconhecimento de objetos *offline*. Foi utilizado esse modelo pré-treinado com *dataset* de frutas, previamente definido para a validação do projeto. Ao utilizá-lo, após o reconhecimento do objeto, há o preenchimento automático do

nome da fruta no campo Nome. Exemplos de reconhecimento de objetos pelo aplicativo são mostrados nas Figuras 16 e 17.

Código 9 – Classe base de integração com o Tensorflow

```
import 'dart:io';

import 'package:flutter/material.dart';
import 'package:image_picker/image_picker.dart';
import 'package:tflite/tflite.dart';

class ImagePickerTflite extends StatefulWidget {
  const ImagePickerTflite({
    Key key,
    @required this.updateName,
  }) : super(key: key);

  final void Function(String) updateName;

  @override
  _ImagePickerTfliteState createState() =>
  _ImagePickerTfliteState();
}

class _ImagePickerTfliteState extends State<ImagePickerTflite> {
  File imageURI;
  String result;
  String path;
  final ImagePicker picker = ImagePicker();

  Future getImageFromCamera() async {
    var image = await picker.getImage(source: ImageSource.camera);
    var imageFile = File(image.path);
    setState(() {
      imageURI = imageFile;
      path = imageFile.path;
    });
    await classifyImage();
  }

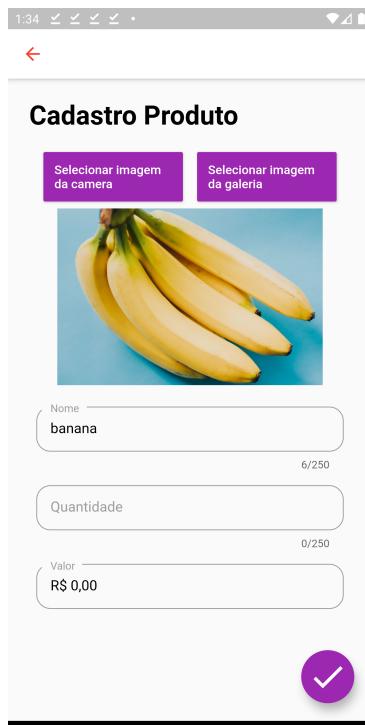
  Future classifyImage() async {
    await Tflite.loadModel(
      model: 'assets/ssd_mobilenet.tflite',
      labels: 'assets/ssd_mobilenet.txt'
    );
    var output = await Tflite.detectObjectOnImage(path: path);
    setState(() {
      result = output.toString();
    });
    widget.updateName(output.first['detectedClass']);
  }

  Future getImageFromGallery() async {
    var image = await picker
      .getImage(source: ImageSource.gallery);
    var imageFile = File(image.path);
    setState(() {
      imageURI = imageFile;
      path = imageFile.path;
    });
  }
}
```

```
    });
    await classifyImage();
}

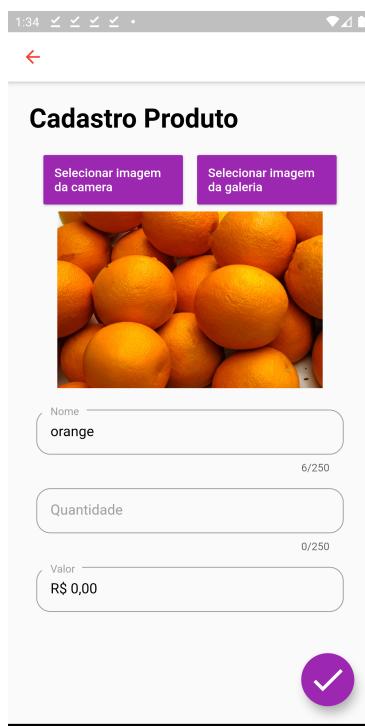
@Override
Widget build(BuildContext context) {
    return Column(
        children: <Widget>[
            Row(
                children: [
                    Expanded(
                        child: Padding(
                            padding: const EdgeInsets.all(8.0),
                            child: RaisedButton(
                                onPressed: getImageFromCamera,
                                child: Text('Selecionar imagem da camera'),
                                textColor: Colors.white,
                                color: Colors.purple,
                                padding: EdgeInsets
                                    .fromLTRB(12, 12, 12, 12),
                            ),
                    ),
                ),
            ),
            Expanded(
                child: Padding(
                    padding: const EdgeInsets.all(8.0),
                    child: RaisedButton(
                        onPressed: getImageFromGallery,
                        child: Text('Selecionar imagem da galeria'),
                        textColor: Colors.white,
                        color: Colors.purple,
                        padding: EdgeInsets.fromLTRB(12, 12, 12, 12),
                    ),
            ),
        ),
    ],
),
if(imageURI != null)
    Image.file(
        imageURI,
        width: 300,
        height: 200,
        fit: BoxFit.cover
    ),
]);
}
```

Figura 16 – Aplicativo reconhecendo bananas



Fonte: Elaborado pelo autor.

Figura 17 – Aplicativo reconhecendo laranjas



Fonte: Elaborado pelo autor.

5 Conclusão

A sociedade atual valoriza o tempo, de forma que otimizá-lo é uma tendência para a funcionalidade das tecnologias que estão sendo desenvolvidas. O uso de *softwares* que podem facilitar a vida do usuário, sendo utilizado em seu cotidiano, refletem os rumos da Ciência da Computação. É imprescindível que a produção científica se volte para as necessidades da sociedade, realizando essa integração entre o mundo acadêmico e o mundo real. Principalmente no momento atual, nunca foi tão necessária a valorização da produção científica, de forma que sua acessibilidade à população contribui para que esta seja reconhecida por sua devida importância.

O presente projeto busca integrar conceitos de *machine learning* com a realização de listas de orçamentos financeiros pelo usuário, sendo um aplicativo que facilita a vida do mesmo. O uso do reconhecimento de imagem busca otimizar tarefas antes executadas manualmente, como o reconhecimento de nomes de objetos. A construção de um aplicativo *mobile* utilizando o Tensorflow para realizar o reconhecimento de objetos e facilitar a experiência da população lidando com suas despesas diárias é uma demonstração de um uso de *machine learning* na melhoria da qualidade de vida da sociedade.

Neste trabalho foi criada uma aplicação capaz de ser executada nativamente em diversas plataformas, como *smartphones* e computadores, através da utilização do *framework* Flutter. Utilizando-se o Laravel em junção com o Lighthouse foi possível realizar a construção de uma API robusta, capaz de executar requisições otimizadas trazendo apenas os dados necessários. A validação do projeto se deu pela realização de testes de requisitos pelo autor, entretanto devido à demanda de tempo para a construção da aplicação e sua estrutura serão necessários testes mais aprofundados de reconhecimento de imagem, a fim de melhorar o desempenho e precisão da ferramenta. Portanto, a utilização dessas tecnologias modernas mostram como o surgimento de novos conceitos e paradigmas são importantes para o desenvolvimento da Ciência da Computação.

6 Próximos Passos

Como próximos passos para o projeto, é importante a realização de mais testes de reconhecimento de imagem, para que a ferramenta tenha uma melhor precisão e desempenho, além disso, é possível utilizar outros *datasets* e treinar o modelo utilizado para reconhecer outras categorias de objetos e aprimorar a presente categoria. Além disso, se faz necessário o estudo da implementação do Tensorflow na API, tendo em vista que o poder de processamento dos celulares ainda é limitado. Há também a possibilidade de publicar o aplicativo nas lojas AppStore e PlayStore. Já em relação a melhorias dentro do aplicativo, pode-se adicionar o *login* com outras plataformas, como Facebook e Google e criar um sistema de categorias de despesas mais robusto.

Referências

- ALPAYDIN, E. *Introduction to machine learning*. Cambridge, Massachusetts: The MIT Press, 2020. 683 p. (4 Ed).
- ALSING, O. *Mobile Object Detection using TensorFlow Lite and Transfer Learning*. 2018. 78 p. — Trabalho de conclusão de curso (Engenharia e Ciência da Computação) – KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Estocolmo, Suécia, 2018.
- ATLASSIAN. *Gitflow Workflow*. 2020. Disponível em: <<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>>. Acesso em: 23 de outubro de 2020.
- BENHAMIDA, A.; VARKONYI-KOCZY, A. R.; KOZLOVSZKY, M. Traffic signs recognition in a mobile-based application traffic signs using tensorflow and transfer learning technics. In: *International Conference of System of Systems Engineering*. Budapest, Hungary: IEEE-SoSE, 2020.
- CHACON, S.; STRAUB, B. *Pro Git*. Los Angeles, Califórnia: Apress, 2016. 421 p. (2 Ed).
- DART. *Dart Programming Language Specification 5th edition draft*. 2019. Disponível em: <<https://dart.dev/guides/language/specifications/DartLangSpec-v2.2.pdf>>. Acesso em: 29 de outubro de 2020.
- DOCKER. *Documentação Docker*. 2020. Disponível em: <<https://docs.docker.com/>>. Acesso em: 23 de outubro de 2020.
- FLUTTER. *Flutter*. 2020. Disponível em: <(<https://flutter.dev/docs/>)>. Acesso em: 29 de outubro de 2020.
- FREITAS, T. *Os três tipos de aprendizado no machine learning, um ramo da inteligência artificial*. 2019. Disponível em: <<https://www.startse.com/noticia/nova-economia/machine-learning-inteligencia-artificial-aprendizado>>. Acesso em 22 de Março de 2020.
- GENESERETH, M.; NILSSON, N. *Logical foundations of Artificial Intelligence*. Palo Alto, Califórnia: Morgan Kaufmann Publishers Inc., 1987. 401 p. (2 Ed).
- KHACHATRYAN, G. *What is GraphQL?* 2018. Disponível em: <<https://medium.com/devgorilla/what-is-graphql-f0902a959e4>>. Acesso em: 29 de outubro de 2020.
- LIGHTHOUSE. *Lighthouse*. 2020. Disponível em: <(<https://lighthouse-php.com>)>. Acesso em: 29 de outubro de 2020.
- MEDVIDOVIC, N.; TAYLOR, R. Software architecture: foundations, theory, and practice. In: *ACM/IEEE International Conference on Software Engineering*. Cape Town, South Africa: ICSE, 2010.
- MILLIGTON, I.; FUNGE, J. *Artificial Intelligence for games*. Boca Raton, Florida: CRC Press, 2009. 847 p. (2 Ed).

MULFARI, D.; MINNOLO, A. L.; PULIAFITO, A. Building tensorflow applications in smart city scenarios. In: *EEE INTERNATIONAL CONFERENCE ON SMART COMPUTING*. n. 3, 2017. Hong Kong, China, Anais, Hong Kong: SMARTCOMP, 2017.

ORACLE. *Banco de dados*. 2020. Disponível em: <<https://www.oracle.com/br/database/what-is-database.html>>. Acesso em: 23 de outubro de 2020.

PHP. *Documentação PHP*. 2020. Disponível em: <https://www.php.net/manual/pt_BR/intro-whatis.php>. Acesso em: 26 de outubro de 2020.

POP, D. P.; ALTAR, A. Designing an mvc model for rapid web application development. *Procedia Engineering*, v. 69, p. 1172–1179, 2014.

POSTGRESQL. *PostgreSQL 13.1 Documentation*. 2020. Disponível em: <<https://www.postgresql.org/files/documentation/pdf/13/postgresql-13-A4.pdf>>. Acesso em: 23 de outubro de 2020.

PREE, W. *Meta Patterns — A Means For Capturing the Essentials of Reusable Object-Oriented Design*. 1994. Disponível em: <<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.7935&rep=rep1&type=pdf>>. Acesso em 1 de setembro de 2019.

ROUSE, M. *State Management*. 2020. Disponível em: <<https://searchapparchitecture.techtarget.com/definition/state-management>>. Acesso em: 23 de outubro de 2020.

SACOLICK, I. *What is CI/CD? Continuous integration and continuous delivery explained*. 2020. Disponível em: <<https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html>>. Acesso em: 23 de outubro de 2020.

SAXENA, K. *What are the core concepts of Laravel*. 2019. Disponível em: <<https://www.resourcifi.com/blog/what-are-the-core-concepts-of-laravel-framework/>>. Acesso em: 29 de outubro de 2020.

TECHOPEDIA. *Query Language*. 2016. Disponível em: <[https://www.techopedia.com/definition/3948/query-language#:~:text=Query%20language%20\(QL\)%20refers%20to,extract%20data%20from%20host%20databases](https://www.techopedia.com/definition/3948/query-language#:~:text=Query%20language%20(QL)%20refers%20to,extract%20data%20from%20host%20databases)>. Acesso em: 23 de outubro de 2020.

TENSORFLOW. *Tensorflow*. 2020. Disponível em: <(<https://www.tensorflow.org/>)>. Acesso em: 29 de outubro de 2020.

TIANKAEW, U.; CHUNPONGTHONG, P.; METTANANT, V. A food photography app with image recognition for thai food. In: *ICT International Student Project Conference*. Nakhon Pathom, Thailand: ICT-ISPC, 2018.