

Universidade da Beira Interior

Departamento de Informática



Departamento de
Informática

Sistema Integrado para Gestão e Entrega ao Domicílio

Elaborado por:

Samuel Silva Dias

Orientador:

Professor Paulo Fazendeiro

22 de janeiro de 2025

Agradecimentos

Este trabalho só é possível devido à ajuda do professor Paulo Fazendeiro, todos os meus professores da Universidade da Beira Interior Universidade da Beira Interior (UBI), tal como os meus familiares, amigos e colegas de curso, uma vez que nos oferecem todo o apoio possível para a realização deste e de muitos outros trabalhos.

Conteúdo

Conteúdo	iii
Lista de Figuras	v
Lista de Tabelas	vii
1 Introdução	1
1.1 Enquadramento e Motivação	1
1.2 Objetivos	1
1.3 Organização do Documento	2
2 Estado da Arte e Tecnologias utilizadas	3
2.1 Introdução	3
2.2 Ferramentas Existentes	3
2.2.1 Principal problema das ferramentas existentes	5
2.2.2 Tecnologias comuns em aplicações relacionadas	6
2.2.2.1 Geolocalização e cartografia	6
2.2.2.2 Sistemas de pagamento	7
2.2.2.3 Infraestrutura de <i>back-end</i>	7
2.2.2.4 Comunicação e notificações	7
2.3 Ferramentas Utilizadas na <i>Delivery APP</i>	7
2.3.1 <i>Flutter</i>	8
2.3.2 <i>Firebase</i>	8
2.3.3 <i>Google Maps Application Programming Interface (API)</i>	8
2.3.4 API para pagamentos online	9
3 Implementação e Testes	11
3.1 Introdução	11
3.2 Arquitetura de Software	12
3.2.1 Diagramas de caso de uso	12
3.2.2 Diagramas de Atividade	14
3.3 Modelo Relacional da base de dados	16
3.4 <i>Bloc's</i>	18

3.4.1	O que é um BLoC?	18
3.4.2	Onde foram utilizado os Business Logic Component (BLoC)'s?	18
3.4.3	<i>Sign Up</i> BLoC's	18
3.4.4	<i>Sign In</i> BLoC's	21
3.4.5	<i>Get Product</i> BLoC	26
3.4.6	Upload Imagem BLoC	32
3.4.7	<i>Create product</i> BLoC	34
3.5	Conexão à base de dados da Firebase	37
3.5.1	Conexão à <i>Firebase</i>	37
3.5.1.1	Aplicação Móvel	38
3.5.1.2	Aplicação WEB	40
3.5.2	Classe <i>User</i>	41
3.5.3	Classe <i>Product</i>	47
3.5.3.1	Funções Extra	52
3.5.4	Google Maps API e <i>urlLauncher</i>	61
3.5.4.1	Google Maps API	62
3.5.5	Dependência <i>urlLauncher</i>	65
3.6	Multi Linguagem	67
3.6.1	Ficheiros XML	67
3.6.2	<i>Azure Translation API</i>	71
3.7	API para pagamentos online	74
3.7.1	<i>Braintree</i>	74
3.8	Testes	77
3.9	Conclusões	79
4	Conclusão	81
4.1	Conclusões Principais	81
4.2	Trabalho Futuro	82
Bibliografia		83
Anexo		85
Resultados dos testes de utilização		85

Lista de Figuras

3.1	Arquitetura do Sistema	11
3.2	Diagrama Caso de Uso Admin	13
3.3	Diagrama Caso de Uso Estafeta	13
3.4	Diagrama Caso de Uso Cliente	13
3.5	Diagrama de Atividade de um Admin	14
3.6	Diagrama de Atividade de um Cliente	15
3.7	Diagrama de Atividade de um Estafeta	16
3.8	Diagrama de Entidade-Associação	17
3.9	Login WEB APP	25
3.10	Login Android	26
3.11	Lista Produtos Android	31
3.12	Lista Produtos WebApp	32
3.13	Upload	34
3.14	Criar Produto	37
3.15	Carrinho	56
3.16	Detalhes e Feedback	58
3.17	Escolha da localização	65
3.18	Resultado	67
3.19	APP em Inglês	73
3.20	APP em Português	73
3.21	Processo de Pagamento do Braintree	77
1	A aplicação funciona bem em diferentes dispositivos	85
2	A interface da aplicação é intuitiva e fácil de usar.	86
3	A aplicação responde rapidamente aos comandos.	86
4	É fácil encontrar as funcionalidades que preciso.	87
5	O processo de registo/login foi simples e eficaz.	87
6	O processo de pesquisa e acesso aos produtos é eficiente.	88
7	O métodos de pagamento disponíveis são práticos e seguros.	88
8	Recomendaria esta aplicação a outras pessoas.	89

Listas de Tabelas

2.1 Comparação das funcionalidades das plataformas de entrega ao domicílio	5
--	---

Acrónimos

GPS	Global Positioning System
UI	<i>User Interface</i>
API	<i>Application Programming Interface</i>
PCI DSS	PCI Security Standards Council
BLoC	Business Logic Component
URL	UUniform Resource Locator
XML	Extensible Markup Language

Capítulo

1

Introdução

1.1 Enquadramento e Motivação

Hoje em dia, a conveniência e a rapidez dos serviços de entrega ao domicílio tornaram-se elementos essenciais para o sucesso das empresas de restauração. Os restaurantes enfrentam o desafio de atender a uma demanda crescente por soluções que integrem geolocalização, gestão de pedidos e entregas em tempo real. Este cenário motivou o desenvolvimento deste projeto, que visa criar uma aplicação móvel para clientes e uma aplicação web para restaurantes, permitindo uma gestão eficiente de produtos, encomendas e entregas.

1.2 Objetivos

O principal objetivo deste projeto é desenvolver um sistema integrado composto por uma aplicação móvel para clientes, que lhes permita pesquisar restaurantes, fazer encomendas e acompanhar as entregas em tempo real, e uma aplicação web para restaurantes, que ofereça funcionalidades de gestão de produtos, acompanhamento de encomendas e organização de entregas de forma prática e eficiente. Além disso, o objetivo é implementar características inovadoras como a geolocalização precisa, sistemas de pagamento seguros e bases de dados em tempo real para sincronização contínua entre dispositivos.

1.3 Organização do Documento

Este relatório está estruturado da seguinte forma:

- **Introdução:** Apresenta o enquadramento, motivação e objetivos do projeto.
- **Tecnologias utilizadas e estado da arte:** Descreve as ferramentas e tecnologias utilizadas, como a geo-localização e cartografia, sistemas de pagamento e bases de dados em tempo real, bem como a exploração dos objetivos existentes na área.
- **Implementação e testes:** Detalha o processo de desenvolvimento, os desafios encontrados e os resultados obtidos durante a fase de testes.
- **Conclusões e Trabalho Futuro:** Reflete sobre os resultados alcançados, as lições aprendidas e as possibilidades de melhorar e expandir o projeto no futuro.

Capítulo

2

Estado da Arte e Tecnologias utilizadas

2.1 Introdução

Neste capítulo, focamos em perceber quais são as ferramentas de entrega ao domicílio que existem e o que diferem entre elas. Esta pesquisa vai permitir obter uma ideia quais as funcionalidades que devem estar na nossa aplicação, mas também perceber onde a aplicação pode se destacar das já existentes. Por fim, direcionamos o foco para as tecnologias utilizadas em cada uma das ferramentas já existentes, que podem ser úteis para a aplicação.

2.2 Ferramentas Existentes

As aplicações de entrega ao domicílio surgiram como resposta à crescente procura de comodidade e agilidade no consumo de alimentos e produtos, impulsionada não só pelo avanço da tecnologia móvel, mas também devido à pandemia. Anteriormente, os serviços de entrega ao domicílio eram limitados, visto que só era possível através de chamadas telefônicas e geralmente restritos a grandes redes de restaurantes. Com o aumento do uso dos *smartphones* e o desenvolvimento de plataformas digitais, empresas como a *iFood* (ifood.com.br), *Uber Eats* (ubereats.com), *Rappi* (rappi.com.br), *Glovo* (glovoapp.com) e a *DoorDash* (www.doordash.com) transformaram o mercado de entregas ao domicílio, tornando o processo mais acessível e eficiente.

O *iFood*, lançado no Brasil em 2011, foi pioneiro na digitalização do processo de pedidos, conectando restaurantes e consumidores numa plataforma centralizada. Logo depois, o *Uber Eats*, lançado em 2015, trouxe a experiência

global do *Uber* para o setor de entrega ao domicílio, usando a sua especialização em logística e transporte para oferecer entregas rápidas e eficientes. A *Rappi*, fundada em 2015 na Colômbia, expandiu o conceito de entrega, permitindo não só a entrega de alimentos, mas também de uma vasta gama de produtos, incluindo medicamentos, eletrônica e serviços personalizados.

Por outro lado, a *Glovo*, lançada em 2015 em Espanha, seguiu uma abordagem semelhante à da *Rappi*, mas com um foco na entrega de “qualquer coisa”. A Glovo destacou-se rapidamente pela sua flexibilidade, permitindo que os utilizadores solicitasse a entrega de qualquer tipo de produto, desde mercearias a medicamentos e pequenos serviços, tornando-se uma referência no mercado europeu e expandindo a sua presença global.

Nos Estados Unidos, a *DoorDash*, fundada em 2013, também ganhou destaque ao liderar o mercado de entregas no país. Inicialmente focada em pequenos negócios e restaurantes locais, a DoorDash cresceu rapidamente, desenvolvendo um modelo eficiente de logística e oferecendo opções como entregas agendadas e parcerias exclusivas com grandes redes de restaurantes. Além disso, a empresa investiu fortemente em tecnologias como otimização de rotas e inteligência artificial, tornando-se um dos principais alvos do cenário global de entregas.

Estas aplicações revolucionaram o setor, oferecendo mais opções aos consumidores, facilitando o funcionamento de pequenos e grandes restaurantes e criando oportunidades de rendimento para estafetas independentes. Além disso, integraram tecnologias como geolocalização, inteligência artificial e sistemas de pagamento digital, tornando o processo de entrega mais rápido e seguro.

	iFood	Uber Eats	Rappi	Glovo	DoorDash
Foco em alimentos	X	X	X	X	X
Entrega de medicamentos			X	X	
Entrega de eletrónica			X	X	
Entrega de qualquer tipo de produto			X	X	
Serviços personalizados			X	X	
Entregas rápidas	X	X	X	X	X
Parcerias com grandes redes	X	X			X
Otimização de rotas		X			X
Geolocalização	X	X	X	X	X
Inteligência artificial		X			X
Sistemas de pagamento digital	X	X	X	X	X

Tabela 2.1: Comparação das funcionalidades das plataformas de entrega ao domicílio

2.2.1 Principal problema das ferramentas existentes

Atualmente, todas as aplicações de entrega ao domicílio cobram taxas entre 15% e 30% por cada encomenda efetuada. Estas taxas são cobradas aos restaurantes como uma comissão sobre o valor total da encomenda, representando uma parte significativa das suas margens de lucro. Para as pequenas empresas e restaurantes locais, esta percentagem elevada torna muitas vezes o modelo de entrega insustentável, uma vez que põe em risco os seus ganhos. Em muitos casos, os restaurantes são obrigados a aumentar os preços dos seus menus para cobrir estes custos adicionais.

Além disso, esta dependência de plataformas externas pode enfraquecer a ligação direta entre o restaurante e os seus clientes, uma vez que a plataforma atua como intermediária e interfere frequentemente na personalização da experiência de consumo. Por esse motivo, muitos estabelecimentos hesitam em aderir a essas plataformas ou optam por limitar a sua presença nelas, principalmente se já possuem o seu próprio sistema de delivery ou estão em busca de alternativas mais viáveis para atender à demanda.

Perante este cenário, a necessidade de um estabelecimento local desenvolver a sua própria aplicação de entrega ao domicílio surge como uma opção viável e estratégica. Ao criar a sua própria solução, os restaurantes podem eliminar as comissões impostas pelas plataformas externas e maximizar os seus lucros. Além disso, esta abordagem permite que o estabelecimento tenha controlo total sobre a experiência do cliente, desde a conceção da *interface* até às opções de personalização da encomenda.

Ao implementar a sua própria plataforma, os estabelecimentos ganham

maior flexibilidade para ajustar os preços, gerir as suas operações conforme a procura e adaptar-se rapidamente às necessidades do mercado, sem depender de intermediários que impõem muitas vezes restrições.

2.2.2 Tecnologias comuns em aplicações relacionadas

Estas ferramentas têm a tendência de utilizar várias ferramentas em comum, por exemplo:

2.2.2.1 Geolocalização e cartografia

1. **APIs de localização:** As plataformas utilizam serviços como o Google Maps e o Mapbox, essenciais para acompanhar as entregas em tempo real, otimizar as rotas e visualizar as localizações. Estes serviços ajudam a determinar a melhor rota para os motoristas de entregas, tendo em conta o tráfego e as condições das estradas.
2. **Geolocalização:** A tecnologia Global Positioning System (GPS) nos *smartphones* permite um seguir precisamente os estafetas e caso necessário, os consumidores, melhorando a o cálculo das estimativas do tempo de entrega.

2.2.2.2 Sistemas de pagamento

1. **Serviços de pagamento:** É realizada a integração da aplicação com serviços de pagamento como o Stripe e o PayPal, bem como os sistemas de pagamento móvel (Apple Pay, Google Pay), garantem transações seguras e rápidas. Estes sistemas facilitam o pagamento *online*, permitindo aos utilizadores efetuar pagamentos através da aplicação.
2. **Encriptação:** As plataformas utilizam fortes práticas de encriptação e cibersegurança para proteger os dados de pagamento e as informações pessoais dos utilizadores.

2.2.2.3 Infraestrutura de *back-end*

1. **Bases de dados em tempo real:** Sistemas de base de dados como o *Firebase* ou o *MongoDB* são utilizadas para garantir que a informação sobre encomendas e entregas é atualizada em tempo real, o que permite uma comunicação eficiente entre restaurantes, estafetas e clientes.
2. **APIs e microsserviços:** É utilizada a arquitetura de microsserviços, que permite que diferentes partes da aplicação, como a gestão de encomendas, o acompanhamento e o sistema de pagamento sejam desenvolvidas e escaladas de forma independente, melhorando a flexibilidade e a eficiência da mesma.

2.2.2.4 Comunicação e notificações

1. **Notificações Push:** As plataformas utilizam serviços de notificação *push* para manter os utilizadores informados sobre o estado das encomendas, promoções e atualizações em tempo real.

2.3 Ferramentas Utilizadas na *Delivery APP*

Ao desenvolver uma aplicação de entrega ao domicílio, a escolha da tecnologia desempenha um papel fundamental para garantir o desempenho, a segurança e, eventualmente, uma boa experiência do utilizador. As ferramentas foram selecionadas não só para demonstrar tudo o que foi aprendido durante a licenciatura, mas também para aprofundar e conhecer novas tecnologias e metodologias de desenvolvimento.

2.3.1 *Flutter*

O Flutter é uma *framework* para desenvolvimento de *User Interface* (UI) criada pela Google, que permite criar aplicações nativas para várias plataformas, como Android e iOS, a partir de um único código-fonte. Utiliza a linguagem *Dart*, conhecida pela sua eficiência e facilidade de aprendizagem. Com o Flutter, é possível desenvolver *interfaces* responsivas e de alto desempenho que mantêm uma aparência consistente em diferentes plataformas. O Flutter também oferece uma vasta biblioteca de componentes visuais e suporte para animações, facilitando a criação de uma experiência de utilizador moderna e interativa. Por fim, o Flutter utiliza o conceito de hot reload, que permite aos programadores verem as alterações realizadas no código em tempo real, sem terem de recompilar toda a aplicação. Isto acelera o processo de desenvolvimento e *debug*.

2.3.2 *Firebase*

Firebase é uma plataforma de desenvolvimento de aplicações móveis fornecida pela Google. Para este projeto, será utilizada como base de dados em tempo real, permitindo a sincronização instantânea de informação entre utilizadores e servidores. Contará com as seguintes coleções:

- *Cart*;
- *feedbacks*;
- *product*;
- *requests*;
- *users*;

O Firebase também oferece uma série de serviços adicionais, como autenticação de utilizadores, armazenamento de arquivos, funções serverless (Cloud Functions) e notificações *push*, possibilitando a criação de uma aplicação escalável e segura. A integração com o Flutter é simples e bem documentada, acelerando o desenvolvimento e reduzindo a complexidade da infraestrutura.

2.3.3 *Google Maps Application Programming Interface (API)*

A API do Google Maps será fundamental para implementar funcionalidades de geolocalização e navegação na aplicação de entregas ao domicílio. Com

esta API é possível apresentar mapas personalizados, traçar rotas de entrega, calcular distâncias entre a localização da encomenda e o cliente e fornecer atualizações de localização em tempo real. A API também permite a integração de funcionalidades como a geocodificação (conversão de endereços em coordenadas geográficas) e a apresentação de marcadores e camadas de dados, essenciais para otimizar a operação de entrega do produto. Neste projeto, a API será utilizada com a finalidade que o cliente consiga dar *pin-point* da sua localização e para que os estafetas do estabelecimento sejam capazes de exportar a morada ou as coordenadas geográficas diretamente da *web app* para o *Google Maps*, ou seja, facilitar o uso da aplicação para todos os utilizadores.

2.3.4 API para pagamentos online

A API de pagamentos via cartão de crédito permitirá que os utilizadores efetuem transações de forma segura e eficiente. Através desta API, a aplicação será capaz de integrar diferentes métodos de pagamento, como cartões de crédito e outros métodos de pagamento locais. A API escolhida deve suportar transações seguras (conforme o PCI Security Standards Council (PCI DSS)), bem como funcionalidades como *tokens* de segurança para proteger os dados dos utilizadores. A integração de uma solução de pagamento fiável é essencial para garantir uma experiência de compra tranquila e segura, aumentando a confiança dos clientes e oferecer mais hipóteses que o pagamento no ato de entrega da encomenda.

Capítulo

3

Implementação e Testes

3.1 Introdução

A fase de implementação e testes tem um papel crucial no ciclo de desenvolvimento de software, uma vez que é aqui que a conceção e o planeamento começam a materializar-se num produto funcional e fiável. A implementação, ou codificação, é o processo de transformar a lógica e o design previamente definidos em código executável, seguindo normas e melhores práticas para garantir eficiência e escalabilidade, logo é necessário não só garantir que existe um plano que a aplicação deve seguir, tal como projetar toda a arquitetura do *software*. Para isso seja possível, foi criado um gráfico que resume o funcionamento do sistema, que permite simplificar o processo de desenvolvimento de ambas as aplicações, mas também da arquitetura de *software*.

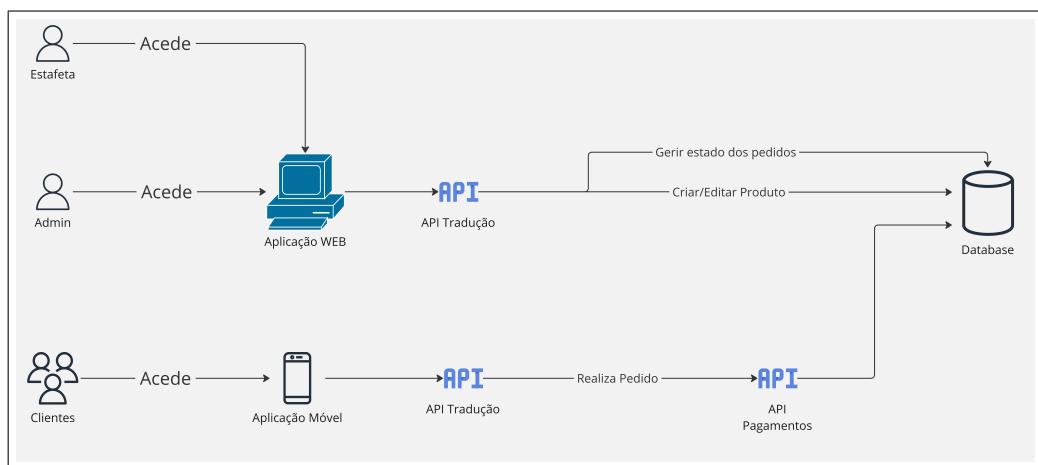


Figura 3.1: Arquitetura do Sistema

A fase de teste é essencial para identificar falhas e validar se o software cumpre os requisitos especificados. Os testes verificam a funcionalidade, o desempenho, a segurança e a facilidade de utilização da aplicação, para detectar e corrigir erros antes do lançamento. Os testes automatizados e manuais garantem que o produto final é robusto, seguro e corresponde às expectativas dos utilizadores. A integração destes processos permite desenvolver sistemas mais seguros e eficientes, essenciais para o sucesso de qualquer aplicação moderna.

Neste capítulo vai ser explicado como foi tudo realizado mas também de que forma foi testado para garantir que não existe nenhum tipo de erro e que o produto é robusto.

3.2 Arquitetura de Software

A arquitetura do software é importante para o desenvolvimento de qualquer sistema complexo. Define a estrutura, os componentes, as *interfaces* e as interações para dar origem a um *software* funcional e eficiente. Sem uma arquitetura bem planeada, um sistema pode tornar-se difícil de manter, escalar e evoluir, além de ser propenso a erros e falhas.

Na questão da aplicação de entregas ao domicílio, é fundamental garantir que planeamos e definimos como cada utilizador da aplicação vai se comportar e agir na aplicação. Para isso foram realizados os diagramas de caso de uso e de atividade para um cliente, um admin e um estafeta. Assim, é possível garantir que todas as funcionalidades de ambas as aplicações sejam eficientes para cada utilizador.

3.2.1 Diagramas de caso de uso

Nos diagramas de caso de uso, foi necessário explicar o que o sistema faz para cada utilizador. Por exemplo, o admin ao aceder à aplicação *WEB* pode criar um produto, editar um produto existente, alterar o estado do pedido e verificar dados estatísticos, enquanto um estafeta está apenas limitado à atualização do estado da encomenda. Já o cliente, após o *login* pode decidir fazer o seu pedido, escrever ou indicar a sua morada e escolher o seu método de pagamento.

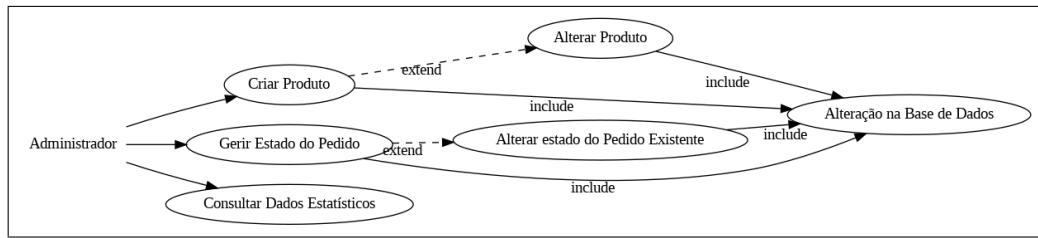


Figura 3.2: Diagrama Caso de Uso Admin

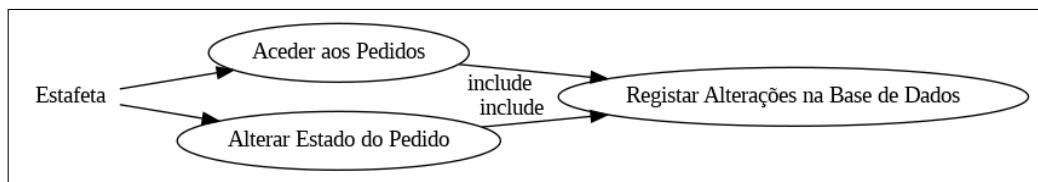


Figura 3.3: Diagrama Caso de Uso Estafeta

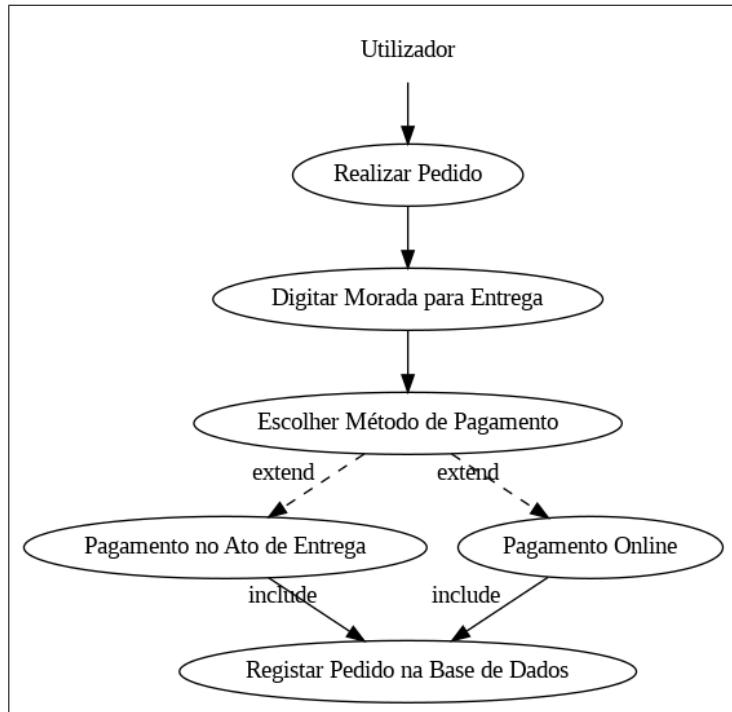


Figura 3.4: Diagrama Caso de Uso Cliente

3.2.2 Diagramas de Atividade

Após os diagramas de caso de uso, é necessário realizar os diagramas de atividade. Os diagramas de atividade servem para explicar ao possível cliente e equipa de desenvolvimento como as funcionalidades são implementadas. Descrevem o fluxo de atividades, as decisões e as interações que ocorrem para realizar um determinado caso de utilização.

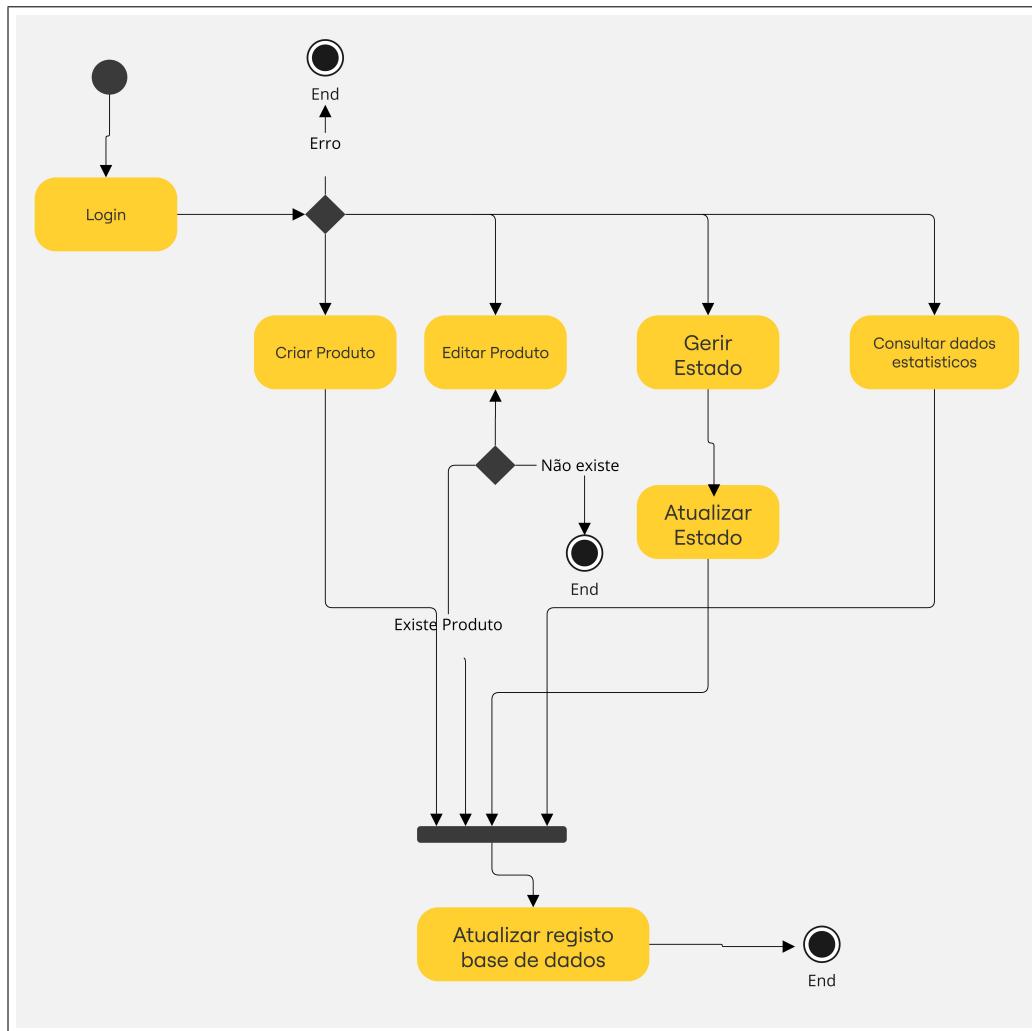


Figura 3.5: Diagrama de Atividade de um Admin

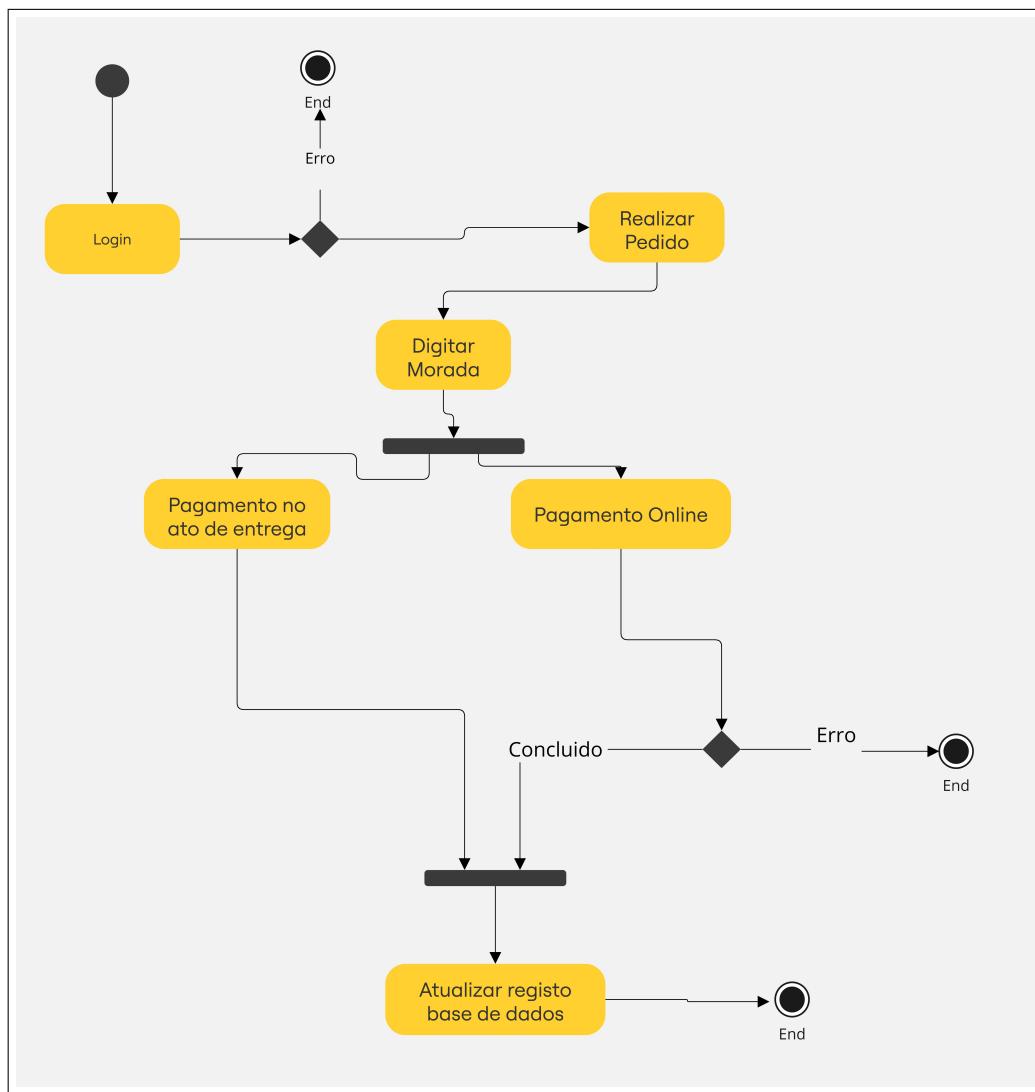


Figura 3.6: Diagrama de Atividade de um Cliente

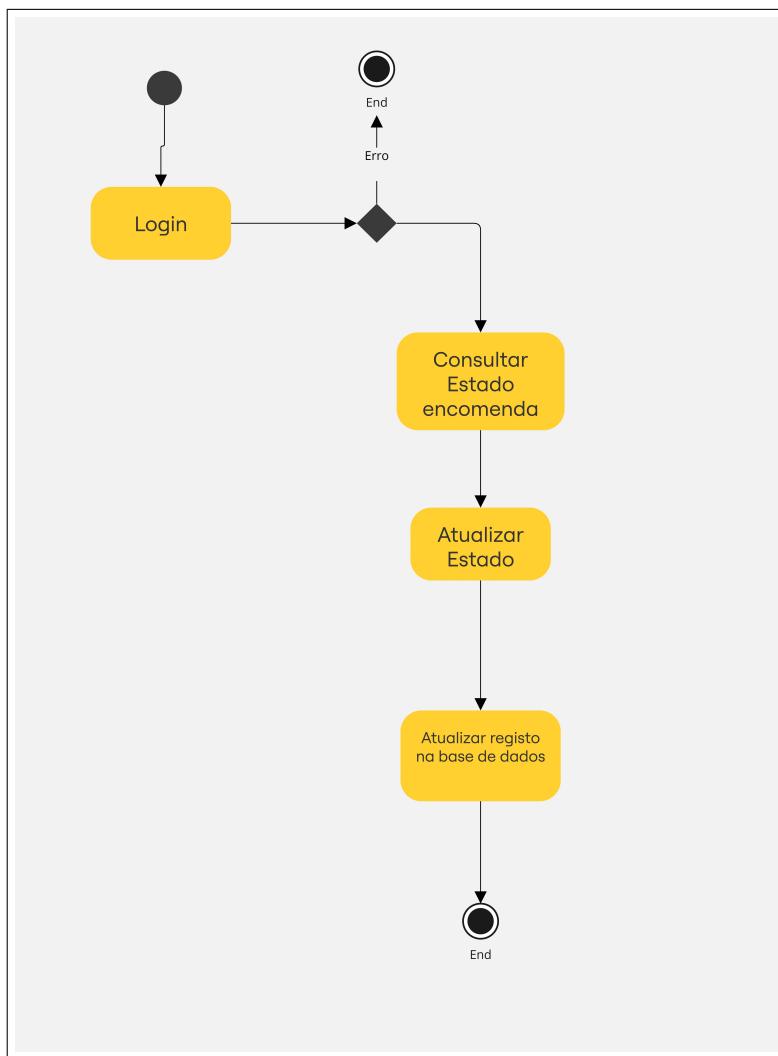


Figura 3.7: Diagrama de Atividade de um Estafeta

3.3 Modelo Relacional da base de dados

Como referido no capítulo 2, a aplicação vai usar o *Firebase* como base de dados para atualizar os dados em tempo real. Para que isso seja possível e que respeite os dados referidos na secção 2.3.2 é necessário criar um diagrama de Entidade-Associação. Esse diagrama vai permitir criar e desenvolver as aplicações de forma eficiente sem ter que realizar várias alterações em caso de futuras implementações.

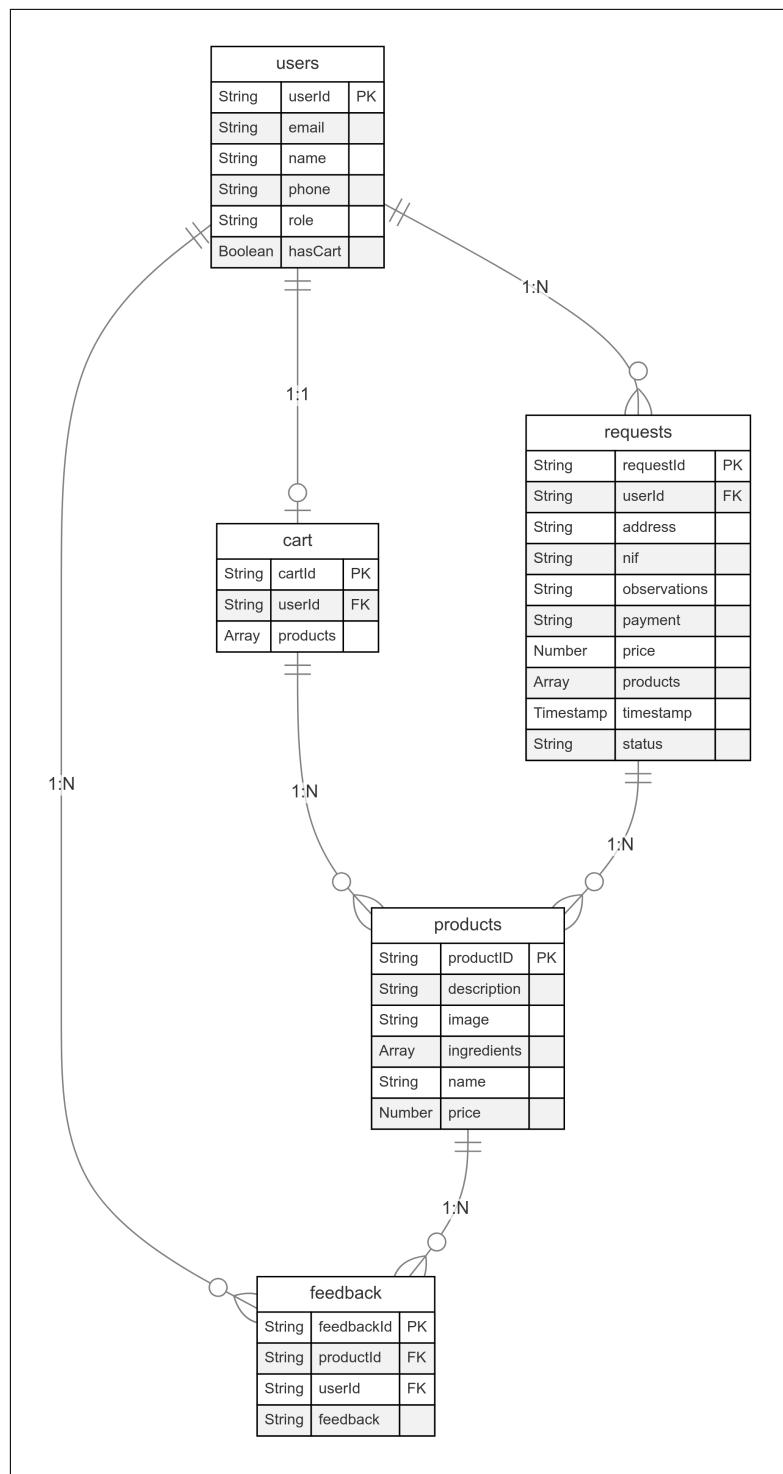


Figura 3.8: Diagrama de Entidade-Associação

3.4 *Bloc's*

3.4.1 O que é um BLoC?

Um Business Logic Component (BLoC) é um padrão de conceção que permite aos programadores gerir de forma eficiente e conveniente o estado das suas aplicações sem um acoplamento estreito entre a apresentação (vista) e a lógica. Também pretende que seja possível utilizar a mesma lógica em vários widgets. Foi mencionado pela primeira vez pelo *Google* no *Google I/O* em 2018 e, desde então, tornou-se o padrão de facto para muitos desenvolvedores quando se trata de soluções de gestão de estado no desenvolvimento de aplicações no Flutter. Ele usa eventos e estados, com o objetivo de manter o código mais modular, facilitando o teste e a reutilização de componentes.

Numa aplicação que use BLoC's, os widgets não contêm lógica diretamente. Eles comunicam com o BLoC através de eventos, que representam ações do utilizador, como pressionar um botão ou carregar dados de uma API. Quando um evento é enviado para o BLoC, este processa esta informação, aplica a lógica empresarial e emite um novo estado. Os widgets, por sua vez, “escutam” esse fluxo de estados, através de *stream's*, que é um recurso nativo do *Dart* e assim que o BLoC emite um estado atualizado, o widget é notificado e reconstrói a *interface* segundo o novo estado. Este mecanismo de comunicação, muitas vezes implementado com um *StreamController*, que separa a gestão do estado da *interface* gráfica, promovendo um código mais limpo e fácil de testar.

3.4.2 Onde foram utilizado os BLoC's?

No âmbito deste projeto, foi essencial utilizar os BLoC's, tanto na aplicação móvel, como na aplicação *web*. Na aplicação móvel, foi necessário utilizar os BLoC's para o processo de *Sign In* e de *Sign Up* de um utilizador, que evita que o próprio esteja sempre a escrever os seus dados de acesso. Além disso, foi utilizado para o processo de obter a lista de produtos presentes na *Firebase Storage*.

Já na aplicação web, foi necessário utilizar de novo os BLoC's para o *Sign In*, para apresentar a lista de produtos existentes na *Firebase Storage*, para a ação de *upload* de imagens, e para o processo de criação de um novo produto.

3.4.3 *Sign Up* BLoC's

O SignUpBloc é uma implementação do padrão BLoC para gerir o processo de *SignUp* do utilizador na aplicação móvel.

```
import 'package:delivery/user/src/models/user.dart';
import 'package:delivery/user/userClass.dart';
import 'package:bloc/bloc.dart';
import 'package:equatable/equatable.dart';

part 'SignUpEvent.dart';
part 'SignUpState.dart';

class SignUpBloc extends Bloc<SignUpEvent, SignUpState> {
    final UserRepository _userRepository;

    SignUpBloc(this._userRepository) : super(SignUpInitial()) {
        on<SignUpRequired>((event, emit) async {
            emit(SignUpProcess());
            try {
                MyUser myUser =
                    await _userRepository.signUp(event.user, event.password);
                await _userRepository.setUserData(myUser);
                emit(SignUpSuccess());
            } catch (e) {
                emit(SignUpFailure());
            }
        });
    }
}
```

Excerto de Código 3.1: SignUpBloc.dart

O ficheiro *SignUpBloc.dart* define o BLoC responsável pelo processo de *Sign Up* do utilizador. A classe *SignUpBloc* estende o *Bloc<SignUpEvent, SignUpState>*, onde *SignUpEvent* representa os eventos de registo e *SignUpState* representa os estados possíveis durante o processo de criação da conta do utilizador.

O construtor *SignUpBloc* recebe um (*UserRepository*) e define o estado inicial como *SignUpInitial*. O evento *SignUpRequired* executa o processo de registo, onde emite:

- *SignUpProcess* enquanto o processo de criação de conta está em andamento,
- *SignUpSuccess* se o processo de criação de conta for bem-sucedido,
- *SignUpFailure* se ocorrer um erro no processo de criação de conta.

```
part of 'SignUpBloc.dart';
```

```

sealed class SignUpEvent extends Equatable {
  const SignUpEvent();

  @override
  List<Object> get props => [];
}

class SignUpRequired extends SignUpEvent {
  final MyUser user;
  final String password;

  const SignUpRequired(this.user, this.password);

  @override
  List<Object> get props => [user, password];
}

```

Exerto de Código 3.2: SignUpEvent.dart

O ficheiro *SignUpEvent* define todos os eventos que o *SignUp* BLoC pode receber:

- *Classe Base SignUpEvent*: *SignUpEvent* é uma classe abstrata que estende *Equatable*, para facilitar a comparação de instâncias e otimizar a atualização de estado do BLoC.
- *Evento SignUpRequired*:
 - Este evento carrega as informações necessárias para efetuar o registo, como o objeto *user* (contendo os dados do utilizador) e a *password* e são armazenados como variáveis finais.
 - É acionado quando o utilizador tenta registrar-se, e essas propriedades são passadas para o *SignUpBloc* para processar o registo.

```

part of 'SignUpBloc.dart';

sealed class SignUpState extends Equatable {
  const SignUpState();

  @override
  List<Object> get props => [];
}

final class SignUpInitial extends SignUpState {}

class SignUpSuccess extends SignUpState {}

```

```
class SignUpFailure extends SignUpState {}  
class SignUpProcess extends SignUpState {}
```

Excerto de Código 3.3: SignUpState.dart

SignUpState é uma classe que define os diferentes estados possíveis para o processo de registo do utilizador para a aplicação *mobile*. Estes estados representam as várias fases do ciclo de vida da ação de registo, permitindo que a *interface* do utilizador reaja conforme o progresso do processo.

- ***SignUpState Base Class***: Esta classe abstrata herda de *Equatable*, facilitando a comparação de instâncias de modo que a *interface* do utilizador seja actualizada apenas quando necessário. A implementação de *props* permite que o Flutter determine automaticamente quando dois estados são iguais.
- ***SignUpInitial***: Representa o estado inicial, onde o processo de registo ainda não foi iniciado.
- ***SignUpProcess***: Indica que o registo está em curso, permitindo que a *interface* apresente um indicador de progresso, por exemplo.
- ***SignUpSuccess***: Representa o sucesso do registo, permitindo que a *interface* apresente uma mensagem de confirmação ou redirecione o utilizador.
- ***SignUpFailure***: Indica uma falha durante o registo, permitindo que a *interface* apresente uma mensagem de erro ou que o utilizador tente novamente.

3.4.4 *Sign In* BLoC's

O *SignInBloc* é uma implementação do padrão BLoC para gerir o processo de *login* do utilizador na aplicação móvel.

No *SignInBloc*, eventos como *login* (*SignInRequired*) e *logout* (*SignOutRequired*) são recebidos e processados. Com base nesses eventos, o BLoC emite estados correspondentes, como *SignInProcess* (autenticação em curso), *SignInFailure* (*login* falhado) ou *SignInSuccess* (*login* bem-sucedido). Esta gestão permite que a *interface* reaja a alterações de estado sem ter de lidar diretamente com a lógica de autenticação, o que oferece vantagens como a capacidade de reutilização de código.

```

import 'package:delivery/user/userClass.dart';
import 'package:bloc/bloc.dart';
import 'package:equatable/equatable.dart';

part 'SignInEvent.dart';
part 'SignInState.dart';

class SignInBloc extends Bloc<SignInEvent, SignInState> {
    final UserRepository _userRepository;

    SignInBloc(this._userRepository) : super(SignInInitial()) {
        on<SignInRequired>((event, emit) async {
            emit(SignInProcess());
            try {
                await _userRepository.signIn(event.email, event.password);
            } catch (e) {
                emit(SignInFailure());
            }
        });
        on<SignOutRequired>((event, emit) async => await _userRepository.
            logOut());
    }
}

```

Excerto de Código 3.4: Ficheiro SignInBloc.dart

O *SignInBloc* é responsável pela lógica de autenticação do utilizador. Ele estende a classe Bloc, indicando que trabalha com eventos (*SignInEvent*) e estados (*SignInState*). O BLoC usa:

- *Atributo UserRepository*: O UserRepository (classe *user*, criado com os dados do utilizador) é injetado no tSignInBloc como um repositório de dados responsável pelas operações de login e logout. Esse padrão facilita o acesso aos métodos de autenticação (como *signIn* e *logOut*).
- *Construtor SignInBloc*: O construtor recebe o *UserRepository* e chama o estado inicial *SignInInitial()*, definindo o estado inicial do BLoC.
- *Eventos de Login (on<SignInRequired>)*:
 - Quando o evento *SignInRequired* é emitido, o *SignInBloc*:
 - * Emite o estado *SignInProcess()* para indicar que o processo de login está em curso.

- * Tenta autenticar o usuário chamando `userRepository.signIn()` com o email e `password`, procurando se ambos existem na `Firebase Authentication`.
- * Caso o `login` seja bem-sucedido, o BLoC permite que avance para a página principal do projeto.
- * Caso haja uma falha, emite o estado `SignInFailure()` para indicar o erro.
- *Eventos de Logout (on<SignOutRequired>):*
 - Chama diretamente o método `logOut()` do repositório de usuário para encerrar a sessão.

```
part of 'SignInBloc.dart';

sealed class SignInEvent extends Equatable {
  const SignInEvent();

  @override
  List<Object> get props => [];
}

class SignInRequired extends SignInEvent {
  final String email;
  final String password;

  const SignInRequired(this.email, this.password);

  @override
  List<Object> get props => [email, password];
}

class SignOutRequired extends SignInEvent {}
```

Exerto de Código 3.5: Ficheiro SignInEvent.dart

O ficheiro `SignInEvent` define todos os eventos que o `SignIn` BLoC pode receber:

- **Classe Base `SignInEvent`:** `SignInEvent` é uma classe abstrata que estende `Equatable`, para facilitar a comparação de instâncias e otimizar a atualização de estado do BLoC.
- **Evento `SignInRequired`:**
 - Este evento carrega as informações necessárias para fazer `login`, como `email` e `password`, armazenadas como variáveis finais.

- É acionado quando o usuário tenta se autenticar, e essas propriedades são passadas para o *SignInBloc* para processar o *login*.
- **Evento *SignOutRequired*:**

- Este evento é acionado quando o utilizador solicita *logout*, indicando que o *SignInBloc* deve processar o *logout* do utilizador da aplicação.

```
part of 'SignInBloc.dart';

sealed class SignInState extends Equatable {
  const SignInState();

  @override
  List<Object> get props => [];
}

final class SignInInitial extends SignInState {}

final class SignInFailure extends SignInState {}

final class SignInProcess extends SignInState {}

final class SignInSuccess extends SignInState {}
```

Excerto de Código 3.6: SignInState.dart

O ficheiro *SignInState.dart* define os diferentes estados que o *SignInBloc* pode emitir durante o processo de *login*.

- **Classe Base *SignInState*:** Assim como *SignInEvent*, *SignInState* estende *Equatable* para facilitar comparações e otimizar a desempenho da aplicação.

- *Estados Concretos:*

- ***SignInInitial*** - Estado inicial, indicando que o usuário ainda não iniciou o *login*.
- ***SignInFailure*** - Indica que houve uma falha no *login*.
- ***SignInProcess*** - Indica que o processo de *login* está em andamento.
- ***SignInSuccess*** - Indica sucesso do *login*. Apenas é utilizado para facilitar tarefas de *debugging*, visto que um *login* com sucesso redireciona o utilizador para a página principal da aplicação



Figura 3.9: Login WEB APP

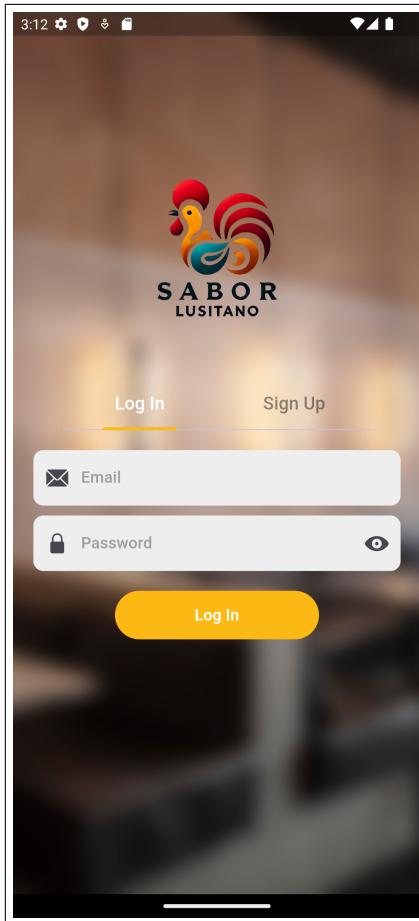


Figura 3.10: Login Android

3.4.5 *Get Product BLoC*

O *Get Product BLoC* pretende obter todos os produtos registados e criados na *storage* da Firebase. É essencial para oferecer aos clientes do restaurante todos os produtos disponíveis, tal como garantir que os trabalhadores do restaurante conseguem aceder a todos os produtos para remover, editar caso seja necessário. Para isso foi necessário criar uma função *getProduct* que permite ir procurar os produtos à *Firebase Storage*:

```
Future<List<Product>> getProduct() async {
    try {
        return await productList.get().then((value) => value.docs
            .map((e) => Product.fromEntity(ProductEntity.fromJson(e.data()
                )))
            .toList());
    } catch (e) {
        print(e);
    }
}
```

```

    } catch (e) {
      log(e.toString());
      rethrow;
    }
}

```

Excerto de Código 3.7: Função getProduct

```

static ProductEntity fromJson(Map<String, dynamic> json) {
  return ProductEntity(
    productID: json['productID'] as String,
    image: json['image'] as String,
    name: json['name'] as String,
    description: json['description'] as String,
    price: json['price'] as double,
    ingredients: Ingredients.fromEntity(
      IngredientsEntity.fromJSON(json['ingredients']) as Ingredients
    ),
  );
}

```

Excerto de Código 3.8: Função fromJson

Esta função vai procurar os documentos presentes na *Firebase Storage* e traduz de um ficheiro *JSON* para uma lista através da função *fromJson* apresentada em cima, para tornar mais fácil a utilização dos dados na *webapp*.

```

import 'package:bloc/bloc.dart';
import 'package:delivery/product/src/models/product.dart';
import 'package:delivery/product/src/productClass.dart';
import 'package:equatable/equatable.dart';

part 'get_product_event.dart';
part 'get_product_state.dart';

class GetProductBloc extends Bloc<GetProductEvent, GetProductState> {
  final ProductClass _productRepo;

  GetProductBloc(this._productRepo) : super(GetProductInitial()) {
    on<GetProduct>((event, emit) async {
      emit(GetProductLoading());
      try {
        List<Product> products = await _productRepo.getProduct();
        emit(GetProductSuccess(products));
      } catch (e) {
        emit(GetProductFailure());
      }
    });
  }
}

```

Excerto de Código 3.9: GetProductBloc.dart

- **Atributo *_productRepo*:**

- A classe possui um atributo privado *_productRepo*, que é uma instância da classe *ProductClass*.
- Esse atributo representa o repositório de produtos, que serve como uma abstração para a fonte de dados.
- O repositório é utilizado para procurar a lista de produtos de forma assíncrona.

- **Construtor *GetProductBloc*:**

- O construtor da classe recebe uma instância do repositório de produtos.
- Chama o construtor da superclasse com um estado inicial *GetProductInitial*, definindo o estado padrão do BLoC ao ser instanciado.

- **Lógica de Eventos e Estados:**

- A lógica principal do *GetProductBloc* é gerenciada através do método *on<GetProduct>*, configurado para reagir ao evento *GetProduct*.

- **Estado *GetProductLoading*:**

- * Quando o evento *GetProduct* é recebido, o BLoC emite o estado *GetProductLoading*.
- * Esse estado indica que o processo de obtenção dos dados foi iniciado e que a aplicação está em modo de carregamento.

- **Estado *GetProductSuccess*:**

- * O BLoC tenta obter a lista de produtos chamando o método *getProduct()* no repositório de produtos.
- * Se a operação for bem-sucedida, o estado *GetProductSuccess* é emitido, acompanhado da lista de produtos obtida.
- * Esse estado representa o sucesso na obtenção dos dados e permite que a *interface* do utilizador seja atualizada com as informações de produtos.

- **Estado *GetProductFailure*:**

- * Caso ocorra algum erro durante a tentativa de obtenção de produtos, o BLoC captura a exceção.
- * Emite o estado *GetProductFailure*, indicando que houve uma falha no processo.
- * Permite que a interface do usuário apresente uma mensagem de erro ou tome outra ação apropriada.

```
part of 'get_product_bloc.dart';

sealed class GetProductEvent extends Equatable {
  const GetProductEvent();

  @override
  List<Object> get props => [];
}

class GetProduct extends GetProductEvent {}
```

Excerto de Código 3.10: GetProductEvent.dart

O ficheiro *GetProductEvent.dart* é uma classe que estende *Equatable*, que como referido anteriormente, vai permitir a comparação de instâncias.

- **Construtor:**

- O construtor é definido como *const*, permitindo que instâncias sejam criadas de forma permanente.

- **Classe *GetProduct*:**

- A classe *GetProduct* estende a classe *GetProductEvent*.
- Essa classe representa um evento específico que será usado no processo de obtenção de produtos.

```
part of 'get_product_bloc.dart';

sealed class GetProductState extends Equatable {
  const GetProductState();

  @override
  List<Object> get props => [];
}

final class GetProductInitial extends GetProductState {}

final class GetProductFailure extends GetProductState {}
```

```
final class GetProductLoading extends GetProductState {}

final class GetProductSuccess extends GetProductState {
    final List<Product> Products;

    const GetProductSuccess(this.Products);

    @override
    List<Object> get props => [Products];
}
```

Exerto de Código 3.11: GetProductState.dart

A classe *GetProductState* é uma classe que tem todos os estados que o BLoC pode passar ao logo do processo.

- **Construtor:**

- O construtor volta a ser definido como *const*, permitindo que instâncias sejam criadas de forma permanente.

- **Classe *GetProductInitial*:**

- A classe *GetProductInitial* estende a classe *GetProductState*.
- Representa o estado inicial do processo de obtenção de produtos.

- **Classe *GetProductFailure*:**

- A classe *GetProductFailure* estende a classe *GetProductState*.
- Indica se houve uma falha na obtenção dos produtos.

- **Classe *GetProductLoading*:**

- A classe *GetProductLoading* estende a classe *GetProductState*.
- Indica se o processo de obtenção dos dados está em andamento.

- **Classe *GetProductSuccess*:**

- A classe *GetProductSuccess* estende a classe *GetProductState*.
- Possui um atributo *Products* que armazena uma lista de objetos do tipo *Product*.
- O construtor é definido como *const* e recebe a lista de produtos.

- **Método *get props*:**

- * O método *get props* retorna a lista de produtos, permitindo que a comparação leve em conta o conteúdo da lista.

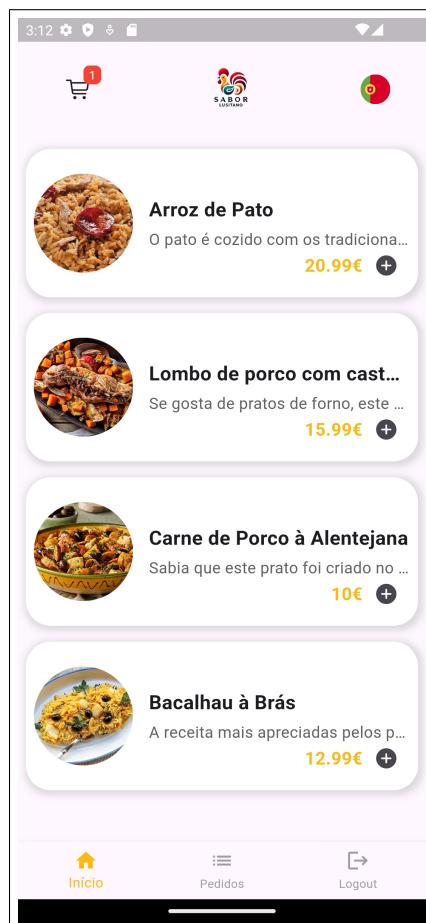


Figura 3.11: Lista Produtos Android

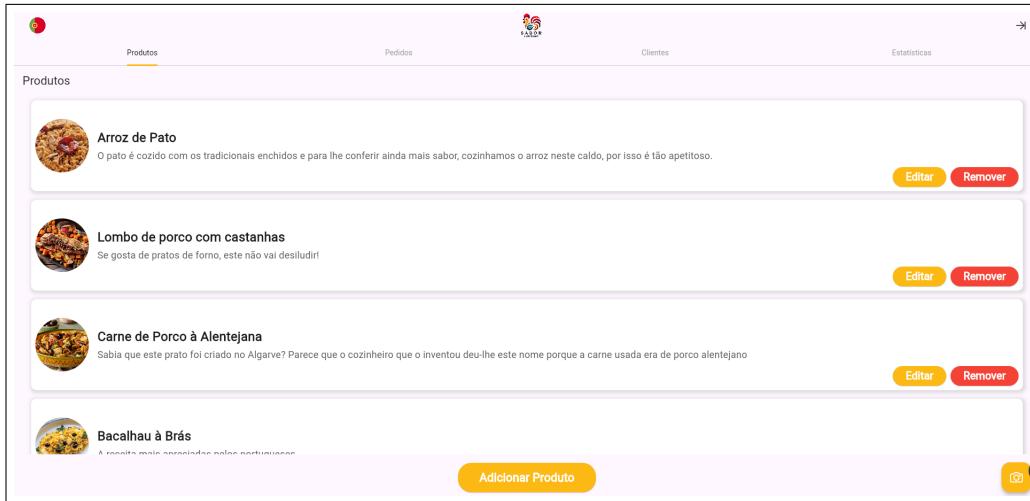


Figura 3.12: Lista Produtos WebApp

3.4.6 Upload Imagem BLoC

Já na *webapp* foi necessário criar um BLoC para a facilitar o processo de *upload* de imagem, para não sobrecarregar o código.

Para isso foi necessário criar uma função *sendImage* que envia a imagem selecionada em *Uint8List*:

```
Future<String> sendImage(Uint8List file , String name) async {
  try {
    Reference firebaseStorageRef = FirebaseStorage.instance.ref() .
      child(name);

    await firebaseStorageRef.putData(
      file ,
      SettableMetadata(
        contentType: 'image/jpeg' ,
      ));
    return await firebaseStorageRef.getDownloadURL();
  } catch (e) {
    log(e.toString());
    rethrow;
  }
}
```

Exerto de Código 3.12: Função *sendImage*

Para além de enviar os dados em *Uint8List*, também garante que o ficheiro é carregado na *Firebase Storage* no tipo *image/jpeg*, para facilitar a sua utilização na aplicação.

Após esta função ser criada, todo o processo de criação do BLoC segue o formato referido em outros BLoC's, com *uploadBloc.dart*, que precisa do *uploadEvent.dart* para registar os eventos que devem acontecer e o *uploadState.dart* que apresenta os estados que podem acontecer durante o processo de *upload* da imagem.

```
class UploadPictureBloc extends Bloc<UploadPictureEvent,
    UploadPictureState> {
    ProductClass productRepo;

    UploadPictureBloc(this.productRepo) : super(UploadPictureLoading()) {
        on<UploadPicture>((event, emit) async {
            try {
                String url = await productRepo.sendImage(event.file, event.name);
                ;
                emit(UploadPictureSuccess(url));
            } catch (e) {
                emit(UploadPictureFailure());
            }
        });
    }
}
```

Exerto de Código 3.13: uploadBloc.dart

```
part of 'upload_bloc.dart';

sealed class UploadPictureEvent extends Equatable {
    const UploadPictureEvent();

    @override
    List<Object?> get props => [];
}

class UploadPicture extends UploadPictureEvent {
    final Uint8List file;
    final String name;

    const UploadPicture(this.file, this.name);

    @override
    List<Object?> get props => [file, name];
}
```

Exerto de Código 3.14: uploadEvent.dart

```
part of 'upload_bloc.dart';

sealed class UploadPictureState extends Equatable {
    const UploadPictureState();
```

```

@Override
List<Object?> get props => [];

final class UploadPictureLoading extends UploadPictureState {}

final class UploadPictureFailure extends UploadPictureState {
  get error => null;
}

final class UploadPictureSuccess extends UploadPictureState {
  final String url;

  const UploadPictureSuccess(this.url);

  @override
  List<Object?> get props => [];
}

```

Excerto de Código 3.15: uploadState.dart

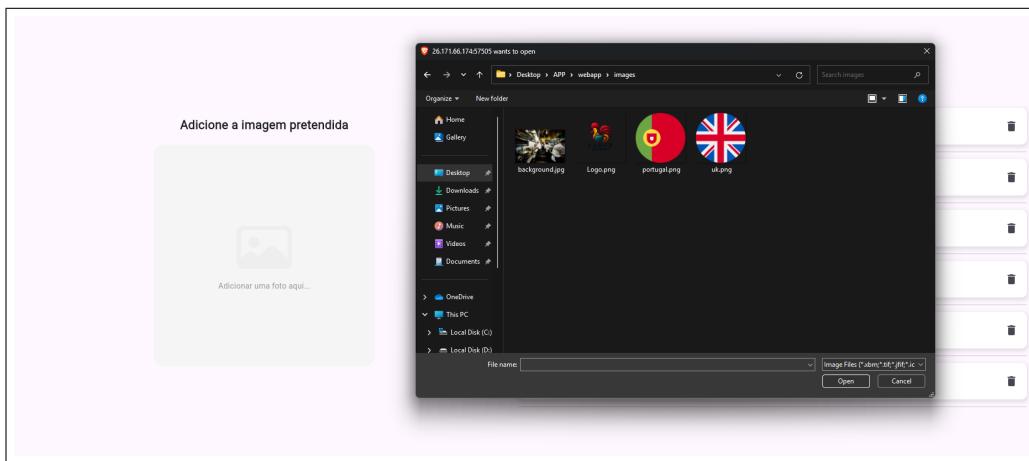


Figura 3.13: Upload

3.4.7 Create product BLoC

Para os serviços de administrador, foi necessário um BLoC para facilitar o processo de criação do produto para ser vendido.

Para ser possível seguir o método implementado em todos os BLoC's anteriores foi necessário criar uma função que enviasse o produto para a coleção *product* associado *Firebase Storage* da aplicação

```

@Override
Future<void> createProduct(Product product) async {
    try {
        return await productList
            .doc(product.productID)
            .set(product.toEntity().toJson());
    } catch (e) {
        log(e.toString());
        rethrow;
    }
}

```

Excerto de Código 3.16: Função createProduct

```

Map<String, dynamic> toJson() {
    return {
        'productID': productID,
        'image': image,
        'name': name,
        'description': description,
        'price': price,
        'ingredients': ingredients.toEntity().toJSON(),
    };
}

```

Excerto de Código 3.17: Função toJson

Esta função lê resultados de um formulário, valida se são válidos e conforme a classe *product*. Se são válidos, transforma a lista para um *JSON* e envia para o produto, caso contrário, o BLoC vai passar o estado de falha.

```

if (_formKey.currentState!.validate()) {
    setState(() {
        product.name = nameController.text;
        product.description = descriptionController.text;
        product.price = double.parse(priceController.text);

        product.ingredients = Ingredients(
            ingredientID: const Uuid().v1(),
            ingredientName1: ingredient1Controller.text,
            ingredientName2: ingredient2Controller.text,
            ingredientName3: ingredient3Controller.text,
            ingredientName4: ingredient4Controller.text,
        );
    });

    print(product.toString());

    context.read<CreateProductBloc>().add(CreateProduct(product));
}

```

Exerto de Código 3.18: Verificação realizada para usar o *CreateProductBloc*

Só com esta função é que é possível realizar o *CreateProductBloc* de forma eficiente.

```
class CreateProductBloc extends Bloc<CreateProductEvent,
    CreateProductState> {
    ProductClass productRepo;

    CreateProductBloc(this.productRepo) : super(CreateProductInitial()) {
        on<CreateProduct>((event, emit) async {
            emit(CreateProductLoading());
            try {
                await productRepo.createProduct(event.product);
                emit(CreateProductSuccess());
            } catch (e) {
                emit(CreateProductFailure());
            }
        });
    }
}
```

Exerto de Código 3.19: *createBloc.dart*

```
part of 'createBloc.dart';

sealed class CreateProductEvent extends Equatable {
    const CreateProductEvent();

    @override
    List<Object> get props => [];
}

class CreateProduct extends CreateProductEvent {
    final Product product;

    const CreateProduct(this.product);

    @override
    List<Object> get props => [product];
}
```

Exerto de Código 3.20: *createEvent.dart*

```
part of 'createBloc.dart';

sealed class CreateProductEvent extends Equatable {
    const CreateProductEvent();
```

```
@override  
List<Object> get props => [];  
}  
  
class CreateProduct extends CreateProductEvent {  
  final Product product;  
  
  const CreateProduct(this.product);  
  
  @override  
List<Object> get props => [product];  
}
```

Exerto de Código 3.21: CreateEvent.dart

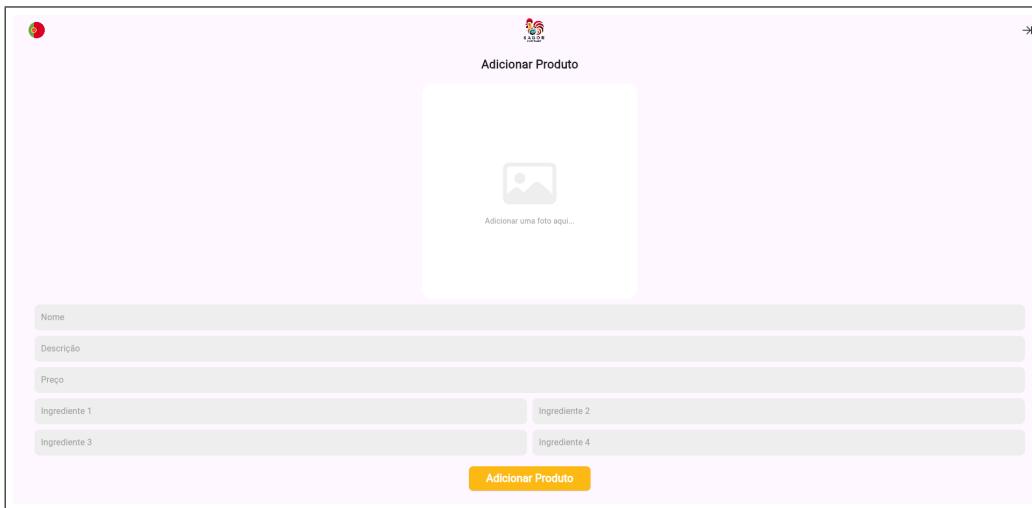


Figura 3.14: Criar Produto

3.5 Conexão à base de dados da Firebase

Nesta secção, abordaremos como conectar a aplicação ao *Firebase*, que será utilizado como base de dados. O *Firebase* fornece uma base de dados em tempo real e de fácil integração com o Flutter, permitindo a sincronização dos dados entre os dispositivos de forma rápida e segura.

3.5.1 Conexão à *Firebase*

Para ser possível utilizar o *Firebase* como base de dados e *storage* da aplicação móvel e aplicação *WEB*, foi necessário realizar alguns procedimentos pedidos

pela própria *Firebase*.

3.5.1.1 Aplicação Móvel

Para realizar a conexão do *Firebase* à aplicação Móvel foi necessário criar um ficheiro JSON para ser possível realizar qualquer tipo de conexão com a aplicação.

```
{
  "project_info": {
    "project_number": "493722328438",
    "project_id": "delivery-68030",
    "storage_bucket": "delivery-68030.appspot.com"
  },
  "client": [
    {
      "client_info": {
        "mobilesdk_app_id": "1:493722328438:android:
          cc239512ddc857ea66ae57",
        "android_client_info": {
          "package_name": "com.example.delivery"
        }
      },
      "oauth_client": [],
      "api_key": [
        {
          "current_key": "AIzaSyCi0rJv-h9q0CG9NtXsbVFmAAp6jKd2G5o"
        }
      ],
      "services": {
        "appinvite_service": {
          "other_platform_oauth_client": []
        }
      }
    }
  ],
  "configuration_version": "1"
}
```

Exerto de Código 3.22: google-services.json

Além disso, foi necessário chamar as dependências necessárias no ficheiro *pubspec.yaml*

```
dependencies:
  .
  .
  .
  cloud_firestore: ^5.4.1
  firebase_storage: ^12.3.0
  .
  .
```

Excerto de Código 3.23: pubspec.yaml

3.5.1.2 Aplicação WEB

Para conectar a *Firebase* à aplicação *WEB* foi necessário obter vários dados fornecidos pela *Firebase*, desde *apiKey*, *authDomain*, *projectId*, *storageBucket*, *messagingSenderId* e *appId*. Além disso, foi preciso criar o ficheiro *firebaseOptions.dart* para aceitar apenas a aplicação *WEB* como a única plataforma disponível, para evitar possíveis conflitos com a aplicação móvel.

```
show defaultTargetPlatform, kIsWeb, TargetPlatform;

class DefaultFirebaseOptions {
  static FirebaseOptions get currentPlatform {
    if (kIsWeb) {
      return webProd;
    }
    switch (defaultTargetPlatform) {
      case TargetPlatform.android:
        throw UnsupportedError(
          'DefaultFirebaseOptions have not been configured for android -',
          'you can reconfigure this by running the FlutterFire CLI again',
          '.',
        );
    }
  }

// Dados obtidos do Firebase Console
static const FirebaseOptions webProd = FirebaseOptions(
  apiKey: "AlzaSyDi39qKrmabGjH5ynyzP24UJAjAAoHWN0E",
  authDomain: "delivery-68030.firebaseio.com",
  projectId: "delivery-68030",
  storageBucket: "delivery-68030.appspot.com",
  messagingSenderId: "493722328438",
  appId: "1:493722328438:web:1d4dce1c198e7ee266ae57");
}
```

Excerto de Código 3.24: firebaseOptions.dart

3.5.2 Classe User

De forma a facilitar todo o processo de integração da aplicação com a base de dados da *Firebase*, foi necessário criar uma classe que permite criar e registar utilizadores, visto que é um elemento essencial da aplicação, pois sem utilizadores, não existe possibilidade de dar o devido uso à aplicação.

Para isso criou-se uma class *User*, com o ficheiro *user.dart* com todas as propriedades de um utilizador, *userEntity.dart* para as funções necessárias para enviar dados para a base de dados, e *userClass.dart* como todos os métodos necessários. Assim vai facilitar o uso dos BLoC's, como referido no capítulo 3.4.

```
class MyUser {  
    String userId;  
    String email;  
    String name;  
    bool hasCart;  
    String role;  
    String phone;  
  
    MyUser({  
        required this.userId,  
        required this.email,  
        required this.name,  
        required this.hasCart,  
        required this.role,  
        required this.phone,  
    });  
  
    static final empty = MyUser(  
        userId: '',  
        email: '',  
        name: '',  
        hasCart: false,  
        role: 'Client',  
        phone: '',  
    );  
  
    MyUserEntity toEntity() {  
        return MyUserEntity(  
            userId: userId,  
            email: email,  
            name: name,  
            hasCart: hasCart,  
            role: role,  
            phone: phone,  
        );  
    }  
}
```

```
static MyUser fromEntity(MyUserEntity entity) {
    return MyUser(
        userId: entity.userId,
        email: entity.email,
        name: entity.name,
        hasCart: entity.hasCart,
        role: entity.role,
        phone: entity.phone,
    );
}
```

Exerto de Código 3.25: userEntity.dart

O código descreve a classe **MyUser**, que representa o utilizador e as suas propriedades.

- **Propriedades da classe:**

- **userId**: Identificador único do utilizador;
- **email**: O endereço de correio eletrónico do utilizador;
- **name**: Nome do utilizador;
- **hasCart**: Booleano que indica se o utilizador tem um carrinho ativo;

- **Construtor:**

- O construtor requer valores para todas as propriedades, garantindo que cada instância de **MyUser** seja criada com dados válidos.

- **Instância vazia:**

- Representa um utilizador vazio, com valores vazios ou predefinidos atribuídos a todas as propriedades.

- **Método *toEntity*:**

- Converte uma instância de **MyUser** num objeto **MyUserEntity**, para upload para o Firebase.

- **Método *fromEntity*:**

- Transforma um objeto **MyUserEntity** numa instância **MyUser**, mapeando os dados recebidos diretamente do *Firebase*, para ser possível utilizá-los na aplicação móvel.

- **Substituição do método *toString*:**

- Define como uma instância **MyUser** é convertida em uma *string*, útil para *debugging* e análise de dados.

```
class MyUserEntity {  
    String userId;  
    String email;  
    String name;  
    bool hasCart;  
    String role;  
    String phone;  
  
    MyUserEntity({  
        required this.userId,  
        required this.email,  
        required this.name,  
        required this.hasCart,  
        required this.role,  
        required this.phone,  
    });  
  
    Map<String, Object?> toJson() {  
        return {  
            'userId': userId,  
            'email': email,  
            'name': name,  
            'hasCart': hasCart,  
            'role': role,  
            'phone': phone,  
        };  
    }  
  
    static MyUserEntity fromJson(Map<String, Object?> json) {  
        return MyUserEntity(  
            userId: json['userId'] as String,  
            email: json['email'] as String,  
            name: json['name'] as String,  
            hasCart: json['hasCart'] as bool,  
            role: json['role'] as String,  
            phone: json['phone'] as String,  
        );  
    }  
}
```

Excerto de Código 3.26: user.dart

A *MyUserEntity* representa a entidade do utilizador da aplicação, utilizada para armazenar e recuperar informações da base de dados do *Firebase*. A classe é composta por propriedades que descrevem os dados do utilizador e métodos que convertem os objetos para e a partir de um formato JSON.

Este processo facilita a integração com serviços *backend* e armazenamento na base de dados.

- **Propriedades da Classe:**

- ***userId***: ID único que representa o utilizador na base de dados;
- ***email***: Endereço de correio eletrónico associado ao utilizador;
- ***name***: Nome do utilizador;
- ***hasCart***: Indica, através de um *boolean*, para definir se o utilizador tem um carrinho ativo;
- ***role***: Indica se o utilizador é um cliente ou se tem alguma função de administrador;
- ***phone***: Número de telefone do utilizador a ser criado.

- **Construtor:**

- O construtor garante que todas as instâncias da classe são criadas com valores válidos para as suas propriedades. Esta abordagem garante a consistência dos dados ao longo do ciclo de vida da aplicação.

- **Método *toJson()*:**

- Este método converte as propriedades da entidade num mapa JSON. As chaves do mapa correspondem aos nomes das propriedades, e os valores são os dados armazenados na entidade. Este formato JSON é utilizado principalmente para enviar dados para o *backend* ou armazená-los na base de dados.

- **Método Estático *fromJson()*:**

- Este método cria uma instância da classe a partir de um mapa JSON. Ele lê os valores no JSON e atribui os valores às propriedades da entidade. Este método serve para transformar dados recebidos da base de dados, ou até mesmo de uma API

```
abstract class UserRepository {
  Stream<MyUser?> get user;
  Future<MyUser> signUp(MyUser myUser, String password);
  Future<void> setUserData(MyUser user);
  Future<void> signIn(String email, String password);
  Future<void> logOut();}
```

Excerto de Código 3.27: userClass.dart

A classe *UserRepository*, criada no ficheiro *userClass.dart*, é utilizada para centralizar e abstrair as operações relacionadas com a gestão de utilizadores da aplicação. importante, pois faz parte de uma implementação baseada no padrão de repositório, que separa a lógica de acesso aos dados da lógica comercial da aplicação.

Após de configurar toda a classe *user*, foi necessário criar todos os métodos definidos em *UserRepository*.

```
Stream<MyUser?> get user {
    return _firebaseAuth.authStateChanges().flatMap((firebaseUser) async
        * {
            if (firebaseUser == null) {
                yield MyUser.empty;
            } else {
                yield await UsersList.doc(firebaseUser.uid).get().then(
                    (value) => MyUser.fromEntity(MyUserEntity.fromJson(value.data()!)));
            }
        });
}
```

Excerto de Código 3.28: Método User

Este método acima fornece um fluxo contínuo que acompanha o estado de autenticação do utilizador em tempo real. Sempre que o estado se altera, verifica se existe um utilizador autenticado:

- Se não houver nenhum utilizador autenticado, devolve um utilizador vazio;
- Se um utilizador estiver autenticado, recupera os dados correspondentes da base de dados e transforma-os num objeto que pode ser utilizado na aplicação.

Este fluxo serve para manter a *interface* do utilizador atualizada automaticamente com base no estado da autenticação.

```
Future<MyUser> signUp(MyUser myUser, String password) async {
    try {
        UserCredential user = await _firebaseAuth.createUserWithEmailAndPassword(
            email: myUser.email, password: password);
        myUser.userId = user.user!.uid;
        return myUser;
    } catch (e) {
        log(e.toString());
        rethrow;
    }
}
```

Excerto de Código 3.29: Método signUp

Este método permite registar um novo utilizador, criando uma conta com o endereço de correio eletrónico e a palavra-passe fornecidos. O processo funciona da seguinte forma: Tenta criar um utilizador no *Firebase Authentication* utilizando as credenciais fornecidas (*email* e *password*), o resto das credenciais referidas na secção 3.5.2 são armazenadas no *Firebase storage*. Além disso, se:

- O registo for concluído, associa o ID de utilizador gerado pelo Firebase ao objeto *MyUser* e devolve-o;
- Ocorrer um erro durante o registo, é lançada uma exceção, não só na aplicação, mas também no terminal, de forma a facilitar processos de *debug*.

O *setUserData* procurar o utilizador que está autenticado. Para isso, não é preciso construir nenhum método em concreto, mas sim utilizá-lo no *SignUp BLoC*, referido na secção 3.4.3. Para isso, é necessário usar o método *user* referido acima. O mesmo acontece com o método *logOut*, mas aqui não é necessário procurar o utilizador autenticado, visto que se trata do processo de desconexão do utilizador na app.

```
Future<Map<String, String>> fetchUserDetails() async {
  User? user = FirebaseAuth.instance.currentUser;
  if (user != null) {
    final userDoc = await FirebaseFirestore.instance
      .collection('users')
      .doc(user.uid)
      .get();

    if (userDoc.exists) {
      return {
        'email': user.email ?? '',
        'name': userDoc.data()?[ 'name'] ?? '',
      };
    }
  }
  return {};
}
```

Exerto de Código 3.30: Função *fetchUserDetails*

A função *fetchUserDetails* é responsável por obter os detalhes do utilizador autenticado no Firebase. Em primeiro lugar, verifica se existe um utilizador autenticado através de *FirebaseAuth.instance.currentUser*. Se existir um utilizador, a função tenta obter os dados adicionais desse utilizador a partir da coleção *users* do *Firestore*, utilizando o *uid* do utilizador autenticado como

identificador do documento. Se o documento do utilizador existir no *Firebase*, a função devolve um mapa com duas chaves: *email* e *name*. O valor *email* é retirado diretamente do objeto *user* e o valor do nome é retirado dos dados do documento no *Firebase*. Caso contrário, devolve um mapa vazio. Esta função é utilizada para preencher os dados do utilizador quando este pretende dar um *feedback*, evitando que o utilizador demore tempo a escrever dados já armazenados na base de dados.

3.5.3 Classe *Product*

Ainda no âmbito de facilitar todo o processo de integração da aplicação com a base de dados da *Firebase*, foi necessário criar uma classe que permite criar e editar Produtos, visto que também é um elemento essencial da aplicação.

Seguindo a mesma linha de pensamento da classe *User*, foi criado um ficheiro com todas as propriedades que o produto que vai ser vendido deve ter, tal como todas as funções necessárias para enviar e obter dados da base de dados.

```
class Product {  
    String productID, image, name, description;  
    double price;  
    Ingredients ingredients;  
  
    Product({  
        required this.productID,  
        required this.image,  
        required this.name,  
        required this.description,  
        required this.price,  
        required this.ingredients,  
    });  
  
    static var empty = Product(  
        productID: const Uuid().v1(),  
        image: '',  
        name: '',  
        description: '',  
        price: 0.0,  
        ingredients: Ingredients.empty,  
    );  
  
    ProductEntity toEntity() => ProductEntity(  
        productID: productID, image: image, name: name,  
        description: description, price: price, ingredients: ingredients,  
    );
```

```
static Product fromEntity(ProductEntity entity) => Product(  
    productID: entity.productID, image: entity.image,  
    name: entity.name, description: entity.description,  
    price: entity.price, ingredients: entity.ingredients,  
)  
  
@override  
String toString() => 'Product{productID: $productID, image: $image,  
    name: $name, description: $description, price: $price, ingredients  
    : $ingredients}'  
}
```

Excerto de Código 3.31: Product Class

Este código define uma classe `Product`, que representa um produto com várias propriedades: `productID`, imagem, nome, descrição, preço e ingredientes. Cada um destes campos descreve as características de um produto, sendo os ingredientes um objeto de uma classe `Ingredientes`, presumivelmente contendo detalhes sobre os ingredientes do produto.

```
class Ingredients {  
    String ingredientID;  
    String ingredientName1;  
    String ingredientName2;  
    String ingredientName3;  
    String ingredientName4;  
  
    Ingredients({  
        required this.ingredientID,  
        required this.ingredientName1,  
        required this.ingredientName2,  
        required this.ingredientName3,  
        required this.ingredientName4,  
    });  
  
    IngredientsEntity toEntity() {  
        return IngredientsEntity(  
            ingredientID: ingredientID,  
            ingredientName1: ingredientName1,  
            ingredientName2: ingredientName2,  
            ingredientName3: ingredientName3,  
            ingredientName4: ingredientName4,  
        );  
    }  
  
    static Ingredients fromEntity(Ingredients entity) {  
        return Ingredients(  
            ingredientID: entity.ingredientID,  
            ingredientName1: entity.ingredientName1,  
            ingredientName2: entity.ingredientName2,  
            ingredientName3: entity.ingredientName3,  
            ingredientName4: entity.ingredientName4,  
        );  
    }  
}
```

Excerto de Código 3.32: Classe Ingredient

A classe também inclui um construtor que requer que todas as propriedades sejam fornecidas. À semelhança da classe *user* existe também um campo estático *empty*, que cria um produto vazio com valores predefinidos, útil para inicializações sem dados ou como um valor predefinido.

Além disso, o ficheiro ainda contém alguns métodos como:

- *toEntity*: Converte o objeto *Product* numa *ProductEntity*, que é provavelmente uma classe utilizada para representar o produto de uma forma mais adequada à persistência ou manipulação de dados numa base de dados.

- *fromEntity*: Faz o oposto, convertendo um *ProductEntity* novamente num objeto *Product*.
- *toString*: Substitui a função padrão de impressão de objetos, proporcionando uma forma legível de apresentar os dados de um produto.

Em suma, a classe *Product* é utilizada para representar um produto num sistema, com métodos de conversão entre objetos e entidades, bem como um valor predefinido.

```
class ProductEntity {
    String productID;
    String image;
    String name;
    String description;
    double price;
    Ingredients ingredients;
    ProductEntity({
        required this.productID,
        required this.image,
        required this.name,
        required this.description,
        required this.price,
        required this.ingredients,
    });

    Map<String, dynamic> toJson() {
        return {
            'productID': productID,
            'image': image,
            'name': name,
            'description': description,
            'price': price,
            'ingredients': ingredients.toEntity().toJSON(),
        };
    }

    static ProductEntity fromJson(Map<String, dynamic> json) {
        return ProductEntity(
            productID: json['productID'] as String,
            image: json['image'] as String,
            name: json['name'] as String,
            description: json['description'] as String,
            price: json['price'] as double,
            ingredients: Ingredients.fromEntity(
                IngredientsEntity.fromJSON(json['ingredients']) as Ingredients
            ),
        );
    }
}
```

Excerto de Código 3.33: Classe ProductEntity

A classe *ProductEntity* contém também todas as propriedades referidas acima, mas possui métodos importantes para facilitar a conversão entre o objeto e o formato de dados, como:

- **Método *toJson*:** Converte a instância *ProductEntity* em um mapa (*Map<String, dynamic>*) que pode ser facilmente transformado em JSON. Esse mapa inclui todos os campos da entidade e, para o campo de ingredientes, chama o método *toEntity().toJson()* da classe *Ingredients* para garantir que os dados dos ingredientes também sejam convertidos corretamente em JSON.
- **Método estático *fromJson*:** Converte um mapa JSON de volta para uma instância de *ProductEntity*. Ele extrai os valores dos campos do JSON e os usa para criar um objeto *ProductEntity*. Para o campo de ingredientes, o código chama o método *fromEntity* da classe *Ingredients* para garantir que os ingredientes são corretamente reconstruídos a partir dos dados JSON.

```
import 'dart:typed_data';

import 'package:webapp/product/src/models/models.dart';

abstract class ProductClass {
    Future<List<Product>> getProduct();
    Future<String> sendImage(Uint8List file, String name);
    Future<void> createProduct(Product product);
}
```

Excerto de Código 3.34: Classe Abstrata ProductClass

Este código define uma classe abstrata *ProductClass*, que especifica os métodos que a implementação desta classe deve fornecer. Esta classe abstrata serve como um contrato para as operações que podem e devem ser realizadas nos produtos.

A classe tem três métodos assíncronos:

```
Future<List<Product>> getProduct() async {
    try {
        return await productList.get().then((value) => value.docs
            .map((e) => Product.fromEntity(ProductEntity.fromJson(e.data())))
            .toList());
    } catch (e) {
        log(e.toString());
        rethrow;
    }
}
```

Excerto de Código 3.35: Método *getProduct*

O método ***getProduct*** devolve uma lista de objetos *Product*. Este método pode ser utilizado para obter os dados do produto, seja por uma API ou na própria base de dados através do método *toJson*, referido anteriormente.

```
Future<String> sendImage(Uint8List file, String name) async {
    try {
        Reference firebaseStorageRef = FirebaseStorage.instance.ref()
            .child(name);

        await firebaseStorageRef.putData(
            file,
            SettableMetadata(
                contentType: 'image/jpeg',
            ));
        return await firebaseStorageRef.getDownloadURL();
    } catch (e) {
        log(e.toString());
        rethrow;
    }
}
```

Excerto de Código 3.36: Método *sendImage*

O método ***sendImage*** recebe um ficheiro de imagem no formato *Uint8List* (uma lista de bytes) e o nome da imagem como parâmetros, e devolve uma *String*. Este método é responsável pelo envio da imagem para o *storage* do *Firebase* e fica disponível para acesso.

```
Future<void> createProduct(Product product) async {
    try {
        return await productList
            .doc(product.productID)
            .set(product.toEntity().toJson());
    } catch (e) {
        log(e.toString());
        rethrow;
    }
}
```

Excerto de Código 3.37: Método *createProduct*

O método ***createProduct*** recebe um objeto *Product* e cria um produto, armazenando-o na base de dados.

3.5.3.1 Funções Extra

Para além destes 3 métodos assíncronos, ainda existem várias funções úteis no processo interação do utilizador com as aplicações. Essas interações vão desde ver a página detalhada do produto, até adicionar o produto ao carrinho.

Além disso, existem mais métodos que precisam de atualizar o estado do widget que contém UI (*Stateful Widget*) de forma apresentar as mudanças.

```
Future<List<Map<String, dynamic>>> fetchCartProducts() async {
    try {
        final user = FirebaseAuth.instance.currentUser;
        if (user == null) {
            throw Exception('No user logged in');
        }

        final userId = user.uid;
        print('Fetching products for user: $userId');
        final querySnapshot =
            await FirebaseFirestore.instance.collection('cart').doc(userId)
                .get();

        if (!querySnapshot.exists) {
            print('No products found in the cart collection for user $userId
                .');
            return [];
        } else {
            print('Products fetched successfully for user $userId.');
        }

        final cartData = querySnapshot.data() as Map<String, dynamic>;
        final products = List<Map<String, dynamic>>.from(cartData['
            products']);

        return products;
    } catch (e) {
        print('Error fetching cart products: $e');
        rethrow;
    }
}
```

Excerto de Código 3.38: Método `fetchCartProducts`

A função **`fetchCartProducts`** é responsável por obter a lista de produtos adicionados pelo utilizador ao carrinho compras, para uma possível compra. Para isso a função tem que ter as seguintes propriedades:

1. Verificação do utilizador autenticado:

- A função verifica se existe atualmente um utilizador autenticado utilizando o *Firebase Authentication*.
- Se não existir um utilizador autenticado, lança uma exceção que indica que nenhum utilizador tem sessão iniciada.

2. Obter ID do utilizador:

- Obtém o *ID* único (*UID*) do utilizador autenticado, que será utilizado para identificar o carrinho deste utilizador.

3. Consultar a base de dados:

- A função acede à *Firestore*, procurando o documento do carrinho na coleção de carrinhos, identificado pelo *UID* do utilizador.

4. Validação da existência dos dados:

- Verifica se o documento existe.
- Se não existir, devolve uma lista vazia e imprime uma mensagem para dar o utilizador a saber que não foram encontrados produtos.

5. Processamento dos dados do carrinho:

- Se o documento existir, extrai os dados do carrinho.
- Assume que os produtos estão armazenados como uma lista de mapas numa propriedade chamada *products*.

6. Retorno da lista de produtos:

- Converte os dados do carrinho na estrutura esperada (lista de mapas) e devolve esta lista.

7. Tratamento de erros:

- Se ocorrer um erro durante o processo, o erro é detetado e envia pelo terminal uma mensagem de registo, para facilitar qualquer processo de *debugging*.

```
Future<void> addToCart(BuildContext context, String productId, Map<
    String, dynamic> productData) async {
    final user = FirebaseAuth.instance.currentUser;
    if (user == null) return print('User not signed in');

    final cartRef = FirebaseFirestore.instance.collection('cart').doc(user
        .uid);
    final cartSnapshot = await cartRef.get();

    final products = cartSnapshot.exists
        ? List.from((cartSnapshot.data()?[ 'products'] as List<dynamic>)..
            addIf(
                cartSnapshot.data()?[ 'products'].any((product) => product[ 'id'] == productId),
                (product) => product[ 'quantity'] += 1,
                () => { 'id': productId, 'data': productData, 'quantity': 1})
            )
        : [
            { 'id': productId, 'data': productData, 'quantity': 1}
        ];

    await cartRef.set({ 'products': products});
    Navigator.pushAndRemoveUntil(
        context,
        MaterialPageRoute<void>(builder: (_) => const HomePage()),
        (route) => false,
    );
}
```

Excerto de Código 3.39: Função addToCart

A função **addToCart** adiciona um produto ao carrinho de compras do utilizador no *Firebase*. Se o produto já existir, incrementa a quantidade; caso contrário, adiciona-o ao carrinho. Esta função realiza os seguintes passos:

1. **Obter utilizador autenticado:** Verifica se há um utilizador autenticado no Firebase. Se não, termina e apresenta uma mensagem de erro.
2. **Referenciar o carrinho:** Obtém a referência ao documento do carrinho do utilizador na coleção *cart*, identificando-o pelo *userID*.
3. **Verificar existência do carrinho:**
 - **Se existir:** Recupera os dados e verifica se o produto já está presente:
 - *Produto no carrinho:* Incrementa a quantidade.
 - *Produto ausente:* Adiciona com quantidade inicial de 1.

- Atualiza o documento no *Firebase*.
 - **Se não existir:** Cria um carrinho com o produto inicial e quantidade 1.
4. **Atualizar informações adicionais:** Atualiza o campo *hasCart* na coleção *users*, indicando que o utilizador tem um carrinho ativo.
 5. **Erro se não autenticado:** Imprime uma mensagem na consola sem alterar o *Firebase*.

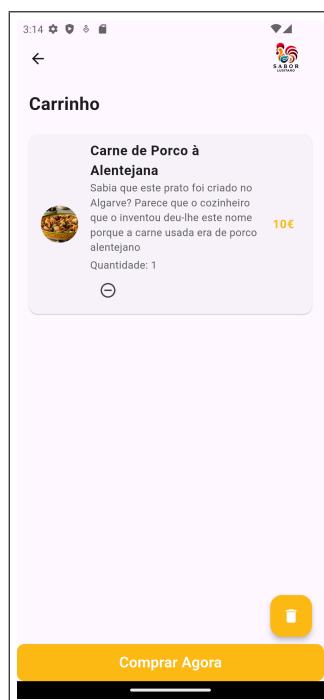


Figura 3.15: Carrinho

```
Future<DocumentSnapshot> getProductDetails(String productId) async {
  return await FirebaseFirestore.instance
    .collection('product')
    .doc(productId)
    .get();
}
```

Exceto de Código 3.40: Função *getProductDetails*

A função ***getProductDetails*** é focada em obter todos os dados de um determinado produto, definido pelo seu *productId*. É utilizada quando o utiliza-

dor entra na página de detalhes do produto, onde contém todas as características mais os *feedbacks* dados acerca do produto.

```
Future<List<Map<String, dynamic>>> getFeedbacks(String productId) async
{
    final feedbacksSnapshot = await FirebaseFirestore.instance
        .collection('feedbacks')
        .where('productId', isEqualTo: productId)
        .get();

    final feedbacks = feedbacksSnapshot.docs
        .map((doc) => doc.data() as Map<String, dynamic>)
        .toList();

    for (var feedback in feedbacks) {
        final userDoc = await FirebaseFirestore.instance
            .collection('users')
            .doc(feedback['userId'])
            .get();

        if (userDoc.exists) {
            feedback['name'] = userDoc.data()?['name'] ?? 'Unknown User';
        } else {
            feedback['name'] = 'Unknown User';
        }
    }

    return feedbacks;
}
```

Excerto de Código 3.41: Função getFeedbacks

A função **getFeedbacks** é responsável por procurar os *feedbacks* associados a um determinado produto. A função acede à coleção de *feedbacks* no Firestore e filtra os documentos em que o campo *productId* fornecido como argumento. Cada documento representa um *feedback* relacionado com esse produto. Após isso, os documentos recuperados são convertidos numa lista de mapas (Map<String, dynamic>), que representam os dados de cada comentário.

Para cada comentário, a função verifica o *userId* associado. Em seguida, acede à coleção de utilizadores para obter o documento desse utilizador, onde procura o nome associado a esse *userId*:

- Se o utilizador existir, o nome é extraído e adicionado ao comentário.
- Se o utilizador não existir ou se o nome não estiver definido, o nome é definido como “*Unknown User*”.

No final, a função devolve a lista de comentários, agora com os nomes dos utilizadores associados incluídos, pronta a ser utilizada na página de detalhes.

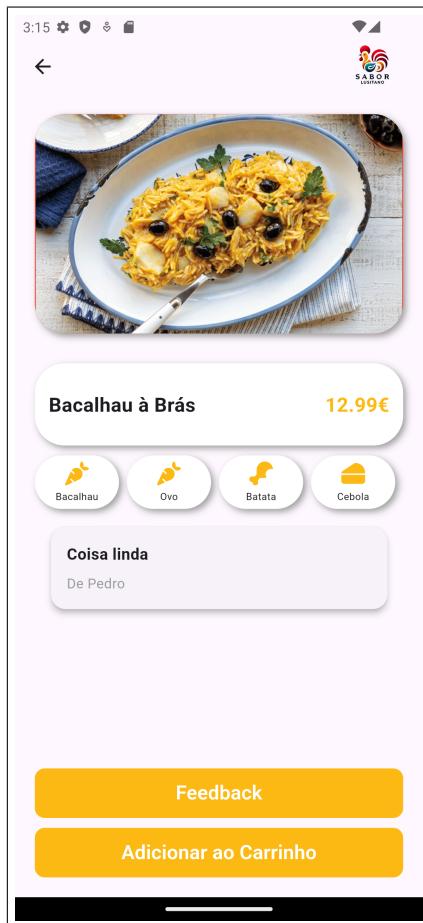


Figura 3.16: Detalhes e Feedback

```
Future<void> removeProductFromCart(String productId, int quantity) async
{
    try {
        final user = FirebaseAuth.instance.currentUser;
        if (user == null) {
            throw Exception('No user is currently signed in.');
        }

        final userId = user.uid;
        final cartRef = FirebaseFirestore.instance.collection('cart').doc(
            userId);

        final cartSnapshot = await cartRef.get();
```

```
if (!cartSnapshot.exists) {
    throw Exception('Cart does not exist for user $userId.');
}

final cartData = cartSnapshot.data() as Map<String, dynamic>;
final products = List<Map<String, dynamic>>.from(cartData[ 'products']);

final productIndex =
    products.indexWhere((product) => product[ 'id' ] == productId);
if (productIndex == -1) {
    throw Exception('Product not found in cart.');
}

if (quantity > 1) {
    products[productIndex][ 'quantity' ] = quantity - 1;
} else {
    products.removeAt(productIndex);
}

await cartRef.update({ 'products': products });
print('Product removed successfully.');
} catch (e) {
    print('Error removing product from cart: $e');
    rethrow;
}
}
```

Exerto de Código 3.42: Função removeProductFromCart

A função **removeProductFromCart** pretende remover um produto do cesto de compras de um utilizador, que está armazenado no *Firebase Firestore*.

Se o carrinho não existir, é lançado um erro, indicando que o carrinho não foi encontrado. Caso contrário, a função recupera a lista de produtos do cesto. Em seguida, tenta localizar o produto a remover, com base no seu *productId*.

Se o produto não for encontrado, é gerado um erro.

Dependendo da quantidade do produto no cesto, a função toma uma ação diferente. Se a quantidade for superior a um, a função reduz a quantidade em um. Caso contrário, se a quantidade for um, o produto é completamente removido da lista. Após esta alteração, a lista de produtos é atualizada no *Firebase*.

Durante todo o processo, se ocorrer um erro, a função captura e imprime a mensagem de erro e, em seguida, retransmite o erro para poder ser tratado noutra parte do código. O objetivo final é garantir que o cesto de compras é atualizado corretamente, removendo ou ajustando a quantidade do produto conforme necessário.

```
Future<void> _fetchImagesFromStorage() async {
    try {
        print('Fetching images from Firebase Storage');
        final ListResult result = await FirebaseStorage.instance
            .refFromURL('gs://delivery-68030.appspot.com')
            .listAll();
        print('Found ${result.items.length} items in storage');

        List<String> names = [];
        for (var ref in result.items) {
            print('Fetching metadata for item: ${ref.fullPath}');
            final FullMetadata metadata = await ref.getMetadata();
            if (metadata.contentType == 'image/jpeg' ||
                metadata.contentType == 'image/png') {
                print('Fetched image name: ${ref.name}');
                names.add(ref.name);
            }
        }
    }
}
```

Excerto de Código 3.43: Função `fetchImagesFromStorage`

A função `fetchImagesFromStorage` tem como objetivo procurar imagens armazenadas no *Firebase Storage*. Ela começa tentando listar todos os itens (ficheiros) presentes no *storage*, identificado pela Uniform Resource Locator (URL) fornecida. Para isso, utiliza o método `listAll()` do *Firebase Storage*. Após obter a lista de itens, a função percorre cada item para obter os seus metadados, como o tipo de conteúdo. Se o tipo de conteúdo for uma imagem JPEG ou PNG, o nome do ficheiro é adicionado a uma lista. Esse processo permite filtrar e selecionar apenas as imagens armazenadas no *Firebase Storage*, descartando outros tipos de ficheiros. No final, a função coleta os nomes das imagens que correspondem aos tipos válidos e os armazena numa lista chamada `names`. Isso pode ser útil, por exemplo, para exibir essas imagens numa interface de utilizador ou realizar alguma outra operação com elas.

```
Future<String> getImageUrl(String imageName) async {
    return await FirebaseStorage.instance
        .refFromURL('gs://delivery-68030.appspot.com/$imageName')
        .getDownloadURL();
}
```

Excerto de Código 3.44: Função `getImageURL`

A função `getImageUrl` tenta obter o URL de uma imagem armazenada no *Firebase Storage*. Ela recebe como parâmetro o nome da imagem (`imageName`), usado para localizar o ficheiro no *Firebase Storage*. A função faz uma chamada assíncrona para o *Firebase Storage*, tentando aceder ao ficheiro através do URL de referência, que inclui o caminho para o ficheiro. Após localizar o ficheiro, a função solicita o URL de *download* para esse ficheiro, permitindo

que a imagem seja acessada em qualquer lugar da aplicação *WEB*. O URL devolvido numa *string*. Esta função é usada para carregar a imagem numa *interface* de administrador, principalmente no processo edição de um produto.

```
Future<void> updateProduct(String productID, String name, String
    description,
    double price, Map<String, String> ingredients, String? imageUrl)
    async {
try {
    await FirebaseFirestore.instance
        .collection('product')
        .doc(productID)
        .update({
            'name': name,
            'description': description,
            'price': price,
            'ingredients': ingredients,
            'image': imageUrl,
        });
} catch (e) {
    log('Error updating product: $e');
    rethrow;
}
}
```

Excerto de Código 3.45: Função *updateProduct*

A função ***updateProduct*** foi desenvolvida para facilitar a atualização de informações de um produto na base de dados. Ela recebe como parâmetros o *productId*, o nome, a descrição, o preço, uma lista de ingredientes, do tipo *map* e o URL da imagem correspondente. A função tenta atualizar os dados do produto associados ao *productId* na coleção *product* da base de dados, substituindo os campos nome, descrição, preço, ingredientes e imagem pelos valores fornecidos, caso estes sejam fornecidos. Se ocorrer um erro durante o processo de atualização, por exemplo, se o produto não for encontrado ou existir um problema com a ligação à base de dados, o erro é capturado e apresentado no terminal, para facilitar qualquer processo de *debugging*.

3.5.4 Google Maps API e *urlLauncher*

Em conjunto com os **BLoC's!** (**BLoC's!**) e o *Firebase*, foi utilizado também a API do Google Maps, para que quando o utilizador realiza um pedido, consiga de forma eficiente indicar a localização específica, através das coordenadas geográficas. Apesar disso, a forma tradicional de indicar a sua localização continua presente, permitindo ao utilizador escrever a sua morada ou indicar com coordenadas geográficas. Além disso, também foi utilizado o *urlLaunch-*

cher na aplicação para abrir que o estafeta consiga, de forma intuitiva, através de um botão, ir diretamente para o *Google Maps* e conseguir obter as direções do estabelecimento para o local de entrega pretendido.

3.5.4.1 Google Maps API

Para que a aplicação móvel consiga usar o *Google Maps*, é necessário importar a dependência associada ao *Google Maps*, tal como a dependência *Geocoding*, que serve para obter as coordenadas geográficas. Ambas as dependências são importadas no ficheiro *pubspec.yaml*.

```
dependencies:
```

```
google_maps_flutter: ^2.0.6
geocoding: ^2.0.0
```

Exerto de Código 3.46: *pubspec.yaml* e dependências necessárias

Após a importação das dependências, podemos prosseguir com o desenvolvimento da página para permitir indicar as coordenadas.

```
class MapView extends StatefulWidget {
  @override
  _MapViewState createState() => _MapViewState();
}

class _MapViewState extends State<MapView> {
  LatLng _selectedLocation = const LatLng(38.71667, -9.13333);

  void _onTap(LatLng location) => setState(() => _selectedLocation =
    location);
  void _onOkPressed() => Navigator.of(context).pop(_selectedLocation);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Center(child: Image.asset('images/Logo.png', height: 70))
      ),
      body: Padding(
        padding: const EdgeInsets.all(20),
        child: Column(
          children: [
            const Text('Pin Point your location', style: TextStyle(
              fontSize: 18, fontWeight: FontWeight.bold)),
            const SizedBox(height: 10),
            Container(
              padding: const EdgeInsets.all(10),

```

```
decoration: BoxDecoration(
    border: Border.all(color: Colors.grey, width: 2),
    borderRadius: BorderRadius.circular(10),
),
height: 400,
width: double.infinity,
child: GoogleMap(
    onMapCreated: _onMapCreated,
    initialCameraPosition: CameraPosition(target:
        _selectedLocation, zoom: 12),
    onTap: _onTap,
    markers: {
        Marker(markerId: const MarkerId('selected-location'),
            position: _selectedLocation),
    },
),
),
Align(
    alignment: Alignment.bottomCenter,
    child: Padding(
        padding: const EdgeInsets.all(16),
        child: SizedBox(
            width: MediaQuery.of(context).size.width,
            child: TextButton(
                onPressed: _onOkPressed,
                style: TextButton.styleFrom(
                    elevation: 3,
                    backgroundColor: const Color.fromRGBO(252, 185,
                        19, 1),
                    shape: RoundedRectangleBorder(borderRadius:
                        BorderRadius.circular(10)),
                ),
                child: const Text("OK", style: TextStyle(color:
                    Colors.white, fontSize: 20, fontWeight:
                    FontWeight.w600)),
            ),
        ),
    ),
),
],
),
),
),
),
);
}
}
```

Excerto de Código 3.47: MapView.dart

Aqui definimos a classe `MapView`, que cria uma *interface* interactiva para escolher uma localização no mapa, utilizando o *Google Maps*. A funcionali-

dade é implementada utilizando um widget de estado (*StatefulWidget*), que permite ao utilizador escolher um ponto no mapa e confirmar a sua seleção.

A classe tem as seguintes funcionalidades principais:

- **Localização inicial:** A localização predefinida é Lisboa, Portugal (latitude: 38,71667, longitude: -9,13333). Esta localização é apresentada no mapa no arranque.
- **Seleção da localização:** O utilizador pode clicar em qualquer parte do mapa para selecionar uma nova localização. Sempre que isto acontece, a variável `_selectedLocation` é atualizada e é colocado um marcador na localização escolhida.
- **Confirmação da seleção:** A *interface* inclui um botão que, quando premido, executa a função `_onOkPressed`. Esta função fecha o ecrã e devolve a localização selecionada à parte da aplicação que chamou esta *interface*. Isto é feito utilizando `Navigator.of(context).pop(_selectedLocation)`.

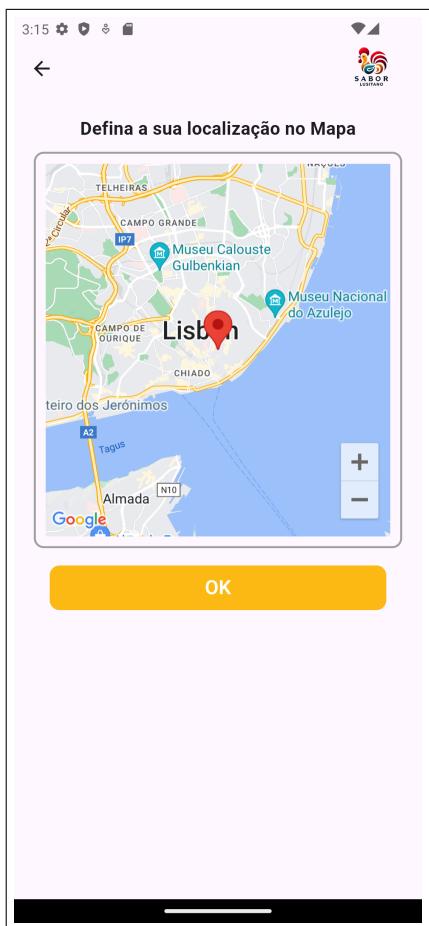


Figura 3.17: Escolha da localização

3.5.5 Dependência *urlLauncher*

Já no processo da aplicação *WEB*, em vez de importar a dependência associada ao *Google Maps*, é mais fácil importar a dependência *urlLauncher* permite abrir URLs de forma simples e eficiente, seja para navegar em páginas *WEB*, fazer chamadas telefónicas, enviar mensagens SMS ou abrir aplicações específicas que suportam determinados URLs (como mapas ou redes sociais).

Neste caso vai ser usado apenas para abrir o *Google Maps* na *WEB*, para usar obter as direções de forma simples e direta. Para complementar, temos que voltar a importar a dependência *geolocator* e *geolocator_web*.

```
dependencies:
```

```
  url_launcher: ^6.0.20
  geolocator: ^8.0.0
```

```
geolocator_web: ^2.0.0
```

Exerto de Código 3.48: pubspec.yaml da aplicação WEB

Após a importação, basta proceder à implementação da função que abra o *Google Maps*.

```
void _openGoogleMaps(String address) async {
    try {
        final query = Uri.encodeComponent(address);
        final googleMapsUrl =
            'https://www.google.com/maps/dir/?api=1&destination=$query';

        if (await canLaunch(googleMapsUrl)) {
            await launch(googleMapsUrl);
        } else {
            throw 'Could not open Google Maps';
        }
    } catch (e) {
        print('Error: $e');
    }
}
```

Exerto de Código 3.49: pubspec.yaml da aplicação WEB

A função *_openGoogleMaps* é utilizada para abrir o Google Maps no dispositivo do utilizador, apresentando o percurso para um endereço específico. Recebe como parâmetro uma cadeia de caracteres chamada *address*, que representa o endereço de destino. Primeiro, codifica o endereço utilizando *Uri.encodeComponent*, assegurando que os caracteres especiais são corretamente representados no URL.

Em seguida, constrói um URL utilizando a API de direções do Google Maps, adicionando o endereço codificado como destino. A função verifica se o dispositivo suporta a abertura do URL gerado utilizando *canLaunch*. Em caso afirmativo, utiliza o *launch* para abrir o Google Maps com as direções para o destino fornecido. Caso contrário, lança uma exceção indicando que o Google Maps não pôde ser aberto.

Os erros durante o processo são tratados e apresentados na consola, permitindo o *debugging*.

Para facilitar todo o processo ao estafeta, a função está a um clique de um botão:

```
IconButton(
    icon: const Icon(Icons.map),
    color: Colors.blue,
    iconSize: 32.0,
    onPressed: () =>
        _openGoogleMaps(request['address']),
```

),

Excerto de Código 3.50: Icon Button

Quando se clica no botão obtém-se este resultado:

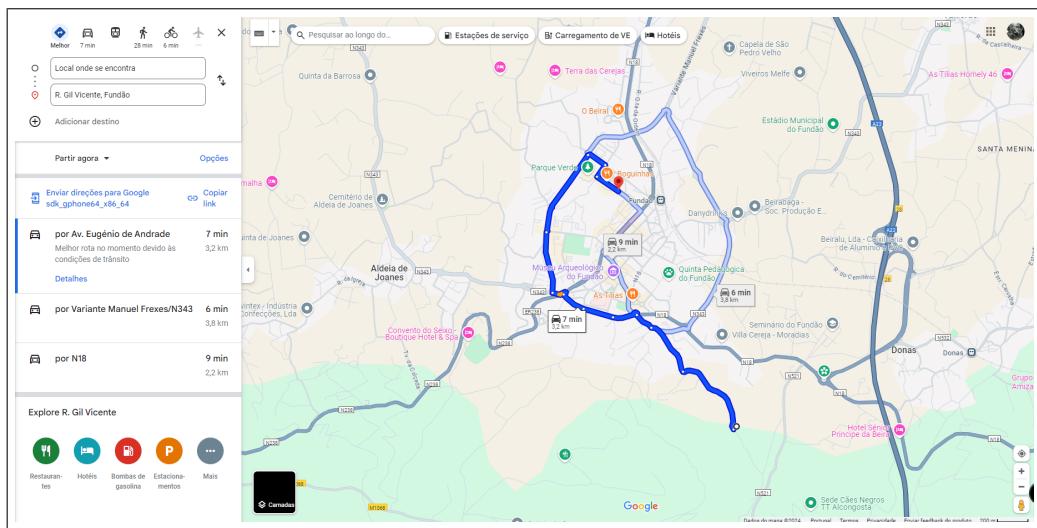


Figura 3.18: Resultado

3.6 Multi Linguagem

3.6.1 Ficheiros XML

Para permitir que ambas as aplicações possam ser utilizadas pelo máximo de utilizadores possíveis, foi necessário realizar a tradução de português para inglês e oferecer a possibilidade ao utilizador de alternar entre as duas línguas.

Para isso, foi necessário criar dois ficheiros Extensible Markup Language (XML) que vão conter todos os *inputs* necessários.

```
<resources>
    <string name="title">Title</string>
    <string name="welcome_message">Welcome to our app!</string>
    <string name="add_to_cart">Add to Cart</string>
    <string name="enter_quantity">Enter quantity:</string>
    <string name="cancel">Cancel</string>
    <string name="home">Home</string>
    <string name="requests">Requests</string>
    <string name="something_went_wrong">Oops! Something went wrong!</string>
    <string name="no_products_available">No products available</string>
    <string name="switch_to_portuguese">Switch to Portuguese</string>
```

```
<string name="switch_to_english">Switch to English</string>
<string name="product_not_found">Product not found</string>
<string name="feedback">Feedback</string>
<string name="no_feedbacks_found">No feedbacks found</string>
<string name="error_loading_feedbacks">Error loading feedbacks</
    string>
<string name="from">From</string>
<string name="user_not_signed_in">User not signed in</string>
<string name="your_requests">Your Requests</string>
<string name="no_requests_found">No requests found</string>
<string name="total_price">Total Price</string>
<string name="date">Date</string>
<string name="in_progress">In Progress</string>
<string name="delivery">Delivery</string>
<string name="completed">Completed</string>
<string name="unknown_status">Canceled</string>
<string name="leave_feedback">Leave your feedback, so we can improve
    things!</string>
<string name="feedback_important">Your feedback is important to us.
    It helps us improve our services and products.</string>
<!-- Continua -->
</resources>
```

Excerto de Código 3.51: stringsEN.xml

```
<resources>
    <string name="title">Titulo</string>
    <string name="welcome_message">Bem-vindo!</string>
    <string name="add_to_cart">Adicionar ao Carrinho</string>
    <string name="enter_quantity">Digite a quantidade:</string>
    <string name="cancel">Cancelar</string>
    <string name="home">Inicio</string>
    <string name="requests">Pedidos</string>
    <string name="something_went_wrong">Ups! Algo deu errado!</string>
    <string name="no_products_available">Nenhum produto disponivel</string>
    <string name="switch_to_english">Mudar para Ingles</string>
    <string name="switch_to_portuguese">Mudar para Portugues</string>
    <string name="product_not_found">Produto nao encontrado</string>
    <string name="feedback">Feedback</string>
    <string name="no_feedbacks_found">Nenhum feedback encontrado</string>
    <string name="error_loading_feedbacks">Erro ao carregar feedbacks</string>
    <string name="from">De</string>
    <string name="user_not_signed_in">Usuario nao conectado</string>
    <string name="your_requests">Seus Pedidos</string>
    <string name="no_requests_found">Nenhum pedido encontrado</string>
    <string name="total_price">Preco Total</string>
    <string name="date">Data</string>
    <string name="in_progress">Em Progresso</string>
    <string name="delivery">Em Entrega</string>
    <string name="completed">Concluido</string>
    <string name="unknown_status">Cancelado</string>
    <string name="leave_feedback">Deixe seu feedback, para que possamos melhorar o nosso servico!</string>
    <string name="feedback_important">Seu feedback e importante para nos . Ele ajuda-nos a melhorar a qualidade dos nossos servicos e produtos.</string>
    <!-- Continua -->
</resources>
```

Exerto de Código 3.52: stringsPT.xml

Após criar os ficheiros XML, foi necessário criar uma função que verifique a linguagem e escolha qual ficheiro deve dar o seu devido uso.

```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:flutter/services.dart' show rootBundle;
import 'package:xml/xml.dart';

class TranslationProvider with ChangeNotifier {
  Map<String, String> _localizedStrings = {};
  Locale _locale = Locale('en');

  Locale get locale => _locale;

  Future<void> load(Locale locale) async {
    _locale = locale;
    String xmlString = await rootBundle
        .loadString('assets/strings_${locale.languageCode}.xml');
    final document = XmlDocument.parse(xmlString);
    final resources = document.findAllElements('string');
    _localizedStrings = {
      for (var element in resources)
        element.getAttribute('name')!: element.text,
    };
    notifyListeners();
  }

  String translate(String key) {
    return _localizedStrings[key] ?? key;
  }
}
```

Excerto de Código 3.53: translationProvider.dart

A função *TranslationProvider* é responsável por verificar a linguagem que o utilizador escolheu e procurar pelo ficheiro XML correspondente. Carrega traduções de ficheiros XML, mapeando as chaves dos atributos de nome para os seus valores traduzidos. O método *load* atualiza o idioma (definido pelo objeto *Locale*), carrega as cadeias de caracteres traduzidas do ficheiro correspondente (por exemplo, strings_en.xml).

Para facilitar todo o processo, foi criado um botão que altera entre português e inglês, passando a constante *Locale* na função *TranslationProvider*.

```
GestureDetector(
  onTap: () async {
    final translationProvider =
        Provider.of<TranslationProvider>(context, listen: false);
    if (translationProvider.locale.languageCode == 'en') {
      await translationProvider.load(const Locale('pt'));
```

```
        } else {
          await translationProvider.load(const Locale('en'));
        }

        setState(() {});
      },
      child: Container(
        decoration: BoxDecoration(
          color: Colors.transparent,
          borderRadius: BorderRadius.circular(30.0),
        ),
        child: Image.asset(
          translationProvider.locale.languageCode == 'en'
            ? 'images/uk.png'
            : 'images/portugal.png',
          height: 30,
          width: 30,
        ),
      ),
    ),
  ),
),
```

Exerto de Código 3.54: Change Language

Quando o utilizador clica no *GestureDetector*, o método *onTap* verifica o idioma atual através do *TranslationProvider* e carrega o idioma oposto através do método *load*. Em seguida, chama *setState* para atualizar a *interface*. O ícone apresentado no botão muda dinamicamente entre as bandeiras do Reino Unido (uk.png) e de Portugal (portugal.png) para indicar ao utilizador a língua ativa.

3.6.2 Azure Translation API

Além disso, é necessário também traduzir as descrições dos pratos de português para inglês. Para isso, visto que a descrição de um produto está inserido no *Firebase*, não é vantajoso colocar num ficheiro XML, visto que cada vez que se adiciona um prato teria que existir atualização do ficheiro com a descrição desse respetivo prato. Logo, o *Azure Translation API* permite traduzir o que a aplicação recebe do campo descrição para inglês.

```
import 'package:http/http.dart' as http;
import 'dart:convert';

class TranslationService {
  static const String _subscriptionKey =
    'API_KEY';
  static const String _endpoint =
    'https://api.cognitive.microsofttranslator.com/translate?api-
version=3.0';
```

```

static Future<String> translateText(String text) async {
  final url = Uri.parse('$_endpoint&from=pt&to=en');

  final response = await http.post(
    url,
    headers: {
      'Ocp-Apim-Subscription-Key': _subscriptionKey,
      'Content-Type': 'application/json',
      'Ocp-Apim-Subscription-Region': 'westeurope', // e.g., 'westus'
    },
    body: json.encode([
      {'Text': text}
    ]),
  );

  if (response.statusCode == 200) {
    final jsonResponse = json.decode(response.body);
    return jsonResponse[0]['translations'][0]['text'];
  } else {
    throw Exception('Failed to translate text');
  }
}
}

```

Excerto de Código 3.55: Translation API

A classe *TranslationService* utiliza a *Azure Translation API* para traduzir texto de português (pt) para inglês (en). A função estática *translateText* recebe um texto como entrada, envia um pedido POST para o ponto final da API com o texto no corpo e, se a resposta for bem sucedida (*statusCode* == 200), devolve o texto traduzido. Caso contrário, lança uma exceção indicando uma falha na tradução.

Após a implementação da classe que realiza a tradução, foi necessário alterar como aplicação se comporta quando recebe o campo descrição diretamente da *Firebase Storage*.

```

FutureBuilder<String>(
  future: translationProvider.locale.languageCode == 'en'
    ? TranslationService.translateText(
        productData['description'] ?? 'Description',
      )
    : Future.value(
        productData['description'] ?? 'Description',
      ),
)

```

Excerto de Código 3.56: FutureBuilder para o campo descrição

O *FutureBuilder* acima carrega dinamicamente a descrição do produto traduzida ou original, consoante o idioma ativo. Se o idioma for o inglês, utiliza o *TranslationService* para traduzir o texto, caso contrário, devolve diretamente o texto original que está na *Firebase Storage*.

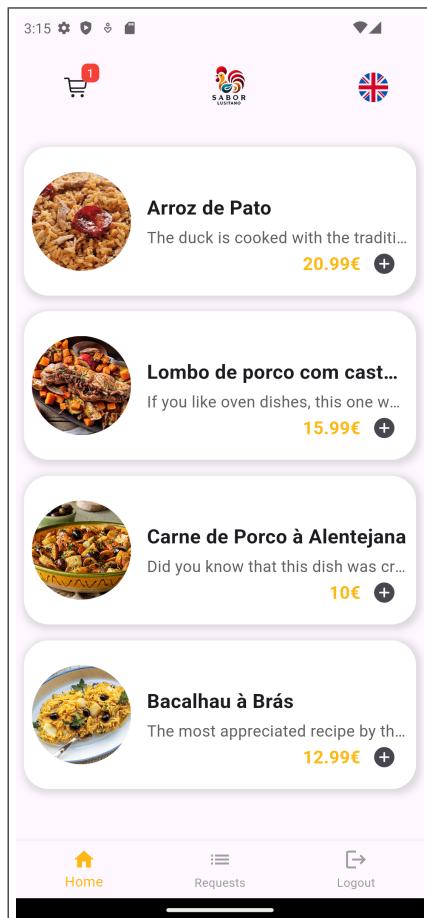


Figura 3.19: APP em Inglês

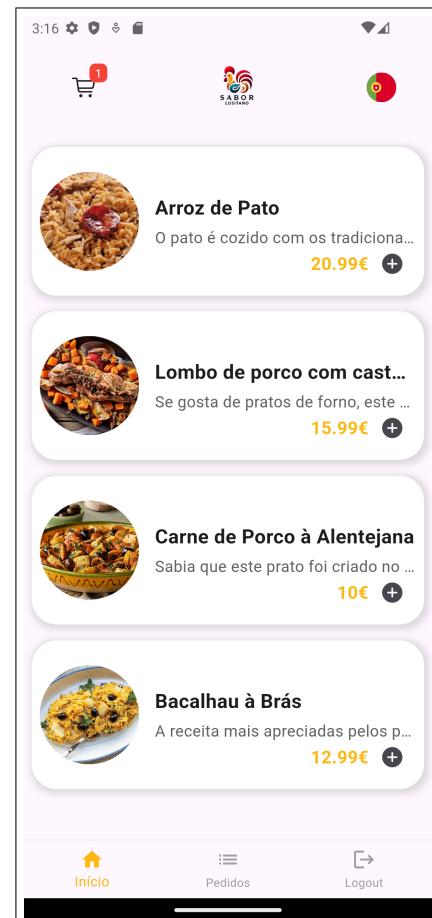


Figura 3.20: APP em Português

3.7 API para pagamentos online

Para além de o utilizador poder pagar o produto pedido no ato de entrega, a aplicação móvel também permite que os utilizadores efetuem pagamentos de forma segura e eficiente. Para isso é necessário a implementação de uma API de pagamentos *online*.

3.7.1 Braintree

Para a implementação do método de pagamentos *online*, foi utilizado o *Braintree*. O *Braintree* é uma plataforma de pagamentos *online* que permite às empresas processar transações de forma segura e eficiente. Como faz parte do grupo PayPal, a Braintree oferece soluções para pagamentos com cartão de crédito, pagamentos com cartão de débito, carteiras digitais, como *PayPal* e *Google Pay*. No caso da aplicação móvel apenas foi utilizado o pagamento mediante cartão de crédito e *PayPal*.

```
import 'package:flutter_braintree/flutter_braintree.dart';
import 'package:flutter/material.dart';

class PaymentService {
  Future<String?> startBraintreePayment(
    BuildContext context, double totalPrice) async {
    var request = BraintreeDropInRequest(
      tokenizationKey: 'sandbox_2438xwxw_rq8zs4nkvb48x775',
      collectDeviceData: true,
      googlePaymentRequest: BraintreeGooglePaymentRequest(
        totalPrice: totalPrice.toStringAsFixed(2),
        currencyCode: 'EUR',
        billingAddressRequired: false,
      ),
      paypalRequest: BraintreePayPalRequest(
        amount: totalPrice.toStringAsFixed(2),
        displayName: 'APP',
      ),
      cardEnabled: true,
    );
    BraintreeDropInResult? result = await BraintreeDropIn.start(request);
    if (result != null) {
      print('Payment method nonce: ${result.paymentMethodNonce.nonce}');
      // Return nonce to be used for processing
      return result.paymentMethodNonce.nonce;
    } else {
      print('Payment cancelled');
      return null;
    }
  }
}
```

```
    }
}
```

Excerto de Código 3.57: Braintree API

A classe *PaymentService* tem um método chamado *startBraintreePayment*, que é responsável por iniciar o fluxo de pagamento e retornar um *nonce*. Este *nonce* é um identificador temporário utilizado para autenticar e processar o pagamento no servidor *Braintree*.

O método recebe dois parâmetros: o contexto, que representa o contexto de aplicação necessário para apresentar a interface de pagamento, e o *totalPrice*, que é o montante total a cobrar. Com estes dados, configura um *BraintreeDropInRequest*, definindo as opções de pagamento disponíveis, tais como *Google Pay*, *PayPal* e cartões de crédito ou débito. A chave de tokenização, fornecida pela *Braintree*, autentica a aplicação no ambiente *sandbox*, enquanto outras configurações, como a recolha de dados do dispositivo e a moeda utilizada, são ajustadas.

Quando o método *BraintreeDropIn.start(request)* é chamado, apresenta uma *interface* para o utilizador selecionar o método de pagamento. Se o pagamento for concluído com êxito, é devolvido o método de pagamento que foi selecionado na consola de debug. Se o utilizador optar por cancelar o processo, o método devolve *null*.

```
TextButton(
    onPressed: () async {
        List<String> productIds = productsToShow
            .map((product) => product[ 'id' ] as String)
            .toList();
        List<Map<String , dynamic>> productList =
            productsToShow
                .map((product) => product[ 'data' ] as Map<String , dynamic
                    >)
                .toList();
        List<int> productQuantities = productsToShow
            .map((product) => product[ 'quantity' ] ?? 1)
            .cast<int>()
            .toList();

        final paymentService = PaymentService();
        final nonce = await paymentService.startBraintreePayment(
            context, double.parse(totalPrice));
        if (nonce != null) {
            sendLocationToFirebase(
                context: context,
                productIds: productIds,
                productList: productList,
```

```
productQuantities: productQuantities,
numberController: _numberController,
observationsController: _observationsController,
addressController: _addressController,
selectedLocation: _selectedLocation,
showDialog: _showDialog,
paymentMethod: 'Credit Card',
price: double.parse(totalPrice),
);
},
),
);
```

Excerto de Código 3.58: Integração do Braintree

O código acima define a ação de um botão que processa um pagamento utilizando o Braintree e depois regista a informação no Firebase. Começa por recolher os dados dos produtos selecionados, criando listas com os seus *IDs*, detalhes e quantidades. Em seguida, integra-se com o Braintree usando o método *startBraintreePayment* da classe *PaymentService*, que inicia o processo de pagamento e obtém o *nonce*. Se o pagamento for concluído com sucesso, ou seja, se o *nonce* não for nulo, a função *sendLocationToFirebase* é chamada para guardar no Firebase as informações sobre os produtos comprados, a localização, as observações adicionais e o método de pagamento utilizado.

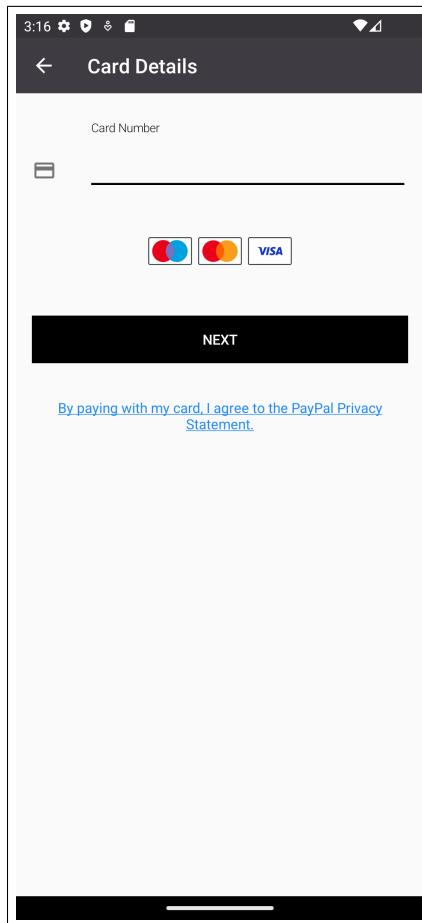


Figura 3.21: Processo de Pagamento do Braintree

3.8 Testes

Após o desenvolvimento de ambas as aplicações, a fim de realizar os testes necessários, foi implementado um formulário baseado numa escala de *Likert* para recolher o *feedback* dos utilizadores sobre a experiência de utilização e a funcionalidade das plataformas. A aplicação móvel destinava-se aos clientes finais, enquanto a aplicação Web foi desenvolvida para permitir aos restaurantes gerir encomendas, produtos, informações sobre os clientes e aceder a dados estatísticos relevantes. O formulário foi atribuído a um grupo de 10 pessoas, cinco utilizadores para a aplicação móvel e cinco para a aplicação Web. No entanto, apenas sete participantes preencheram o formulário, o que permitiu obter informações valiosas para avaliar o estado, identificar áreas a melhorar e garantir a satisfação dos utilizadores em ambos os contextos. As

perguntas feitas aos utilizadores foram baseadas numa escala de um a cinco, onde um significa 'Discordo totalmente' e cinco seria 'Concordo totalmente'.

As questões colocadas foram:

- A interface da aplicação é intuitiva e fácil de usar.
- É fácil encontrar as funcionalidades que preciso.
- O processo de registo/login foi simples e eficaz.
- A aplicação responde rapidamente aos comandos.
- A aplicação funciona bem em diferentes dispositivos (telemóvel, tablet, computador).
- O processo de pesquisa e acesso aos produtos é eficiente.
- Os métodos de pagamento disponíveis são práticos e seguros.
- Estou satisfeito com a experiência geral da aplicação.
- Recomendaria esta aplicação a outras pessoas.
- Voltaria a utilizar esta aplicação no futuro.
- Deixa a tua opinião naquilo que pode ser melhorado.

Após a análise dos resultados do formulário, obteve-se uma opinião globalmente positiva sobre as aplicações Web e móveis. A interface foi amplamente elogiada pela sua facilidade de utilização, recebendo classificações maioritariamente entre 4 e 5 na escala de *Likert*. As funcionalidades foram bem avaliadas, especialmente no que diz respeito à eficiência do acesso e da resposta aos comandos, refletindo uma implementação robusta.

No entanto, foram identificadas áreas específicas de melhoria, como na aplicação móvel, onde houve relatos de textos desalinhados em dispositivos como o Samsung S10, bem como pedidos de métodos de pagamento mais diversificados, como o MB Way.

Na aplicação web, alguns utilizadores sugeriram melhorias para cenários com ligação limitada à internet. A satisfação geral foi positiva, com a maioria dos utilizadores a indicar que recomendaria as aplicações e a mostrar uma elevada intenção de as reutilizar. No entanto, a taxa de resposta ao formulário, que foi de 70%, indica que há margem para um maior envolvimento dos utilizadores em futuros processos de feedback.

Todas as respostas estão armazenadas num ficheiro *Excel* acessível ao público. (<https://tinyurl.com/296ykg9f>). Todos os dados pessoais foram ocultados para manter a privacidade do utilizador. Todos os dados estatísticos podem ser consultados no anexo associado aos resultados (anexo)

Com as opiniões e *feedback's* dados, foi possível detetar e resolver problemas que não foram identificados antes, como os textos desalinhados, a velocidade de conexão à base de dados e melhorar consequentemente a resposta dos comandos, tal como trabalhos futuros a realizar nas aplicações, como implementações de novos métodos de pagamento.

3.9 Conclusões

Em suma, este capítulo detalhou o processo de implementação e teste das aplicações móveis e Web, demonstrando como normas de arquitetura como os BLoC's, com a utilização do *Firebase* para persistência de dados e autenticação, foram cruciais para o desenvolvimento de um produto robusto. A utilização de serviços de terceiros, como *Google Maps*, *urlLauncher*, *Azure Translate API* e *Braintree* acrescentaram funcionalidades importantes que melhoraram a experiência do utilizador. Os BLoC's foram essenciais na gestão de estados nas *interfaces* de utilizador, permitindo uma separação clara entre a apresentação e a lógica comercial. A gestão da base de dados através do *Firebase* permitiu a integração perfeita e um acesso eficiente aos dados, bem como a autenticação. A dependência de outros serviços, como o *Google Maps* e o *Braintree*, também desempenhou um papel crucial na usabilidade e funcionalidade da aplicação

Por fim, os testes efetuados através de formulários baseados numa escala *Likert* forneceram informações valiosas, destacando áreas de sucesso e outras que requerem melhorias.

Capítulo

4

Conclusão

Neste capítulo final são apresentadas as principais conclusões da realização deste projeto e os trabalhos futuros que podem ser realizados para elevar a aplicação

4.1 Conclusões Principais

O desenvolvimento da *Delivery APP* permitiu proporcionar uma solução integrada para clientes de um determinado restaurante, destacando-se pela gestão eficiente de produtos, encomendas e entregas. O uso de tecnologias como *Flutter*, *Firebase* e *Google Maps API* garantiu funcionalidades modernas e de alto desempenho, incluindo geolocalização precisa, sincronização de dados em tempo real e pagamentos seguros. Além disso, a separação da lógica através de *BLoC's* facilita a manutenção do sistema.

Esta aplicação também teve um impacto prático significativo, indo ao encontro das necessidades específicas dos restaurantes ao eliminar as elevadas taxas de terceiros, como foi referido no capítulo 2.2, permitindo também um maior controlo sobre a experiência do cliente. A utilização de testes automatizados e manuais assegurou um produto robusto e estável, enquanto a escolha de tecnologias modernas garantiu a escalabilidade e abriu caminho para futuras expansões. Estes resultados reforçam a capacidade da aplicação para competir diretamente com as grandes plataformas existentes, proporcionando autonomia aos restaurantes e melhorando a experiência do cliente final.

4.2 Trabalho Futuro

No futuro, o objetivo é vender os direitos de utilização da aplicação aos restaurantes, permitindo-lhes reduzir os custos com plataformas de terceiros e reforçar a relação com os seus clientes. Para tal, será necessário adaptar a aplicação às necessidades de cada restaurante, incluindo personalizações e integração com métodos de pagamento locais. Um exemplo relevante seria a implementação do *MBWAY*, caso a aplicação fosse comercializada em Portugal, a pedido de um dos vários utilizadores que testaram a aplicação. Esta característica não foi desenvolvida atualmente devido ao foco colocado nas funcionalidades principais e também devido à necessidade de investigar os requisitos técnicos e legais específicos de cada mercado.

Outra melhoria futura é o desenvolvimento de uma versão iOS. Por enquanto, a aplicação móvel foi apenas desenvolvida para Android devido a limitações de recursos, visto que é necessário um sistema *macOS* para a compilação da aplicação. A expansão para iOS aumentaria o público-alvo e tornaria a aplicação mais competitiva.

Estas melhorias representam passos importantes para fazer crescer a aplicação e torná-la ainda mais eficaz no mercado.

Bibliografia

Flutter Documentation [Online]

Link: <https://docs.flutter.dev>

Último acesso a: 16/12/2024.

Pub.dev [Online]

Link: <https://pub.dev>

Último acesso a: 4/12/2024.

Flutter Braintree [Online]

Link: https://pub.dev/packages/flutter_braintree

Último acesso a: 4/12/2024.

Braintree Documentation [Online]

Link: <https://developer.paypal.com/braintree/docs/>

Último acesso a: 4/12/2024.

Google Maps Flutter [Online]

Link: https://pub.dev/packages/google_maps_flutter

Último acesso a: 6/11/2024.

Geocoding Package [Online]

Link: <https://pub.dev/packages/geocoding>

Último acesso a: 6/11/2024.

Flutter Bloc [Online]

Link: https://pub.dev/packages/flutter_bloc

Último acesso a: 15/10/2024.

Flutter BLoC for Beginners [Online]

Link: <https://medium.com/flutter-community/flutter-bloc-for-beginners-839e2>

Último acesso a: 15/10/2024.

Firebase Core [Online]

Link: https://pub.dev/packages/firebase_core

Último acesso a: 2/12/2024.

Firebase Authentication [Online]

Link: https://pub.dev/packages/firebase_auth

Último acesso a: 2/12/2024.

Cloud Firestore [Online]

Link: https://pub.dev/packages/cloud_firestore

Último acesso a: 2/12/2024.

Firebase Storage [Online]

Link: https://pub.dev/packages/firebase_storage

Último acesso a: 2/12/2024.

Azure Documentation [Online]

Link: <https://learn.microsoft.com/en-us/azure/?product=popular>

Último acesso a: 4/12/2024.

Firebase Documentation [Online]

Link: <https://firebase.google.com/docs>

Último acesso a: 4/12/2024.

Link: <https://miro.com/>

Último acesso a: 20/1/2025.

Anexo

Resultados dos testes de utilização

Neste anexo estão presentes todos os gráficos de barras referentes às questões colocadas aos utilizadores que testaram ambas as aplicações, tal como referido na secção de testes (3.8).

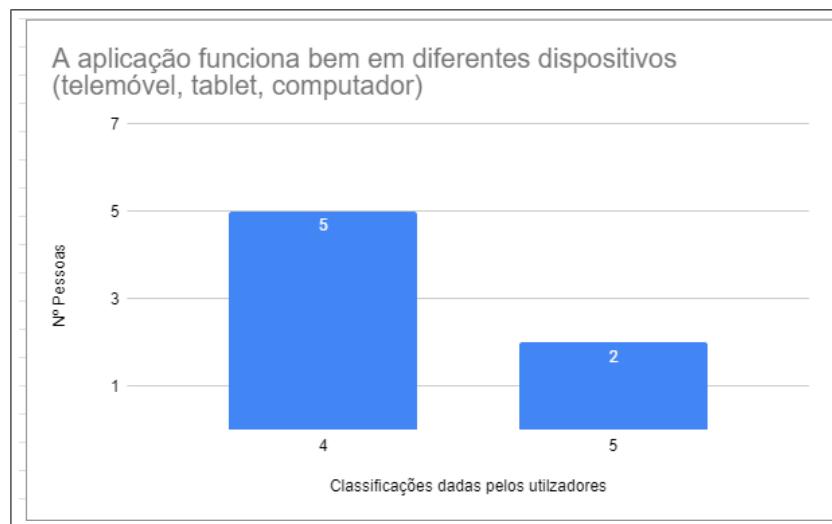


Figura 1: A aplicação funciona bem em diferentes dispositivos

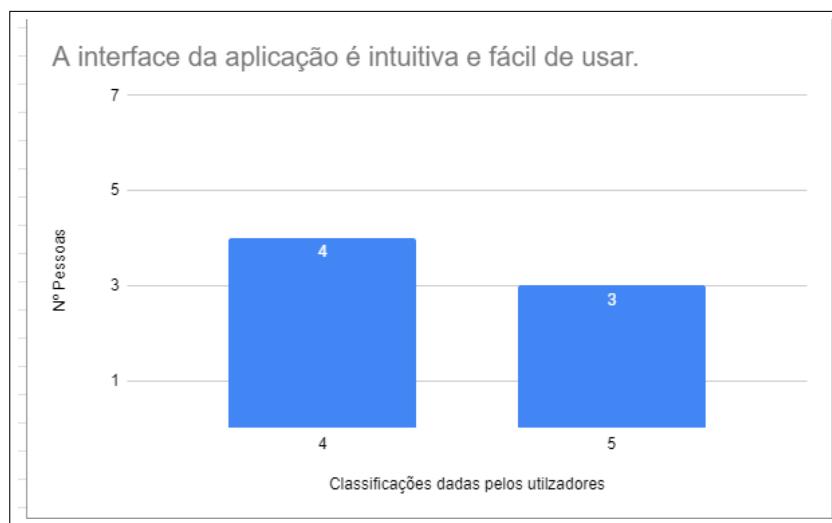


Figura 2: A interface da aplicação é intuitiva e fácil de usar.



Figura 3: A aplicação responde rapidamente aos comandos.

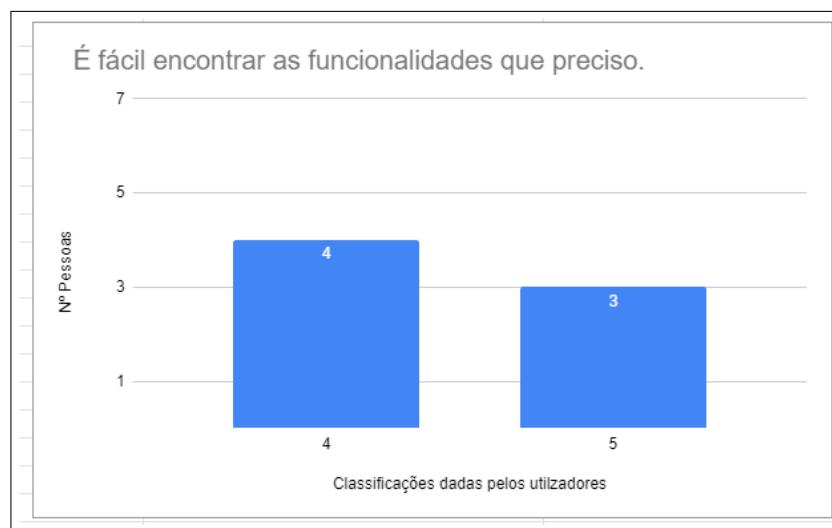


Figura 4: É fácil encontrar as funcionalidades que preciso.

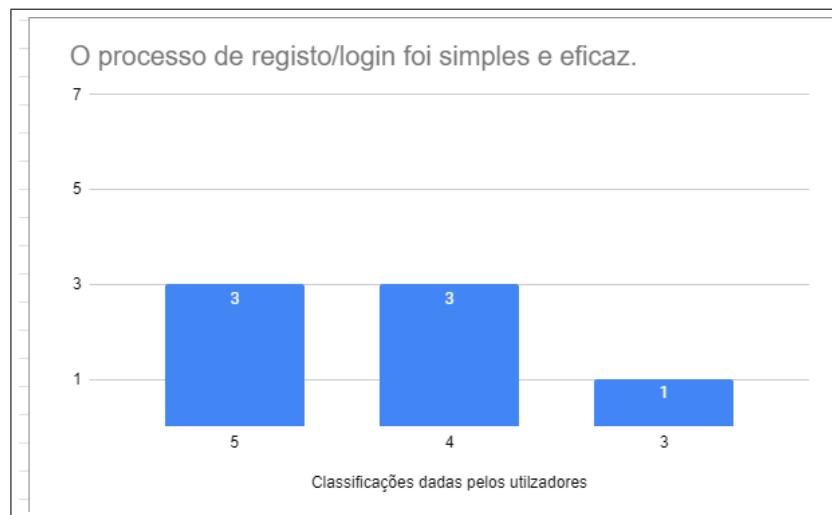


Figura 5: O processo de registo/login foi simples e eficaz.

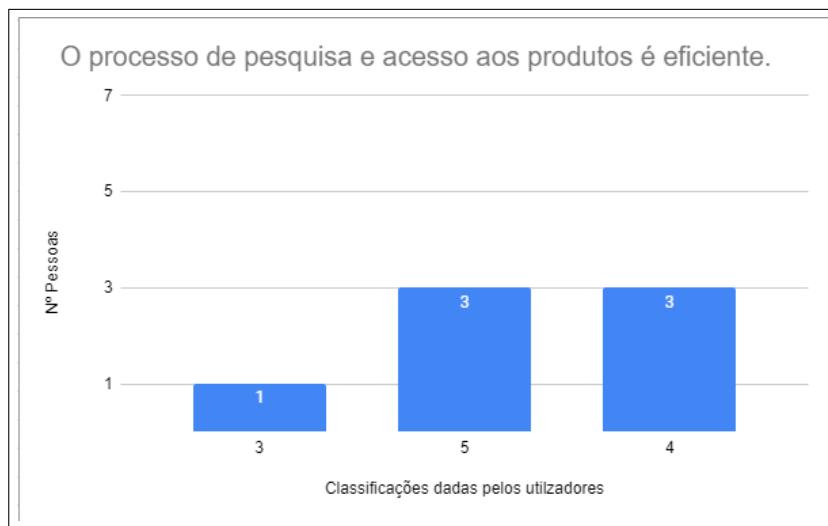


Figura 6: O processo de pesquisa e acesso aos produtos é eficiente.

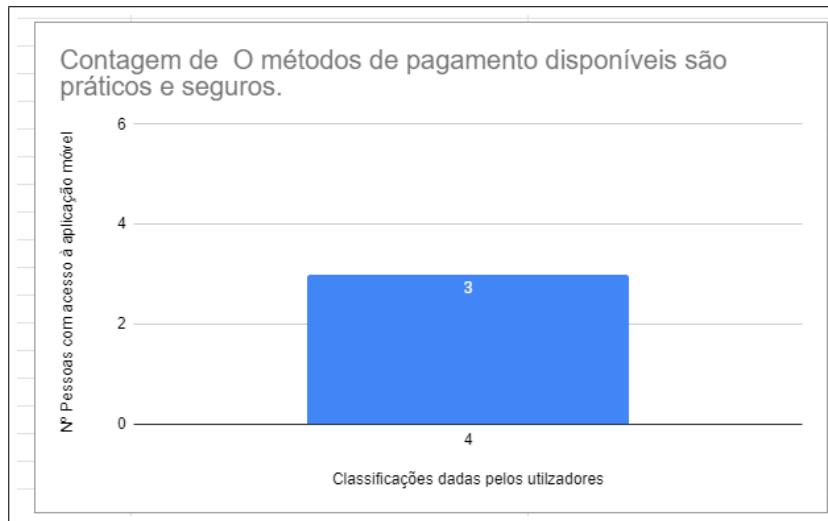


Figura 7: O métodos de pagamento disponíveis são práticos e seguros.

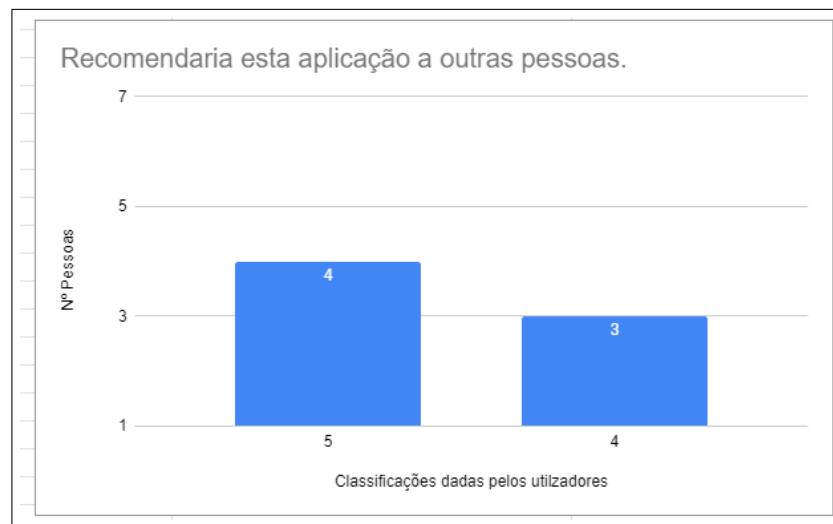


Figura 8: Recomendaria esta aplicação a outras pessoas.