Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Thiago Martiusi Moura

DESENVOLVIMENTO DE APLICATIVO MÓVEL PARA PIZZARIA

Americana
2021

Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Thiago Martiusi Moura

DESENVOLVIMENTO DE APLICATIVO MÓVEL PARA PIZZARIA

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do Professor Dr. Kleber de Oliveira Andrade.

Área de concentração: Programação para dispositivos móveis.

Americana
2021

Fatec Americana Ministro Ralph Biasi

Centro Paula Souza

SÃO PAULO GOVERNO DO ESTADO

Secretaria de Desenvolvimento Econômico

Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Thiago Martiusi Moura

DESENVOLVIMENTO DE APLICATIVO MÓVEL PARA PIZZARIA

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do Professor Dr. Kleber de Oliveira Andrade.

Área de concentração: Programação para dispositivos móveis.

Americana, 06 de dezembro de 2021.

Banca Examinadora:

Kleber de Oliveira Andrade
Doutor em Engenharia Mecânica
FATEC-AM

José Luiz Zem
Doutor em Física

Clerivaldo José Roccia
Mestre em Tecnologia

AGRADECIMENTOS

Em primeiro, minha gratidão à minha esposa, meu grande apoio em todos os momentos. À minha família, meu alicerce ao longo de todo o tempo. Agradeço ao professor orientador Kleber por toda atenção e dedicação que contribuíram sobremaneira para a concretização deste trabalho. Ao professor Antonio Lacerda cujos ensinamentos foram de grande valia nesta obra. Ao colega de classe André Nunes por sua relevante contribuição aos resultados alcançados. À FATEC – Americana pela elevada qualidade técnica e consideração ímpar aos discentes para a conclusão das atividades acadêmicas em meio a uma época de grandes provações. Por fim, agradeço à pizzaria Alto Gusto pela parceria nesta obra.

DEDICATÓRIA

Dedico este trabalho acima de tudo à Deus. À minha esposa e filhos, fontes de minha alegria e aos meus pais que a duras penas me oportunizaram a conclusão dos estudos.

Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Aprendemos a voar como os pássaros e a nadar como os peixes, mas não aprendemos a conviver como irmãos.

Martin Luther King

Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

RESUMO

Além dos desafios empresariais inerentes a cada negócio, as micro e pequenas empresas, assim como os profissionais autônomos, se deparam com as inovações que a tecnologia da informação insere no mercado, usualmente implementadas pelas grandes corporações. Dentre as quais se destaca o comércio online impulsionado pelos dispositivos móveis. Se fazer presente nesse canal de comunicação, contudo, requer investimento para oferecer um ambiente seguro e funcional, apto a oferecer uma boa experiência de uso a seus usuários. Nessa senda o presente trabalho busca desenvolver uma aplicação voltada ao comércio de entregas em domicílio, com foco nos pequenos estabelecimentos comerciais. Para tanto trabalha-se com a metodologia Scrum/Ágil para a construção de um aplicativo móvel para dispositivos com sistema operacional android, uma interface de programação de aplicações – API e um banco de dados relacional. As soluções desenvolvidas integram-se a uma interface de retaguarda – BackOffice voltada ao lojista. Essa interface, desenvolvida em parceria para este projeto, visa prover uma facilidade de visualização e controle pelo estabelecimento comercial e não integra o escopo deste trabalho. Nesse desenvolvimento, são definidas estratégias de segurança para as soluções como a criptografia de senha de acesso com algoritmo de função hash de 256 bits e definidos padrões de apresentação e usabilidade para obtenção de boa experiência de usuário, baseando-se nos protocolos das heurísticas de Nielsen, objetivando um bom engajamento dos clientes com a solução para que os lojistas consigam concretizar suas vendas pelo aplicativo, oferecendo facilidade, confiança e tranquilidade aos clientes ao longo de toda navegação. Por fim, faz-se uma análise de viabilidade, custos e mercado potencial para o fornecimento das soluções tecnológicas implementadas como produto ou como serviço de acordo com as necessidades, restrições e capacidades do público-alvo.

Palavras-chave: Aplicativo; Banco de Dados; Interface de Programação de Aplicações.

ABSTRACT

In addition to the business challenges inherent to each business, micro and small companies, as well as self-employed professionals, are faced with the innovations that information technology inserts into the market, usually implemented by large corporations. Among which, online commerce driven by mobile devices stands out. To be present in this communication channel, however, requires investment to offer a safe and functional environment, able to provide a good user experience to its users. In this context, the present work seeks to develop an application focused on home delivery commerce, with a focus on small commercial establishments. For this purpose, it works with the Scrum/Agile methodology to build a mobile application for devices with android operating system, an application programming interface – API and a relational database. The developed solutions are integrated with a BackOffice interface aimed at the retailer. This interface, developed in partnership for this project, aims to provide an easy visualization and control by the commercial establishment and is not part of the scope of this work. In this development, security strategies are defined for the solutions, such as password encryption with a 256-bit hash function algorithm, and presentation and usability standards are defined to obtain a good user experience, based on Nielsen's heuristics protocols, aiming at a good engagement of customers with the solution so that retailers can make their sales through the application, offering ease, confidence, and tranquility to customers throughout the navigation. Finally, an analysis is made of feasibility, costs, and potential market for the supply of technological solutions implemented as a product or as a service according to the needs, restrictions, and capabilities of the target audience.

Keywords: Application; Database; Application Programming Interface.

LISTA DE ILUSTRAÇÕES

Figura 1. Diagrama de caso de uso – credenciamento 19
Figura 2. Diagrama de caso de uso – seleção de produto 20
Figura 3. Diagrama de caso de uso – especificação 20
Figura 4. Diagrama de caso de uso – carrinho 21
Figura 5. Diagrama de caso de uso – pedido 22
Figura 6. Diagrama de classes 23
Figura 7. Model View Controller – MVC 24
Figura 8. Diagrama Entidade Relacionamento – DER 25
Figura 9. Variáveis de ambiente 27
Figura 10. Conexão com banco de dados 28
Figura 11. Middleware cors 29
Figura 12. Middlewares 29
Figura 13. Criptografia 31
Figura 14. Método saveUser 31
Figura 15. Método saveOrder 32
Figura 16. Splash_controller e Splash_page 34
Figura 17. Tela de login 35
Figura 18. Validações 36
Figura 19. Login 37
Figura 20. Tela de cadastro 38
Figura 21. Tela home_page 39
Figura 22. Cardápio 40
Figura 23. Método openMenu 41
Figura 24. Detalhes 42
Figura 25. Classe ItemController 43
Figura 26. Sacola 44
Figura 27. Pedido 45

Figura 28. Formas de pagamento 46
Figura 29. Página de pedido 47
Figura 30. Pedidos 48

LISTA DE TABELAS

Tabela 1: cronograma de entrega de *sprints*.....50
Tabela 2: Planejamento da *sprint* 1.....50
Tabela 3: Planejamento da *sprint* 2.....52
Tabela 4. Planejamento da *sprint* 3.....52
Tabela 5. Planejamento da *sprint* 4.....54
Tabela 6. Planejamento da *sprint* 5.....55
Tabela 7. Planejamento da *sprint* 6.....56
Tabela 8. Planejamento da *sprint* 7.....57

SUMÁRIO

1 INTRODUÇÃO 13
2 ESPECIFICAÇÃO DO PROGRAMA 15
2.1 Escopo 15
2.2 Clientes do software 15
3 REQUISITOS DO SISTEMA 16
3.1 Requisitos Funcionais 16
3.2 Requisitos não funcionais 16
4 DEFINIÇÃO DO PROJETO 18
4.1 Casos de Uso 19
4.2 Diagrama de Classe 22
4.3 Model View Controller 23
4.4 Banco de Dados 24
4.5 Diagrama Entidade Relacionamento 25
5 DESENVOLVIMENTO 26
6 METODOLOGIA 49
7 CONSIDERAÇÕES FINAIS 59
REFERÊNCIAS 61

1 INTRODUÇÃO

Aplicações voltadas a dispositivos móveis representam um mercado em ampla expansão com grande demanda global. O desenvolvimento de aplicativos no segmento mobile não representa apenas uma tendência, mas uma realidade na qual tais dispositivos estão completamente integrados ao dia a dia das pessoas, trazendo a facilidade de contar com diferentes soluções conectadas na palma da mão em praticamente qualquer lugar em que estiver.

Nesse mercado abre-se uma oportunidade para que as empresas, profissionais autônomos e prestadores de serviços possuam um canal de comunicação direto e permanente com seus clientes, podendo oferecer seus produtos e serviços em um ambiente digital em conjunto aos meios tradicionais de comunicação ou de forma exclusiva. Essa via provê facilidades de operação e controle ao comerciante ao passo que os clientes encontram suas necessidades em poucos cliques, dispensando-se os deslocamentos e telefonemas comuns antes do surgimento deste mercado.

A possibilidade de lançar mão de aplicações digitais pelos clientes agora, em tempos de pandemia e consequente distanciamento social, tornou-se necessidade. Nesse contexto poder ofertar uma solução digital ganha relevância para que empresas, comércios e autônomos continuem a oferecer seus produtos, em muitos casos estar no meio digital restou imperativo, uma condição *sine qua non* para a continuidade do negócio.

Diante dessa necessidade, o mercado de tecnologia da informação corre na contramão de outras áreas com o aumento da demanda no mercado digital. Nessa seara observa-se uma demanda ainda reprimida por parte de pequenos comerciantes e prestadores de serviços que necessitam adentrar ao mercado digital, porém não dispõem de recursos financeiros suficientes para a contratação de soluções complexas ou grandes empresas do meio.

Com esse contexto o presente trabalho visa oferecer uma solução simples e funcional, apta a atender as necessidades de um público-alvo que ainda não se encontra inserido no mercado digital e que busca manter suas atividades numa época de grandes dificuldades, sem renunciar à segurança e às boas práticas em tecnologia da informação. Ao mesmo tempo este trabalho objetiva cumprir com a função social da academia, oferecendo o produto deste estudo de forma gratuita a um comércio de

bairro, integrando-o ao mercado digital como um meio adicional de vendas, além das tradicionais formas de comercialização por telefone e balcão.

Para essa proposta foi oferecida a construção de uma solução mobile a uma pequena pizzaria de bairro sem custos de desenvolvimento.

Este Projeto está distribuído em 7 títulos, conforme segue: título 1 – Introdução, título 2 – Especificação do programa, título 3 – Requisitos do sistema, título 4 – Definição do projeto, título 5 – Desenvolvimento, título 6 – Metodologia e título 7 – Referências bibliográficas.

2 ESPECIFICAÇÃO DO PROGRAMA

2.1 Escopo

O trabalho compreende um aplicativo de compra de pizzas em domicílio para dispositivos móveis com sistema operacional android desenvolvido em linguagem Dart e framework Flutter, uma API também desenvolvida em linguagem Dart, apta a receber requisições nos métodos HTTP de diferentes origens e salvar as informações em banco de dados e um banco de dados relacional mySql. Dois desenvolvimentos a parte complementam o presente projeto, embora não integrem este escopo. O primeiro consiste em uma solução de retaguarda desenvolvida em linguagem java para operação em servidor dentro do conceito de Progressive Web App – PWA, com objetivo de prover uma interface amigável ao lojista para melhor controle de produtos e pedidos. O segundo utiliza o serviço Firebase Cloud Messaging para o envio de notificações – Push Notifications, capaz de prover um canal de comunicação ativa do servidor para a aplicação.

2.2 Clientes do software

O público-alvo das soluções desenvolvidas são pequenos comerciantes que trabalham com entregas de produtos em domicílio. O aplicativo objeto deste trabalho está customizado para atender as necessidades de uma pizzaria com serviço de entrega. A delimitação do público-alvo visa suprir à necessidade de pequenos estabelecimentos comerciais cuja capacidade de investimento torna inviável a contratação de grandes desenvolvedoras de software para construção de soluções customizadas.

3 REQUISITOS DO SISTEMA

O planejamento do projeto parte da definição requisitos e necessidades para a partir de então construir um padrão de projeto de software. O projeto de interfaces se baseia nos conceitos de design system e nas heurísticas de Nielsen para a definição de layouts amigáveis e funcionais, com vistas a oferecer uma boa experiência de uso aos usuários.

3.1 Requisitos Funcionais

O primeiro requisito funcional consiste na realização de um cadastro de cada cliente, com a coleta de informações essenciais e definição de senha de acesso. A seguir o aplicativo deve buscar e exibir as categorias de itens disponíveis no cardápio, carregar o cardápio e exibir os detalhes de cada item selecionado pelo usuário. Listar os itens que o usuário desejar adquirir e realizar a compra com o registro das informações pertinentes em banco de dados. Por fim deve listar os pedidos realizados e atualizar seus respectivos status.

A edição, adição e remoção de itens do cardápio, contudo, não faz parte dos requisitos destas soluções, tendo em vista que, conforme detalhado no escopo, referidas atividades compõem a competência de uma solução agregada.

3.2 Requisitos não funcionais

Tendo em mente que o aplicativo poderá ser utilizado por dispositivos com diferentes padrões de tamanho de tela e resolução, faz-se necessário estabelecer um requisito de adaptabilidade para o correto funcionamento em diferentes hardwares. Ainda decorrente dessa realidade, o aplicativo adota uma construção voltada ao baixo consumo de processamento e memória, utilizando-se de páginas sem controle de estados, apenas indicando quando algum campo necessitar ser observável, reduzindo-se a necessidade de se manter toda tela em constante atualização, bem como optando-se por manter a menor quantidade de instâncias abertas simultaneamente. Na conexão com a API faz-se a verificação prévia de conexão e na

conexão com o banco de dados adota-se o protocolo de verificação, controle de inserção de dados por meio de transações e encerramento de conexão aberta. Para o envio e armazenamento de senha de acesso junto ao banco de dados adota-se um requisito de segurança, promovendo o encapsulamento da senha em um algoritmo hash criptografado de 256 bits.

4 DEFINIÇÃO DO PROJETO

O comércio de gêneros alimentícios de rápido preparo encontra-se disseminado ao longo das vilas, vilarejos e bairros das cidades brasileiras. Grande parte dessas lojas são comércios de pequeno porte com foco na refeição local e entrega no balcão. Com o advento da tecnologia de internet móvel um novo ramo de comércio em domicílio ganha forma. Os serviços de *delivery* que antes estavam relegados aos atendimentos telefônicos se tornam cada vez mais expressivos conforme a tecnologia *mobile* se populariza. Nesse contexto, enquanto grandes redes dispõem de capacidade de investimentos para prover suas próprias plataformas aptas a operar nesse novo canal de comunicação, os comerciantes locais encontram dificuldade de adentrar nesse mercado, comumente limitando-se a aplicativos comerciais coletivos nos quais encontram-se restritos às condições, taxas e limitações do serviço enquanto se veem em meio a grande concorrência dentro das plataformas.

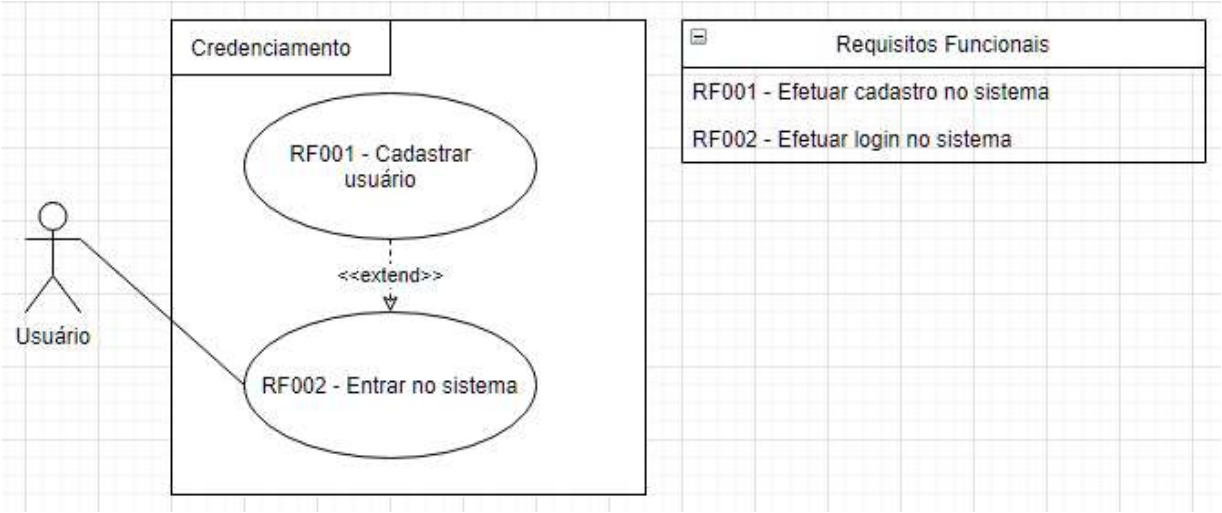
O desenvolvimento de um aplicativo de entrega de pizzas envolve três frentes de trabalho: um banco de dados para guardar informações de clientes, produtos e pedidos, optando pela tecnologia SQL - Standard Query Language que atende adequadamente as necessidades do projeto e utilizando-se da ferramenta MySQL, uma solução gratuita já consolidada no mercado; Uma solução de back-end, desenvolvendo-se uma Interface de Programação de Aplicações ou API - Application Programming Interface, em linguagem DART e um front-end mobile desenvolvido em linguagem DART com a SDK - Software Development Kit Flutter.

O projeto da aplicação será modelado de acordo com o paradigma da orientação a objeto, permitindo a adoção de modelos a serem herdados e a implementação de polimorfismo nas classes derivadas conforme a necessidade. O banco de dados seguirá o modelo relacional, de fácil implementação e manutenção, permitindo a adoção de chaves primárias e chaves estrangeiras na organização do relacionamento entre as tabelas.

4.1 Casos de Uso

Com a definição de requisitos e necessidades elabora-se o diagrama de caso de uso UML – Unified Modeling Languade. Na linguagem de modelagem unificada o diagrama de classe de uso define as funcionalidades propostas de forma simples, permitindo identificar os atores envolvidos e as funções que o sistema se propõe a executar.

Figura 1: Diagrama de caso de uso – credenciamento.



Fonte: o autor.

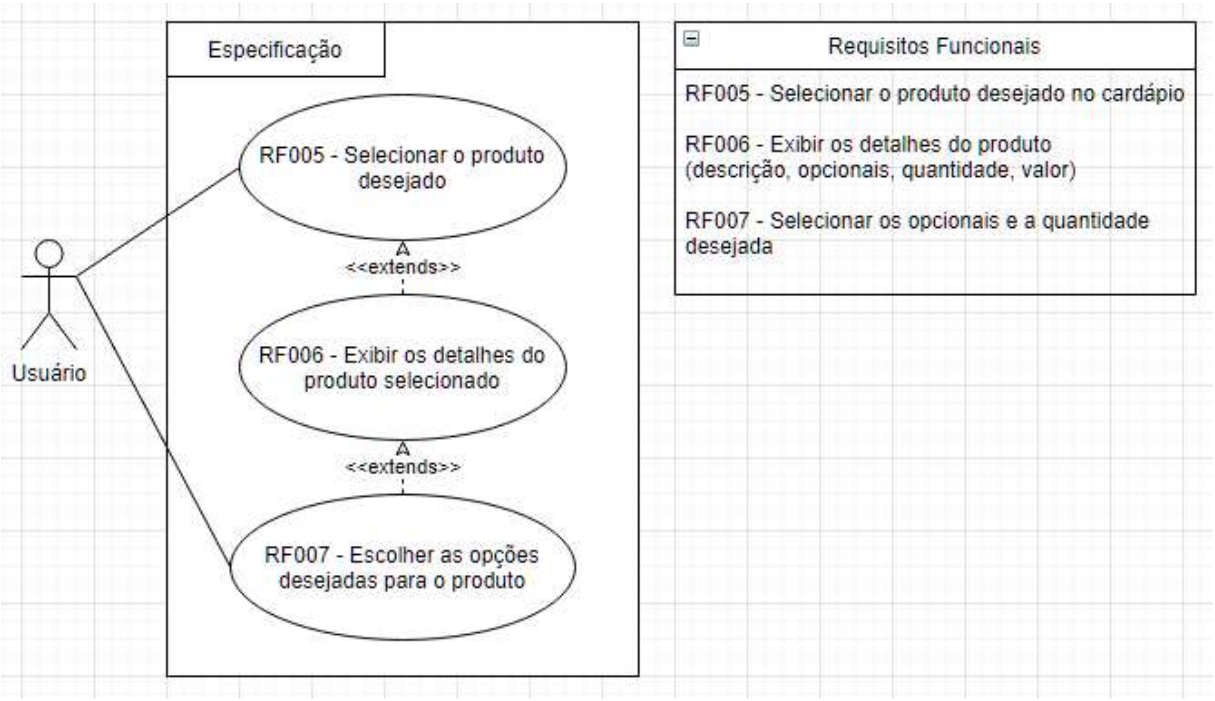
Na figura 1 é apresentado o diagrama de caso de uso construído na plataforma draw.io para as funcionalidades de credenciamento de usuário. Neste diagrama o usuário realiza o login de acesso ao sistema, podendo realizar a atividade de cadastro no caso de primeiro acesso.

Figura 2: Diagrama de caso de uso – seleção de produto.

Fonte: o autor.

O diagrama da figura 2 apresenta as funcionalidades relacionadas com a seleção de produto. Neste diagrama o usuário escolhe qual a categoria de produto que deseja (pizzas salgadas, pizzas doces, pizzas brotinhos ou bebidas). Após a definição a aplicação lista o cardápio indexado para iniciar na categoria selecionada.

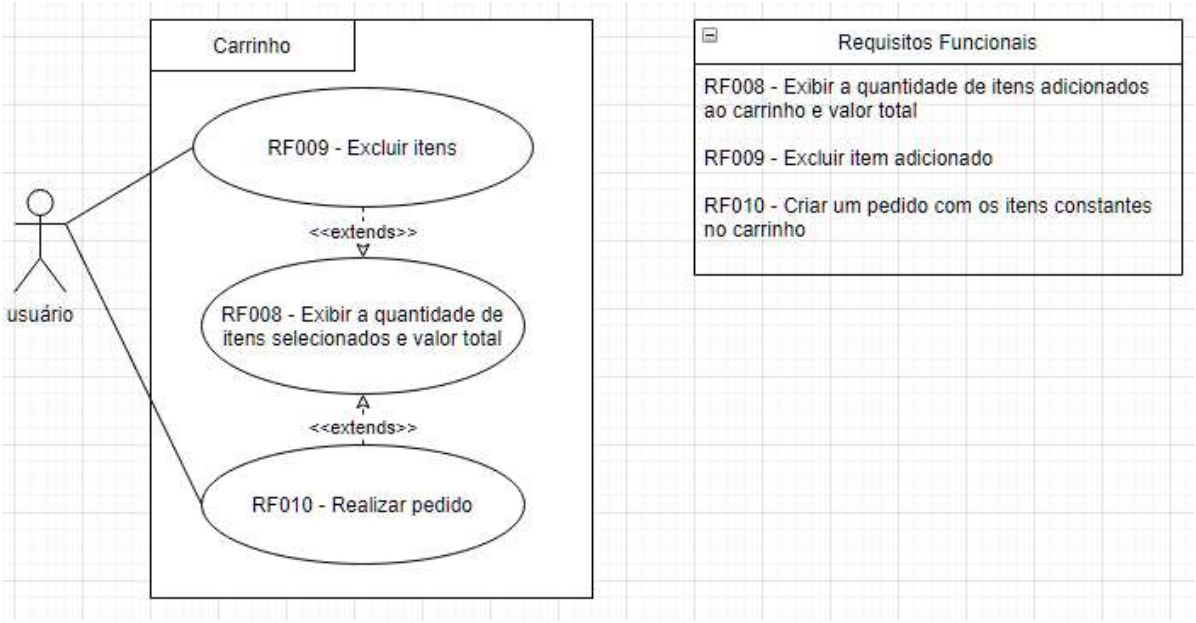
Figura 3: Diagrama de caso de uso – especificação.



Fonte: o autor.

No cardápio o usuário irá selecionar o produto desejado, figura 3, carregando a tela de detalhes do item selecionado. Nesta tela o usuário seleciona os opcionais disponíveis para aquele item e a quantidade desejada para aquele produto.

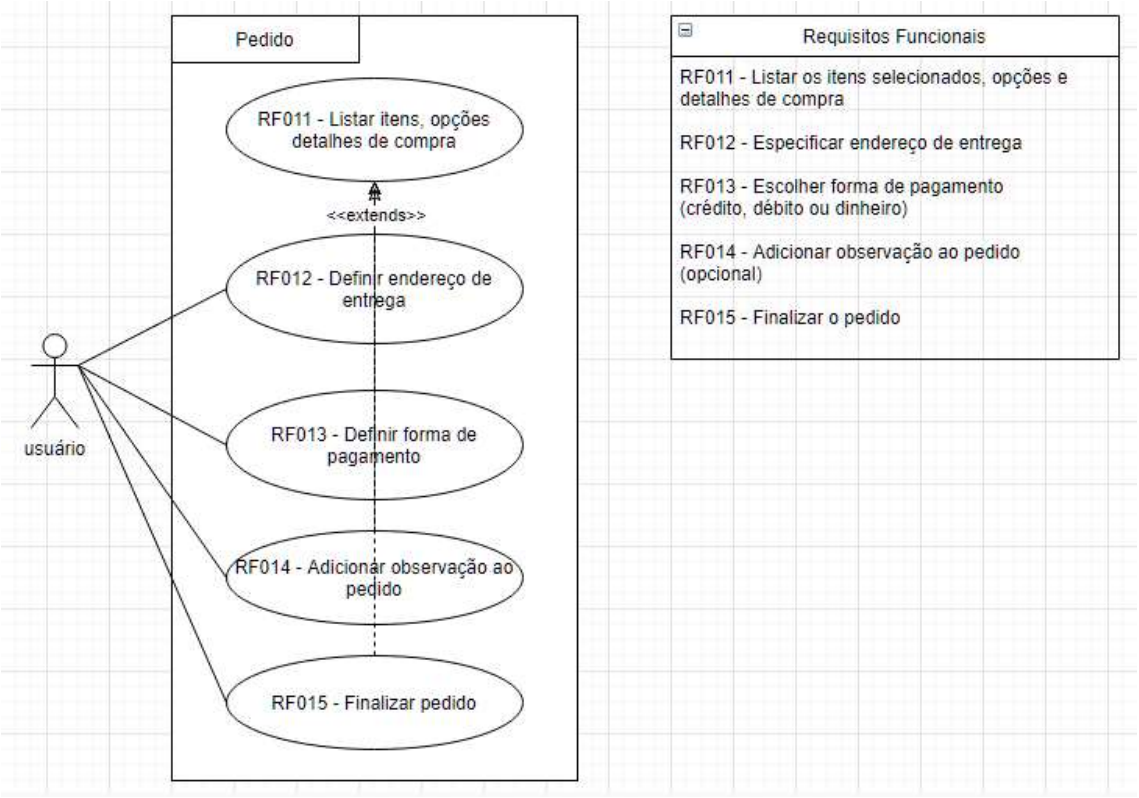
Figura 4: Diagrama de caso de uso – carrinho.



Fonte: o autor.

No carrinho, ou sacola de compras, figura 4, é exibida a quantidade de itens que foram selecionados e o valor total. O usuário pode excluir itens ou realizar um pedido com os itens que escolheu.

Figura 5: Diagrama de caso de uso – pedido.



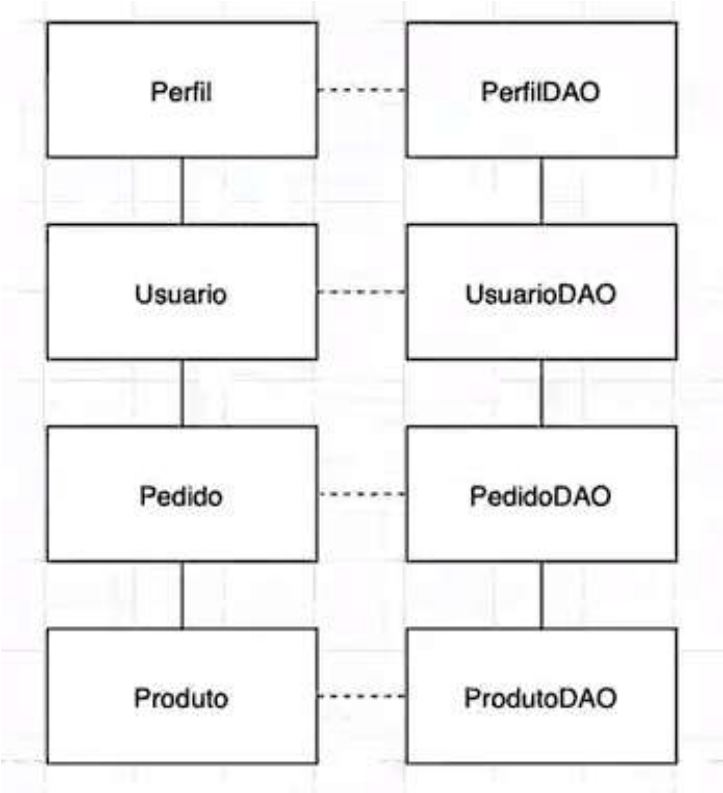
Fonte: o autor.

Para a realização do pedido, figura 5, são listados os itens do cardápio escolhidos e valor da compra. O usuário deve especificar o endereço de entrega, selecionar a forma de pagamento (cartão de crédito, cartão de débito ou dinheiro) e pode opcionalmente adicionar alguma observação. Após as especificações o usuário poderá finalizar o pedido.

4.2 Diagrama de Classe

Para a confecção do diagrama de classes foram especificadas as entidades orientadas a objeto que serão convertidas pela API para entidades relacionais no banco de dados por meio de Data Access Object – DAO.

Figura 6: Diagrama de classes.



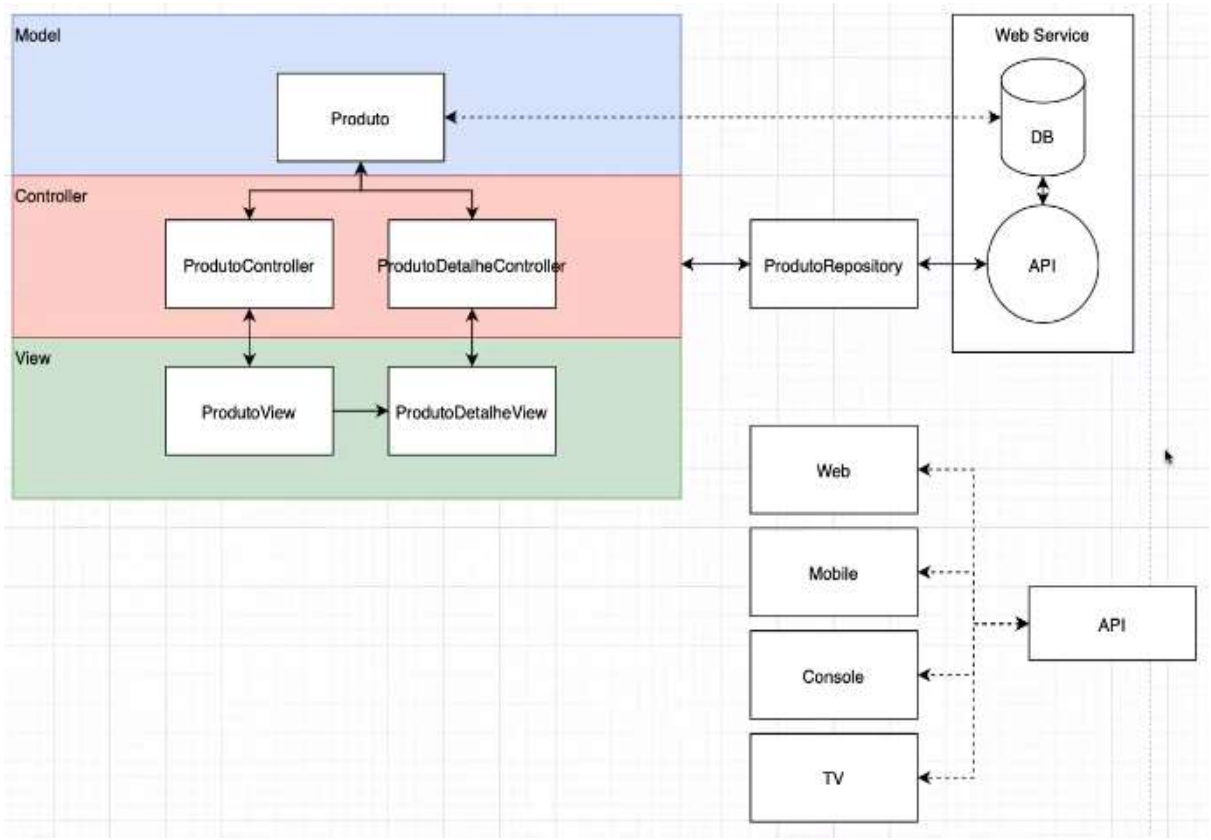
Fonte: o autor.

De acordo com o diagrama de classes apresentado na figura 6 as entidades com informações baseadas na orientação a objeto (Perfil, Usuário, Pedido e Produto) são convertidas pela API em entidades relacionais (DAO) e vice-versa.

4.3 Model View Controller

A arquitetura projetada compreende uma API apta a trabalhar com diferentes aplicações *front-end,* um banco de dados e uma aplicação *front-end mobile* com quatro camadas principais: *Repository* (comunicações com *back-end*)*; Model* (modelos)*; Controller* (controladoras) *e View* (interfaces), além de uma camada auxiliar de utilitários (*helpers*).

Figura 7: *Model View Controller* – MVC.



Fonte: o autor.

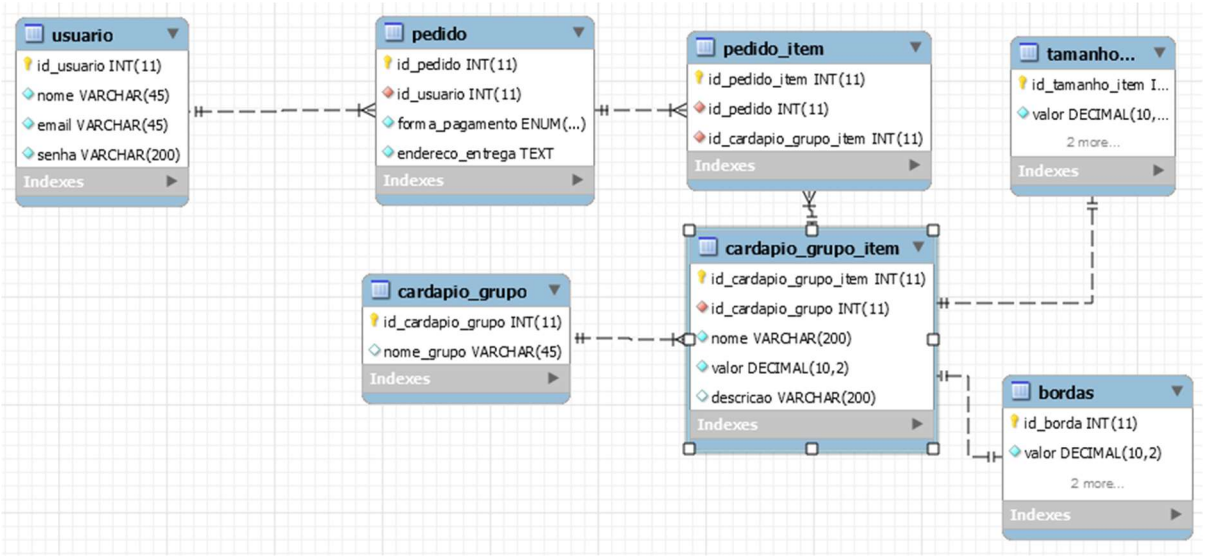
Na figura 7 está o MVC - *Model View Controller*, com a representação dos componentes que integram o projeto e suas interações. O segmento web-service apresenta o banco de dados e a API, a aplicação está segmentada em suas respectivas camadas e a API está demonstrada abaixo apta a se comunicar com diferentes aplicações.

4.4 Banco de Dados

O banco de dados relacional foi estruturado para que atenda às normalizações, apresentando 7 entidades inter-relacionadas.

4.5 Diagrama Entidade Relacionamento

Figura 8: Diagrama Entidade Relacionamento – DER.



Fonte: o autor.

Conforme figura 8, a entidade “usuário” possui relação do tipo “um para muitos” com a entidade “pedido”, a entidade “pedido” tem relação “um para muitos” com a entidade “pedido_item”, a entidade “cardapio_grupo” se relaciona “um para muitos” com “cardapio_grupo_item” que também se relaciona “um para muitos” com “pedido_item”, a entidade “cardapio_grupo_item” também possui relacionamento “um para um” com as entidades “tamanho_item” e “bordas”.

5 DESENVOLVIMENTO

A construção do backend parte de projeto dart para web server (API) utilizando o pacote shelf. A comunicação com o banco de dados foi configurada a partir do arquivo docker-compose.yaml e os parâmetros foram adicionados ao arquivo env. Esses parâmetros ficam em variáveis de ambiente do servidor para que os dados de produção fiquem restritos ao backend, sem a necessidade de compartilhar esses dados com os programadores, mantendo acesso somente aos administradores do banco de dados para maior segurança das informações.

A configuração do arquivo pubspeck.yaml adotou os seguintes pacotes (libs): mysql1 para conexão com o banco de dados; injectable para fazer as injeções de dependências e get_it para gerenciar as dependências, operando como service locator para controlar as instâncias das classes; dotenv para carregar as variáveis de ambiente; shelf servidor web adotado para a construção do backend em linguagem dart; build_runner para geração de código ao ser executado juntamente com o injectable; shelf_router para requisições de rotas com o shelf.

No arquivo server.dart criou-se um método que chama o comando load() do pacote dotenv para carregar as configurações. A seguir trabalha-se as configurações para conexão com o banco de dados criando-se um arquivo de modelo de configuração em lib/application/config/database_connection_configuration.dart. Optou-se por definir as variáveis host, user, port, password e dabaseName com a tipologia “final” tendo em vista que seus valores não serão alterados após a primeira inserção. A seguir implementa-se o método do arquivo de modelo de configuração em server.dart a partir do comando env do pacote dotenv. Para que os dados sejam buscados a partir das variáveis de ambiente em um ambiente de produção adota-se a biblioteca Platform.

Figura 9: variáveis de ambiente.

```
Future<void> loadConfigApplication() async {
  await load();

  final databaseConfig = DatabaseConnectionConfiguration(
    host: Platform.environment['DATABASE_HOST'] ?? env['databaseHost'],
    user: Platform.environment['DATABASE_USER'] ?? env['databaseUser'],
    port: Platform.environment['DATABASE_PORT'] ?? env['databasePort'],
    password:
        Platform.environment['DATABASE_PASSWORD'] ?? env['databasePassword'],
    databaseName: Platform.environment['DATABASE_NAME'] ?? env['databaseName'],
  );

  GetIt.I.registerSingleton(databaseConfig);
  configureDependencies();
}
```

Fonte: o autor.

A figura 9 do arquivo server.dart exibe a chamada para o comando load() do pacote dotenv e a utilização preferencial dos dados constantes nas variáveis de ambiente, no caso do ambiente de desenvolvimento os dados serão carregados a partir do comando env do pacote dotenv, que buscará as configurações salvas no arquivo env.

Os dados de conexão são carregados na memória instanciando-se o databaseConfig com o pacote Get_it: “GetIt.I.registerSingleton(databaseConfig);” que mantém uma única instância na memória para toda a aplicação.

A conexão com o banco de dados ocorre a partir de uma interface e sua implementação criadas em lib/application/database/. A interface apenas define o método do tipo “MySqlConnection” que é instanciado na implementação. A implementação recebe os dados de configuração importados de server.dart em um atributo privado e o instancia no método construtor. O método de “MySqlConnection” retorna a conexão com os dados recebidos no atributo privado.

Figura 10: conexão com banco de dados.

```dart
class DatabaseConnection implements IDatabaseConnection {
  final DatabaseConnectionConfiguration _configuration;

  DatabaseConnection(this._configuration);

  @override
  Future<MySqlConnection> openConnection() {
    return MySqlConnection.connect(ConnectionSettings(
      host: _configuration.host,
      port: int.parse(_configuration.port),
      user: _configuration.user,
      password: _configuration.password,
      db: _configuration.databaseName,
    ));
  }
}
```

Fonte: o autor.

A figura 10 extraída do arquivo database_connection.dart que implementa a interface criada mostra o atributo privado “_configuration” que recebe os dados de server.dart (figura 9), o método construtor e o método de conexão com o banco de dados.

O pacote injectable fará o controle das instâncias a partir do comando: “@Injectable(as: IDatabaseConnection)”. Para tanto cria-se um arquivo para sua configuração em lib/application/config/ que executará com comando: “@InjectableInit()”. Ao executar o comando pub run build_runner watch no terminal o injectable resolve das dependências e cria um arquivo de mesmo nome com final “.config” na mesma pasta em que é gerado um registro dos dados de DatabaseConnection, permitindo a geração de novas instâncias conforme ocorrem alterações no código. O método configureDependencies() do arquivo de configuração é implementado no server.dart dentro do método loadConfigApplication() para que ocorra a chamada do método durante a execução.

Para a chamada das requisições junto ao banco de dados são configurados comandos em uma função cors a partir de *middlewares* criados em

lib/application/middlewares/middlewares.dart. A execução antecipada pelo middleware objetiva alterar a resposta a partir dos parâmetros que foram passados.

Figura 11: *middleware* cors.

```
const _defaultCorsHeader = {
  'Access-Control-Allow-Origin': '*',
  'Access-Control-Allow-Methods': 'GET, POST, PATCH, PUT, DELETE, OPTIONS',
  'Access-Control-Allow-Header':
      '${HttpHeaders.contentTypeHeader}, ${HttpHeaders.authorizationHeader}'
};

Middleware cors({Map<String, String> headers = _defaultCorsHeader}) {
  return (Handler innerHandler) {
    return (Request request) async {
      if (request.method == 'OPTIONS') {
        return Response(HttpStatus.ok, headers: headers);
      }
      final mapHeaders = {...request?.headers, ...headers};
      final response = await innerHandler(request.change(headers: mapHeaders));
      return response.change(headers: headers);
    };
  };
}
```

Fonte: o autor.

A figura 11 mostra a implementação do *middleware* a partir dos parâmetros _defaultCorsHeader, retornando duas funções anônimas que permitem a alteração dos headers conforme a resposta recebida. O middleware é adicionado em server.dart, figura 12.

Outro middleware é necessário para criar um retorno padrão do tipo json. Esse middleware é adicionado em server.dart como padrão “application/json”.

Figura 12: *middlewares.*

```
var handler = const shelf.Pipeline()
    .addMiddleware(shelf.logRequests())
    .addMiddleware(cors())
    .addMiddleware(
        defaultResponseContentType('application/json;charset=utf-8'))
    .addHandler(appRouter);
```

Fonte: o autor.

Na figura 12 observa-se os middlewares adicionados em server.dart. Os retornos padronizados estão configurados como defaultResponseContentType(‘application/json;charset=utf-8’), fazendo com que as respostas retornem formatadas como json. A rota é adicionada ao “addHandler”.

A configuração das rotas utiliza o pacote shelf_router_generator. Para tanto é criada uma interface em lib/application/routers que possui apenas um método que

recebe um “Router” do pacote shelf_router. A implementação da interface contém uma lista de rotas e o método herdado configurará as rotas para cada rota integrante da lista. A chamada do método é adicionado em server.dart que ao ser executado irá varrer as rotas constantes da lista, configurando-as. Essa configuração permite manter o arquivo server.dart limpo de configurações de rotas, evitando-se uma construção que tende a um monolito e possibilitando a adoção de uma estratégia de “dividir para conquistar” ao separar cada função.

A API foi segmentada em 3 funcionalidades distintas. A primeira funcionalidade é “user” e objetiva efetivar o cadastro e login de usuário, a segunda é “orders” e visa controlar os pedidos realizados e a terceira é “menu” e se propõe a listar o cardápio.

Cada funcionalidade é subdividida em 4 camadas. A primeira camada é “modules” e se encarrega dos mapas de rotas e suas view models, a segunda é “entities”, é o núcleo responsável por conter aquilo que cada funcionalidade se destina, representando a entidade da tabela do banco de dados, a terceira é “repositories” e encarrega-se do envio e recebimento de dados com o banco de dados e a quarta camada é “services” responsável pelas regras implementadas, evitando o acúmulo dessas implementações na camada “repositories”.

Um cuidado necessário ocorre com a segurança das senhas cadastradas, criptografando-as. Para que essa regra seja implementada na camada “services”, cria-se uma classe “utilitário” em lib/application/helpers que ficará responsável pela chamada do algoritmo de criptografia sha-256 da biblioteca crypto para a execução de criptografia em *secure hash algorith* de 256 bits.

A implementação da criptografia para senhas compreende um requisito da aceitabilidade prática, enquanto parâmetro de confiabilidade, conforme Atributos de Aceitabilidade de Sistemas de Nielsen (1993).

Figura 13: criptografia.

```dart
import 'dart:convert';
import 'package:crypto/crypto.dart';

class CriptyHelper {
  static String generateSHA256Hash(String password) {
    final bytes = utf8.encode(password);
    return sha256.convert(bytes).toString();
  }
}
```

Fonte: o autor.

O utilitário cripty_helper.dart, figura 13, chama a função de hash para criptografia no padrão sha-256, gerando uma saída de tamanho fixo com 256 bits.

A camada “services” envia a senha já criptografada para a camada “repositories”. Nessa camada usa-se um atributo privado do tipo IDatabaseConnection para aproveitar a lógica construída em lib/application/database.

Figura 14: método saveUser.

```dart
class UserRepository implements IUserRepository {
  final IDatabaseConnection _connection;

  UserRepository(
    this._connection,
  );

  @override
  Future<void> saveUser(RegisterInputModel model) async {
    final conn = await _connection.openConnection();

    try {
      await conn.query(
          'insert into usuario(nome,email,senha,telefone) values(?,?,?,?)', [
        model.name,
        model.email,
        model.password,
        model.phone,
      ]);
    } on MySqlException catch (e) {
      print(e);
      throw DatabaseErrorException(message: 'Erro ao registrar usuário');
    } finally {
      await conn?.close();
```

Fonte: o autor.

Observa-se na figura 14 que o atributo _connection é chamado no método assíncrono saveUser para abrir a conexão com o banco de dados. O cadastro dos dados no banco ocorre com a execução de uma consulta (query) que está inserida em um bloco try/catch para tratamento de exceções, retornado uma mensagem sem

os dados do banco de dados, preservando informações dessa camada. Por fim o método fecha a conexão se estiver aberta.

Para as comunicações (query) do pedido (order) com o banco de dados usa-se uma transação, *transaction,* para que caso ocorra algum erro na inserção dos dados no banco haja a reversão, *rollback,* de todo o comando.

Figura 15: método saveOrder.

```
Future<String> saveOrder(SaveOrderInputModel saveOrder) async {
  final conn = await _connection.openConnection();
  var resultOrder;

  try {
    await conn.transaction((_) async {
      resultOrder = await conn.query('''
    insert into pedido(
    id_usuario,
    forma_pagamento,
    endereco_entrega,
    observacao,
    valor
    ) values(?,?,?,?,?)
    ''', [
        saveOrder.userId,
        saveOrder.paymentType,
        saveOrder.address,
        saveOrder.note,
        saveOrder.value,
      ]);

      final orderId = resultOrder.insertId;
      await conn.queryMulti(
          '''
        insert into pedido_item(
          id_pedido,
          id_cardapio_grupo_item,
          id_borda,
          id_2sabores
        ) values(?,?,?,?)
      ''',
          saveOrder.save.map((item) =>
              [orderId, item.itensId, item.bordasId, item.secondItemId]));
    });

    return resultOrder.insertId.toString();
  } on MySqlException catch (e) {
    print(e);
    throw DatabaseErrorException(message: 'Erro ao inserir order');
  } finally {
    await conn?.close();
```

Fonte: o autor.

A figura 15 exibe a chamada com o banco de dados para a realização de um pedido (order) com a transaction que irá executar uma query para inserção de dados na tabela “pedido” e uma queryMulti para a tabela “pedido_item”, tendo em vista que

cada pedido pode ter mais de um item. A transaction está inserida em um try/catch para tratamento de erros com o fechamento da conexão ao final.

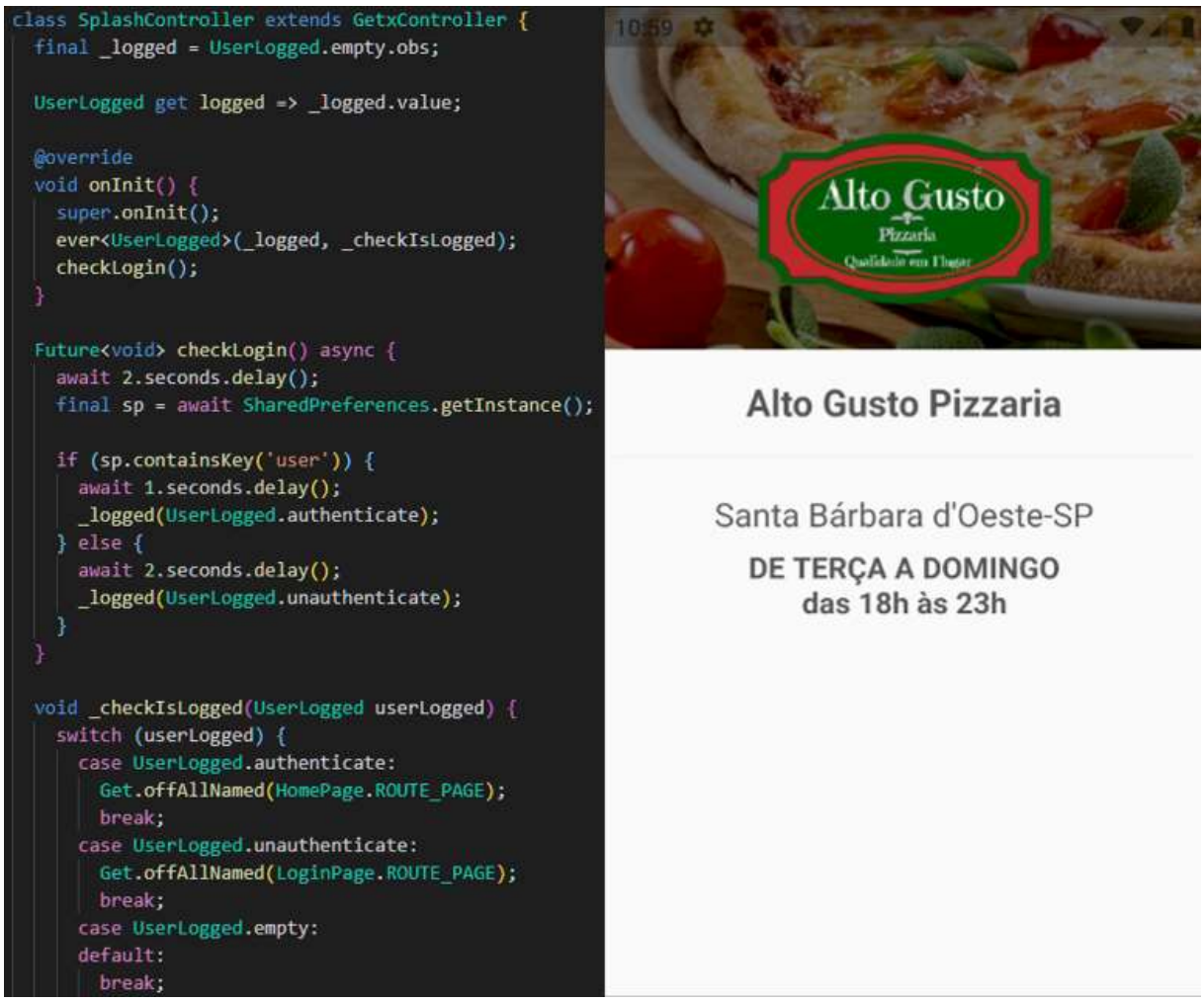
O aplicativo (*front-end)* desenvolvido a partir da SDK Flutter utiliza os pacotes get para gerenciar a aplicação e a comunicação com o back-end, flutter_icons para utilização de diferentes ícones, grouped_buttons para adoção de check-boxes, intl para formatações de valores no padrão brasileiro, scroll_to_index para indexar o menu na posição desejada, shared_preferences para guardar informações do usuário loggado e modal_bottom_sheet para construção de menus do tipo “scrollController”.

Para o layout de interfaces foram estabelecidos padrões de cor e estilo, mantendo-se uma harmonia visual nas diferentes telas de acordo com as definições de *design system*. Nessa premissa as telas apresentam design em tons quentes com ênfase na cor vermelha para botões e *appbars*, valendo-se na associação existente entre o vermelho e a alimentação, conforme estudo realizado pelo professor coordenador do Núcleo de Estudos da Embalagem da Escola Superior de Propaganda e Marketing Fábio Mestriner. Também foram definidos tons neutros para manter a harmonia de cor e contraste, além da opção de apresentar menos recursos em cada tela, abrindo-se telas adicionais se necessário, evitando-se o acúmulo de funções em um mesmo ambiente, preservando-se as características de simplicidade e facilidade de uso. Optou-se por fundos brancos que preservam o contraste com o estilo escolhido e mantém o estilo numa tendência minimalista, contribuindo para que não ocorra a poluição visual em cada página.

A aplicação se inicia a partir de uma tela de splash que possui uma estrutura composta por uma spash_page que será executada ao abrir o aplicativo como tela de abertura do tipo “StatelessWidget” tendo em vista que o controle de estados será feito pelo get, uma splash_controller gerenciado pelo pacote get para controlar os dados de atualização de tela, sendo responsável pelo controle de estados e uma splash_bindings para ser previamente executada com as dependências do service_locator do pacote get.

A splash_controller é responsável por verificar a autenticação do usuário e direcioná-lo para tela inicial caso logado ou para tela de login caso não logado.

Figura 16: splash_controller e splash_page.



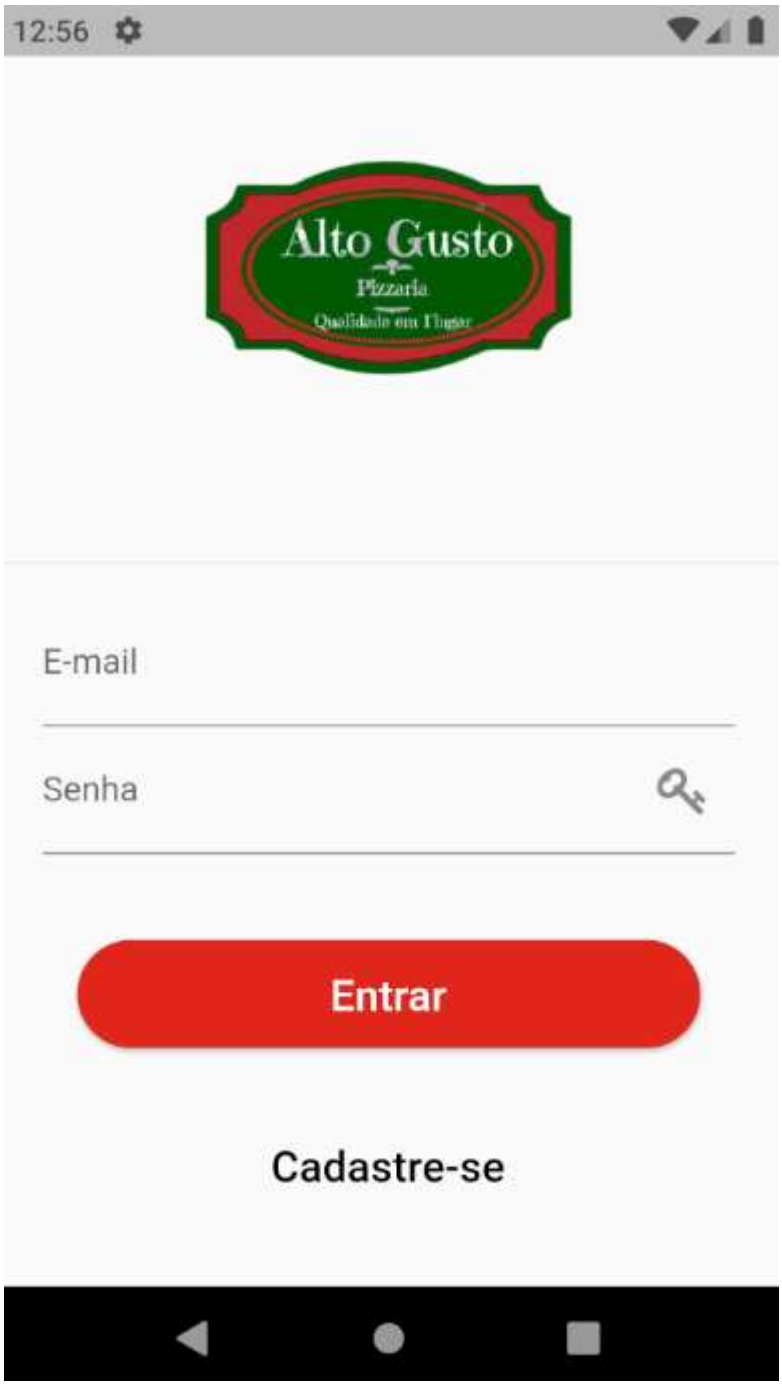
Fonte: o autor.

Na figura 16, à esquerda exibe a splash_controller responsável pelo controle de estados com a verificação de login e o direcionamento de tela sem a necessidade de utilização de um "*StatefullWidget*". O atributo privado _logged recebe o valor observável de UserLogged fazendo com que opere com a reatividade do pacote Get como se fosse uma variável do tipo "Rx". O encapsulamento objetiva manter esse controle privativo do arquivo splash_controller. Além da verificação de login e direcionamento de página, outra definição implementada visa controlar o tempo de exibição da splash_page. Para tanto foi construída uma lógica distinta para usuário logado e usuário não logado. Essa distinção objetiva manter um tempo maior de exibição da splash_page, à direita, ao usuário deslogado para melhor compreensão das informações constantes na tela como os dias e horários de funcionamento. Para o usuário logado, contudo, manter um tempo demasiadamente longo de exibição

implicaria em uma má experiência com o aplicativo, vez que teria que esperar mais tempo para ter o aplicativo disponível toda vez que o carregasse.

Essa implementação busca prover os atributos de aceitabilidade social acerca do comportamento do sistema, conforme os atributos de aceitabilidade de sistemas de Nielsen (1993).

Figura 17: tela de login



Fonte: o autor.

A figura 17 exibe o design definido para a tela de login do aplicativo. Nessa construção busca-se um layout simples e limpo com vistas a facilitar a operação pelo usuário. Os botões foram padronizados com o tema em tom vermelho e tonalidades neutras.

A tela de login apresenta uma verificação de e-mail válido, uma verificação de tamanho mínimo de senha e um ícone funcional de chave para exibição dos caracteres do campo senha.

Figura 18: validações.

```
PizzaDeliveryInput(
  controller: emailEditingController,
  label: 'E-mail',
  validator: (String value) {
    if (value == null ||
        value.isBlank ||
        !value.isEmail) {
      return 'E-mail inválido';
    }
    return null;
  },
),
Obx(() => PizzaDeliveryInput(
      controller: passwordEditingController,
      label: 'Senha',
      suffixIcon: Icon(FontAwesome.key),
      suffixIconOnPressed: controller.showHidePassword,
      obscureText: controller.obscureText,
      validator: (String value) {
        if (value.length < 6) {
          return 'Senha deve ter pelo menos 6 caracteres';
        }
        return null;
```

Fonte: o autor.

A figura 18 apresenta as verificações de e-mail válido e de caracteres da senha da tela de login. Uma vez que a tela de login não guarda estados, para que o estado mude ao clicar é necessário que o botão de senha seja observável, motivo pelo qual referido campo está inserido em um Obx ( ).

O login é realizado a partir da comunicação com a API com um comando POST por meio da funcionalidade restClient do pacote get. O POST envia um mapa de chave e valor com e-mail e senha.

Figura 19: login.

```
Future<UserModel> login(String email, String password) async {
  final response = await restClient.post('/user/auth', {
    'email': email,
    'password': password,
  }, decoder: (resp) {
    if (resp != '') {
      return UserModel.fromMap(resp);
    }
  });

  if (response.hasError) {
    String message = 'Erro ao autenticar usuário';

    if (response.statusCode == 403) {
      message = 'Usuário ou senha inválidos';
    }

    throw RestClientException(message);
  }
  return response.body;
}

Future<void> saveUser(RegisterUserInputModel model) async {
  final response = await restClient.post('/user/', {
    'name': model.name,
    'email': model.email,
    'password': model.password,
    'phone': model.phone,
  });

  if (response.hasError) {
    throw RestClientException('erro ao registrar usuário');
```

Fonte: o autor.

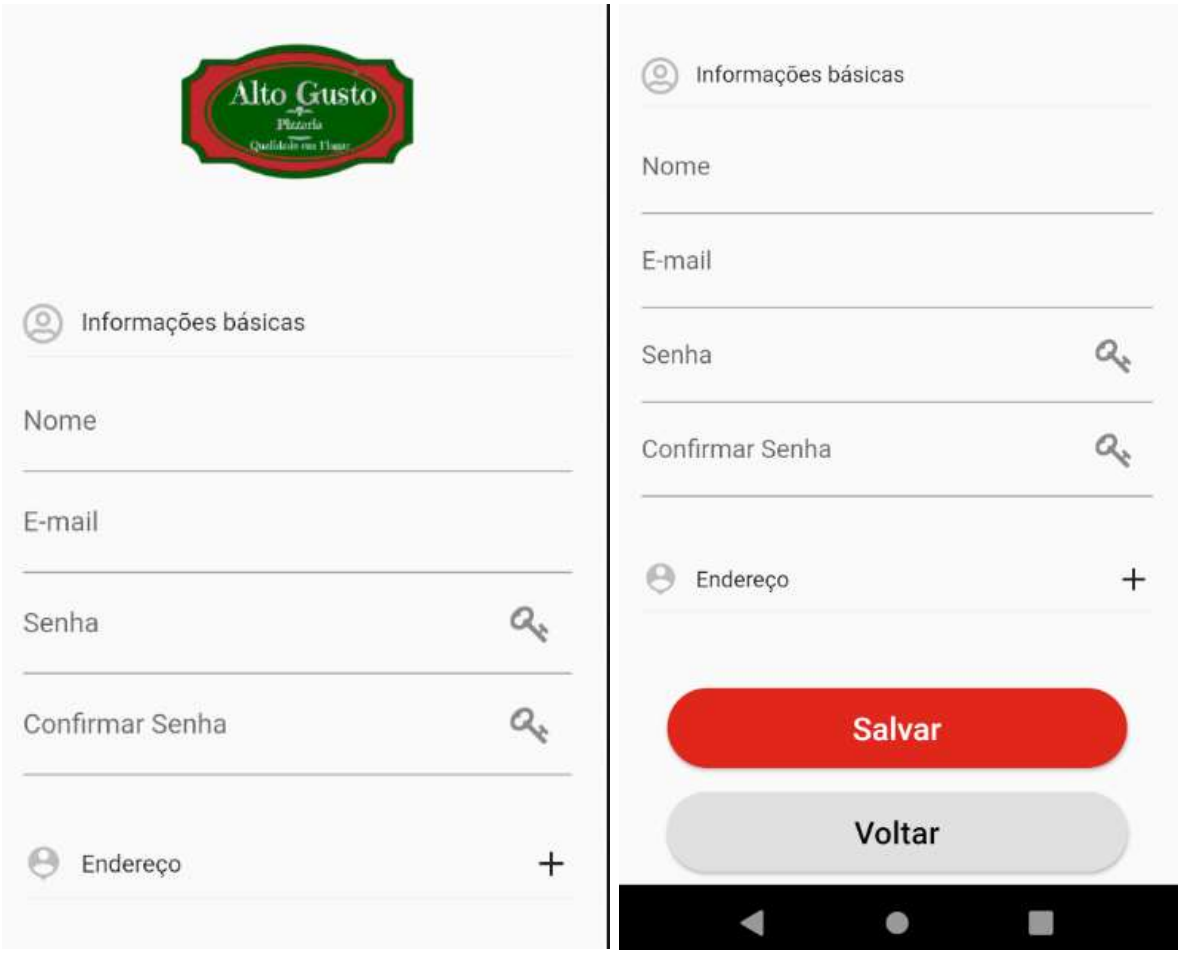
A figura 19 exibe o método de envio dos dados de login. Ao utilizar o restClient do Get connect não é necessário construir um bloco try/catch tendo em vista tratar-se de uma funcionalidade “*error-safed*”, na qual os erros são capturados e encapsulados, retornando sob a forma de “hasError”.

Para que o usuário perceba que o comando de login foi efetivamente acionado foi implementado um loader que mantém um indicador circular rodando na tela enquanto aguarda um retorno do serviço. Ao retornar será apresentada uma mensagem ao usuário na cor azul caso login realizado com sucesso ou na cor vermelha caso retornou erro de login. Essas implementações também serão aproveitadas em outras telas da aplicação.

Um cadastro é necessário ao novo usuário. Para tanto é informado nome, e-mail e senha por meio de uma tela de cadastro com validações para que o nome seja

diferente de nulo e em branco, e-mail válido, senha com pelo menos 6 caracteres e confirmação de senha para prevenção de erro de digitação.

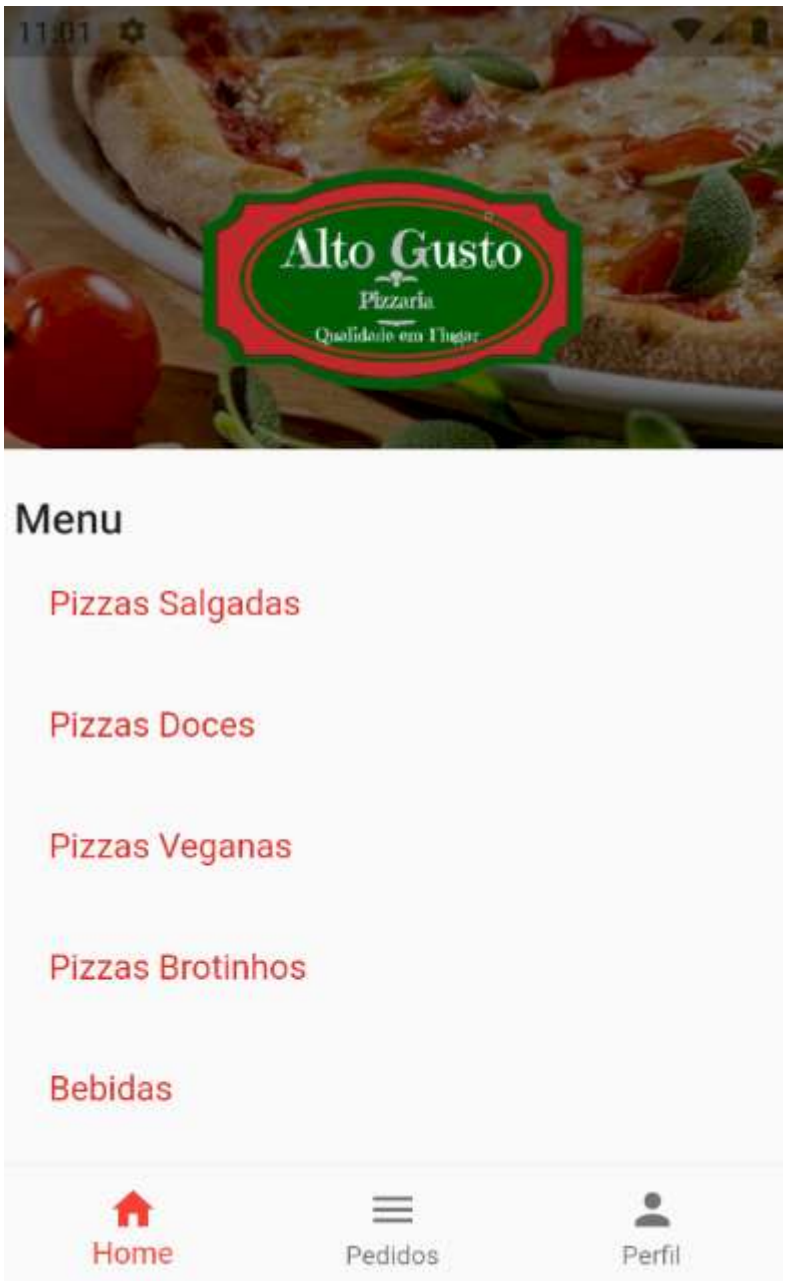
Figura 20: tela de cadastro.



Fonte: o autor.

O design da tela de cadastro, figura 20, segue o padrão pré-definido para a tela de login com os campos verificáveis para cadastro de nome, e-mail, senha e confirmação de senha, os dois últimos com ícones funcionais para exibição dos caracteres digitados. O botão salvar segue o padrão de cor do tema e o botão voltar tem padrão de cor neutro.

Figura 21: home_page.

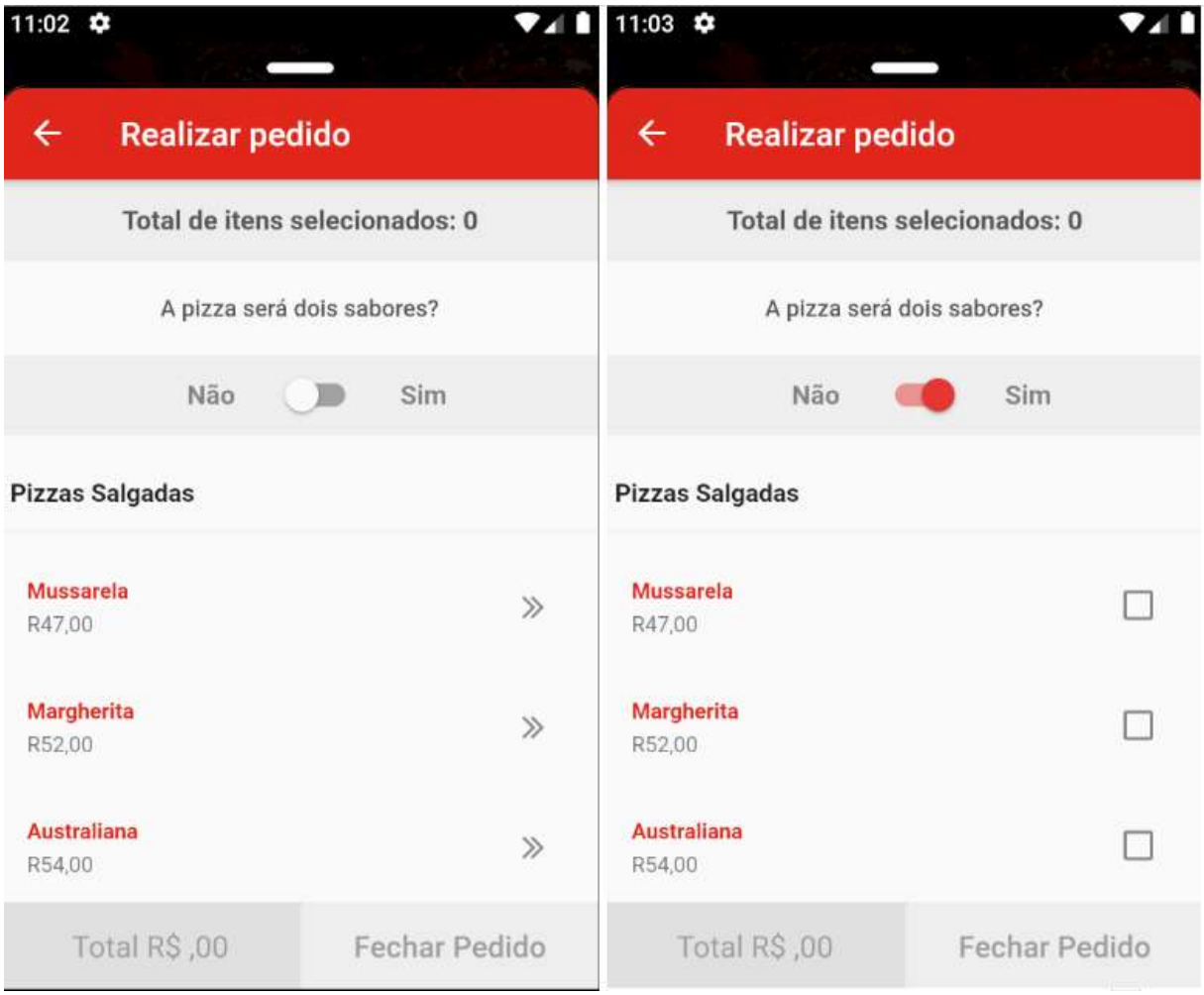


Fonte: o autor.

A tela home, figura 21, exibe as opções de pizzas salgadas, pizzas doces, pizzas veganas, pizzas brotinhos e bebidas. Optou-se por uma construção na qual essas opções, do tipo ListTile, carregam todos os itens do menu, porém indexados para exibir a partir do grupo selecionado. As opções correspondem à entidade cardápio_grupo do banco de dados, imagens 1 e 2. Para a indexação utilizou-se do pacote scroll_to_index, passando o parâmetro de referência ao clicar no botão correspondente.

A tela de cardápio, figura 22, utiliza o pacote modal_bottom_sheet para construção de menus do tipo “scrollController”. A exibição do cardápio ocorre por meio de um ListView.builder() com physics: NeverScrollableScrollPhysics() para que não tenha scroll em cada item exibido e com shrinkWrap: true para que o tamanho da tela a ser construída corresponda a quantidade de itens a serem exibidos. Cada item é carregado em um ListTile() com title: Text(item.name) e com subtitle: Text(‘R\${formatter.format(item.price)}’, sendo que formatter corresponde à formatação de moeda no padrão brasileiro utilizando o pacote intl.

Figura 22: cardápio.



Fonte: o autor.

Tendo em vista a possibilidade de o usuário escolher uma pizza com um ou dois sabores, a tela de cardápio, figura 22, apresenta essa recursividade, possibilitando selecionar 2 itens elencáveis, conforme imagem à direita.

Foi criado um menu_model para receber os itens do cardápio. A API retorna os itens com os parâmetros id, nome e preço. Os dados do backend são recepcionados

pelo menu_repository que verifica se a resposta é uma lista e retorna os dados para a página menu_model.

Figura 23: método openMenu.

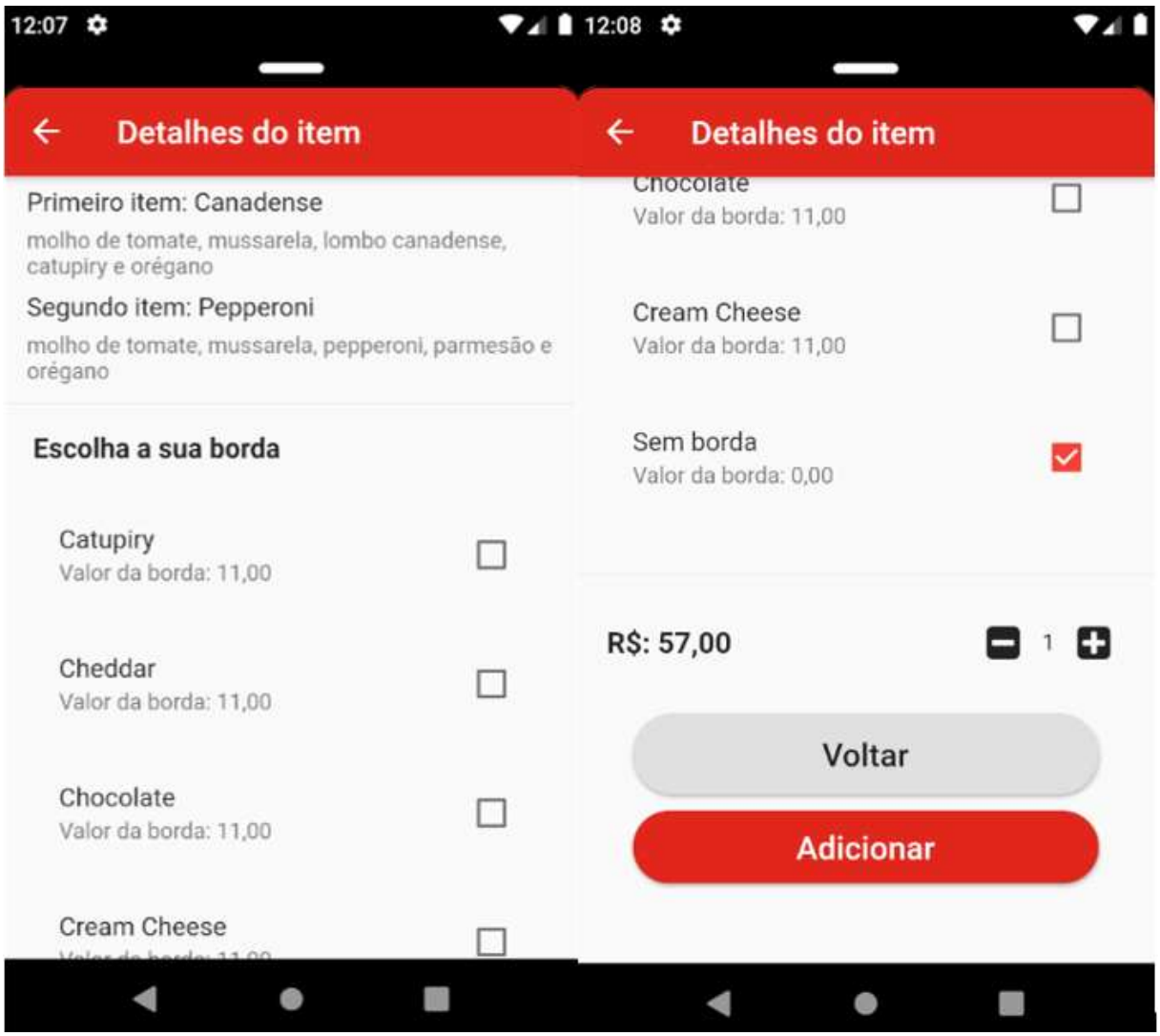
```
class HomeController extends GetxController {
  Future<void> openMenu(String itemMenu, int indexCardapio) async {
    Get.put(MenuRepository(Get.find()));
    Get.put(MenuController(Get.find(), indexCardapio));

    await showBarModalBottomSheet(
      context: Get.context,
      builder: (context) {
        return MenuPage(itemMenu);
      },
    );

    Get.delete<MenuController>();
    Get.delete<MenuRepository>();
  }
}
```

Fonte: o autor.

Na figura 23 o método openMenu da página home_controller executa Get.put para a chamada da do MenuRepository, página menu_repository, e outro Get.put para MenuController, página menu_controller, passando o parâmetro indexCardapio para indexar de acordo com o grupo selecionado.

Uma aplicação de compra de pizzas apresenta especificidades que geralmente não são comuns em outros aplicativos de entrega de comida. Em geral há opções para compra de pizzas de diferentes tamanhos e com diferentes opções de bordas. Essas características somadas à necessidade de apresentar detalhes para que o cliente possa observar os ingredientes que compõe cada pizza levou a um reestudo do projeto para escolha da melhor forma de apresentar esses recursos ao usuário. Deste estudo chegou-se a construção de uma tela de detalhes que permite especificar todas as características inerentes ao produto.

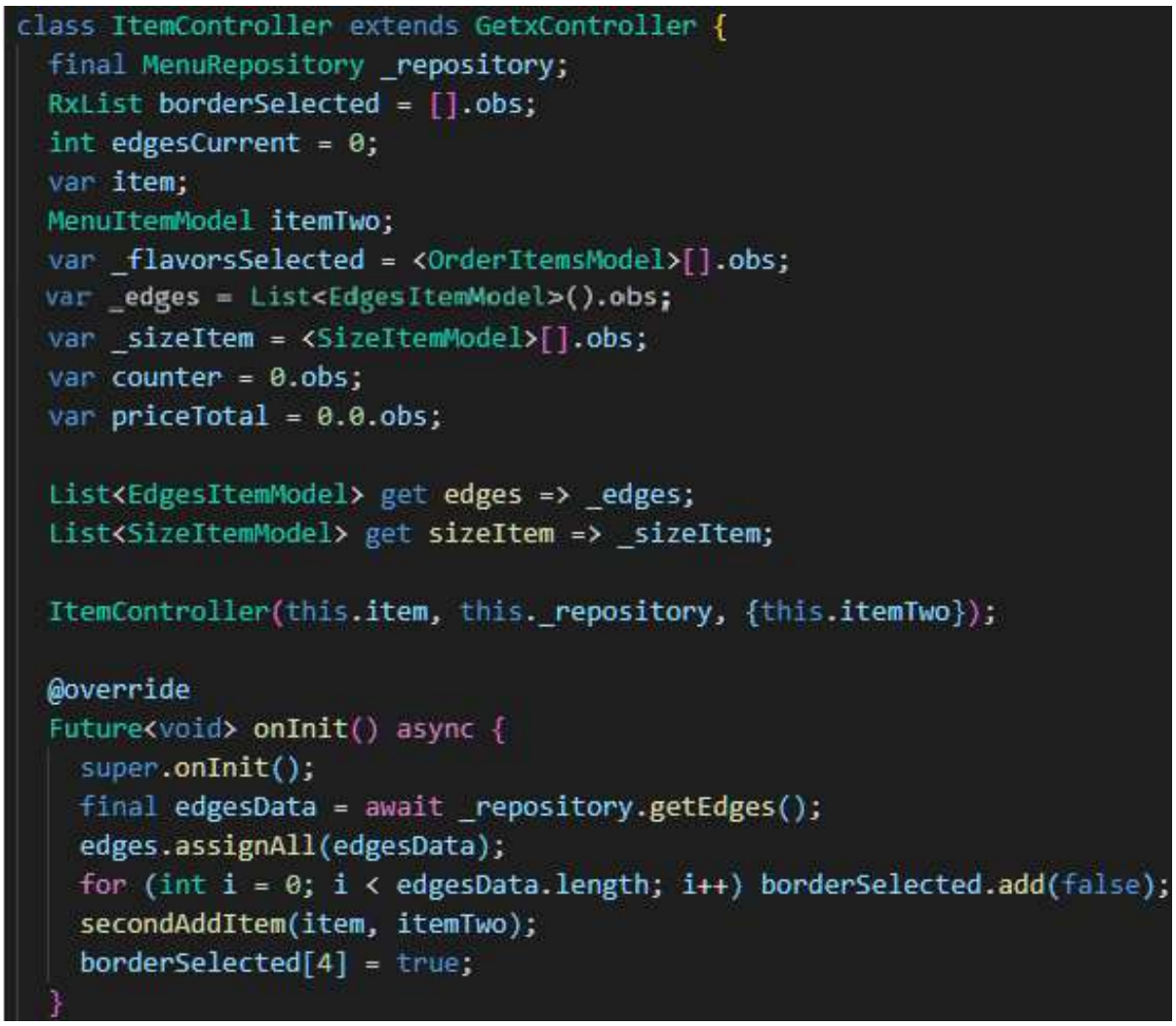
Figura 24: tela de detalhes



Fonte: o autor.

Na tela de detalhes do item cuja figura 24 exibe a parte superior à esquerda e a parte inferior à direita, é carregada a descrição daquele item a partir da coluna “descricao” que foi adicionada na tabela cardapio_grupo_item. Nessa tela há opções para escolha de borda, bem como possibilita ao cliente especificar quantas pizzas daquele sabor deseja adquirir, apresentando ao usuário a quantidade selecionada e o valor total.

Figura 25: classe ItemController.



```
class ItemController extends GetxController {
  final MenuRepository _repository;
  RxList borderSelected = [].obs;
  int edgesCurrent = 0;
  var item;
  MenuItemModel itemTwo;
  var _flavorsSelected = <OrderItemsModel>[].obs;
  var _edges = List<EdgesItemModel>().obs;
  var _sizeItem = <SizeItemModel>[].obs;
  var counter = 0.obs;
  var priceTotal = 0.0.obs;

  List<EdgesItemModel> get edges => _edges;
  List<SizeItemModel> get sizeItem => _sizeItem;

  ItemController(this.item, this._repository, {this.itemTwo});

  @override
  Future<void> onInit() async {
    super.onInit();
    final edgesData = await _repository.getEdges();
    edges.assignAll(edgesData);
    for (int i = 0; i < edgesData.length; i++) borderSelected.add(false);
    secondAddItem(item, itemTwo);
    borderSelected[4] = true;
  }
```

Fonte: o autor.

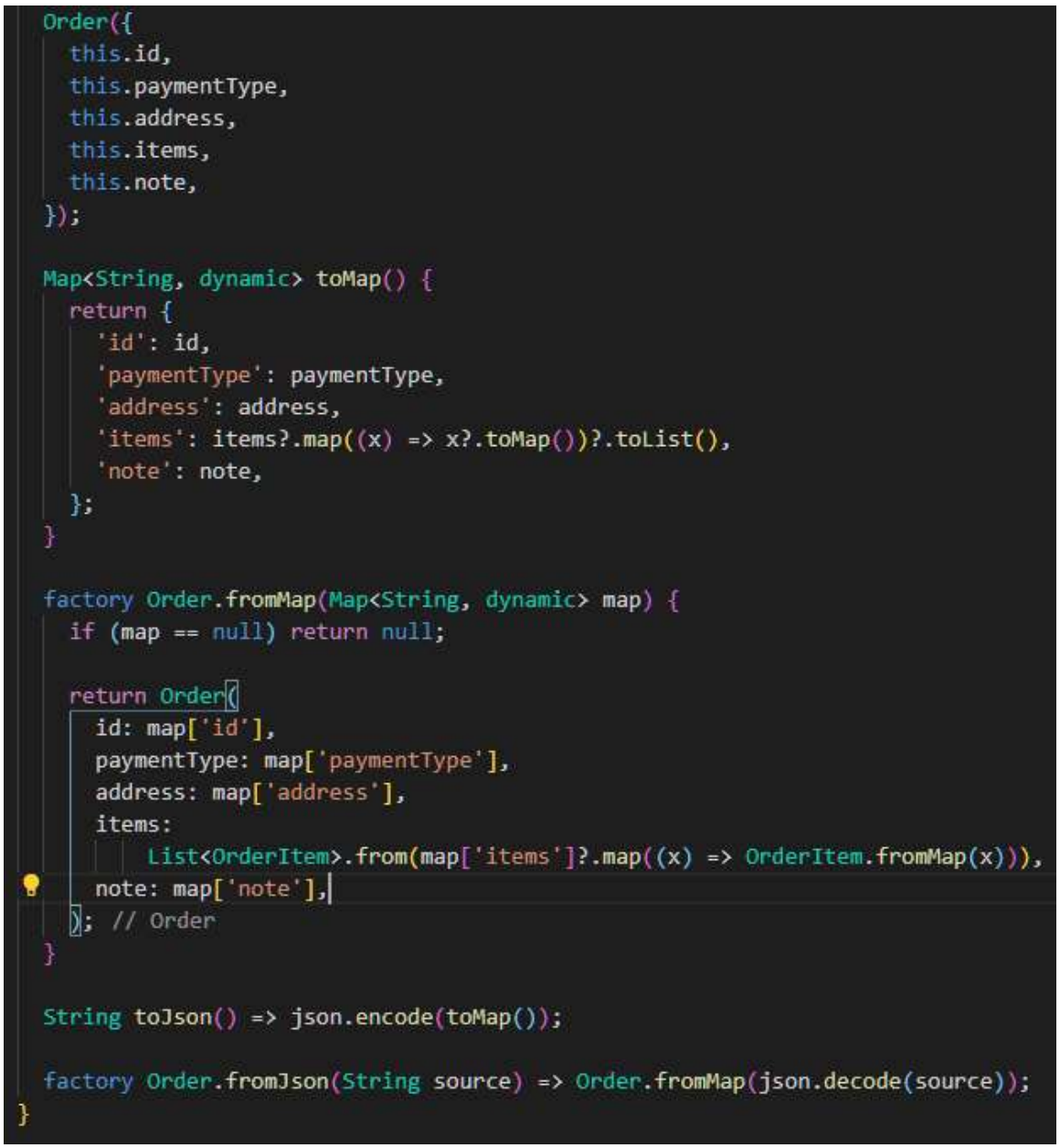
Na figura 25 está a controladora com as variáveis item, segundo item, borda, tamanho, contador de itens adicionados e valor total. Como a aplicação está sendo construída com o pacote getx, sem utilização de “StatefullWidget”, as variáveis são monitoradas pelo pacote por meio da extensão “.obs” para controle e atualização de estados.

Figura 26: Sacola.

Fonte: o autor.

A exibição do carrinho ou sacola de compra, figura 26, lista todos os pedidos adicionados e exibe os campos necessários à finalização do pedido. Além dos campos de endereço de entrega e forma de pagamento, um campo opcional foi introduzido com uma variável do tipo String que retorna ao banco de dados informação sobre alguma observação que o usuário deseja acrescentar.

Figura 27: pedido.



```
Order({
  this.id,
  this.paymentType,
  this.address,
  this.items,
  this.note,
});

Map<String, dynamic> toMap() {
  return {
    'id': id,
    'paymentType': paymentType,
    'address': address,
    'items': items?.map((x) => x?.toMap())?.toList(),
    'note': note,
  };
}

factory Order.fromMap(Map<String, dynamic> map) {
  if (map == null) return null;

  return Order(
    id: map['id'],
    paymentType: map['paymentType'],
    address: map['address'],
    items:
        List<OrderItem>.from(map['items']?.map((x) => OrderItem.fromMap(x))),
    note: map['note'],
  ); // Order
}

String toJson() => json.encode(toMap());

factory Order.fromJson(String source) => Order.fromMap(json.decode(source));
}
```

Fonte: o autor.

A figura 27 exibe a API com as variáveis do pedido: id; paymentType (forma de pagamento); address (endereço); items (lista de itens que compõem o carrinho) e note (observação).

A forma de pagamento retorna obrigatoriamente uma de três opções possíveis, conforme figura a seguir.

Figura 28: formas de pagamento.

```
  Future<void> saveOrder(CheckoutInputModel model) async {
    switch (model.paymentType) {
      case 'Cartão de Crédito':
        model.paymentType = 'credito';
        break;
      case 'Cartão de Débito':
        model.paymentType = 'debito';
        break;
      case 'Dinheiro':
        model.paymentType = 'dinheiro';
        break;
    }

    final response = await _restClient.post('/order/', model.toMap());

    if (response.hasError) {
      print(response.statusText);
      throw RestClientException('Erro ao registrar pedido');
    }
  }
}
```

Fonte: o autor.

As três opções possíveis para forma de pagamento disponíveis (cartão de crédito; cartão de débito e dinheiro) são demonstradas na figura 28. A Figura também exibe um tratamento de exceção caso haja algum erro na comunicação entre a aplicação e a API ao realizar o registro do pedido.

Além da forma de pagamento, outro item de preenchimento obrigatório é a indicação de endereço, do tipo String.

Figura 29: página de pedido.

```
ListTile(
  title: Text('Endereço de Entrega'),
  subtitle: Obx(() => Text(controller.adress)),
  trailing: TextButton(
    onPressed: () => controller.changeAdress(),
    child: Text(
      'Alterar',
      style: TextStyle(color: Get.theme.primaryColor),
    ),
  ),
),
ListTile(
  title: Text('Forma de Pagamento'),
  subtitle: Obx(() => Text(controller.paymentType)),
  trailing: TextButton(
    onPressed: () => controller.changePaymentType(),
    child: Text(
      'Alterar',
      style: TextStyle(color: Get.theme.primaryColor),
    ),
  ),
),
ListTile(
  title: Text('Observação/precisa de troco?'),
  subtitle: Obx(() => Text(controller.note)),
  trailing: TextButton(
    onPressed: () => controller.changeNote(),
    child: Text(
      'Alterar',
      style: TextStyle(color: Get.theme.primaryColor),
    ),
  ),
),
```

Fonte: o autor.

Na figura 29 estão os botões para informação de endereço, indicação de forma de pagamento e um botão de preenchimento opcional para indicar alguma observação, permitindo ao cliente acrescentar observação sobre os itens, sobre o endereço ou mesmo sobre a forma de pagamento.

Após a realização do pedido, o aplicativo disponibiliza ao usuário uma tela com seus pedidos realizados.

Figura 30: Pedidos.

Fonte: o autor.

O histórico de pedidos, figura 30, integra-se à aplicação de back-office, responsável pelo envio de notificações – Push Notifications, capazes de disparar mensagens ao usuário e atualizar o status do pedido.

6 METODOLOGIA

O projeto foi estruturado dentro da metodologia Scrum/ágil com o cronograma de apresentação de *sprints* ou ciclos nas quais um conjunto de atividades devem ser executadas com vistas a entregar um MVP - *Minimum Viable Product* em cada *sprint*. Para tanto foram estabelecidas seis etapas gerais a serem empregadas em cada entrega:

I. Planejamento;
II. Elaboração de layout de interfaces;
III. Desenvolvimento de banco de dados;
IV. Desenvolvimento de API;
V. Desenvolvimento de aplicação;
VI. Testes.

De acordo com Ribeiro, et al., 2015, o Scrum é um método ágil empírico, interativo e com entregas incrementais. Nessa linha, as etapas não são necessariamente sequenciais, não tendo a obrigatoriedade de conclusão de uma para início de outra, permitindo um melhor aproveitamento da distribuição de carga horária de trabalho. Cada *sprint* acompanha a documentação além do desenvolvimento apresentado.

Foram definidas sete entregas que compõem o *Product Backlog,* conforme lista de *sprints* a seguir:

I. Login;
II. Cadastro;
III. Cardápio;
IV. Detalhes de itens;
V. Sacola de compras;
VI. Realização de pedidos.
VII. Histórico de pedidos e tela de *Splash*

As entregas das sprints seguem o cronograma conforme o quadro a seguir:

Tabela 1: cronograma de entrega de *sprints*

SPRINT	DATA DE ENTREGA
I - Login	18/03/2021
II - Cadastro	01/04/2021
III - Cardápio	15/04/2021
IV - Detalhe de itens	06/05/2021
V - Sacola de compras	27/05/2021
VI - Realização de pedidos	10/06/2021
VII - Histórico de pedidos e tela de Splash	24/06/2021

Fonte: o autor.

O cronograma geral de entrega das *sprints,* tabela 1, está detalhado para a execução de cada atividade que integra cada entrega, conforme demonstrado nos gráficos de Burndown apresentados a seguir. Após a conclusão do cronograma de desenvolvimento o projeto terá foco na entrega das três soluções, banco de dados, API e aplicação, acompanhamento e coleta de *feedbacks* e documentação.

Na *sprint* 1 foi entregue um MVP apto a realizar as rotinas de login de usuário, abrangendo os desenvolvimentos necessários para toda a concepção dos softwares e processos a serem executados no aplicativo (*front-end*), API e banco de dados (*back-end*). Na tabela a seguir estão as atividades planejadas da *sprint* e respectivos pesos para o gráfico de *burndown*.

Tabela 2: Planejamento da *sprint* 1.

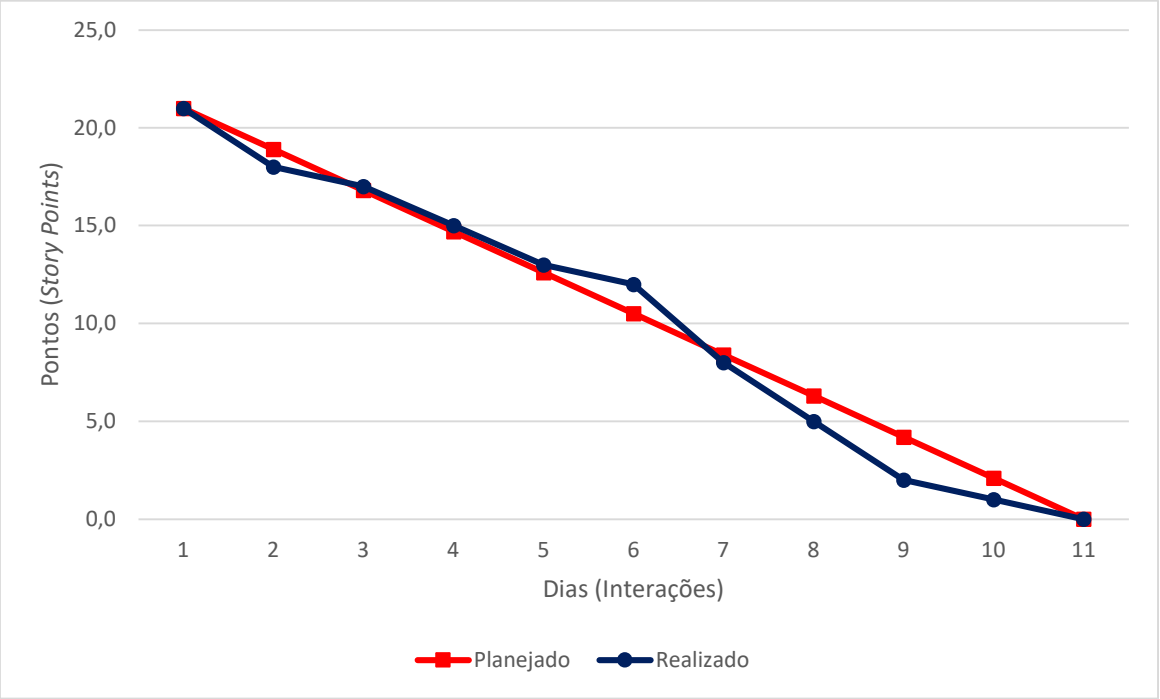
Atividade	Pontos
Levantamento de requisitos	3
Definição de camadas e Model View Controller - MVC	3
Elaboração de modelo de dados	2
Implementação de variáveis e ambiente e lógica de conexão (DB)	2
Implementação de criptografia - SHA-256 para senhas	2

Desenvolvimento do método de login	4
Definição de layout de telas	2
Construção de tabelas (DB)	1
Testes	1
Documentação	1
Total	**21**

Fonte: o autor.

Conforme tabela 2, para a primeira entrega foram planejadas 10 tarefas com peso total de 21 pontos.

Gráfico 1: *Burndown* conforme execução da *sprint* 1.



Fonte: o autor.

O gráfico de *burndown* da primeira entrega, gráfico 1, detalha a execução das atividades propostas na tabela 2 ao longo dos dias úteis durante as duas semanas de trabalho. Nessa entrega houve uma quantidade maior de atividades em razão da necessidade de melhor compreensão do projeto como um todo, possibilitando uma abordagem sistêmica de proêmio para posteriores especificidades. No foco da entrega foi desenvolvida a funcionalidade de login do usuário na aplicação.

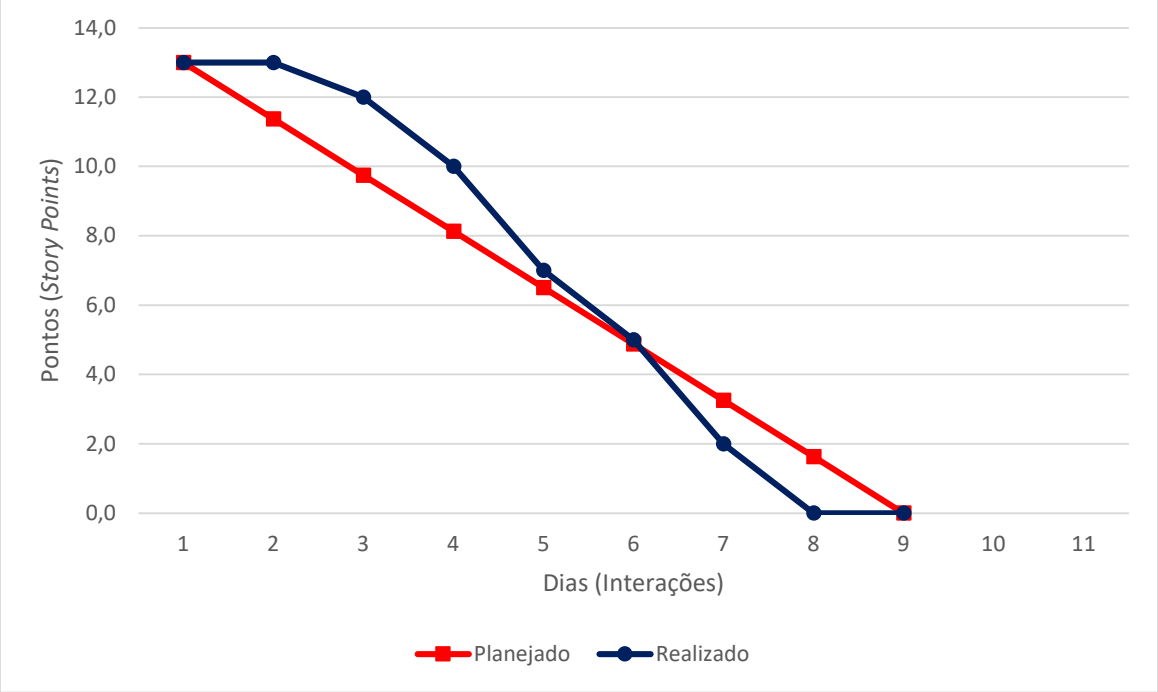
Tabela 3: Planejamento da *sprint* 2.

Atividade	Pontos
Desenvolvimento do método de cadastro de usuário (saveUser)	4
Interface tela cadastro	3
Interface tela home	2
Implementação de tela - home_page	2
Testes	1
Documentação	1
Total	**13**

Fonte: o autor.

Na segunda entrega, tabela 3, foram planejadas 6 entregas totalizando peso de 13 pontos.

Gráfico 2: *Burndown* conforme execução da *sprint* 2.



Fonte: o autor.

Conforme se observa no gráfico de *burndown* da segunda entrega a execução das atividades planejadas findou-se 1 dia antes da data prevista. Nesta entrega foi desenvolvida a funcionalidade de cadastro de novos usuários.

Tabela 4: Planejamento da *sprint* 3.

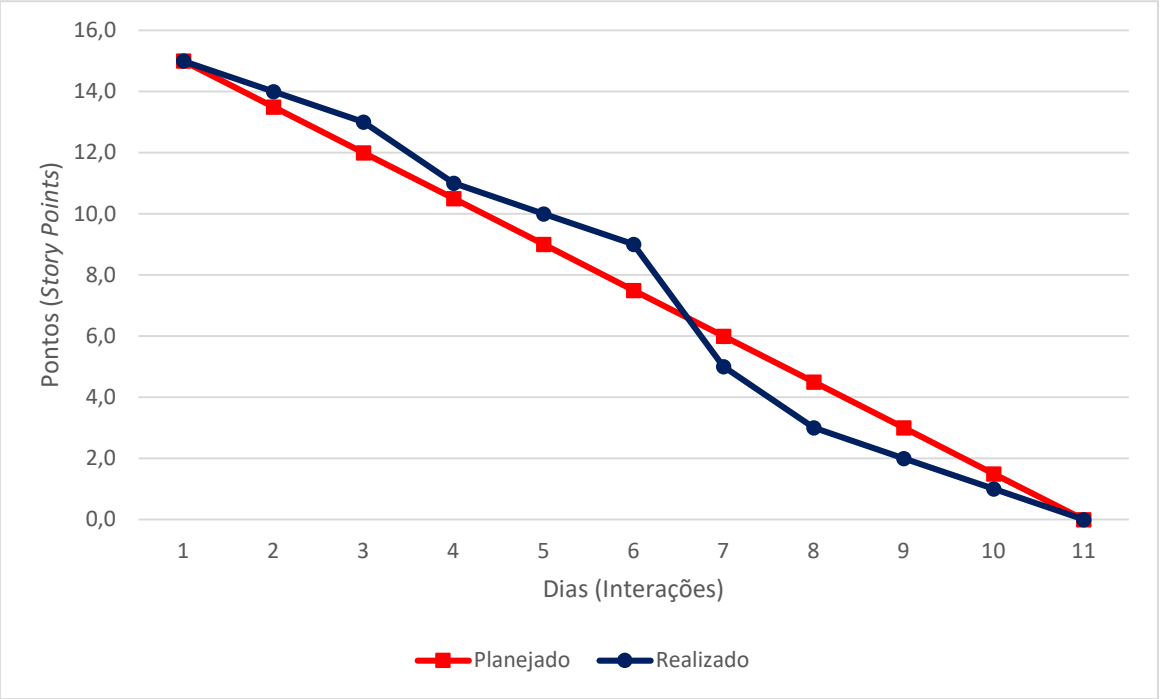
Atividade	Pontos
Interface tela cardápio	2

Categoria pizzas salgadas	1
Categoria pizzas doces	1
Categoria pizzas brotinho	1
Categoria pizzas veganas	1
Categoria bebidas	1
Lógica de 2 sabores	5
Implementação do Scroll Index	1
Testes	1
Documentação	1
Total	**15**

Fonte: o autor.

A tabela 4 exibe as atividades planejadas para a *sprint* 3, com peso de 15 pontos ao final.

Gráfico 3: *Burndown* conforme execução da *sprint* 3.



Fonte: o autor.

A terceira entrega objetivou listar o cardápio ao usuário. O gráfico 3 exibe um momento de inflexão entre os dias 6 e 7 em razão da complexidade no desenvolvimento da lógica de 2 sabores, conforme regra de negócio, haja vista que o cliente pode optar por escolher um item do cardápio para o qual referida regra se aplica, a exemplo uma pizza salgada com 2 sabores distintos, bem como também desejar adquirir um item cuja lógica é distinta, por exemplo uma bebida.

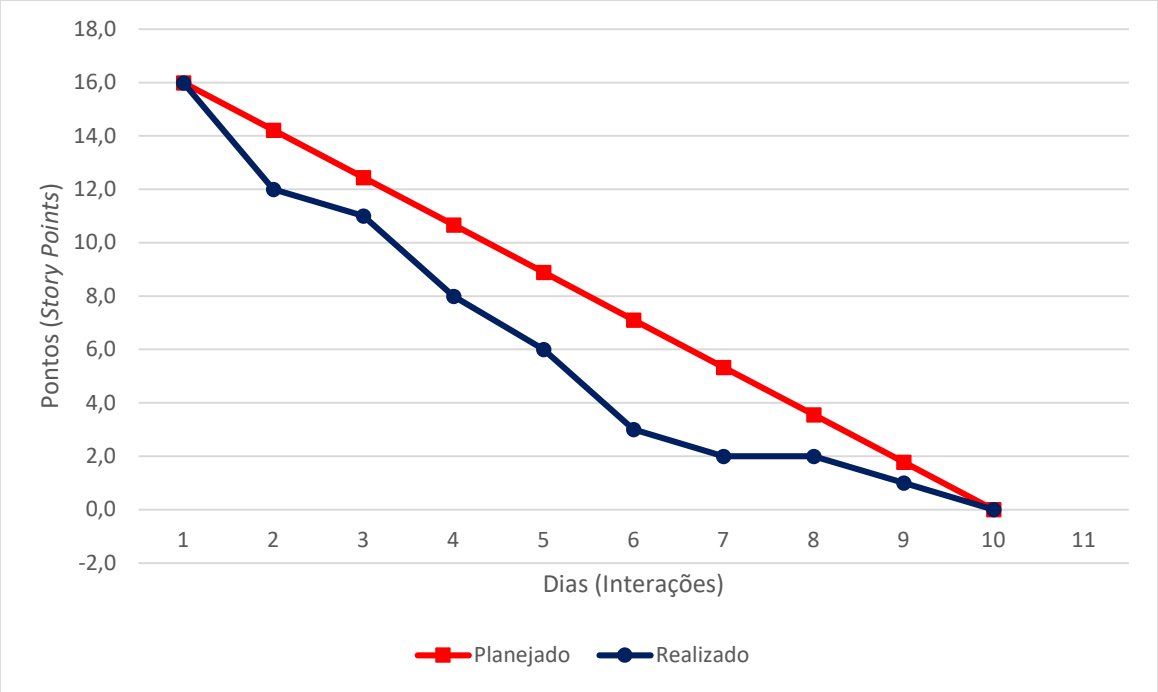
Tabela 5: Planejamento da *sprint* 4.

Atividade	Pontos
Interface da tela de item	3
Desenvolvimento de lógica de negócio (controllers)	4
Implementação de opcionais do item (bordas)	5
Construção de comandos finalizar e retornar	1
Funções somar itens e totalizar valores	1
Testes	1
Documentação	1
Total	**16**

Fonte: o autor.

Na tabela 5 estão descritas as atividades previstas na quarta entrega, com foco em detalhar cada item do cardápio, somando peso de 16 pontos.

Gráfico 4: *Burndown* conforme execução da *sprint* 4.



Fonte: o autor.

A execução da entrega 4 encontrou dificuldade inicial para se definir uma interface apta a atender as necessidades do usuário sem incorrer em uma tela demasiadamente complexa, com excesso de informação, porém capaz de fornecer todas as informações e comandos para que o usuário consiga navegar e adicionar os itens ao carrinho em quantidade e com todos os opcionais, conforme desejar. Referida

adversidade gerou um atraso inicial que se prolongou ao longo de todo o período, a conclusão dos trabalhos, contudo, pôde ser realizada dentro do cronograma estabelecido, conforme demonstrado no gráfico 4.

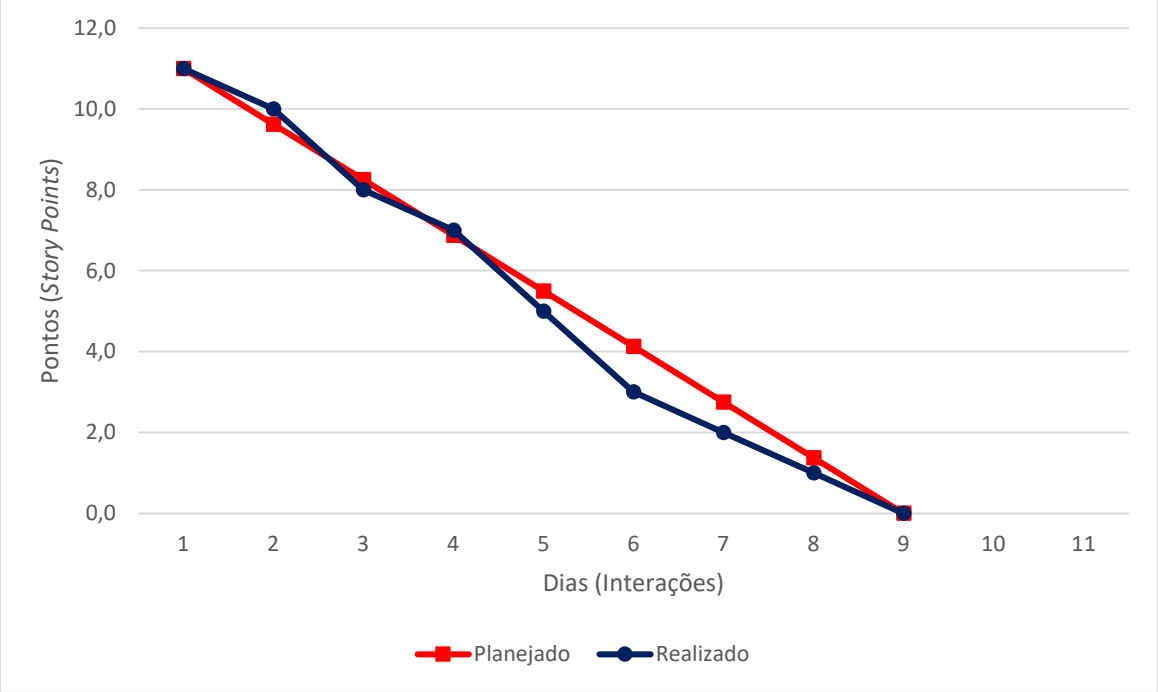
Tabela 6: Planejamento da *sprint* 5.

Atividade	Pontos
Desenvolvimento de lógica de carrinho de compras (shopping_card_controller)	3
Implementação da tela de sacola de compras (shopping_card_page)	3
Especificações obrigatórias e opcionais de pedido	2
Botões finalizar pedido e voltar	1
Testes	1
Documentação	1
Total	**11**

Fonte: o autor.

Para a entrega 5 foi definida a execução da sacola ou carrinho de compras cujas atividades estão detalhadas na tabela 6, somando-se 11 pontos.

Gráfico 5: *Burndown* conforme execução da *sprint* 5.



Fonte: o autor.

De acordo com o gráfico 5 a execução da *sprint* 5 conseguiu desenvolver as atividades previstas ao seu tempo, apresentando a curva de realização bem próxima da curva de planejamento.

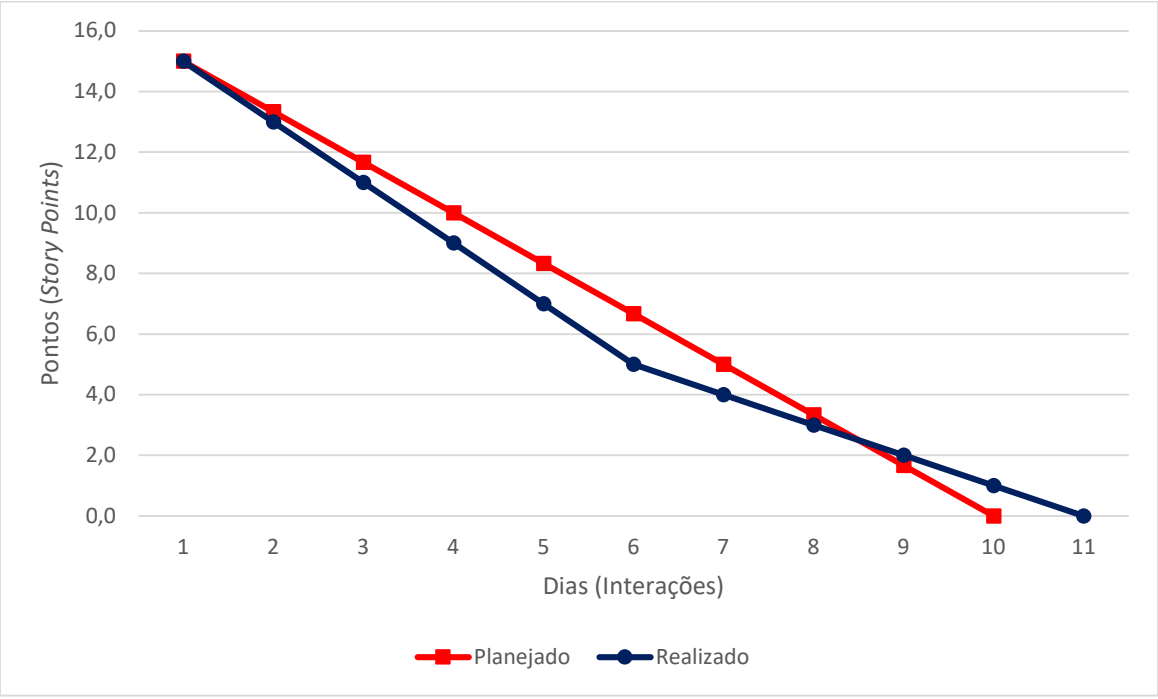
Tabela 7: Planejamento da *sprint* 6.

Atividade	Pontos
Desenvolvimento do método de cadastro de pedido (saveOrder)	4
Desenvolvimento de parâmetros de pedido	2
Desenvolvimento de modelo abstrato (order_item_model)	2
Desenvolvimento de modelo abstrato (order_model)	2
Implementação lógica - order_repository	3
Testes	1
Documentação	1
Total	**15**

Fonte: o autor.

A tabela 7 lista as atividades previstas para a sexta entrega, dentro do objetivo de implementar a realização do pedido a ser registrado no banco de dados.

Gráfico 6: *Burndown* conforme execução da *sprint* 6.



Fonte: o autor.

Conforme demonstrado no gráfico 6, a execução das atividades se prolongou por tempo maior que o inicialmente previsto, com a entrega sofrendo atraso de 1 dia.

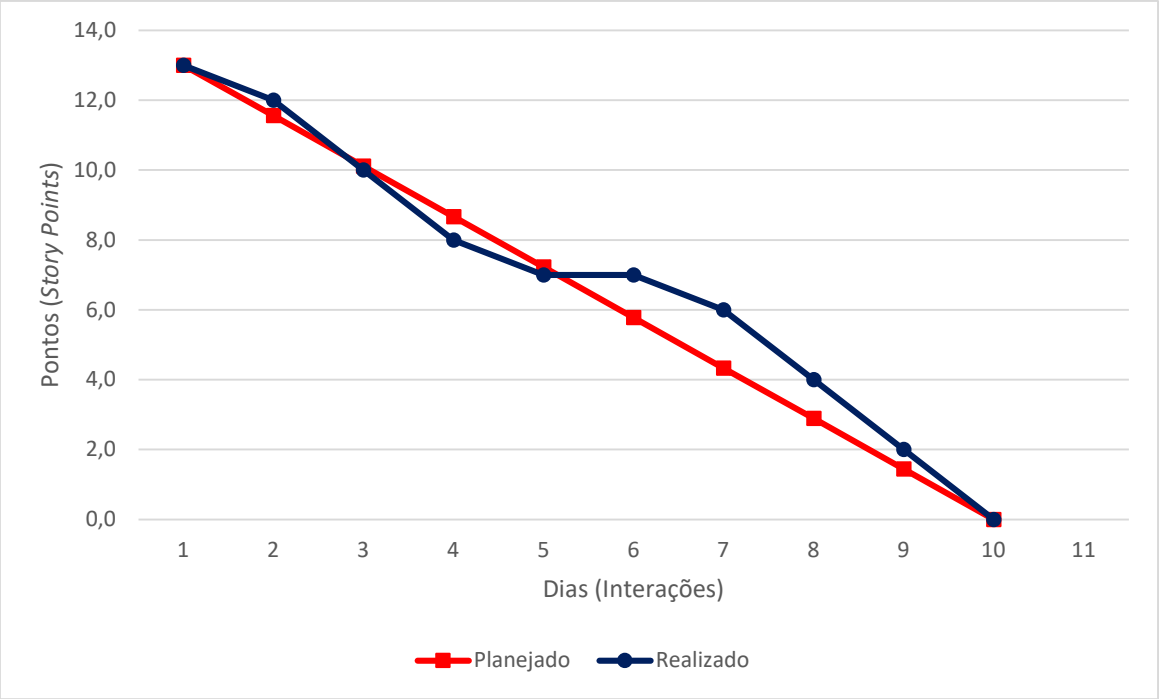
Tabela 8: Planejamento da *sprint* 7.

Atividade	Pontos
Desenvolvimento do método de busca de pedidos (ordersHystory)	3
Interface de tela - meus pedidos	1
Desenvolvimento lógico - orders_bindings	1
Desenvolvimento lógico - orders_controller	1
Implementação de tela - orders_page	1
Interface e implementação de tela - splash_page	2
Desenvolvimento lógico - splash_controller	2
Testes	1
Documentação	1
Total	**13**

Fonte: o autor.

A última *sprint* foi planejada para a entrega de duas funcionalidades, sendo a primeira a exibição dos pedidos realizados pelo usuário e a segunda uma tela temporária a ser exibida ao se inicializar o aplicativo. As atividades dessa entrega estão listadas na tabela 8 e sua execução foi realizada conforme o gráfico a seguir.

Gráfico 7: *Burndown* conforme execução da *sprint* 7.



Fonte: o autor.

Os trabalhos realizados para a concretização das funcionalidades da última entrega necessitaram de uma pausa de 1 dia sem desenvolvimento, conforme

demonstrado no vetor de realização entre os dias 5 e 6 do gráfico 7, em razão de indisponibilidade de tempo hábil para execução ocasionada por conflito de compromissos na agenda no mês de junho, mas sem prejuízo do cronograma previsto.

As ferramentas utilizadas para a confecção do *back-end* foram o Visual Studio Code – editor utilizado para o desenvolvimento em linguagem dart; Postman – Ferramenta utilizada para testar as requisições do servidor e MySql Workbench – Ferramenta adotada para construção de modelo de dados e consulta ao banco de dados. Para o *front-end,* além das ferramentas já apresentadas, foram empregadas as seguintes ferramentas: Adroid Studio para o desenvolvimento com a SDK flutter e Figma para o planejamento e definição de layouts.

7 CONSIDERAÇÕES FINAIS

A construção de uma aplicação para serviço de entregas em domicílio envolve várias soluções interativas para que se possa alcançar boa confiabilidade e praticidade nas operações. Nesse contexto, o trabalho tem por base oferecer uma boa experiência de uso aos clientes ao longo de todo processo de compra.

A experiência do usuário, contudo, vai além das facilidades que o software oferece. Segundo Rocha (2017), a avaliação de usabilidade é mais abrangente:

> "Para avaliar a usabilidade de sistemas, devem ser levados em consideração os aspectos comuns a todos os sistemas computacionais e as características particulares deles. Por exemplo, se um sistema é web, a avaliação da usabilidade deve considerar a facilidade de aprendizado e a satisfação do usuário, mas também aspectos como disponibilidade de rede e tempo de resposta de uma requisição."

Para que a solução possa oferecer uma experiência de uso positiva, portanto, a alocação dos softwares em ambiente de produção também necessita de estrutura adequada, apta a oferecer bom tempo de resposta e capacidade suficiente para atender as necessidades de cada cliente.

Desta feita, em uma análise de mercado, a comercialização dos softwares como produto final implica na necessidade do cliente realizar a aquisição de licenças de uso em conjunto com a contratação de servidores web balanceados para sua necessidade, assinatura para publicação em loja de aplicativos, além do custeio de mão de obra especializada para manutenção. Outra boa prática seria o registro de domínio próprio para apontar o aplicativo, simplificando eventuais futuras alterações de servidores sem a necessidade de lançar atualização, bastando redirecionar o domínio registrado.

Depreende-se que a aquisição no modelo de produto licenciado requer elevado investimento financeiro e contato direto com diferentes fornecedores de softwares e serviços. No presente trabalho, tendo como público-alvo o pequeno comerciante e lojista, referido modelo apresenta elevada complexidade e alto investimento financeiro, comprometendo a viabilidade da proposta.

Outra abordagem consiste em adotar o modelo de serviço, com a infraestrutura de servidores e custos de manutenção embarcados, dentro do conceito de *Software as a Service* – SaaS. Esse modelo provê simplicidade ao cliente, além de exigir menor

investimento inicial, possibilitando melhor estimativa de sucesso comercial dentro desse nicho de mercado. A adoção do modelo SaaS, contudo, envolve estabelecer uma precificação com menor margem de lucro, geralmente sem retorno suficiente para cobertura de custos a curto prazo, visando alcançar maior escalabilidade, ampliando a carteira de clientes no médio e longo prazo.

REFERÊNCIAS

BLOKDYK, Gerardus. **API: application programming interface**. 3. ed. [*S. l.*]: Emereo Pty Limited, 2018. 288 p. ISBN 978-0655430360.

NIELSEN, J. **Usability Engineering**. New York, NY: Academic Press, 1993.

GEHANI, Narain. **The Database Book**: Principles & Practice Using MySQL. 2. ed. [*S. l.*]: Silicon Press, 2011. 375 p. ISBN 978-0929306452.

MESTRINER, Fábio. Design de embalagem. Curso Avançado. 1ł. ed. São Paulo: Prentice Hall, 2002.

RIBEIRO, Rafael Dias; RIBEIRO, Horácio da Cunha e S. **Métodos ágeis em gerenciamento de projetos**. Rio de Janeiro: Horácio da Cunha e Sousa Ribeiro, 2015.

ROCHA, Larissa Castro. **HUbis: heurísticas de usabilidade para avaliar sistemas ubíquos**. 2017. 132 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal do Ceará, Fortaleza, 2017.

SOMMERVILLE, I. **Engenharia de Software.** 8ª Edição. São Paulo: Pearson Addison-Wesley, 2007.