

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-
GRANDENSE - CÂMPUS PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET**

CRISLEY ALVES DE MOURA

**FASTER-FOOD: SISTEMA DE GERENCIAMENTO DE PEDIDOS EM
FAST-FOODS POR MEIO DE UM DISPOSITIVO MÓVEL**

Élder F. F. Bernardi

**PASSO FUNDO
2018**

CRISLEY ALVES DE MOURA

**FASTER-FOOD: SISTEMA DE GERENCIAMENTO DE PEDIDOS EM
FAST-FOODS POR MEIO DE UMA APLICAÇÃO MÓVEL**

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-rio-grandense, Câmpus Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Élder F. F. Bernardi

PASSO FUNDO

2018

CRISLEY ALVES DE MOURA

**FASTER-FOOD: SISTEMA DE GERENCIAMENTO DE PEDIDOS EM FAST-
FOODS POR MEIO DE UMA APLICAÇÃO MÓVEL**

Trabalho de Conclusão de Curso aprovado em ____/____/____ como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

Élder F. F. Bernardi

Jorge Luis Boeira Bavaresco

José Antônio Oliveira de Figueiredo

Rafael Marisco Bertei

PASSO FUNDO

2018

DEDICATÓRIA

*Aos meus pais
que independente da distância
sempre me apoiaram.*

AGRADECIMENTOS

Primeiramente agradeço pelo apoio de minha família que, apesar da distância durante o período, sempre demonstrou apoio e incentivo durante toda a trajetória do curso.

Agradeço ao IFSUL por fornecer as condições necessárias para tornar a conclusão possível e ao orientador pela orientação prestada durante o desenvolvimento do projeto.

Agradeço também ao Google e Stackoverflow e sua comunidade pelas dúvidas sanadas durante todo o desenvolvimento do projeto, com certeza o projeto não teria saído do papel caso algumas dúvidas não fossem sanadas.

EPÍGRAFE

“O sangue gera parentes, mas somente a
lealdade faz uma família”
Desconhecido

RESUMO

O presente estudo teve como objetivo o desenvolvimento de um aplicativo que permitisse aos usuários realizar pedidos em um estabelecimento através de um dispositivo móvel sem a necessidade de seguir uma fila física para tal. O aplicativo foi desenvolvido utilizando Ionic 3, Angular 5, HTML, CSS3, JAVASCRIPT e um servidor rodando em NODEJS para realizar o webpack da aplicação. O backend da aplicação foi desenvolvido em Java, utilizando os recursos da JPA para realizar o mapeamento objeto-relacional da aplicação, do Hibernate Validator para validação das informações que seriam persistidas e do framework Spring para o desenvolvimento do serviço Rest. O resultado deste estudo foi permitir que o cliente realizasse pedidos em um estabelecimento carregando a listagem de produtos através de um código QRCODE e ser notificado em tempo real sobre o progresso do pedido realizado.

Palavras-chave: Java, Ionic, Angular, Rest

ABSTRACT

This study aims the development of an application which allows the customer to order at an establishment through the smartphone without needing to stand in line for that. The application was developed using Ionic 3, Angular 5, HTML, CSS3, JAVASCRIPT and a server running with NODEJS to make the webpack of the application. The backend of the application was developed in Java, using the resources of the JPA to make de map of the relational object of the application, the Hibernate Validator to validate the data which were going to be persisted and the Spring framework to develop de Rest webservice. The result of this study was to allow the customer to make orders at an establishment loading the products list through a QRCODE and be notified in real time about the progress of the order.

Keywords: Java, Ionic, Angular, Rest.

LISTA DE FIGURAS

Figura 1 - Casos de Uso	33
Figura 2 - Diagrama de atividade realizar login	38
Figura 3 - Diagrama de atividade gerar relatório	39
Figura 4 - Diagrama de atividade realizar pagamento.....	40
Figura 5 - Diagrama de atividade realizar pedido.....	41
Figura 6 - Diagrama de atividade listar produtos.....	42
Figura 7 - Diagrama de classes.....	43
Figura 8 - Diagrama de implantação	44
Figura 9 - Arquivo POM	47
Figura 10 - Bibliotecas do Projeto	48
Figura 11 - Conexão com banco de dados	49
Figura 12 - Dependências da aplicação.....	51
Figura 13 - Arquitetura do Sistema	53
Figura 14 - Camada modelo	54
Figura 15 - Classe Cliente	55
Figura 16 - Classe Pedido.....	57
Figura 17 - Camada Repository	59
Figura 18 – CORSFilter	60
Figura 19 – Controllers	61
Figura 20 - Serviço Rest.....	61
Figura 21 - Serviço Rest – Classe Pedido.....	64
Figura 22 - Camada modelo do aplicativo	66
Figura 23 - Conexão com o serviço	68
Figura 24 - Views	70
Figura 25 - Views – Listagem de produtos	71
Figura 26 - JSON de internacionalização	72
Figura 27 - Definição do local dos arquivos de internacionalização	73
Figura 28 - Diretiva Translate	74
Figura 29 – Rotas	75
Figura 30 - Atualização do pedido	77
Figura 31 - Pedido servisse	78

Figura 32 – Tutorial	80
Figura 33 – Pedido	82
Figura 34 - Notificações	83
Figura 35 - Painel administrativo	84
Figura 36 - Histórico e avaliação	85

LISTA DE TABELAS

Tabela 1 - Anotações de segurança do Spring.....	26
Tabela 2 - Verbos utilizados em serviços REST	29
Tabela 3 - Caso de uso realizar login.....	33
Tabela 4 - Caso de uso realizar pagamento	35
Tabela 5 - Caso de uso realizar pedido	36
Tabela 6 - Caso de uso gerar relatório.....	37

|

LISTA DE ABREVIATURAS E SIGLAS

APP – Aplicativo
CSS – Cascading Stylesheet
HTML – Hyper Text Markup Language
API – Application Programming Interface
JSON – Javascript Object Notation
JPA – Java Persistence API
ORM – Object Relational Mapping
REST – Representational State Transfer
WWDC – Apple Worldwide Developers Conference
GPS – Global Position System
SDK – Software Development Kit
MVC – Model-View-Controller
CRUD – Create, Read, Update and Delete
PHP – Personal Home Page
API – Application Programming Interface
JSP – JavaServer Pages
JSF – JavaServer Faces
EJB – Enterprise JavaBeans
ARO – Army Research Office
NFS – National Science Foundation
SGBD – Data Base Management System
HTTP – HyperText Transfer Protocol
Rest – Representational State Transfer
JDBC – Java Database Connectivity
POM – Project Object Model
NPM – Node Package Manager
URI – Uniform Resource Identifiers
UUID – Unique Universal Identifier

SUMÁRIO

1	INTRODUÇÃO.....	15
1.1	OBJETIVOS.....	16
1.1.1	Objetivo geral	16
1.1.2	Objetivos específicos.....	16
2	REFERENCIAL TEÓRICO	17
2.1	FORMAS DISPONÍVEIS DE DESENVOLVIMENTO DE APLICATIVOS MÓVEIS	17
2.1.1	Desenvolvimento Nativo	17
2.1.1.1	Android.....	17
2.1.1.2	Windows Phone	18
2.1.1.3	iOS	19
2.1.2	Desenvolvimento Híbrido.....	20
2.1.2.1	Apache Cordova	20
2.1.2.2	Outros	21
2.1.2.3	Tecnologias para desenvolvimento de aplicações móveis híbridas	21
2.2	SERVIDOR	23
2.2.1	NodeJS.....	23
2.2.2	Java	23
2.2.3	Java EE	24
2.2.4	Spring	25
2.3	PERSISTÊNCIA.....	26
2.3.1	PostgreSQL	27
2.3.2	Java Persistence API.....	27
2.4	COMUNICAÇÃO.....	28
2.4.1	Json	28
2.4.2	Rest	28
2.4.3	QRCODE.....	29
3	METODOLOGIA	31
3.1	ESTUDO DE CASO	31
3.2	GESTÃO DO NEGÓCIO.....	31
3.3	FUNCIONAMENTO	31

3.4	REQUISITOS FUNCIONAIS.....	32
3.5	REQUISITOS NÃO FUNCIONAIS	32
3.6	DIAGRAMA DE CASOS DE USO.....	32
3.7	DESCRIÇÃO DO CASO DE USO	33
3.7.1	Caso de uso realizar login	33
3.7.2	Caso de uso realizar pagamento.....	35
3.7.3	Caso de uso realizar pedido	36
3.7.4	Caso de uso gerar relatório	37
3.8	DIAGRAMA DE ATIVIDADES.....	38
3.9	DIAGRAMA DE CLASSES	42
3.10	DIAGRAMA DE IMPLANTAÇÃO	44
4	DESENVOLVIMENTO DO PROTÓTIPO DA APLICAÇÃO	46
4.1	PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO.....	46
4.1.1	Backend do serviço	46
4.1.2	Aplicativo Móvel.....	49
4.1.3	Painel administrativo.....	51
4.2	ARQUITETURA DO SISTEMA	52
4.3	SERVIDOR WEB	53
4.3.1	Camada Model	53
4.3.2	Camada Repository	58
4.3.3	Camada Controller.....	59
4.4	APLICATIVO MÓVEL	64
4.4.1	Arquitetura	65
4.4.2	Camada modelo	66
4.4.3	Serviço.....	67
4.4.4	Views.....	69
4.4.5	Internacionalização.....	72
4.5	PAINEL ADMINISTRATIVO.....	74
4.5.1	Arquitetura	74
4.5.2	Views	76
4.5.3	Serviço.....	77
5	RESULTADOS.....	79
5.1	TUTORIAL	79

5.2	REALIZAÇÃO DE PEDIDO.....	81
5.3	PAINEL ADMINISTRATIVO E NOTIFICAÇÕES.....	82
5.4	HISTÓRICO E AVALIAÇÃO DOS PRODUTOS	84
6	AVALIAÇÃO DO PROJETO	86
7	CONSIDERAÇÕES FINAIS.....	88
8	REFERÊNCIAS	91

1 INTRODUÇÃO

Nos últimos anos houveram avanços tecnológicos que permitiram às pessoas instalar aplicativos em seus dispositivos móveis para usufruir de um determinado recurso que o aplicativo oferecia, seja este recurso o acesso a uma rede social, a um aplicativo que permita ter um controle sobre seus gastos financeiros ou mesmo aplicativos que automatizem um determinado serviço.

A realização de pedidos em Fast-Foods geralmente obriga o cliente a seguir por uma fila física composta por N pessoas para a realização de um pedido, sendo necessário obter informações sobre os produtos, sobre as formas de pagamento e até mesmo a possibilidade de o atendente ter a capacidade de se comunicar em outro idioma com o cliente, impactando diretamente no tempo de espera das demais pessoas e na qualidade do atendimento do estabelecimento.

O desenvolvimento de um aplicativo que facilite este processo faz com que o cliente não necessite mais seguir uma fila física para a realização de pedidos e pagamentos, garante que clientes de outros países possam ser atendidos em seu idioma nativo evitando assim confusões idiomáticas e pelas informações que serão armazenadas pelo estabelecimento também será possível realizar o mapeamento dos interesses dos clientes, para assim poder realizar ações de marketing em cima de um cliente em específico ou um grupo de clientes.

A partir destas informações, o presente trabalho possui como objetivo principal o desenvolvimento de um aplicativo que permita aos clientes realizar pedidos em um estabelecimento através de um dispositivo móvel, sendo possível também definir o idioma que será utilizado para realizar o seu atendimento, garantindo assim uma maior integridade das informações passadas ao cliente.

Para realizar o desenvolvimento da aplicação foi realizada uma análise sobre as tecnologias que seriam necessárias para desenvolver o projeto, tendo como base as principais características que foram identificadas no estudo do caso, para assim poder desenvolver algo que estivesse de acordo com o estudo que foi realizado.

1.1 OBJETIVOS

Nesta seção serão apresentados os objetivos gerais e específicos do projeto.

1.1.1 Objetivo geral

Desenvolver uma aplicação híbrida que auxilie na realização de pedidos em um Fast-food através do dispositivo particular do cliente.

1.1.2 Objetivos específicos

- Diminuição da fila física na realização de pedidos e pagamentos
- Armazenamento dos interesses dos clientes para futuras ações de marketing
- Autonomia do cliente na realização de pedidos
- Internacionalização da aplicação
- Realizar pagamentos através de uma plataforma digital

2 REFERENCIAL TEÓRICO

Nesta seção serão abordados os conceitos essenciais para o desenvolvimento do projeto.

2.1 FORMAS DISPONÍVEIS DE DESENVOLVIMENTO DE APLICATIVOS MÓVEIS

Para o desenvolvimento de aplicativos móveis, há a possibilidade de desenvolver a aplicação para a plataforma nativa do dispositivo ou desenvolvê-la de forma híbrida.

2.1.1 Desenvolvimento Nativo

Nesta seção do projeto serão apresentadas, de forma breve, as três principais plataformas para desenvolvimento de aplicativos móveis nativos que atualmente estão no mercado.

Vale salientar que os aplicativos que são desenvolvidos de forma nativa são desenvolvidos utilizando o SDK específico da plataforma para o qual ele está sendo construído, ou seja, ele apenas estará disponível em tal plataforma (BRAVI, 2015).

O desenvolvimento específico para a plataforma garante que o usuário tenha uma maior facilidade em relação à usabilidade do aplicativo por utilizar recursos gráficos do próprio dispositivo que o usuário já está acostumado e também permite ao aplicativo ter acesso a todos os recursos que o dispositivo tem a oferecer (BRAVI, 2015).

2.1.1.1 Android

O Android é uma plataforma de desenvolvimento de aplicativos para dispositivos móveis, como celulares e Tablets, escritos em Java. O Android foi criado com o objetivo de criar um padrão em uma plataforma de código aberto, para assim poder

garantir que todas as tendências e tecnologias estejam disponíveis nos celulares para o maior número de pessoas possíveis (LECHETA, 2013).

O sistema operacional é baseado em Linux e teve o desenvolvimento iniciado em 2003 por uma empresa chamada Android Inc. Em 2005, o Google adquiriu a empresa e é até hoje responsável pelo desenvolvimento e a manutenção da tecnologia (MONTEIRO, 2012).

Embora o Google seja o principal responsável por trás da plataforma, há outras grandes empresas que acompanham o desenvolvimento da dela, tanto é que em 2007 foi criado a OHA (*Open Handset Alliance*) que se trata da união de grandes empresas do mercado mundial de telefonia móvel do mundo, grupo este que é composto por grandes nomes do mercado, como: Motorola, Samsung, Intel, Acer e muitas outras, sendo tal grupo liderado pelo Google (LECHETA, 2013).

Atualmente o Android encontra-se na versão 8.0 (ANDROID, 2017), oferecendo o poder da plataforma não apenas a dispositivos móveis, mas a relógios, televisores e até em acessórios que podem ser utilizados pelo usuário, como: roupas, óculos e entre outros (ANDROID, 2017).

2.1.1.2 Windows Phone

Com o crescimento de dispositivos móveis nas últimas décadas e um novo mercado tecnológico se expandindo no mundo, a Microsoft viu a oportunidade de desenvolver a sua própria plataforma para dispositivos móveis para competir com as duas principais empresas que estavam se estabelecendo neste novo mercado: a Apple com seu sistema operacional iOS e o Google com o Android (MÔNACO e DO CARMO, 2012).

Em 2010, a Microsoft realizou o lançamento de sua primeira plataforma mobile, o Windows Phone 7 – na época chamado de Windows Phone 7 Series – que buscava trazer novos recursos aos seus clientes, não apenas realizar um plágio dos sistemas que já estavam estabelecidos na época (MÔNACO e DO CARMO, 2012).

O Windows Phone veio com a proposta de não apenas solucionar os problemas dos usuários no seu dia a dia, mas também de levar em consideração a experiência do usuário ao utilizar a plataforma, oferecendo uma interface gráfica intuitiva, de fácil

utilização e de alto desempenho. A plataforma foi desenvolvida pensando no usuário profissional que necessita de celular para o seu dia a dia nos negócios e também no usuário comum que o utiliza para o lazer (MÔNACO e DO CARMO, 2012).

Atualmente a plataforma encontra-se na versão Windows 10 Mobile e vem acompanhado dos aplicativos do Office nativos, o armazenamento em nuvem do *OneDrive*, o seu próprio assistente pessoal, a Cortana, que é capaz fornecer ajuda ao usuário com instruções, lembretes, tarefas e outros recursos do dispositivo (MICROSOFT, 2017).

2.1.1.3 iOS

O sistema operacional iOS foi desenvolvido pela Apple e o sistema é utilizado por todos os dispositivos desenvolvidos pela empresa. A princípio, a empresa desenvolveu o sistema para iPhone, mas por conta de sua popularidade e de sua boa performance durante a execução, o iOS passou a ser o sistema padrão de todos os dispositivos que foram lançados por ela, entre eles pode-se citar o iPod, iPad e até mesmo a *Apple TV* (MILANI, 2012).

Uma característica interessante do iOS é que o sistema não pode ser executado por um dispositivo que não contenha o hardware específico da empresa, ou seja, somente os dispositivos que foram fabricados e desenvolvidos pela empresa é que são capazes de executar o sistema (MILANI, 2012).

Em fevereiro de 2007, durante o WWDC (*Apple Worldwide Developers Conference*), foi anunciado o primeiro iPhone. No primeiro ano da empresa, o iPhone por oferecer aos seus clientes uma tela *multitouch* com uma ótima sensibilidade ao toque, e também pelo fato de todos os seus recursos serem integrados, como GPS, internet e entre outros, o que na época foi uma evolução muito grande no mercado, aliás, a Apple (LECHETA, 2012).

Embora o iPhone tenha feito muito sucesso no seu lançamento, ele não permitia aos desenvolvedores a criação de aplicativos para a plataforma pois, o SDK responsável por proporcionar tal recurso aos desenvolvedores somente foi lançado um ano depois, em 2008, junto com a abertura da *App Store*, que é um serviço oferecido pela Apple para que o usuário possa realizar o download de aplicativos,

jogos, músicas e entre outros. E o lançamento do SDK impulsionou significativamente o desenvolvimento de aplicativos, pois os aplicativos poderiam ser instalados facilmente em qualquer aparelho e ainda proporcionar uma remuneração aos desenvolvedores (LECHETA, 2012).

Atualmente o iOS encontra-se na versão 11 e é considerado o sistema operacional mais avançado do mundo, trazendo, nesta última versão, novas possibilidades para realidade aumentada em aplicativos e jogos, execução de tarefas com facilidade e rapidez, um novo *Dock* e uma série de outros recursos (Apple, 2017).

2.1.2 Desenvolvimento Híbrido

Nesta seção serão apresentadas de forma breve algumas das principais alternativas no desenvolvimento de aplicativos híbridos que atualmente estão no mercado.

Aplicativos híbridos possuem como característica a possibilidade de executar o app em mais de uma plataforma utilizando apenas um código fonte para as plataformas para o qual ele será disponibilizado, fazendo com que seja exigido um menor tempo de trabalho, de investimento e pode-se desenvolvê-lo utilizando linguagens web (BRAVI, 2015).

Entretanto, eles não permitem acesso a todos os recursos do dispositivo, não oferecem a experiência de usuário que o aplicativo nativo oferece e não garante acesso aos recursos individuais de cada plataforma (BRAVI, 2015).

2.1.2.1 Apache Cordova

O Apache Cordova é um framework gratuito, de código aberto e que oferece aos desenvolvedores a possibilidade de desenvolver aplicativos multiplataformas utilizando HTML5, JAVASCRIPT e CSS sem a necessidade de escrever a aplicação na linguagem nativa da plataforma (APACHE CORDOVA, 2017).

O Apache Cordova possui como principal atrativo a capacidade de desenvolver o aplicativo uma única vez e permitir que o mesmo seja executado em diversas

plataformas, combinando os recursos nativos da plataforma e os recursos que o Apache Cordova tem a oferecer. As aplicações que são desenvolvidas utilizando este padrão de desenvolvimento são consideradas aplicações híbridas (WARGO, 2013).

Além de o Apache Cordova permitir desenvolver aplicativos utilizando apenas HTML, CSS e JAVASCRIPT, ele também oferece a possibilidade de acesso aos recursos que o dispositivo poderia oferecer caso fosse desenvolvido em linguagem nativa, como, por exemplo: acesso a lista de contatos, GPS, câmera, acelerômetro e entre outros (WARGO, 2013).

O Apache Cordova é capaz de executar a aplicação independente da plataforma, pois a aplicação desenvolvida é executada dentro de um componente nativo, chamado *webview*, que é responsável por renderizar o conteúdo do aplicativo na tela do usuário. Como o Apache Cordova permite a utilização de HTML, CSS e JAVASCRIPT, as lógicas da aplicação – navegação entre páginas, execução de vídeos, efeitos, estilos, conexão com banco e entre outros – podem ser aplicadas normalmente que o framework é capaz de tratá-las e renderiza-las ao usuário. (WARGO, 2013).

2.1.2.2 Outros

Além da utilização do Apache Cordova também existem outras ferramentas que possibilitam o desenvolvimento híbrido, como: Phonegap (PHONEGAP, 2017), Xamarin (XAMARIN, 2017) e entre outros. Entretanto, tais ferramentas não fazem parte do projeto, portanto, não serão descritas.

2.1.2.3 Tecnologias para desenvolvimento de aplicações móveis híbridas

Nesta seção serão apresentadas as tecnologias que farão parte do escopo do projeto e que serão aplicadas no desenvolvimento do aplicativo. As tecnologias aqui descritas serão utilizadas por oferecerem recursos e funcionalidades que serão necessários durante o projeto.

2.1.2.3.1 Ionic

O Ionic é um framework de código aberto que foi fundado pela empresa Drifty Co em 2012 que é utilizado para o desenvolvimento de aplicações móveis híbridas utilizando HTML, CSS e JAVASCRIPT (IONIC, 2017). O principal foco do framework é proporcionar um conjunto de componentes visuais que facilitem no desenvolvimento da interface gráfica da aplicação (DRIFTY, 2017).

Além de facilitar o desenvolvimento da interface gráfica a equipe por trás do projeto visa aperfeiçoar seus componentes, conforme Vasconcellos (2017): “A equipe do Ionic procura otimizar a performance de todos os componentes que eles criam, pois eles sabem que a fluidez e a responsividade de uma *app* híbrida são fatores chaves.”

A versão mais atual do Ionic exige que o mesmo seja desenvolvido em conjunto com Angular, pois assim o potencial máximo do framework pode ser atingido, permitindo com que diversas interações, gestos, animações e entre outros possam ser aplicados à sua aplicação híbrida (IONIC, 2017).

2.1.2.3.2 Angular 5

O Angular é um framework baseado em JAVASCRIPT que é utilizado para realizar o desenvolvimento de aplicações Web. Ele foi desenvolvido pelo Google e é mantido até os dias atuais. Em 2018 ele foi eleito o *framework front-end* mais popular entre os desenvolvedores por ser considerado mais rápido, leve e fácil de utilizar em relação aos seus concorrentes (RELEVANT SOFTWARE, 2018).

Ele possui um otimizador que é responsável por tornar a compilação do projeto mais leve do que as versões anteriores, ele este compilador é responsável por remover os códigos que são desnecessários durante a execução da aplicação, tornando-o mais leve e mais rápido (RELEVANT SOFTWARE, 2018).

Apesar de o framework ser baseado em JAVASCRIPT, a linguagem de programação utilizada para o desenvolvimento das aplicações é o *TypeScript*. O *TypeScript* é uma linguagem tipada que permite aos desenvolvedores identificar erros na aplicação de acordo com a escrita do código, algo que o *TypeScript* nativo

não consegue fazer. Posteriormente o código escrito em *TypeScript* será compilado para JAVASCRIPT e, para isso, se é utilizado a versão 2.3 do *TypeScript* tornando o processo de compilação mais rápido do que das versões anteriores (RELEVANT SOFTWARE, 2018).

2.2 SERVIDOR

Nesta seção serão apresentadas as tecnologias que serão utilizadas no lado do servidor da aplicação.

2.2.1 NodeJS

NodeJS é uma plataforma desenvolvida utilizando o JAVASCRIPT V8 – que é o motor a Google utiliza atualmente em seu navegador - e um laço de eventos escritos dentro do servidor. Ele foi apresentado por Ryan Dahl, no evento JSConf, em 2009. Ele surgiu com a proposta de facilitar o desenvolvimento de aplicações de rede rápidas e escaláveis que necessitavam transmitir uma grande quantidade de dados em tempo real (NODEBR, 2016).

Ao invés de criar uma nova *thread* a cada conexão que é realizada no servidor, ele é responsável por disparar um evento dentro da *engine* de processos do Node para cada conexão, o que impacta no desempenho do servidor, pois o modelo atual utilizado em linguagens de programação populares, como PHP ou Java, dispara uma *thread* que possui cerca de 2MB de memória anexadas à ela a cada conexão que é criada (NODEBR, 2016).

Portanto, o Node oferece o recurso de desenvolver programas de redes escaláveis que necessitarão trafegar uma grande quantidade de dados em tempo real (NODEBR, 2016).

2.2.2 Java

O Java é uma linguagem de programação que foi oficialmente lançado em maio de 1995, pela empresa *Sun Microsystems*. A Sun viu no Java a possibilidade de

adicionar conteúdos dinâmicos nas páginas web, como interatividade e animações, que eram desenvolvidas na época, o que fez com que a linguagem se destacasse com facilidade por conta da explosão da popularidade da Web que ocorreu no ano de 1993 (DEITEL e DEITEL, 2009).

Os programas que são desenvolvidos em Java consistem em partes chamadas de classes. Dentro de cada classe serão encontrados os métodos de uma determinada classe que realizam tarefas e retornam os resultados obtidos após a execução daquela tarefa, assim permitindo ao desenvolvedor cada parte necessária de seu sistema Java. Entretanto, o Java além de permitir que sejam criadas novas classes, também fornece ao desenvolvedor uma série de bibliotecas de classe Java que são conhecidas como Java APIs (*Application Programming Interfaces*) que realizam alguns tratamentos para o desenvolvedor sem fazer com que o mesmo tenha a necessidade de desenvolvê-las (DEITEL e DEITEL, 2009).

Atualmente o Java continua se destacando na Web, fornecendo aos desenvolvedores uma série de recursos para o desenvolvimento de aplicativos e sistemas. O mesmo também permite a execução de jogos, realização de uploads, realização de transações bancárias, a utilização de bate papo online, mapas interativos e diversos outros recursos presentes na web.

2.2.3 Java EE

O Java EE (*Java Platform Enterprise Edition*) é uma plataforma que possui um conjunto de bibliotecas e funcionalidades que possibilitam tornar a aplicação desenvolvida distribuída. O Java EE consegue preencher diversos requisitos de aplicações corporativas ou de grande porte, pois o mesmo baseia-se em componente modulares que executam em servidores de aplicações que são capazes de suportar escalabilidade, segurança, integridade e entre outras exigências da web (FARIA, 2015).

A plataforma Java EE, como dito anteriormente, possui uma série de bibliotecas e funcionalidades que possuem objetivos específicos que auxiliam no desenvolvimento da aplicação. Dentro de tais recursos, os mais comumente utilizados nos projeto são:

- *Servlets*: são componentes Java executados no servidor para gerar conteúdo dinâmico para a web, como HTML e XML.
- JSP (JavaServer Pages): JSP é uma especialização de Servlets que permite que o conteúdo da página possa ser gerado de forma dinâmica, similar às tecnologias ASP e PHP, porém mais robusta por ter todas as facilidades da plataforma Java.
- JSF (JavaServer Faces): é um framework web baseado em Java que tem como objetivo simplificar o desenvolvimento de interfaces (telas) de sistemas para a web, através de um modelo de componentes reutilizáveis.
- JPA (Java Persistence API): é uma API padrão do Java, com conceito objeto-relacional, que é utilizada para a realização de persistência de dados, que oferece alta produtividade em sistemas que necessitam de integração com um banco de dados.

Até o presente momento o Java está na versão 8 e pode ser baixado gratuitamente no site oficial da Oracle Corporation (JAVA, 2017).

2.2.4 Spring

Spring é um framework Java que foi originalmente escrito por Rod Johnson, em 2002, que visava facilitar o desenvolvimento de aplicações corporativas, diminuir a complexidade das mesmas e permitir o uso de *JavaBeans* para alcançar resultados que antes somente poderiam ser atingidos através de EJB (WALLS, 2014).

O framework oferece uma série de funcionalidades que facilitam no desenvolvimento de aplicações Java. Dentro de todos os recursos que o Spring oferece, pode-se destacar o fato do *framework* ser baseado em injeção de dependências e programação orientada a aspectos (WALLS, 2014).

O *framework* oferece a possibilidade de aplicar anotações especiais de segurança aos métodos que compõem o sistema, como: *Secured*, *PreAuthorize*, *PostAuthorize* e entre outros. Cada anotação utilizada possui uma finalidade, conforme apresentado na Tabela 1.

Tabela 1 - Anotações de segurança do Spring

Anotação	Descrição
<i>@Secured</i>	Recebe um <i>array</i> de <i>String</i> informando as permissões que são necessárias para a execução de um determinado método
<i>@PreAuthorize</i>	Restringe o acesso a um método antes da invocação com base no resultado da avaliação de uma expressão
<i>@PostAuthorize</i>	Permite que o método seja executado, entretanto gera uma exceção caso a expressão passada como parâmetro retorne falso.
<i>@PostFilter</i>	Permite a execução do método, entretanto, filtra o resultado baseado na expressão passada como parâmetro
<i>@PreFilter</i>	Permite a execução do método, entretanto, filtra a entrada antes de entrar no método

Fonte: Adaptado de WALLS, CRAIG. Spring in Action, 2014

Tais recursos do *framework* garantem um melhor controle no gerenciamento de permissões no sistema, garantindo que o usuário somente tenha acesso aos métodos disponibilizados a ele (WALLS, 2014).

O framework atualmente encontra-se na versão 5.0 e continua oferecendo diversas funcionalidades que auxiliam no desenvolvimento de sistemas web (SPRING, 2017).

2.3 PERSISTÊNCIA

Nesta seção serão apresentadas as tecnologias que serão utilizadas para o desenvolvimento da camada de persistência das informações que serão necessárias no projeto.

2.3.1 PostgreSQL

O SGBD PostgreSQL teve origem na Universidade Berkley, na Califórnia (EUA), em 1986, em um projeto chamado de POSTGRES. O professor Michael Stonebraker e sua equipe tiveram como objetivo o desenvolvimento de um sistema de armazenamento de dados, o projeto foi apoiado por diversos órgãos da época, entre eles está *Army Research Office* (ARO) e o *National Science Foundation* (NFS).

Em 1994, ocorreu a primeira grande mudança no projeto, pois a ferramenta tornou-se popular em seu ramo, assim o projeto foi encerrado, dando origem ao programa Postgres95 que trouxe a possibilidade de incorporação da linguagem SQL, pois anteriormente utilizava-se a linguagem PostQUEL (Milani, 2008).

Em 1996, o nome Postgres95 estava desatualizado, assim, a denominação da ferramenta foi novamente modificado, passando a se chamar da forma como se chama hoje: PostgreSQL (Milani, 2008).

O PostgreSQL não possui um limite para o tamanho da base de dados, as tabelas podem ter um tamanho de até 32 TB, capacidade de armazenar, por tabela, entre 250 e 1600 colunas por tabela, não possui um limite de indexes por tabela e, além de tudo, é uma ferramenta Open Source (PostgreSQL, 2017).

2.3.2 Java Persistence API

Java Persistence API (JPA) é uma API padrão do Java que mapeia suas informações utilizando o conceito de mapeamento objeto-relacional (*object-relational mapping, ORM*) que é uma técnica de mapeamento que fornece ao desenvolvedor a possibilidade de converter dados entre banco de dados relacionais e linguagens de programação que trabalham utilizando com orientação a objetos. O mapeamento que é realizado entre o banco de dados e o Java ocorre através de metadados que descreve qual é a relação entre os objetos que constam no sistema e o banco de dados (FARIA, 2015).

Apesar da implementação da ORM ser considerada um pouco mais complexa em relação aos *frameworks* que estão disponíveis na web, os benefícios que a mesma oferece em longo prazo são grandes. Ela facilita a realização de

manutenções no sistema, faz com que menos linhas de códigos sejam escritas para realizar uma determinada tarefa, o que facilita a compreensão do código e torna o sistema mais fácil de ser alterado e, o banco de dados que for modelado pode ser colocado em produção em outros bancos de dados, sem a necessidade de realizar qualquer alteração no código-fonte da aplicação (FARIA, 2015).

2.4 COMUNICAÇÃO

Nesta seção serão apresentadas as tecnologias que serão utilizadas para realizar a comunicação entre o aplicativo e o servidor.

2.4.1 Json

Json (*Javascript Object Notation*) é uma formatação leve que foi desenvolvida com a proposta de permitir com que diferentes sistemas possam trocar dados entre si independentes da plataforma, da arquitetura e até mesmo da linguagem de programação utilizada durante o desenvolvimento da aplicação. É uma formatação que é facilmente gerada e lida pelas máquinas, para os humanos é fácil de ler e escrever e é independente de qualquer outra linguagem de programação, usa convenções que são familiares entre diversas linguagens de programação, fazendo com que o Json seja o formato ideal para a troca de dados (JSON, 2017).

O Json é composto de duas estruturas, sendo a primeira chamada de: coleção de pares e valores que relacionando com linguagens de programação pode ser caracterizado como um objeto, *struct*, *hash table* e entre outros. E a segunda estrutura é considerada uma lista ordenada de valores que, vinculando com a programação, pode ser caracterizada como *array*, vetor, lista e entre outros (JSON, 2017).

2.4.2 Rest

Rest (*Representational State Transfer*) é uma abstração da arquitetura *World Wide Web* que consiste em um conjunto restrições arquiteturais aplicadas a

componentes e dados dentro de um sistema distribuído (RESTAPITUTORIAL, 2017).

Rest é baseado em recursos e realiza a sua comunicação com o banco de dados através de um arquivo Json ou XML que represente as informações que serão manipuladas dentro do banco. Cada comunicação que é realizada com o banco possui conteúdo o suficiente para descrever como a informação deve ser processada (RESTAPITUTORIAL, 2017).

Rest realiza as requisições utilizando o protocolo HTTP e tais requisições são compostas pelo *endpoint* que será consumido, o método/verbo que será processado e o corpo que deverá ser executado (Deering, 2017).

Os quatro tipos de verbos mais comuns que são utilizados ao realizar uma requisição são demonstrados na Tabela 2.

Tabela 2 - Verbos utilizados em serviços REST

Verbo	Descrição
Get	Utilizado para obter as informações
Post	Utilizado para adicionar informações
Put	Utilizado para atualizar o dado
Delete	Utilizado para excluir informações

Fonte: Do autor

2.4.3 QR CODE

O termo QR CODE que significa “Código de Resposta Rápida” é um código de barra quadrado que funciona como Códigos de Barra UPC avançado e que podem ser lidos por máquinas. Seu desenvolvimento ocorreu no Japão e foi primeiramente utilizado para a indústria automotiva, entretanto, hoje ele pode ser utilizado para armazenar informações, como: números de telefone, detalhes de produtos, mensagens de e-mail e entre outros (TRUSTTHISPRODUCT, 2017).

Há dois tipos de códigos QR CODE: dinâmicos e estáticos. Os códigos dinâmicos permitem que o conteúdo para o qual o usuário será redirecionado possa ser alterado sem a necessidade de interferir no código gerado. Já os códigos

estáticos armazenam dentro de si a informação em forma de texto – seja um texto ou um redirecionamento para um servidor -, ou seja, as informações não podem ser alteradas ou editadas, neste caso será necessário que um novo código seja gerado para a nova informação (TRUSTTHISPRODUCT, 2017).

Dita as tecnologias que serão utilizadas no projeto, agora será apresentada a metodologia do projeto.

3 METODOLOGIA

Este projeto possui como objetivo o desenvolvimento de uma aplicação híbrida que auxilie na realização de pedidos em Fast-foods genéricos, assim permitindo que o processo de atendimento ao cliente e de realização de pagamento possa se tornar mais ágil.

Antes de abordar o desenvolvimento da aplicação será realizado um levantamento de todos os conceitos que serão utilizados como base para o desenvolvimento da aplicação, explicando a origem, o funcionamento e o porquê de tal tecnologia ser a mais adequada.

Após o levantamento dos conceitos necessários para o desenvolvimento, um estudo sobre as linguagens de programação será iniciado para que se possa observar o funcionamento das mesmas para assim poder aplicá-las no desenvolvimento do projeto.

3.1 ESTUDO DE CASO

Nesta seção do projeto será apresentado um estudo de caso levando em consideração as características que atualmente são mais comuns na maioria dos Fast-foods.

3.2 GESTÃO DO NEGÓCIO

Para este projeto, foi levado em consideração que o estabelecimento não possui nenhum sistema computacional responsável pelo gerenciamento dos pedidos, sendo assim todo o processo de realização de atendimento é realizado através de papel e caneta.

3.3 FUNCIONAMENTO

O estabelecimento possui um cardápio físico onde constam os produtos que são oferecidos e a realização de pedidos ocorre com o cliente indo diretamente no caixa, após o cliente seguir uma fila física. Após a realização do pedido, lhe é entregue

uma nota contendo o código do pedido. Em seguida o cliente escolhe uma das mesas que estiverem disponíveis, se desejar, e aguarda até que o atendente informe que o produto solicitado já está disponível para retirada.

Por fim, o cliente irá até o caixa realizar a retirada do produto informando o código do seu pedido.

3.4 REQUISITOS FUNCIONAIS

- RF1. O aplicativo deve possuir uma listagem dos produtos com imagem;
- RF2. O aplicativo deve permitir a realização de Login na aplicação, entretanto, não é obrigatório;
- RF3. O aplicativo deve permitir que o cliente realize pedidos;
- RF4. Executar o aplicativo em mais de um idioma;
- RF5. Realizar pagamento pelo aplicativo;
- RF6. Acompanhar o pedido;
- RF7. Notificação sobre o acompanhamento do pedido;
- RF8. Avaliar o pedido;

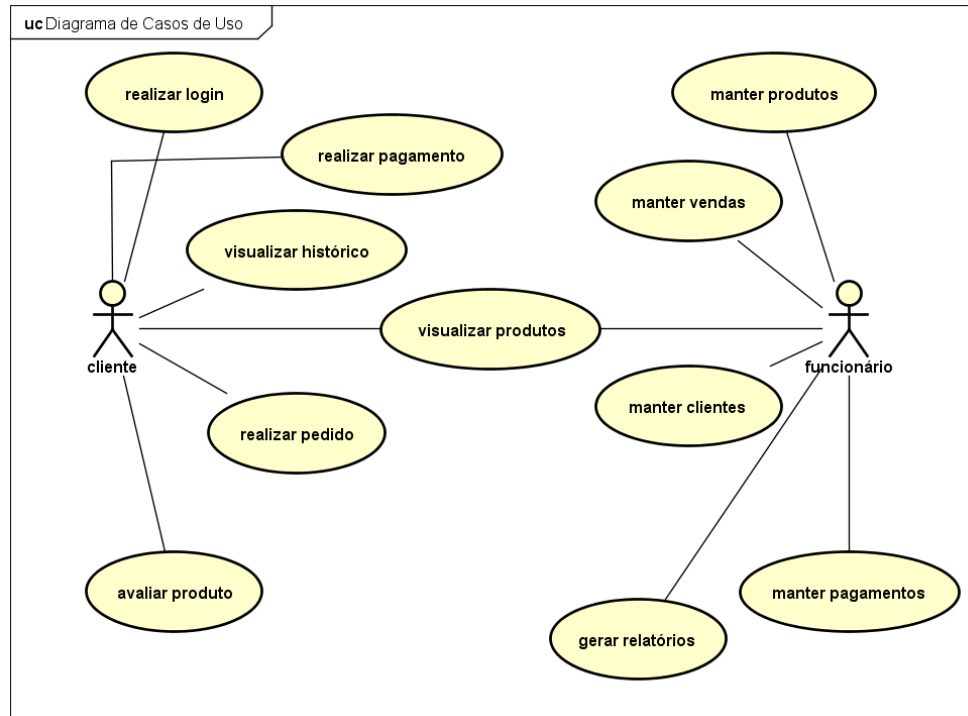
3.5 REQUISITOS NÃO FUNCIONAIS

- RNF1. O aplicativo deve ser compatível com o maior número de plataformas possíveis;
- RNF2. Possibilidade de realização de pedidos sem a necessidade de seguir uma fila física;

3.6 DIAGRAMA DE CASOS DE USO

O Diagrama de Casos de Uso tem como objetivo demonstrar o cenário que a aplicação funcionará e apresentar as funcionalidades que o sistema irá oferecer do ponto de vista do usuário. O diagrama apresentado na Figura 1 representa as funcionalidades que serão desenvolvidas.

Figura 1 - Casos de Uso



Fonte: Do autor

3.7 DESCRIÇÃO DO CASO DE USO

A descrição dos casos de uso tem como objetivo detalhar as funções do autor com os casos de uso.

3.7.1 Caso de uso realizar login

Tabela 3 - Caso de uso realizar login

Descrição do Caso de Uso	Realizar Login
Ator Principal	Cliente
Resumo	Este caso de uso visa descrever como é realizado o Login no aplicativo pelo cliente

Pré-Condição	O cliente deverá ter o aplicativo instalado no celular
Pós condição	Visualizar os produtos disponíveis
Fluxo Principal	
Ações do Ator	Ações do Sistema
Informar as credenciais de Login do Facebook/Google	
	2. Verifica se as credenciais estão corretas e retorna o resultado
	3. Carrega a lista de produtos
4. Visualiza a lista de produtos	
Fluxo Secundário	
Ações do Ator	Ações do Sistema
Seleciona a opção de realizar login anônimo	
	Carrega a lista de produtos
Visualiza a lista de produtos	

Fonte: Do autor

3.7.2 Caso de uso realizar pagamento

Tabela 4 - Caso de uso realizar pagamento

Descrição do Caso de Uso	Realizar Pagamento
Ator Principal	Cliente
Resumo	Este caso de uso visa descrever como é realizado o Pagamento pelo cliente
Pré-Condição	O cliente deverá ter finalizado um pedido
Pós condição	
Fluxo Principal	
Ações do Ator	Ações do Sistema
Cliente escolhe a forma de pagamento e informa dados do pagamento	
	Conecta-se ao sistema responsável pelo pagamento, tenta realizar a operação e retorna o resultado da operação
Cliente verifica se o pagamento foi realizado	
Fluxo Secundário	
Ações do Ator	Ações do Sistema

Fonte: Do autor

3.7.3 Caso de uso realizar pedido

Tabela 5 - Caso de uso realizar pedido

Descrição do Caso de Uso	Realizar Pedido
Ator Principal	Cliente
Resumo	Este caso de uso visa descrever como ocorre a realização de pedidos
Pré-Condição	O cliente deverá ter realizado login
Pós-condição	
Fluxo Principal	
Ações do Ator	Ações do Sistema
Cliente realiza leitura de QRCode do estabelecimento através de seu dispositivo móvel	
	O sistema carrega a listagem de produtos de acordo com o estabelecimento
Cliente adiciona o produto desejado ao carrinho	
	Sistema irá adicionar o produto ao carrinho
Visualiza os produtos que estão no carrinho e finaliza o pedido	
	Salva o pedido no sistema
Fluxo Secundário	

Ações do Ator	Ações do Sistema

Fonte: Do autor

3.7.4 Caso de uso gerar relatório

Tabela 6 - Caso de uso gerar relatório

Descrição do Caso de Uso	Gerar Relatório
Ator Principal	Funcionário
Resumo	Este caso de uso visa descrever como ocorre a realização de relatório
Pré-Condição	O funcionário deverá ter realizado login no sistema e deverá ter pelo menos um pedido registrado no sistema
Pós condição	
Fluxo Principal	
Ações do Ator	Ações do Sistema
O funcionário seleciona o produto, mês/ano e/ou cliente desejado	
	2. O sistema mostra os pedidos que foram realizados referentes ao filtro aplicado
3. Cliente gera o relatório	
	4. Mostra o relatório que foi gerado
Fluxo Secundário	
Ações do Ator	Ações do Sistema

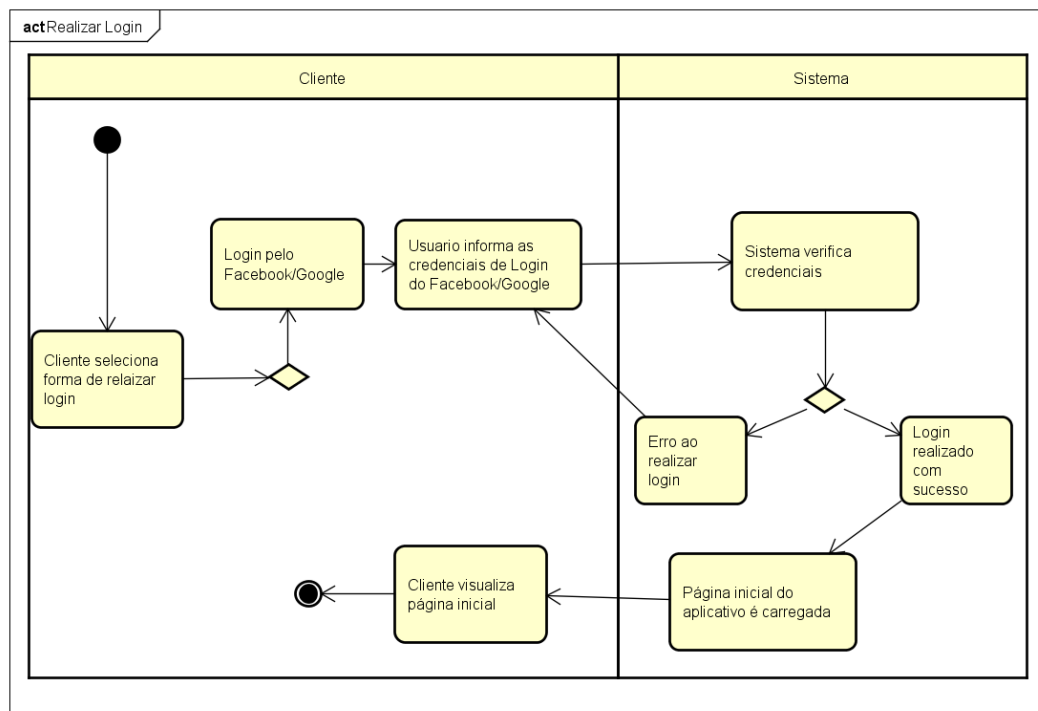
Fonte: Do autor

3.8 DIAGRAMA DE ATIVIDADES

Os diagramas de atividades são responsáveis por demonstrar as etapas necessárias de um determinado processo para que ele atinja o objetivo desejado.

O diagrama a seguir na Figura 2 demonstrará como funcionará a realização de login no sistema.

Figura 2 - Diagrama de atividade realizar login



Fonte: Do autor

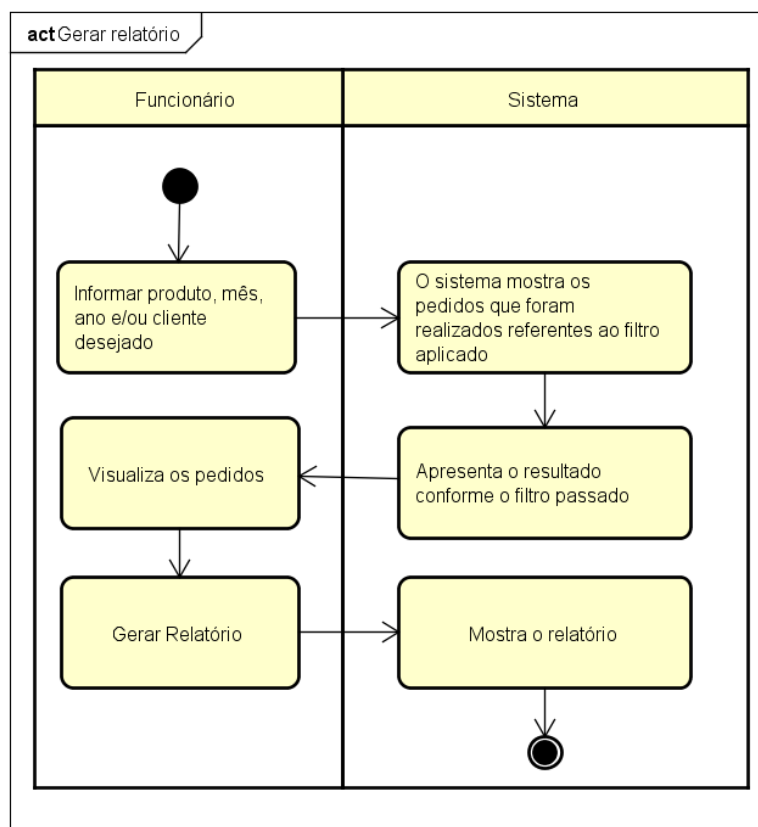
O diagrama demonstra que o cliente irá visualizar uma tela solicitando que seja realizado o *Login*. Serão oferecidas duas formas de realização de *Login*, utilizando uma conta do Google ou uma conta do Facebook.

Após selecionar a forma que ele de realizar o *Login*, o cliente deve informar as suas credenciais da conta para que o sistema verifique se elas coincidem. Caso ocorra algum problema durante o *Login*, a aplicação irá retornar uma mensagem de erro informando que houve uma falha na realização do *Login*.

Caso o *Login* seja realizado com sucesso, o sistema carrega a página inicial da aplicação.

O próximo diagrama irá demonstrar o que ocorre quando o funcionário solicitar que os relatórios sejam gerados. Demonstrado na Figura 3.

Figura 3 - Diagrama de atividade gerar relatório

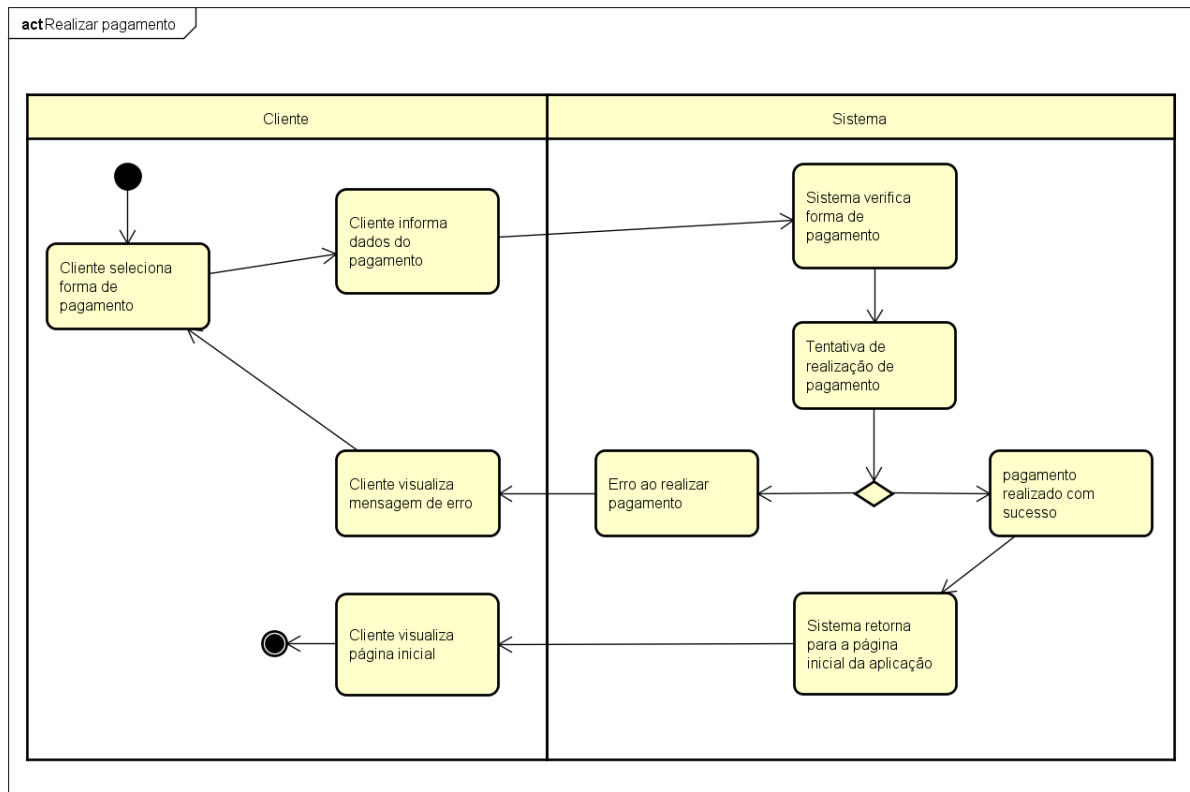


Fonte: Do autor

O diagrama demonstra que o funcionário irá estabelecer alguns parâmetros para serem aplicados no relatório que será gerado. Após isto, o sistema realizará uma busca em sua base de dados e retornará para o funcionário a lista das informações que foram obtidas na pesquisa. Após o funcionário observar as informações recebidas, ele poderá gerar o relatório.

O diagrama ilustrado na Figura 4 visa demonstrar como ocorre a realização de pagamento no estabelecimento.

Figura 4 - Diagrama de atividade realizar pagamento

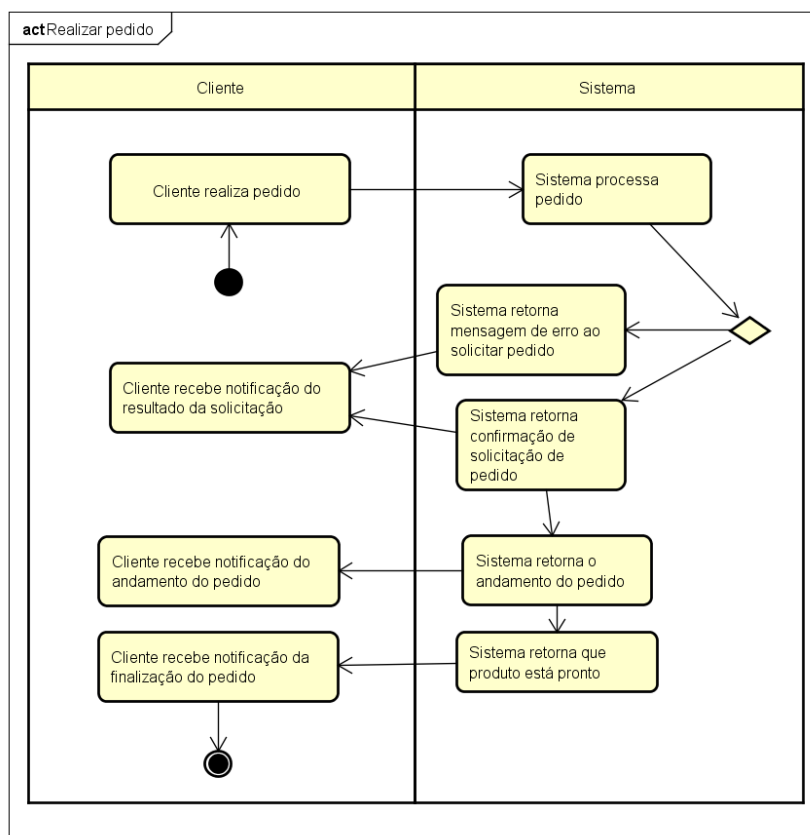


Fonte: Do autor

Após selecionar os produtos desejados o cliente deve informar os dados do cartão que ele utilizará para que elas possam ser validadas e a realização do pagamento possa ser efetuada. O sistema retornará uma mensagem para o aplicativo informando se o pagamento foi realizado com sucesso ou se ocorreu algum erro durante o procedimento.

O diagrama ilustrado na Figura 5 demonstra como ocorrerá a realização de pedido no estabelecimento.

Figura 5 - Diagrama de atividade realizar pedido



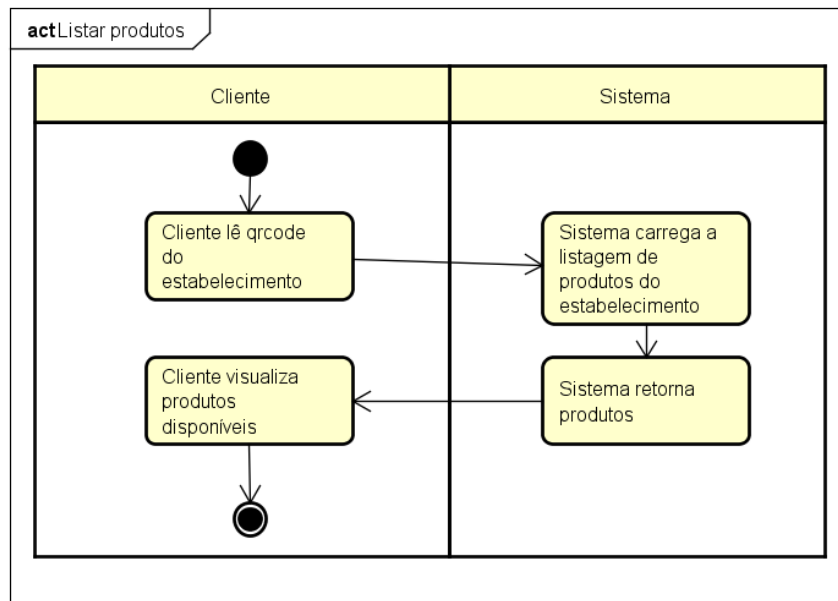
Fonte: Do autor

O cliente irá selecionar os produtos desejados e irá realizar o pedido. O sistema irá processar o pedido solicitado e retornará uma mensagem confirmando a realização do pedido ou informando que houve algum erro durante o a execução. Caso o pedido seja executado com sucesso o sistema irá informar o cliente através de notificações o andamento do pedido.

Quando o pedido for finalizado, o cliente receberá uma mensagem informando que o pedido está disponível para retirada, assim o cliente apenas deve ir até o balcão e informar o número do pedido para que o receba. Após isto, o sistema irá finalizar o pedido e enviará uma notificação ao cliente de que o pedido foi finalizado.

O diagrama ilustrado na Figura 6 demonstra como ocorrerá a listagem de pedidos do estabelecimento.

Figura 6 - Diagrama de atividade listar produtos



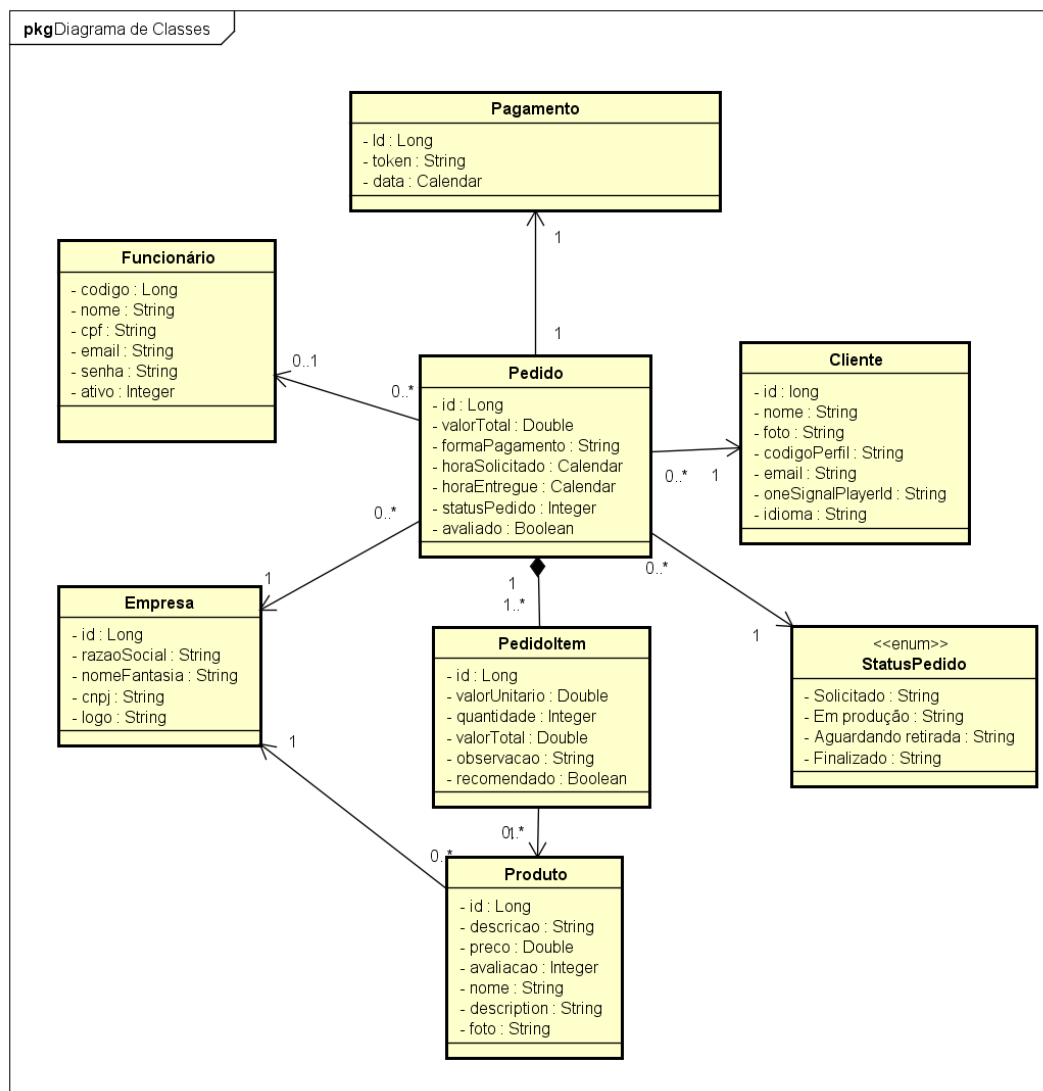
Fonte: Do autor

O Cliente irá acessar o aplicativo e assim que acessá-lo o sistema irá carregar a listagem de produtos disponíveis e retornará tal lista para que o cliente possa visualizá-la.

3.9 DIAGRAMA DE CLASSES

A Figura 7 apresenta o diagrama de classes que será utilizado no desenvolvimento do aplicativo.

Figura 7 - Diagrama de classes



Fonte: Do autor

A classe “Funcionário” será utilizada para armazenar os dados do funcionário que realizará *Login* no sistema.

A classe “Cliente” armazenará as informações do cliente que realizará pedidos no estabelecimento. O cliente não possuirá um usuário e senha, pois seu *Login* será realizado utilizando uma API de *Login* nativa da rede desejada, no caso, do Google ou do Facebook.

As classes “Pedido” e “PedidoItem” serão responsáveis por armazenar os pedidos que foram realizados pelo cliente.

A classe “Pagamento” é responsável por armazenar o *token* de pagamento que se é gerado quando opta-se por realizar os pagamentos através de um *gateway* de

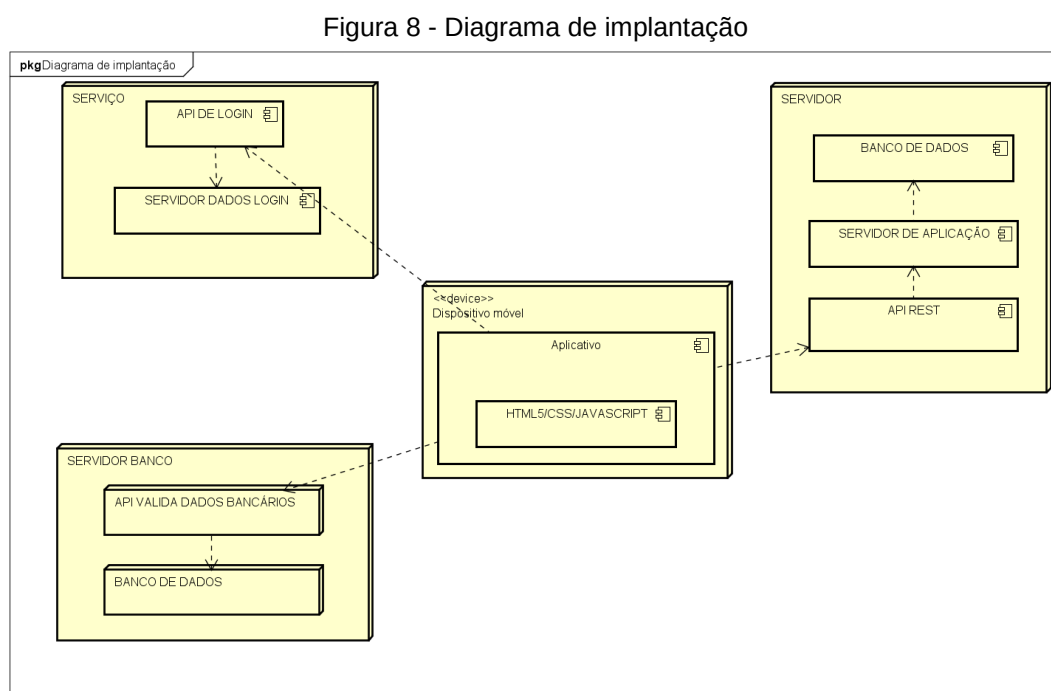
pagamento, no caso, o *gateway* de pagamento irá gerar um *token* que será utilizado para referenciar aquela compra em específico.

As classes “Produto” e “Empresa” armazenam os produtos que estão disponíveis em um determinado estabelecimento.

O Enum “StatusPedido” é responsável por armazenar o status do pedido.

3.10 DIAGRAMA DE IMPLANTAÇÃO

Na UML, os diagramas de implantação possuem como objetivo realizar a modelagem da arquitetura física do sistema, demonstrando como ocorre o relacionamento entre os componentes de software e hardware no sistema e a distribuição física do processamento. A Figura 8 demonstra como ocorrerá tal relacionamento no sistema.



Fonte: Do autor

A figura acima demonstra que o aplicativo será executado no dispositivo móvel do cliente onde tal aplicação estará sendo executada basicamente utilizando HTML, CSS e JAVASCRIPT. A aplicação realizará conexões HTTP com a API REST que será responsável por realizar a comunicação entre o servidor da aplicação e o banco de dados do sistema.

Além de conectar-se com o serviço onde estarão armazenadas as informações da aplicação, o aplicativo também se conectará com uma API de *Login* que será responsável por realizar a validação de *Login* nos servidores do Facebook e do Google.

4 DESENVOLVIMENTO DO PROTÓTIPO DA APLICAÇÃO

Esta seção descreve o processo de desenvolvimento do projeto, abordando os aspectos mais significativos e relevantes para a conclusão, como: sua arquitetura, a preparação do ambiente de desenvolvimento tanto no *frontend* quanto no *backend* e os componentes mais importantes da aplicação, explicando sua estrutura, funcionamento e configuração. Também optou-se pela explicação detalhada do processo, como uma forma de exercitar a atividade de documentação do sistema.

4.1 PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

O aplicativo foi projetado a partir de uma arquitetura composta por um servidor backend para o armazenamento das informações, do aplicativo móvel utilizado pelo usuário e de um painel administrativo para visualização de pedidos. Tais funcionalidades serão abordadas nos tópicos seguintes.

4.1.1 Backend do serviço

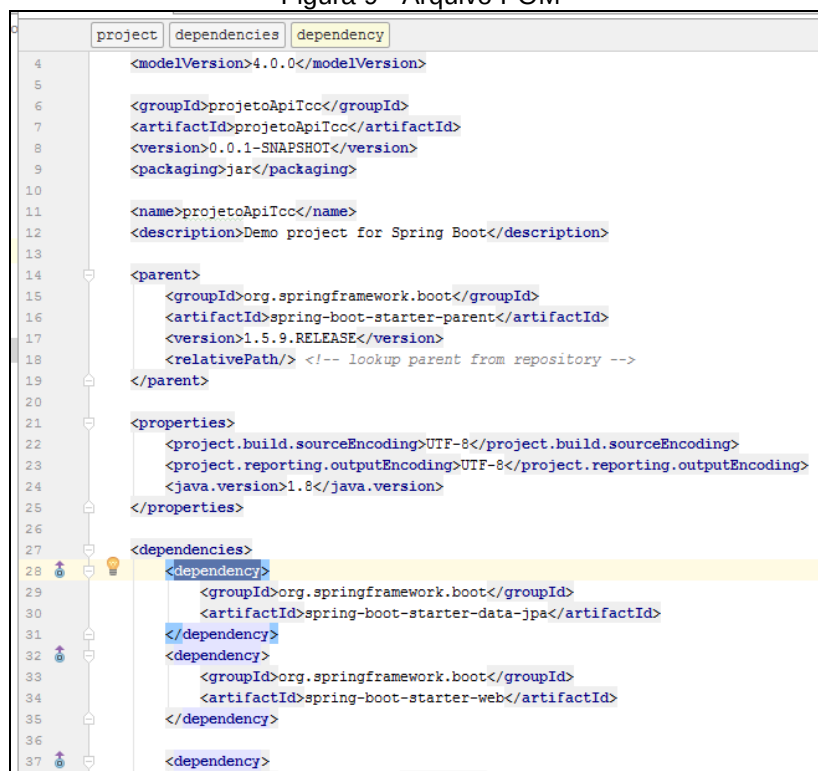
Para o desenvolvimento do *backend* do serviço, optou-se pela utilização da IDE IntelliJ IDEA 2017.1.1 por conta da familiaridade com a IDE. O SGBD utilizado para o armazenamento das informações foi o PostgreSQL versão 9.4, utilizando como auxílio a ferramenta PGAdmin III para verificação e validação das persistências que foram realizadas na aplicação e das estruturas que estavam sendo geradas no banco. O *Hibernate* 5.0.12 que utiliza a versão 2.1 da JPA foi adicionado ao projeto em conjunto com a versão 5.3.6 do *Hibernate Validator* que aplica a especificação da *Bean Validation* API para realizar a validação das informações que estão sendo enviadas pelo usuário no lado do servidor. Os serviços Rest que foram desenvolvidos no projeto foram criados a partir do framework *Spring Boot* 5.9 e para a comunicação com o banco foi utilizado o driver JDBC do PostgreSQL.

O IntelliJ permite a criação e configuração de projetos como qualquer outra IDE, entretanto, para um melhor gerenciamento do projeto, optou-se pela criação de um

projeto do tipo Maven através do site *Spring Initializr*, pois ele permite ao desenvolvedor criar um projeto já possuindo algumas das bibliotecas que serão utilizadas na aplicação, como bibliotecas para persistência de dados, comunicação com o banco, serialização de objetos e entre outros.

Por se tratar de um projeto do tipo Maven, ele possui um arquivo XML de configuração chamado POM que é responsável por armazenar e gerenciar as informações e as dependências que o projeto possui – bibliotecas e plugins –, conforme apresentado na Figura 9.

Figura 9 - Arquivo POM



Fonte: Do autor

O projeto gerado possui dois diretórios principais e um arquivo XML de configuração que podem ser considerados os mais importantes.

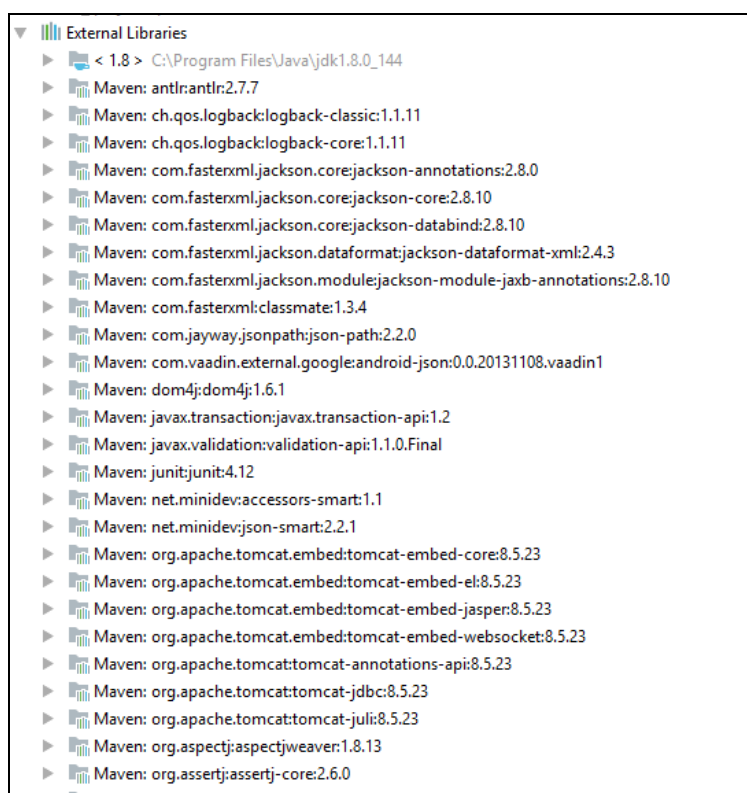
O diretório *src* é o diretório onde ficará todo o código fonte da aplicação, suas classes, interfaces, entidades, relacionamentos, comunicação com o banco e etc.

O diretório *target* é responsável por armazenar o contexto ao qual a aplicação está sendo executada. Ele realiza uma cópia de todo o diretório *src* e realiza o carregamento da aplicação conforme o que foi especificado no diretório *src*.

O arquivo POM é responsável por armazenar e gerenciar as informações do projeto, ele é composto por *tags* que são utilizadas para definir a versão do modelo de POM que está sendo utilizado. O *groupid* seria algo como o prefixo utilizado para estrutura dos pacotes no projeto. A *tag artifactId* define qual será o nome do artefato final que será gerado após o empacotamento. O *version* é utilizado para definir a versão do projeto que irá complementar no nome do pacote. O *packing* é responsável por definir qual será o tipo de empacotamento que terá após todo o processo de build do projeto. O *name* é utilizado para definir o nome do projeto. O *description* permite adicionar uma descrição ao projeto. O armazena todas as dependências que o projeto possui para execução e o *dependency* informa uma dependência que o projeto possui.

O resultado gerado a partir do que foi definido no arquivo POM do projeto pode ser observado na Figura 10.

Figura 10 - Bibliotecas do Projeto

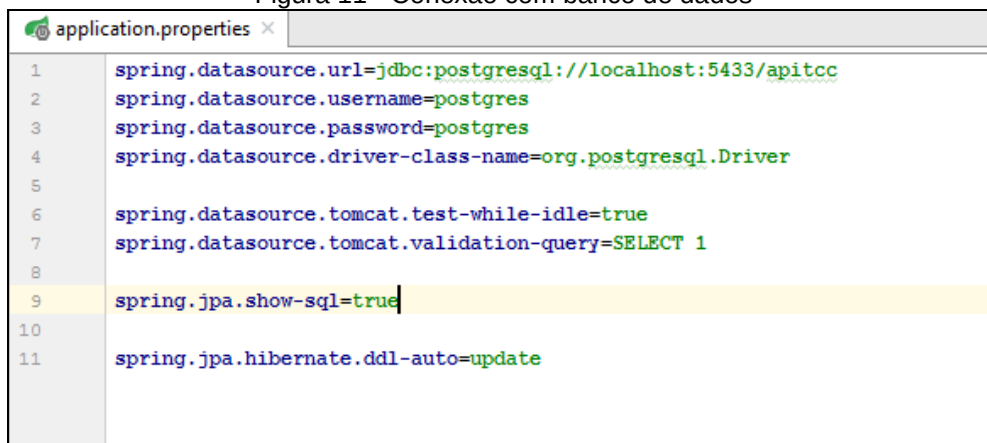


Fonte: Do autor

Dentro do diretório *resources*, que está dentro do diretório *src*, foi criado um arquivo de configuração que é responsável por definir onde está localizado o banco

de dados que será utilizado no projeto e qual deve ser o comportamento da aplicação em relação a este banco, conforme apresentado na Figura 11.

Figura 11 - Conexão com banco de dados



```
application.properties x
1  spring.datasource.url=jdbc:postgresql://localhost:5433/apitcc
2  spring.datasource.username=postgres
3  spring.datasource.password=postgres
4  spring.datasource.driver-class-name=org.postgresql.Driver
5
6  spring.datasource.tomcat.test-while-idle=true
7  spring.datasource.tomcat.validation-query=SELECT 1
8
9  spring.jpa.show-sql=true
10
11  spring.jpa.hibernate.ddl-auto=update
```

Fonte: Do autor

Pode-se observar que o arquivo acima está informando para a aplicação onde está localizado o banco de dados através do atributo `url`, informando posteriormente o usuário através do atributo `username`, a senha pelo `password` e o driver que deve ser utilizado para realizar a comunicação com o banco com o `driver-class-name`. Em seguida há a linha `spring.jpa.show-sql` da configuração da JPA que define que todas as queries que forem executadas dentro da aplicação devem aparecer na saída da IDE para fins de debug. E por último o `ddl-auto` que define qual a operação que deve ser realizada sempre quando o serviço for executado, no caso, com o `update`, será realizado um `update` nas tabelas e em seus atributos sem perder as informações que lá estão armazenadas.

Depois que a aplicação é executada pelo menos uma vez, ele verifica se as informações que foram definidas nos arquivos de configuração são válidas, caso elas sejam válidas, ele realizará a comunicação com o banco de dados que foi informado no arquivo.

4.1.2 Aplicativo Móvel

Após a configuração básica do *backend* da aplicação, iniciou-se a configuração que seria utilizada para o desenvolvimento da aplicação no front-end.

Para o desenvolvimento do *frontend* da aplicação optou-se por utilizar o Ionic em sua versão 3, Angular na versão 5 e Node JS, versão 8.9.3. Vale salientar que o Node foi utilizado apenas como servidor para a execução da aplicação e como gerenciador de dependências de pacotes que o projeto possui.

Para o desenvolvimento de aplicações híbridas optou-se por utilizar um gerenciador de pacotes que, basicamente, fica responsável por armazenar e gerenciar as dependências que o projeto possui para o seu correto funcionamento.

Um dos gerenciadores que mais tem sido utilizado recentemente em conjunto com o Node é o NPM. O NPM utiliza como referência um arquivo *package.json* que é responsável por armazenar as dependências de pacotes que o projeto possui para o seu funcionamento.

Cada pacote é tratado como um módulo do sistema que é utilizado em alguma funcionalidade do conjunto Ionic e tais módulos ficam armazenados em um diretório chamado *node_modules*.

Para realizar a instalação e configuração do Ionic e de suas funcionalidades, utilizou-se como referência a documentação oficial do *framework*, que pode ser acessada através do site oficial. Após realizar a instalação e a configuração do Ionic seguindo a documentação, foi gerado o arquivo responsável por armazenar as dependências que o projeto possui, conforme apresentado na Figura 12.

Figura 12 - Dependências da aplicação

```

1  {
2    "name": "aplicacaoMenuLateral",
3    "version": "0.0.1",
4    "author": "Ionic Framework",
5    "homepage": "http://ionicframework.com/",
6    "private": true,
7    "scripts": {
8      "clean": "ionic-app-scripts clean",
9      "build": "ionic-app-scripts build",
10     "lint": "ionic-app-scripts lint",
11     "ionic:build": "ionic-app-scripts build",
12     "ionic:serve": "ionic-app-scripts serve"
13   },
14   "dependencies": {
15     "@angular/common": "5.0.0",
16     "@angular/compiler": "5.0.0",
17     "@angular/compiler-cli": "5.0.0",
18     "@angular/core": "5.0.0",
19     "@angular/forms": "5.0.0",
20     "@angular/http": "5.0.0",
21     "@angular/platform-browser": "5.0.0",
22     "@angular/platform-browser-dynamic": "5.0.0",
23     "@ionic-native/barcode-scanner": "^4.7.0",
24     "@ionic-native/core": "4.3.2",
25     "@ionic-native/facebook": "^4.5.3",
26     "@ionic-native/google-plus": "^4.6.0",
27     "@ionic-native/local-notifications": "^4.6.0",
28     "@ionic-native/onesignal": "^4.6.0",
29     "@ionic-native/splash-screen": "4.3.2",
30     "@ionic-native/status-bar": "4.3.2",
31     "@ionic/storage": "^2.1.3",
32     "@ngx-translate/core": "^9.1.1",
33     "@ngx-translate/http-loader": "^3.0.1",

```

Fonte: Do autor

4.1.3 Painel administrativo

Para o desenvolvimento do painel administrativo da aplicação foi utilizado o editor de texto Visual Studio Code, versão 1.21.3, o NodeJS versão 8.9.3 como servidor e o framework Angular em sua versão 5.

O processo de configuração do Angular 5 ocorreu seguindo a documentação oficial do framework que, basicamente, funciona da mesma forma que quando utilizado em conjunto com o Ionic. Ele possui um arquivo JSON que é responsável por definir as dependências do projeto e utiliza o gerenciador de pacotes NPM para gerenciá-las.

Os pacotes que compõem as dependências do projeto são armazenados dentro do diretório *node_modules*, que é responsável por armazenar todas as dependências que o Angular 5 possui pra o seu funcionamento básico.

4.2 ARQUITETURA DO SISTEMA

O projeto desenvolvido possui dois componentes que são essenciais para o seu funcionamento. Um deles é responsável por apresentar através dos componentes do Ionic as funcionalidades que a aplicação tem a oferecer para o usuário. O outro componente é responsável por realizar a persistências das informações que estão sendo obtidas na aplicação móvel.

A aplicação móvel armazena em objetos todas as informações que são necessárias para a persistência da informação no banco e as envia para um *RestFull* webservice que realiza a comunicação e a persistência de tais informações no banco de dados.

Para obter um melhor gerenciamento de cada funcionalidade do sistema, ambos os componentes do sistema foram divididos em camadas. No lado do servidor, foram desenvolvidas quatro camadas: *controller*, *model*, *repository* e útil.

A camada *model* é responsável pelo mapeamento objeto-relacional de cada entidade na aplicação, definindo quais são os seus atributos e como o ocorrerá a relação com os demais modelos que compõem o sistema. As definições que forem moldadas nesta camada posteriormente se tornarão tabelas, relacionamentos e atributos no banco de dados.

A camada *repository* é responsável por tratar as persistências que ocorrerão na aplicação. Ela define as consultas e as operações que serão realizadas no banco de dados tendo como referência o *model* que está sendo utilizado como referência para cada repositório criado.

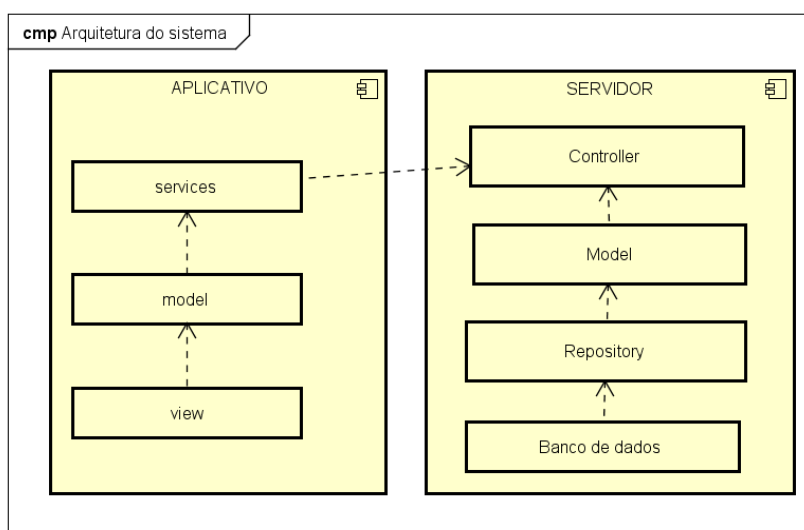
A camada *controller* disponibilizará os serviços REST que ficarão responsáveis por realizar a comunicação entre a aplicação e o banco de dados.

No lado da aplicação foram definidas três camadas principais: *model*, *services* e *views*. A camada *model* da aplicação é responsável por armazenar classes que possuam os mesmos atributos que foram definidos no lado do servidor, isto para que a persistência da informação seja realizada de forma correta. A camada *services* é responsável por realizar requisições através do protocolo HTTP a um determinado URI no serviço REST que está sendo disponibilizado ao usuário. A camada *view* é composta pelos componentes que farão parte da interface gráfica que será disponibilizada para o usuário interagir. Ela será composta pelos componentes que o

Ionic oferece, por arquivos HTML, CSS, JAVASCRIPT e de imagens. Uma das propostas iniciais do projeto era realizar a internacionalização da aplicação, permitindo ao usuário definir em qual idioma a aplicação deverá ser executada. As configurações de tal recurso também serão aplicadas nesta camada.

A Figura 13 ilustra como ocorrerá o funcionamento da aplicação e a relação de um componente com o outro.

Figura 13 - Arquitetura do Sistema



Fonte: Do autor

4.3 SERVIDOR WEB

Nesta seção serão explicadas as funcionalidades que são consideradas mais relevantes para o funcionamento sistema, visando detalhar aquilo que julgar indispensável para uma melhor compreensão do projeto como um todo.

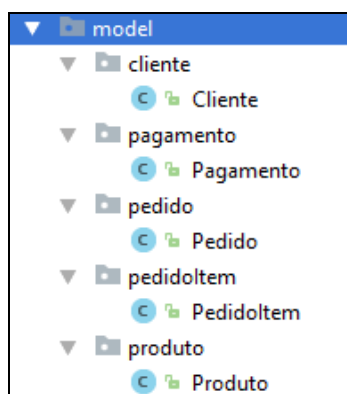
4.3.1 Camada Model

A camada *model* que foi desenvolvida na aplicação visa realizar o mapeamento objeto-relacional que será utilizado como referência para realizar o envio dos dados ao banco de dados. Este mapeamento é utilizado para definir os objetos que posteriormente serão convertidos em tabelas no banco de dados, definindo sua estrutura, suas relações, seu comportamento, seus atributos e entre outros.

Vale salientar que todas as classes aqui criadas implementam o padrão *JAVABEAN* que determina algumas boas práticas de programação que são aceitas e permitidas a nível mundial, assim, todas as classes que foram criadas no projeto implementam a interface *Serializable*, que é necessária caso deseja-se realizar a transferência da entidade em uma comunicação cliente X servidor. Todos os atributos devem possuir métodos públicos chamados de *Getters* e *Setter* para realizar manipulação das informações destas propriedades e que todas as classes devem possuir um construtor vazio para realizar a sua instância.

As classes essenciais para o funcionamento da aplicação que foram definidas após realizar uma análise são: cliente, pagamento, pedido, pedidoitem e produto, conforme apresentado na Figura 14.

Figura 14 - Camada modelo



Fonte: Do autor

A classe *Cliente* é responsável por armazenar informações sobre o cliente que está utilizando o sistema, lembrando que o *Login* e o registro na aplicação por parte do cliente ocorrerá através de redes sociais – *Google Plus* e *Facebook* –, armazenando suas informações básicas no banco.

Uma das propostas iniciais do projeto foi a possibilidade de notificar o usuário sobre o andamento do pedido de acordo com a mudança do status. Tal funcionalidade foi implementada utilizando o serviço *OneSignal*, pois ela permite ao administrador enviar notificações para os dispositivos cadastrados através de um *UUID (Unique Universal Identifier)* – chamado de *PlayerId* - que é gerado pelo serviço com base em informações do dispositivo. Assim, além das informações

básicas do usuário, este valor gerado também é armazenado no banco de dados para que notificações possam ser enviadas ao cliente.

Na Figura 15 é apresentada parte do código da classe *Cliente* que foi criada para representar as informações do cliente que serão armazenadas a partir das informações vindas da rede social utilizada para *Login*. Nesta classe também constará as anotações da JPA que realizaram o mapeamento objeto-relacional da classe.

Figura 15 - Classe Cliente

```
/**
 * Created by Crisley Alves on 07, dez, 2017
 */
@Entity
@Table(name = "clientes")
public class Cliente implements Serializable{

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "CODIGOPERFIL", nullable = false)
    private String codigoPerfil;

    @Column(name = "NOME", nullable = false)
    private String nome;

    @Column(name = "EMAIL", nullable = false)
    @Email
    private String email;

    @Column(name = "FOTO", nullable = false)
    private String foto;

    @Column(name = "ONESIGNALPLAYERID", nullable = false)
    private String oneSignalPlayerId;

    @Column(name = "IDIOMA", nullable = false)
    private String idioma;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "cliente", fetch = FetchType.LAZY)
    @JsonManagedReference
    private List<Pedido> pedidoList;
```

Fonte: Do autor

As anotações que são da JPA e do *Hibernate Validator* são identificadas pelo caractere “@” seguido da funcionalidade que deverá ser aplicada a aquele elemento da classe. A anotação *@Entity* é responsável por determinar que esta classe se torne uma tabela no banco de dados. A anotação *@Table* permite especificar os detalhes da tabela que será utilizada ao criar a entidade no banco. Com o parâmetro

name="clientes" está sendo definido que o nome da tabela que será gerado no banco de dados se chamará clientes.

O *@ID* define qual será o atributo da classe cliente que será utilizado como chave primária, no caso, será o atributo *id*. A anotação *@Column* é responsável por definir valores e regras que devem ser atribuídas a esta coluna que será gerada a partir da JPA, no caso, passando como parâmetro o *name = "ID"* e o *nullable = false* está sendo definido que o nome deste campo no banco de dados deverá possuir o nome ID e que o campo não pode ser nulo ao realizar uma persistência no banco.

A anotação *@GeneratedValue* é utilizada para informar como ocorrerá a geração do valor único da entidade ao realizar uma persistência no banco de dados.

O parâmetro *strategy = GenerationType.AUTO* é responsável por deixar com o provedor de persistência qual a estratégia que deve ser aplicada ao gerar os valores primários desta classe.

A anotação *@OneToMany* é responsável por definir a relação de um para muitos. Neste caso, define-se que há uma lista do objeto Pedido, onde um cliente possui vários pedidos, mas um pedido deve possuir á apenas um cliente. O parâmetro *cascadeType.ALL* define que quando ocorrer alguma operação no banco de dados que envolva a classe Cliente, as modificações também deverão ser aplicadas à classe Pedido. O parâmetro *mappedBy* é responsável por definir qual será o atributo da classe Pedido que será utilizado para realizar a relação entre as classes.

O parâmetro *FetchType* é responsável por definir como deverá ocorrer o carregamento da lista de pedidos do usuário, na imagem o *FetchType.LAZY* define que a listagem dos pedidos somente deve ser carregada quando ela for requisitada.

A anotação *@JsonManagedReference* é responsável por definir que as informações contidas na listagem de pedidos deve ser serializada normalmente. Na classe Pedido no campo onde estabelece-se a relação com o cliente, deve-se informar a anotação *@JsonBackReference*, para que não ocorra a serialização daquela entidade. Estas anotações são necessárias, pois a aplicação entrará em um loop infinito onde o objeto Cliente solicitará o objeto Pedido e o objeto Pedido solicitará o objeto Cliente dentro dele e assim por diante, então, esta anotação bloqueia este loop recursivo.

Na Figura 16 são apresentados os elementos que compõem a classe Pedido que vai ser persistida no banco de dados.

Figura 16 - Classe Pedido

```

@Entity
@Table(name = "pedidos")
public class Pedido implements Serializable{

    @Id
    @SequenceGenerator(name = "seq_pedido", sequenceName = "seq_pedido_id", allocationSize = 1)
    @GeneratedValue(generator = "seq_pedido", strategy = GenerationType.SEQUENCE)
    private Long id;

    @Column(name = "horaSolicitado", updatable = false)
    @Temporal(TemporalType.TIMESTAMP)
    @NotNull
    private Calendar horaSolicitado;

    @Column(name = "horaEntregue")
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar horaEntregue;

    @Column(name = "formaPagamento")
    private String formaPagamento;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "pedido_id", nullable = false)
    private List<PedidoItem> produtoItemList;

    @ManyToOne(fetch = FetchType.LAZY)
    @JsonBackReference
    private Cliente cliente;

    @OneToOne(fetch = FetchType.LAZY, mappedBy = "pedido", cascade = CascadeType.ALL)
    @JoinColumn(name = "pagamento", updatable = false)
    @JsonManagedReference
    private Pagamento pagamento;

```

Fonte: Do autor

A anotação `@SequenceGenerator` é utilizada para definir a uma sequencia que será utilizada em uma ou mais classes da aplicação. No caso, foram utilizados os parâmetros `name = "seq_pedido"` para definir um nome para esta sequencia que está sendo definida, o `sequenceName = "seq_pedido_id"` define o nome da sequencia que será atribuído na base de dados e o parâmetro `allocationSize = 1` define de quanto em quanto será o incremento que será aplicado na sequência.

O parâmetro `generator = "seq_pedido"` que foi utilizado na anotação `@GeneratedValue` é utilizado para informar qual o nome da sequência que foi criada que deverá ser utilizada na geração das informações do campo, e o

strategyType.SEQUENCE é responsável por definir que ele será executado de forma sequencial.

A anotação *@Temporal* é utilizada quando é preciso realizar a persistência de campos ou propriedades do tipo *Date* ou *Calendar*. O parâmetro *TemporalType.TIMESTAMP* que foi utilizado está definindo que o tipo que será mapeado será do tipo *TIMESTAMP*.

A anotação *@JoinColumn* é responsável por definir comportamentos que devem ser aplicados ao persistir esta informação no banco de dados. Neste caso, o parâmetro *name="pedido_id"* está definindo que este será o nome da chave estrangeira que será criada na classe *PedidoItem*.

A anotação *@OneToOne* define que ocorrerá uma relação de um para um com o elemento que for especificado, neste caso, com a entidade *Pagamento*.

O parâmetro *updatable=false* que foi utilizado na anotação *@JoinColumn* é responsável por definir que quando ocorrer atualizações nesta classe, o valor que já tiver sido definido para este campo não poderá ser modificado.

4.3.2 Camada Repository

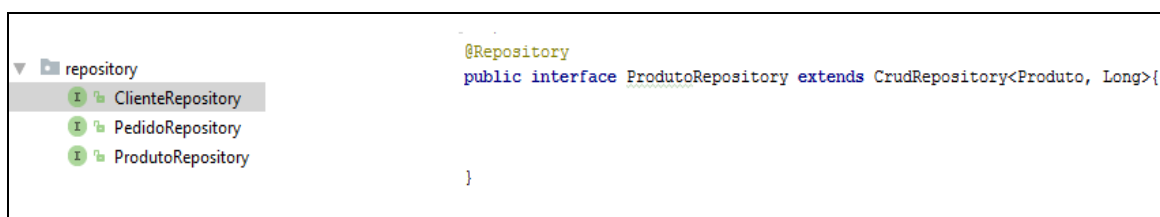
A camada *repository* que foi criada é responsável por realizar as persistências e operações no banco que a aplicação deverá realizar. Na aplicação foi utilizado o Spring Framework, pois ele oferece uma série de recursos que visa automatizar o desenvolvimento da aplicação, como: facilidade no desenvolvimento de um serviço *Rest*, definição das permissões que uma aplicação deverá possuir, interfaces que facilitam a persistência da informação no banco de dados e entre outros.

Um destes recursos disponibilizados é a interface *CrudRepository* que permite ao desenvolvedor realizar as operações básicas de CRUD – inserir, editar, excluir e listar – passando como parâmetro a classe que será mapeada e o tipo da chave primária que foi criada no banco.

Para o desenvolvimento do projeto foram criadas três interfaces que são responsáveis por mapear suas respectivas classes, sendo elas: *Cliente*, *Pedido* e *Produto*. Todas elas implementam a interface *CrudRepository* e passam como parâmetro a classe que será utilizada como referência pela interface e qual o tipo de

dados que é a chave primária daquela classe. Entretanto, para que o Spring possa realizar as persistências e utilizar a interface, é necessário utilizar a anotação `@Repository` na interface criada para que o Spring saiba que aquela interface deve ser tratada como um repositório que será responsável pela persistência das informações na base de dados da entidade passada como parâmetro, conforme apresentado na Figura 17.

Figura 17 - Camada Repository



Fonte: Do autor

4.3.3 Camada Controller

A camada *Controller* é responsável por disponibilizar uma interface pela qual o aplicativo realizará as operações no banco de dados. A comunicação do servidor web e da aplicação ocorrerá através de um serviço *Rest*, que permite aos aplicativos e softwares obterem acesso às informações de um determinado servidor através de Uris que são disponibilizadas para a realização de tais requisições no serviço.

A comunicação entre o serviço *Rest* e as demais aplicações ocorre por meio de um arquivo texto que represente as informações que estão sendo solicitadas ou que devem ser persistidas no banco de dados. Neste projeto a representação de tais informações ocorrerá através de um arquivo no formato JSON, pois a aplicação que foi desenvolvida possui como base a linguagem JAVASCRIPT e os objetos nesta linguagem são representados no formato JSON.

Antes de iniciar a explicação sobre como ocorre o acesso às informações do banco através do serviço, há a necessidade de realizar uma configuração no lado do servidor que define que outros domínios que sejam diferentes do servidor podem obter acesso às informações que estão armazenadas. Tal configuração é necessária, pois, por medidas de segurança, existe uma especificação chamada *Cross-Origin Resource Sharing* que por *default* bloqueia o acesso às informações quando as requisições ocorrem de outro domínio. Para solucionar tal questão, foi criada uma

classe chamada *CORSFilter* que é responsável por definir um cabeçalho que autoriza o acesso, conforme demonstrado na Figura 18.

Figura 18 – CORSFilter

```

13
14  /**
15   * Created by Crisley Alves on 27, fev, 2018
16   */
17  @Component
18  public class CORSFilter extends OncePerRequestFilter{
19
20      private static final Log LOG = LoggerFactory.getLog(CORSFilter.class);
21
22      @Override
23      protected void doFilterInternal(HttpServletRequest request,
24                                     HttpServletResponse response,
25                                     FilterChain filterChain) throws ServletException, IOException {
26
27          response.addHeader("Access-Control-Allow-Origin", "*");
28          if (request.getHeader("Access-Control-Request-Method") != null && "OPTIONS".equals(request.getMethod())) {
29              LOG.trace("Sending Header...");
30              // CORS "pre-flight" request
31              response.addHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
32              response.addHeader("Access-Control-Allow-Headers", "Content-Type");
33              response.addHeader("Access-Control-Max-Age", "1");
34          }
35          filterChain.doFilter(request, response);
36      }
37  }
38

```

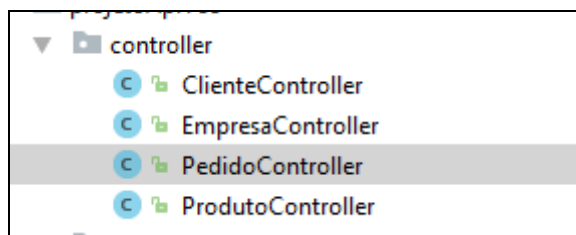
Fonte: Do autor

Como pode ser observado na imagem, a classe *CORSFilter* possui uma anotação *@Component* que é um recurso fornecido pelo Spring. Ela é utilizada para identificar que tal classe é um componente da aplicação, assim sendo, esta classe será automaticamente identificada e executada pelo Spring.

Também se faz necessário a implementação da interface *OncePerRequestFilter* do Spring, que define o que deve ser executado a cada uma das requisições que ocorrem no serviço. Dentro desta interface há o método *doFilterInternal* que trata o filtro que deve ocorrer a cada uma das requisições. Este método foi sobrescrito utilizando a anotação *@Override* para que se possa definir regras customizadas em relação às requisições. Esta classe, basicamente, define no cabeçalho que requisições do tipo *GET*, *POST*, *PUT* e *DELETE* podem ser realizadas no serviço e que o acesso aos recursos podem ser realizadas a partir de outros domínios.

A Figura 19 demonstra os *controllers* que foram desenvolvidos para realizar as operações no banco de dados.

Figura 19 – Controllers



Fonte: Do autor

Para realizar a explicação sobre a configuração e o funcionamento de um serviço *Rest* utilizando o Spring, será utilizada como referência a classe *Pedido*, pois, além da mesma possuir os mesmos recursos dos demais serviços e ser uma das mais importantes do projeto, nela também está sendo tratado o envio de notificações para o dispositivo do cliente e a abstração da geração do *token* de pagamento de um pedido.

A Figura 20 apresenta parte do código fonte que foi utilizado para realizar a configuração do serviço *Rest*.

Figura 20 - Serviço Rest

```
@RestController()
@RequestMapping("pedidos")
public class PedidoController {

    private final PedidoRepository pedidoRepository;

    private final ProdutoRepository produtoRepository;

    private final ClienteRepository clienteRepository;

    public PedidoController(PedidoRepository pedidoRepository, ProdutoRepository produtoRepository,
        ClienteRepository clienteRepository) {
        this.pedidoRepository = pedidoRepository;
        this.produtoRepository = produtoRepository;
        this.clienteRepository = clienteRepository;
    }

    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<?> listAll(Pageable pageable){
        return new ResponseEntity<>(pedidoRepository.findAll(pageable), HttpStatus.OK);
    }

    @RequestMapping(method = RequestMethod.GET, path =("/{id}")
    public ResponseEntity<?> getPedidoById(@PathVariable("id") Long id){
        return new ResponseEntity<>(pedidoRepository.findOne(id), HttpStatus.OK);
    }

    @RequestMapping(method = RequestMethod.GET, path = "/empresas/{id}")
    public ResponseEntity<?> getPedidoByEmpresaId(@PathVariable("id") Long id){
        return new ResponseEntity<>(pedidoRepository.findAllByEmpresaId(id), HttpStatus.OK);
    }
}
```

Fonte: Do autor

A anotação *@RestController* é uma anotação do Spring Framework que combina duas funcionalidades necessárias ao se trabalhar com Rest, o *@Controller* para definir que uma determinada classe é um *Controller* e o *@ResponseBody*, que é responsável por habilitar a serialização de um objeto que está sendo retornado a partir do *HttpResponse*.

Também foram criadas instâncias dos repositórios que foram criados anteriormente e tais repositórios serão os responsáveis por realizar as operações no banco de dados através dos métodos públicos que a interface *CrudRepository* do Spring disponibiliza.

A anotação *@RequestMapping* é responsável por mapear as requisições web que ocorrem aos métodos disponíveis dentro do serviço, podendo definir algumas configurações através de parâmetros para a anotação, como o método, o que ele irá produzir, consumir, caminho e dentre outros.

Em relação aos métodos, os mais comuns são: *POST*, *GET*, *DELETE* e *Remove*. Requisições do tipo *GET* são utilizadas para obtenção de dados, o *POST* para realizar a persistência de novas informações, o *PUT* para realizar a atualização de uma determinada informação ou um grupo de informações e o método *DELETE* é utilizado para realizar a remoção da informação do banco de dados.

O parâmetro *path* que é passado dentro do *RequestMapping* é utilizado para definir a URI que será responsável por disponibilizar a funcionalidade do método para as aplicações. Caso uma URI que utilize o método *GET* necessite de um determinado parâmetro para o seu funcionamento, tal parâmetro deve ser definido através de colchetes, conforme apresentado no método *getPedidoById*.

O parâmetro *consumes* é utilizado para definir o tipo de documento que será consumido por este método do serviço, no caso, conforme dito anteriormente, trabalhou-se com arquivos do tipo JSON, portanto, faz-se necessário definir que o método irá consumir arquivo do tipo JSON, para tal, utiliza-se a classe *MediaType*, que permite definir o tipo de arquivo que será consumido e produzido, neste caso, utilizou-se o *APPLICATION_JSON_UTF8_VALUE*.

O tipo de retorno *ResponseEntity* utilizado nos métodos que foram disponibilizados pelo serviço é utilizado para obter o retorno da operação que foi realizada no banco e para obter o Status HTTP do resultado requisição, assim

podendo tratar o resultado da operação de acordo com o status retornado pelo serviço.

A anotação *@PathVariable* é definida como parâmetro de uma função quando necessita-se realizar uma determinada operação no serviço de acordo com o critério que está sendo enviado por parâmetro pela aplicação que está realizando a requisição ao serviço. Por exemplo, no caso da função *getPedidoById* a função está recebendo como parâmetro uma variável *id* do tipo *Long* que é passada como parâmetro para o método *findOne* que está disponível no *pedidoRepository*, tal função é responsável por obter um determinado pedido de acordo com o seu código.

Sempre quando ocorre a realização ou a atualização de um pedido no sistema, é realizada uma instância de uma classe chamada *NotificationUtil* que foi desenvolvida exclusivamente para realizar o tratamento de notificações sobre o andamento do pedido de acordo com as modificações que ocorrem no status.

A realização do envio de notificações ocorre através do serviço do *OneSignal* que permite realizar o envio de notificações para todos os usuários que estiverem cadastrados em uma determinada aplicação, sendo possível também enviar as notificações para grupos de pessoas ou mesmo para usuários específicos da aplicação.

A realização do envio de notificações na aplicação ocorre através de um recurso do serviço do *OneSignal* que obtém o código do dispositivo que está realizando o pedido e armazena-o em um variável chamada *oneSignalPlayerId*, e com base neste código que é armazenado pelo serviço e armazenado como atributo na classe *Cliente* é possível identificar quem realizou o pedido e a partir de qual dispositivo, assim permitindo o envio de notificações para este cliente.

A classe *NotificationUtil* realiza uma requisição HTTP do tipo *POST* no serviço do *OneSignal* passando como parâmetro a chave da API que foi gerada para a aplicação, o id do aplicativo que foi gerado, os idiomas e as mensagens que devem ser apresentadas para o usuário ou para o grupo de usuários em seus respectivos idiomas e o *oneSignalPlayerId* que foi obtido do dispositivo do usuário. As mensagens que estão sendo enviadas para o serviço para posteriormente serem encaminhadas para o usuário estão sendo definidas de acordo com o status do pedido e, mantendo o padrão de internacionalização da aplicação, elas estão sendo definidas nos idiomas Português e Inglês.

A realização do pagamento do pedido seria tratada utilizando o gateway de pagamento do pagseguro, entretanto, para realizar operações financeiras utilizando o gateway é necessário adquirir uma conta do tipo vendedor que é paga. Portanto, optou-se por não aplicar na aplicação, mas deixar a aplicação preparada para uma futura integração. Para tal, criou-se uma classe chamada Pagamento que é responsável por armazenar um *token* de pagamento que seria gerado através do gateway, e este *token* de pagamento está sendo gerado através da classe *SecureTokenGenerator*, que é responsável por gerar um *token* randômico com o mesmo tamanho que seria gerado pelo gateway.

A Figura 21 ilustra o restante do código fonte que foi utilizado na classe Pedido, abordando os pontos mais relevantes que foram explicados acima.

Figura 21 - Serviço Rest – Classe Pedido

```
@RequestMapping(path = "/insert/", method = RequestMethod.POST, consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
@Transactional
public ResponseEntity<?> insertPedido(@Valid @RequestBody Pedido pedido) {
    if(pedido.getCliente().getOneSignalPlayerId() == null){
        Cliente cliente = clienteRepository.findFirstByEmailIgnoreCaseContaining(pedido.getCliente().getEmail());
        pedido.getCliente().setOneSignalPlayerId(cliente.getOneSignalPlayerId());
    }
    pedido.getPagamento().setToken(new SecureTokenGenerator().nextToken().toUpperCase());
    new NotificationUtil().notificaUsuario(pedido.getStatusPedido(), pedido.getCliente());
    return new ResponseEntity<>(this.pedidoRepository.save(pedido), HttpStatus.CREATED);
}

@RequestMapping(path = "/update/", method = RequestMethod.PUT, consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> updatePedido(@RequestBody Pedido pedido) {
    Pedido full = new Pedido();
    full = pedidoRepository.findOne(pedido.getId());
    Hibernate.initialize(full.getCliente().getPedidoList());
    Hibernate.initialize(full.getEmpresa().getPedidoList());
    if(pedido.getAvaliado() == Boolean.TRUE) {
        for (PedidoItem p : pedido.getProdutoItemList()) {
            if (p.isRecomendado() == true) {
                Produto produto = this.produtoRepository.findOne(p.getProduto().getId());
                produto.setAvaliacao((produto.getAvaliacao() + 1));
            }
        }
    }
    if(pedido.getAvaliado() == Boolean.FALSE)
        new NotificationUtil().notificaUsuario(pedido.getStatusPedido(), full.getCliente());
    pedido.setCliente(full.getCliente());
    pedido.setEmpresa(full.getEmpresa());
    return new ResponseEntity<>(this.pedidoRepository.save(pedido), HttpStatus.OK);
}
```

Fonte: Do autor

4.4 APLICATIVO MÓVEL

Nesta seção serão apresentadas algumas informações sobre o desenvolvimento do aplicativo, destacando os pontos mais relevantes da aplicação.

4.4.1 Arquitetura

A arquitetura de uma aplicação desenvolvida com Ionic, Angular e Cordova é composta por arquivos que gerenciam o seu funcionamento híbrido básico, outros que são responsáveis por tratar os serviços que são utilizados na aplicação e outros que cuidarão da renderização dos conteúdos e do comportamento atribuído a cada uma destas renderizações.

Como o aplicativo foi desenvolvido utilizando a combinação dos frameworks Angular e Ionic, a aplicação em si é baseada no desenvolvimento de componentes, sendo um componente o isolamento de uma determinada funcionalidade da aplicação com finalidades específicas, e tal componente é escrito utilizando HTML, CSS, Javascript e as *tags* específicas que são fornecidas pelo Ionic.

Ao realizar a criação de um novo componente na aplicação, tal componente possuirá três arquivos com funcionalidades únicas que serão responsáveis por gerenciar as configurações deste componente, sendo um destes arquivos responsável por armazenar as *tags* que são utilizadas para renderizar o conteúdo deste componente, outro por tratar as estilizações e o último por tratar os eventos que serão atribuídos à ele, e a *tag* que será utilizada para a importação deste componente na aplicação.

Os componentes que são utilizados na aplicação devem ser obrigatoriamente importados em seu módulo principal, o *app.module.ts*. Este módulo que é gerado automaticamente após a criação de um projeto Ionic em conjunto com o Angular é o módulo principal da aplicação que é gerencia todos os componentes, serviços, diretivas, rotas, *plugins* e todos os recursos que fazem parte da aplicação.

Nele é possível definir qual será o componente que deverá ser utilizado ao iniciar o aplicativo, por padrão um componente cujo nome é *App* é gerado automaticamente ao criar um novo projeto e tal componente se faz necessário não só por ser o componente principal da aplicação, mas também por tratar configurações específicas ao se trabalhar com Ionic e Angular.

Neste componente ocorre a identificação do dispositivo que está acessando a aplicação através do módulo *Platform* que faz parte do módulo Ionic & Angular, nele é possível identificar se o usuário está acessando a aplicação através de um

dispositivo que possui android, ios ou *Windows Phone*, além também de permitir identificar se o acesso ocorreu através de um *smartphone*, navegador ou *tablet*.

Além de realizar a identificação do dispositivo também se é carregado todos os plugins que foram utilizados na aplicação.

4.4.2 Camada modelo

O desenvolvimento da camada modelo na aplicação foi realizado tendo como base os objetos que precisam ser persistidos no *backend* da aplicação. Os modelos que foram criados no front nada mais são do que objetos que possuem os mesmos atributos que as classes de persistência da camada *Model* do *backend*. Optou-se por realizar a cópia dos atributos para facilitar na hora de realizar as operações no front da aplicação já mantendo o padrão de objeto que é válido tanto no serviço que foi desenvolvido quando no modelo que foi criado no *backend*.

A Figura 22 ilustra o objeto que foi criado para o armazenamento e persistência dos dados de um pedido.

Figura 22 - Camada modelo do aplicativo

```
import { PagamentoModel } from './pagamento.model';
import { ClienteModel } from './cliente.model';
import { PedidoItemModel } from './pedidoitem.model';

export class PedidoModel{
    codigo: number;
    horaSolicitado: any;
    horaEntregue: any;
    formaPagamento: any;
    valorTotal: number;
    produtoItemList: Array<PedidoItemModel> = new Array();
    cliente: ClienteModel = new ClienteModel();
    statusPedido: number;
    pagamento: PagamentoModel = new PagamentoModel();
    avaliado: boolean;
    empresa: number;
}
```

Fonte: Do autor

No topo do arquivo realiza-se a importação de outros modelos que também foram criados no front e que são necessários ao realizar a persistência de um novo Pedido. Após a importação destes modelos, define-se uma classe chamada `PedidoModel` que possui como finalidade o armazenamento dos pedidos que foram realizados, já armazenando-os em um formato válido para posteriormente realizar a persistência da informação. Após a declaração de um atributo, é possível definir o tipo deste atributo, sendo, *number* para número, *boolean* para verdadeiro ou falso e *any* para que a variável armazene qualquer tipo de dado que for atribuído a ela.

4.4.3 Serviço

A camada de serviço da aplicação é responsável por realizar as requisições no serviço Rest que foi desenvolvido, passando como parâmetro os seus respectivos parâmetros. Nela também ocorre a manipulação dos eventos que podem ser solicitados em relação ao pedido, como: remover item do carrinho, adicionar um produto ao carrinho, realizar um pedido e inserir a avaliação do cliente sobre os itens adquiridos. Por se tratar de uma classe extensa, somente serão abordados os pontos mais relevantes da classe, como a importação dos objetos, as injeções dos serviços e a persistência da informação. A Figura 23 ilustra parte do código fonte que foi desenvolvido nesta classe.

Figura 23 - Conexão com o serviço

```

import { Storage } from '@ionic/storage';
import { ClienteModel } from '../model/cliente.model';
import { ProdutoModel } from '../model/produto.model';
import { PedidoItemModel } from '../model/pedidoitem.model';
import { PedidoModel } from '../model/pedido.model';
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { EmpresaService } from '../empresa.service';
import 'rxjs/add/operator/map';
@Injectable()
export class PedidoService{
  configUrl = 'http://192.168.1.103:8080/pedidos/insert/';
  configUrlUpdate = 'http://192.168.1.103:8080/pedidos/update/';
  pedido: PedidoModel;
  PedidoItemModel: PedidoItemModel;
  produtosCarrinho: Array<ProdutoModel> = new Array<ProdutoModel>();
  constructor( private http: HttpClient, public storage: Storage, private empresaService: EmpresaService) {
    this.PedidoItemModel = new PedidoItemModel();
    this.pedido = new PedidoModel();
    this.pedido.cliente = new ClienteModel();
  }
  addProduto(produto){
    this.PedidoItemModel = new PedidoItemModel();
    this.PedidoItemModel.observacao;
    this.PedidoItemModel.quantidade = 1;
    this.PedidoItemModel.valorUnitario = produto.preco;
    this.PedidoItemModel.valorTotal = (produto.preco * 1);
    this.PedidoItemModel.produto = produto;
    this.pedido.horaEntregue = '';
    this.pedido.statusPedido = 1;
    this.pedido.produtoItemList.push(this.PedidoItemModel);
  }
}

```

Fonte: Do autor

Na primeira linha realiza-se a importação do módulo *Storage* que vem junto com o Ionic. Este módulo foi importado para que ao realizar um pedido seja obtido os dados do cliente que foi armazenado no celular quando ele realizou o *Login* ou registro na aplicação.

Os modelos importados na aplicação são responsáveis por armazenar o Pedido, os itens do pedido que foram selecionados e os produtos que foram selecionados pelo cliente. O módulo *HttpClient* é responsável por fornecer a possibilidade de realizar requisições HTTP ao serviço que foi desenvolvido e foi através dele que as requisições foram realizadas. A importação do módulo *Injectable* se faz necessária ao se trabalhar com serviços que realizam requisições, assim tratando as dependências que podem ocorrer ao utilizar o serviço criado.

O serviço *EmpresaService* é importado pois o cliente realizou a leitura de um código QRCODE que realizou uma requisição HTTP ao serviço solicitando o carregamento das informações do estabelecimento. A anotação *@Injectable* define

que esta classe que é um serviço será injetada em outros serviços, caso isto não fosse realizado este serviço deveria ser importado em cada um dos módulos que utilizarem esta classe e no construtor da classe foram inicializados todos os módulos e serviços que serão utilizados dentro da aplicação.

A função *addProduto* que foi desenvolvida recebe como parâmetro o produto que foi selecionado pelo cliente e adiciona-o ao *pedidoItemModel* que foi instanciado na classe. Além de adicionar o produto, também define-se o restante dos parâmetros da classe *pedidoItem* e adiciona-o a lista de produtos selecionados pelo cliente. Após a adição dos produtos realiza-se uma requisição do tipo *POST* ao serviço passando como parâmetro o objeto *Pedido* criado. A requisição ocorre no endereço que está sendo armazenado na variável *configUrl*.

4.4.4 Views

As views são componentes compostos por códigos HTML e CSS que são responsáveis por renderizar o conteúdo para o usuário. Cada componente possui um arquivo que trata os eventos, outro que trata os estilos e outro que armazena as tags utilizadas naquele componente. A Figura 24 ilustra o código fonte que é responsável por tratar os eventos que podem ocorrer no componente de listagem de produtos.

Figura 24 – Views

```

import { EmpresaService } from './../../services/empresa.service';
import { TranslateService } from '@ngx-translate/core';
import { PedidoService } from './../../services/pedido.service';
import { ProdutoService } from './../../services/produtos.service';
import { ProdutoInfo } from '../produto-info/produto.info';
import { Component } from '@angular/core';
import { NavController, ModalController } from 'ionic-angular';
import { CurrencyPipe } from '@angular/common';
@Component({
  selector: 'page-produtos',
  templateUrl: 'produtos.html'
})
export class ProdutosPage {
  produtos: any;
  constructor(public navCtrl: NavController, public modalCtrl: ModalController,
    private produtosService: ProdutoService, private pedidoService: PedidoService,
    private translate: TranslateService, private empresaService: EmpresaService) {
    this.produtosService
      .getProdutosByEmpresaId(this.empresaService.empresa.id).subscribe( res => {
        this.produtos = res; })
  }

  infoProduto(produto){
    let modal = this.modalCtrl.create(ProdutoInfo, {produto : produto});
    modal.present();
  }

  adicionaCarrinho(produto){
    this.pedidoService.addProduto(produto);
  }
}

```

Fonte: Do autor

No arquivo responsável por tratar os eventos de um componente é possível realizar a importação de *plugins*, componentes da aplicação, serviços e módulos específicos que permitem acesso aos recursos do celular. Neste componente de listagem de produtos, realizou-se a importação do serviço *EmpresaService*, que trata da listagem dos produtos do estabelecimento carregado através do código QRCODE.

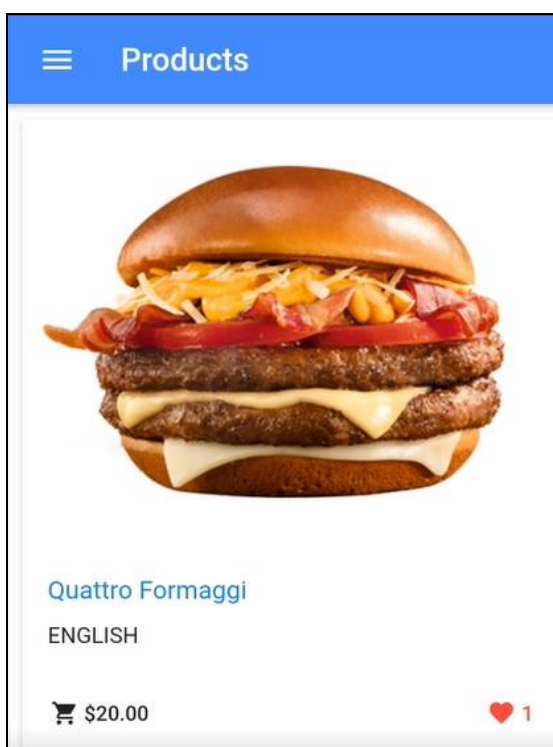
O módulo *CurrencyPipe* foi importado, pois na listagem de produtos será apresentado o preço do produto, como a aplicação foi internacionalizada, o valor que deve ser apresentado para o usuário deverá variar de acordo com a idioma selecionado, sendo importante salientar que apenas a parte visual do preço foi

tratada, o valor em si será o mesmo para todos, entretanto, tal melhoria pode ser aplicada futuramente na aplicação.

A importação do módulo *Component* se faz necessária, pois este é um componente da aplicação e o carregamento dos arquivos que compõem tal componente estão sendo informados dentro da anotação *@Component*, sendo o *selector* a *tag* que será utilizada para chamar o componente para aplicação e o *templateUrl* responsável por informar qual o arquivo HTML que corresponde à este componente. Também é possível adicionar um arquivo responsável pela estilização do componente através do seletor *styleUrls*, que recebe como parâmetro um *array* de arquivos de estilização, entretanto, a interface gráfica foi desenvolvida utilizando os componentes do Ionic, que já supria todas as necessidades dos componentes que foram criados.

Também realizou-se a importação do módulo *TranslateService* que é utilizado para tratar a internacionalização da aplicação, ele será melhor explicado no próximo tópico. A Figura 25 ilustra como ficou a interface gráfica deste componente.

Figura 25 - Views – Listagem de produtos



Fonte: Do autor

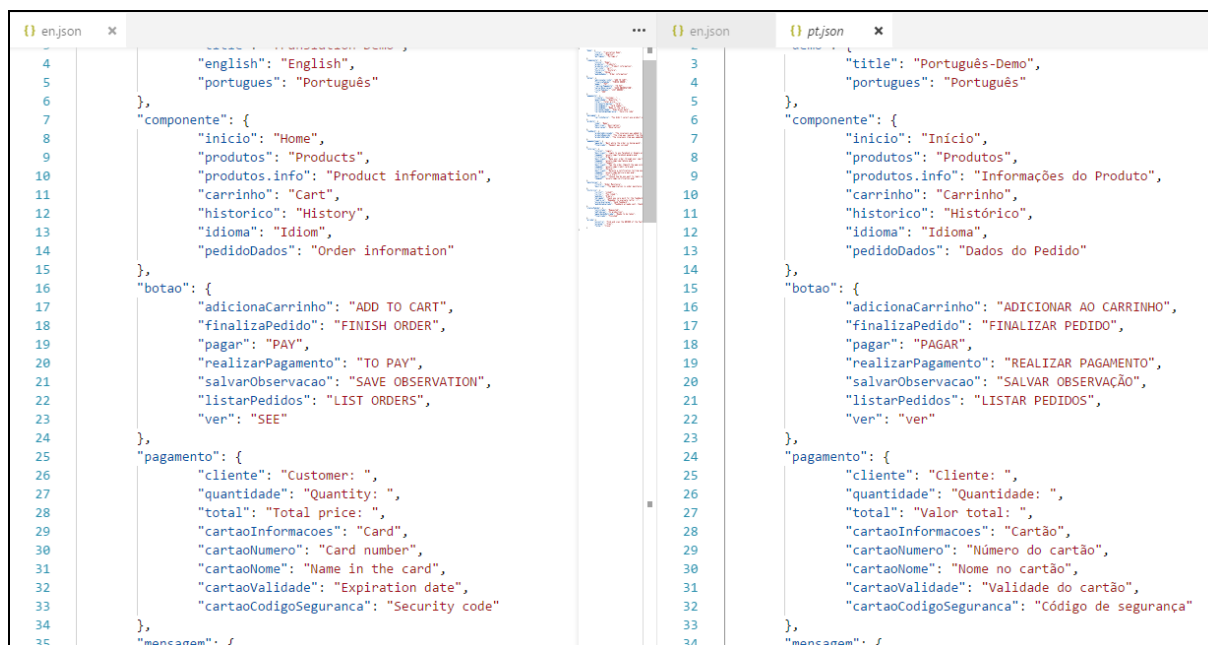
4.4.5 Internacionalização

Uma das principais propostas do projeto foi realizar a internacionalização da aplicação, para assim assegurar que os estabelecimentos possam oferecer seus produtos para as pessoas de outros países em seu idioma nativo.

Para realizar a internacionalização da aplicação foi utilizada a biblioteca *ngx-translate* que permite configurar arquivos no formato JSON contendo todas as informações que devem ser apresentadas ao cliente em seu respectivo idioma. O processo de instalação da biblioteca ocorre através do gerenciador de pacote NPM conforme apresentado na documentação oficial da biblioteca.

Para utilizar a biblioteca deve-se realizar sua importação no *app.module.ts*, que, conforme dito anteriormente, é responsável por armazenar todos os recursos que serão utilizados na aplicação e também deve-se criar arquivos JSON que armazenarão os conteúdos que serão traduzidos, conforme apresentado na Figura 26.

Figura 26 - JSON de internacionalização



Fonte: Do autor

Pode-se observar na imagem que o arquivo de tradução é composto por objetos que possuem chave e valor, sendo o primeiro valor definido a chave e o segundo o

valor, que estão dentro de um objeto em JAVASCRIPT que será utilizado para acessar a informação.

Além de definir os arquivos, também realiza-se a importação do módulo *TranslateService*, que será responsável por acessar este arquivo e disponibilizar as informações que estão sendo armazenadas nele. Entretanto, conforme explicado anteriormente, todos os recursos que são utilizados na aplicação devem ser devidamente configurados no *app.module.ts*, portanto, a importação da biblioteca neste componente é necessária, e também deve-se informar ao serviço onde está localizado os arquivos de configuração.

A Figura 27 ilustra como ocorre a definição do local que deve ser acessado para obter a informação no idioma correto, sendo que a configuração ocorre através de um dos módulos que foi importado no topo do arquivo, o *HttpLoaderFactory* é uma função que foi criada para realizar o tratamento da internacionalização, retornando como resultado o módulo *TranslateHttpLoader* que é disponibilizado pelo *ngx-translate* para definir o arquivo que deve ser carregado quando houver uma requisição – de componente – na aplicação, passando como parâmetro o módulo *HttpClient* que realiza uma requisição ao arquivo informado nos parâmetros da função.

Figura 27 - Definição do local dos arquivos de internacionalização

```
export function HttpLoaderFactory(http: HttpClient) {  
  return new TranslateHttpLoader(http, './assets/i18n/', '.json');  
}
```

Fonte: Do autor

Para apresentar a informação na aplicação utiliza-se uma diretiva. Diretivas são regras específicas que são aplicadas a um determinado componente ou trecho de código. No caso a biblioteca *ngx-translate* disponibiliza a diretiva *translate* que quando utilizada permite acessar o arquivo JSON de internacionalização que foi criado. A Figura 28 ilustra como ocorre o acesso à informação traduzida e como utilizar a diretiva disponibilizada pela biblioteca.

Figura 28 - Diretiva Translate

```

<button *ngIf="!empresaService.empresa"
        ion-button block color="secondary"
        (click)="getInfoEmpresa()">
    <span translate> qrcode.escanear </span>
</button>
<button *ngIf="empresaService.empresa"
        ion-button block color="danger" (click)="fechar()">
    <span translate> qrcode.fechar </span>
</button>

```

Fonte: Do autor

Dentro da tag que está sendo usada para executar a diretiva *translate*, pode-se observar que está sendo acessado um dos objetos que fazem parte do arquivo de configuração JSON, neste caso, o objeto QRCODE e informando o atributo que deve ser acessado dentro deste objeto, assim retornando o valor conforme o idioma que foi selecionado pelo cliente.

4.5 PAINEL ADMINISTRATIVO

Nesta seção serão apresentadas algumas informações sobre o desenvolvimento do painel administrativo do sistema, destacando apenas o que for distinto do que foi explicado na seção anterior, pois o funcionamento do painel administrativo é semelhante ao do aplicativo.

4.5.1 Arquitetura

A arquitetura do painel funciona da mesma forma que a do aplicativo, entretanto, no aplicativo a renderização das páginas e as rotas que estavam disponíveis eram configuradas pelo Ionic. No caso do painel administrativo foi utilizado as configurações de rotas do Angular 5 conforme apresentado na Figura 29.

Vale salientar que o trecho de código que está sublinhado em vermelho está funcionando normalmente, ele apenas aparece desta forma, pois este projeto que está sendo utilizado para explicação ainda não possui as dependências da aplicação instaladas.

Figura 29 – Rotas

```

TS app.routing.module.ts
1  import { PedidosUpdateComponent } from './pages/pedidos/pedidos-update/pedidos.update.component';
2  import { PedidosComponent } from './pages/pedidos/pedidos.component';
3  import { PaginaNaoEncontradaComponent } from './pages/pagina-nao-encontrada/pagina-nao-encontrada.component';
4  import { InicioComponent } from './pages/inicio/inicio.component';
5  import { NgModule } from '@angular/core';
6  import { Routes, RouterModule } from '@angular/router';
7
8  //constante responsável por armazenar as rotas que farão parte do projeto
9  const appRoutes: Routes = [
10   { path: '', component: InicioComponent},
11   { path: 'pedidos', component: PedidosComponent},
12   { path: 'pedidos/update/:id', component: PedidosUpdateComponent},
13   { path: 'pagina-nao-encontrada', component: PaginaNaoEncontradaComponent},
14   { path: '**', redirectTo: '/pagina-nao-encontrada'}
15 ];
16
17
18 @NgModule({
19   imports: [
20     RouterModule.forRoot(appRoutes)
21   ],
22   exports: [RouterModule]
23 })
24 export class AppRoutingModule{
25
26 }

```

Fonte: Do autor

No topo do arquivo pode-se observar a importação do componente *PedidoUpdateComponent* que é responsável por realizar a atualização no pedido que foi solicitado pelo cliente. O componente *PedidosComponent* é utilizado para realizar a listagem de todos os pedidos que foram realizados no estabelecimento.

O componente *PaginaNaoEncontradaComponent* é um componente que foi criado para ser renderizado sempre quando uma rota que não existe for solicitada. O *InicioComponent* é renderizado quando o administrador do estabelecimento realizar o login no painel, este componente está responsável por renderizar as informações básicas da empresa pós *Login*, basicamente, a tela de boas vindas.

Logo abaixo ocorre a importação dos módulos *Routes* e *RouterModule*, sendo o *Routes* responsável por armazenar as rotas que estão disponíveis na aplicação e o *RouterModule* o módulo que é responsável por configurar as rotas recebidas como parâmetro na aplicação.

Em seguida foi definida uma constante chamada *appRoutes* que recebe um array de objetos que armazena as informações das rotas e dos componentes que devem ser renderizados naquela respectiva rota.

O parâmetro `path` define o nome da rota que será disponibilizada na aplicação, podendo também definir o recebimento de parâmetros naquela rota. No caso da rota `pedidos/update/:id` será recebido como parâmetro o `id` do pedido que será manipulado.

O parâmetro `redirectTo` define um componente default que deverá ser executado caso uma rota inválida seja solicitada, neste caso o usuário será redirecionado para a rota 'pagina-não-encontrada' que executa o componente referenciado.

Logo abaixo ocorre a utilização do módulo `@NgModule` que é responsável por configurar a aplicação, definindo como rota as configurações de rotas que foram definidas nas linhas acima. Com isto apenas deve-se realizar a importação deste arquivo `app.routing.modules.ts` no `app.modules.ts`, pois todos os componentes e configuração da aplicação devem estar presentes lá, conforme explicado na seção anterior.

4.5.2 Views

As *views* possuem o mesmo padrão de funcionamento do aplicativo, nesta seção, julgo apenas necessário informar o método responsável por realizar a persistência da atualização do pedido.

Ao realizar o acesso à rota responsável pela atualização do pedido, espera-se receber como parâmetro o `id` do pedido que deve ser persistido. Recebendo como parâmetro o `id`, realiza-se uma consulta no serviço para obter as informações do pedido que possua o `id` recebido.

Em seguida armazena-se o resultado desta consulta no modelo "pedido" que foi importado no topo deste componente. Em seguida os valores que foram recebidos desta requisição são atribuídos ao formulário que está sendo visualizado pelo usuário, isto ocorre através do método `patchValue` do Angular, que permite a atribuição de dados à um determinado formulário.

Posteriormente ao realizar a atualização do pedido, verifica-se se o código do status do pedido é igual a 4 – pedido finalizado -, caso a atualização do pedido possua este código, obtêm-se a data atual e armazena-a dentro do atributo `horaEntregue` que está presente no objeto pedido que será persistido no banco, conforme apresentado na Figura 30.

Figura 30 - Atualização do pedido

```
TS pedidos.update.component.ts ●
38         'avaliado': new FormControl(null),
39         'empresa': new FormControl(null)
40     });
41
42     this.route.params
43     .subscribe(
44         (params: Params) => {
45             this.pedidoService.getPedidoById(params['id'])
46             .subscribe((val) =>{
47                 this.pedido = val;
48                 this.pedidoForm.patchValue(val);
49                 this.itens = this.pedidoForm.value.produtoItemList;
50             });
51         }
52     )
53 }
54
55 onSubmit(){
56
57     //caso esteja finalizando o pedido, vai registrar a hora que foi entregue
58     if(this.pedidoForm.value.statusPedido == 4){
59         this.pedidoForm.value.horaEntregue = Date.now();
60     }
61
62     this.pedidoForm.value.empresa = this.empresaService.empresa;
63
64     this.pedidoService.updatePedido(this.pedidoForm.value).subscribe((res) => {
65         this.router.navigate(['pedidos']);
66     });
67 }
68
69 }
```

Fonte: Do autor

4.5.3 Serviço

O serviço da aplicação possui a mesma estrutura de funcionamento do aplicativo, realiza-se importação dos módulos responsáveis por tratar as requisições que serão realizadas e ao realizar as requisições no serviço são passados os parâmetros esperados, conforme apresentado na Figura 31.

Figura 31 - Pedido servisse

```
TS pedido.service.ts x
1  import { HttpClient } from '@angular/common/http';
2  import { Observable } from 'rxjs/Observable';
3  import { Injectable } from '@angular/core';
4
5  @Injectable()
6  export class PedidoService{
7
8      configUrlGetPedido = "http://localhost:8080/pedidos/";
9      configUrlUpdate = "http://localhost:8080/pedidos/update/";
10
11      listarPedidosByEmpresa = "http://localhost:8080/pedidos/empresas/";
12
13      constructor(private http: HttpClient){}
14
15      listarPedidosPorEmpresa(id){
16          id = parseInt(id);
17          return this.http.get(this.listarPedidosByEmpresa+id, id).map(res => res);
18      }
19
20      getPedidoById(id){
21          id = parseInt(id);
22          return this.http.get(this.configUrlGetPedido+id, id).map(res => res);
23      }
24
25      updatePedido(pedido){
26          return this.http.put(this.configUrlUpdate, pedido).map((res) => res);
27      }
28  }
```

Fonte: Do autor

5 RESULTADOS

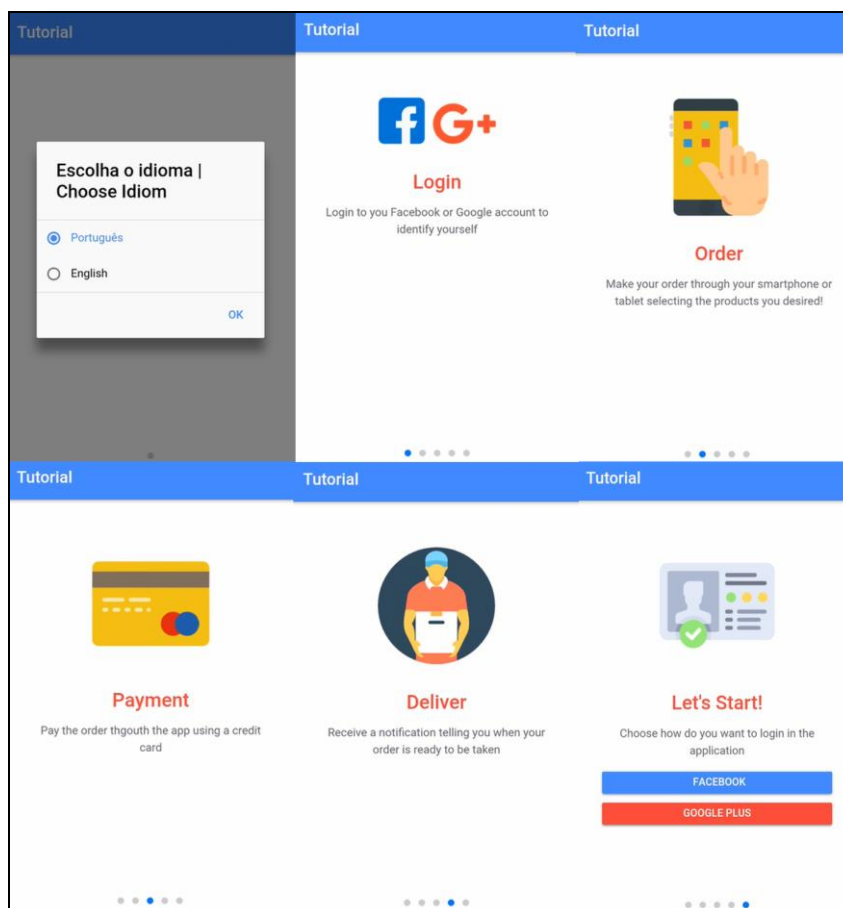
Nesta seção será apresentado o resultado do sistema após o seu desenvolvimento, abordando os acontecimentos que ocorrem durante todo o processo de atendimento, desde a sua solicitação até a sua entrega.

5.1 TUTORIAL

Assim que o usuário executar o aplicativo ocorrerá uma consulta no armazenamento local do celular para verificar se o usuário está executando o aplicativo pela primeira vez ou não. Caso o aplicativo esteja sendo executado pela primeira vez, será perguntado qual o idioma que ele deseja utilizar para a realização do pedido, oferecendo a possibilidade de executá-lo no idioma Inglês ou Português. Caso o usuário já tenha utilizado o aplicativo uma única vez, ele será redirecionado automaticamente para a página pós *Login*.

Após a seleção do idioma será apresentado um tutorial de como funciona a realização de pedidos através do celular, explicando que o cliente deve realizar Login através de uma rede social, selecionar os produtos que desejados, efetuar o pagamento do pedido através de seu cartão de crédito e aguardar até o recebimento de uma notificação informando que o pedido está disponível para retirada, conforme apresentado na Figura 32.

Figura 32 – Tutorial



Fonte: Do autor

No background da aplicação, quando o usuário acessar o dispositivo é realizada uma requisição ao serviço do *OneSignal* informando o id do dispositivo que está acessando para assim poder gerar o *playerId* que corresponderá à este dispositivo. Caso o dispositivo já tenha acessado o sistema aplicativo, o mesmo procedimento iria acontecer, entretanto, ao invés de gerar um novo *playerId* para este dispositivo, ele apenas validaria que o dispositivo já está cadastrado na base de dados do serviço.

Quando o cliente realizar o *Login* pela rede social desejada, será realizado o cadastro dele no banco de dados da aplicação, realizando uma conexão direta com o serviço *Rest* que foi desenvolvido para assim realizar o armazenando de seu

nome, email, idioma escolhido para atendimento e o *playerId* que foi gerado pelo *OneSignal*.

5.2 REALIZAÇÃO DE PEDIDO

Em seguida aparecerá uma tela que orientará o cliente a encontrar o código QRCODE do estabelecimento que ele deseja realizar o pedido e realizar a leitura dele. Neste momento, o código lido será enviado para o serviço *Rest* informando o CNPJ do estabelecimento escolhido, retornando para a aplicação o JSON que armazena a listagem dos produtos que o estabelecimento está disponibilizando e as informações básicas do estabelecimento, como código, logo e o nome.

No menu superior esquerdo serão habilitadas as opções produtos e carrinho, sendo o menu produtos para realizar a listagem dos produtos do estabelecimento e o carrinho para que o usuário possa verificar todos os produtos que constam no seu pedido.

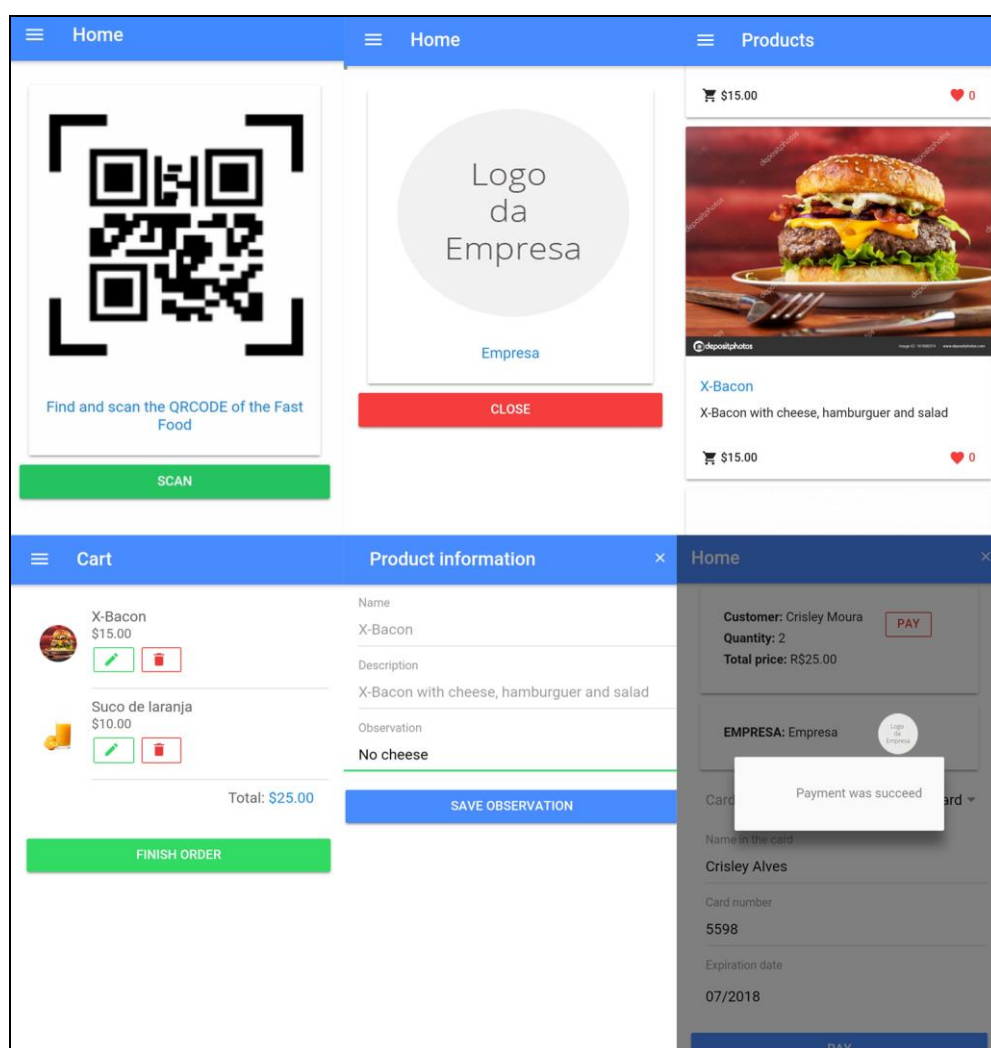
Na listagem dos produtos será apresentado o nome do produto, a foto, a descrição e a quantidade de pessoas que o adquiriram e o recomendam. Quando o cliente adicionar um produto ao seu carrinho será apresentado uma notificação informando que o produto foi adicionado ao carrinho, sendo a notificação e todo o conteúdo apresentado na tela renderizados de acordo com o idioma escolhido no início.

Posteriormente o cliente deverá acessar o menu carrinho, nesta página o cliente poderá verificar os produtos que serão solicitados, remover os produtos que não forem mais desejados no pedido, validar o valor que será cobrado pelo serviço e também informar em produtos específicos alguma observação que julgar necessária, como, por exemplo: solicitar um X Salada, mas sem salada.

Na página de pagamento, no topo da página, será apresentado o nome do cliente que está realizando o pedido, a quantidade de itens que constam nele, o valor total do pedido e o logo da empresa que está recebendo o pedido. Logo abaixo, serão solicitados os dados da conta pela qual o pagamento será realizado, sendo necessário informar o banco responsável pelo pagamento, o nome presente no cartão, o número da conta e a data de expiração.

Quando o usuário realizar o pedido pelo aplicativo os dados utilizados durante o *Login* e as informações do pedido serão enviadas ao serviço *Rest* que realizará a persistência das informações. Ao persistir as informações do pedido será realizada uma conexão com o serviço do *OneSignal*, obtendo deste cliente o *playerId* correspondente, enviando-o para o serviço do *OneSignal* para que uma notificação de confirmação de recebimento de pedido seja enviada para o cliente. A Figura 30 ilustra parte dos procedimentos explicados acima.

Figura 33 – Pedido



Fonte: Do autor

5.3 PAINEL ADMINISTRATIVO E NOTIFICAÇÕES

Um painel administrativo foi desenvolvido para poder gerenciar os pedidos que foram realizados no estabelecimento, para acessá-lo, o administrador do

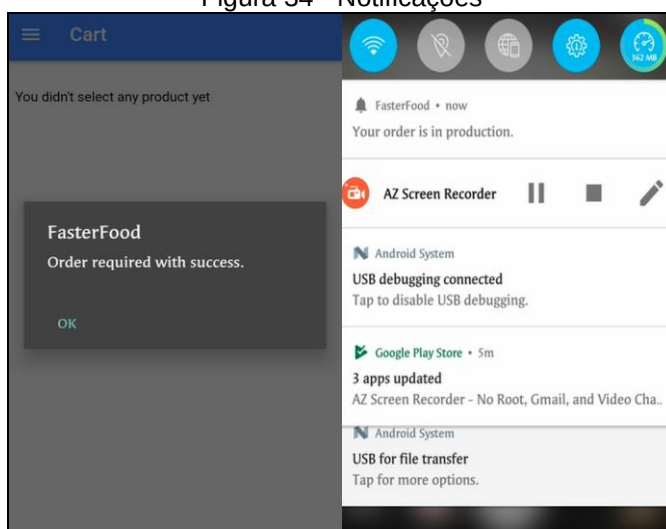
estabelecimento deve realizar o *Login* no painel informando o CNPJ do estabelecimento desejado. Ao acessar o painel administrativo lhe é apresentado o nome da empresa, seu logo e a data que está sendo realizado o acesso. No canto superior esquerdo existe a opção pedidos que permite acessar todos os pedidos que foram realizados naquele estabelecimento.

Quando ocorrer um clique em editar o administrador terá acesso as informações do pedido, sendo apresentada para ele a data que o pedido foi solicitado, a data que ele foi entregue – caso já tenha sido finalizado -, os itens que foram solicitados - apresentando em vermelho os itens que possuírem uma observação especial - e a opção status que permite ao administrador alterar o status do pedido conforme a situação atual dele.

Quando o administrador alterar o status do pedido e realizar a persistência no banco de dados da aplicação, uma nova requisição será realizada ao serviço do *OneSignal* passando como parâmetro o *playerId* do cliente que realizou aquele pedido em específico e o status do pedido, para assim poder controlar qual será a mensagem de status do pedido que será enviada para o cliente, lembrando que a notificação será enviada de acordo com o idioma do cliente.

A Figura 31 confirma o recebimento das notificações pelo usuário e a Figura 32 ilustra como ficou o painel administrativo.

Figura 34 - Notificações



Fonte: Do autor

Figura 35 - Painel administrativo

FasterFood Pedidos Sair

Logo da Empresa

SEJA BEM VINDO, **Empresa**

Data: 01/07/2018

FasterFood Pedidos Sair

Listagem de Pedidos

Código	Solicitado	Qtd. Itens	Total	Status	Ações
12	13:50:57	2	R\$25.00	Solicitado	editar

FasterFood Pedidos Sair

Atualização de Pedidos

Data: 01/07/2018 Data Solicitado: 13:50:57 Data Entrega: 14:01:10

Itens

- X-Bacon - R\$15.00
Obs: **Completo**
- Suco de laranja - R\$10.00
Obs: **No cheese**

Status: Finalizado

[Atualizar](#)

Fonte: Do autor

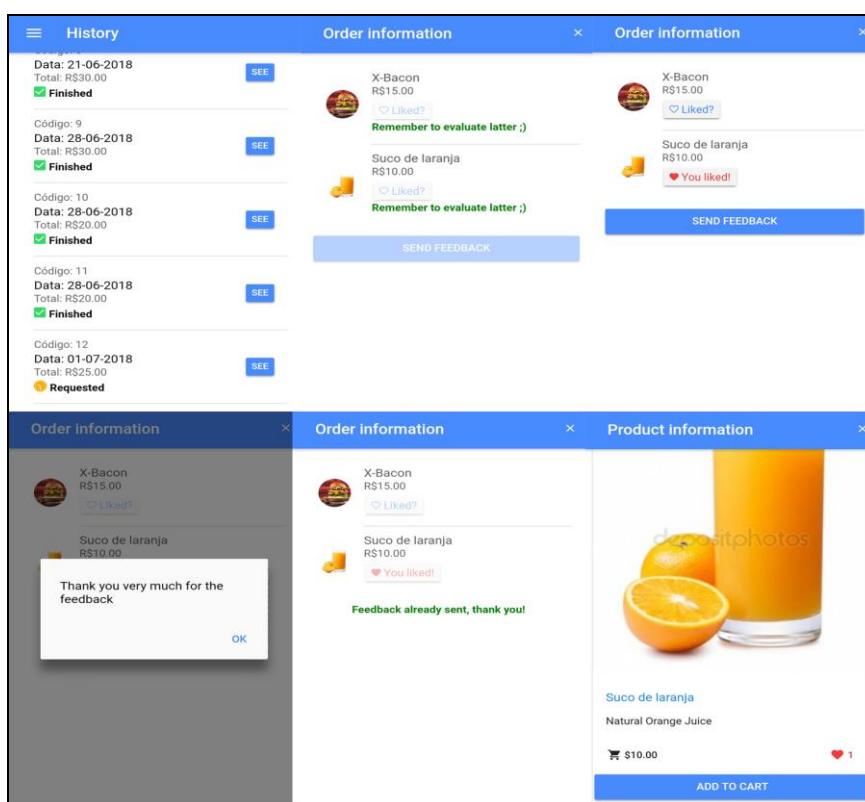
5.4 HISTÓRICO E AVALIAÇÃO DOS PRODUTOS

No menu do canto superior esquerdo há a opção histórico que disponibiliza a listagem de todos os pedidos que o cliente realizou até a presente data. Neste menu o cliente poderá visualizar cada um dos pedidos e os itens que foram solicitados nele. Caso o cliente tenha realizado um pedido, mas ainda não o tenha recebido, a visualização dos pedidos e dos itens estará disponível, entretanto, ele não poderá avaliar os produtos solicitados.

Caso o cliente já tenha recebido o pedido, ele poderá dar um *feedback* sobre cada um dos produtos adquiridos, dizendo se o recomenda ou não. Quando o usuário enviar o *feedback* ele receberá uma notificação na aplicação de que sua avaliação foi enviada para o estabelecimento.

Cada produto que o cliente recomenda recebe um incremento no banco de dados que posteriormente será apresentado na listagem de produtos, para que assim os clientes possam verificar quais são os produtos que mais são recomendados para consumo. A figura 32 ilustra parte do funcionamento descrito.

Figura 36 - Histórico e avaliação



Fonte: Do autor

6 AVALIAÇÃO DO PROJETO

O presente estudo teve como objetivo principal o desenvolvimento de uma aplicação híbrida que permitisse aos clientes a realização de pedidos através de um dispositivo móvel, permitindo também ao usuário a possibilidade de definir o idioma que seria utilizado na aplicação. Não se vê atualmente muitas empresas populares oferecendo tais recursos aos seus usuários, portanto, acredita-se que a aplicação acabou por apresentar funcionalidades que seriam diferenciais relevantes para os estabelecimentos que adquirissem o aplicativo.

Um dos pontos mais relevantes do projeto foi o acervo de tecnologias que foram utilizadas durante o desenvolvimento, manipulando recursos e funcionalidades tanto do *backend* quanto do *frontend*, destacando, principalmente, as conexões com serviços externos, como o do *Facebook*, *Google* e do *OneSignal*.

Ao desenvolver o aplicativo para realização de pedidos - que por si só já exigiu bastante tempo de estudo e análise - não foram analisadas a fundo as funcionalidades que deveriam ser implementadas no projeto, fazendo com que alguns atributos fossem adicionados ou retirados durante o desenvolvimento.

Tais alterações impactaram de forma negativa na aplicação, pois a aplicação, o serviço e toda a camada modelo do sistema possuíam classes ou atributos que se relacionavam, ou seja, quando havia a necessidade de adicionar um determinado atributo ou mesmo adicionar ou remover uma determinada classe, seja no *frontend* ou no *backend* da aplicação, sempre tinha que rever a lógica por trás de cada uma das funcionalidades do sistema que tinham alguma relação com aquela classe que foi modificada.

A implementação da listagem e realização de pedidos por empresa é um bom exemplo deste problema, pois, primeiramente a aplicação foi desenvolvida para funcionar em apenas um estabelecimento, posteriormente foi necessário torna-la aplicável à todos os estabelecimentos, necessitando adicionar referências do estabelecimento na classe Pedido e Produto, alterando assim a estrutura no serviço, na aplicação e na modelagem.

Por fim, apesar das adversidades observadas durante o projeto, foi possível alcançar todos os objetivos que foram estabelecidos no início. A realização de pedidos utilizando o dispositivo móvel do cliente em um estabelecimento através da

leitura de um código QR CODE para obter a lista de produtos que o estabelecimento tem a oferecer.

Fornecer ao cliente a possibilidade de escolher o idioma que será utilizado durante a realização do pedido e enviar notificações ao cliente em tempo real sobre o andamento do pedido são os pontos mais fortes da aplicação. Com o *OneSignal* também é possível realizar o envio de notificações para dispositivos específicos que utilizam a aplicação, assim sendo possível para o estabelecimento realizar ações de marketing para um cliente ou um grupo de clientes.

7 CONSIDERAÇÕES FINAIS

O presente estudo teve como objetivo o desenvolvimento de um aplicativo que permitisse a realização de pedidos em um estabelecimento a partir do dispositivo móvel do cliente, sendo possível também definir o idioma que seria utilizado durante o atendimento e realizar o envio de notificações para o usuário sobre o andamento do pedido realizado.

Primeiramente foi realizado um estudo de caso que visava identificar quais são as características em comum que existiam entre os estabelecimentos de Fast-Food, para assim poder realizar um mapeamento do comportamento e das necessidades básicas destes estabelecimentos e aplicá-los em uma modelagem genérica que seria posteriormente utilizada como base para o desenvolvimento do aplicativo.

Para realizar o desenvolvimento do *backend* da aplicação optou-se por desenvolvê-lo utilizando a linguagem JAVA, mais especificamente os recursos oferecidos pela JPA e pelo *Hibernate Validator*, que visam facilitar o mapeamento objeto-relacional da aplicação e validar as informações que seriam persistidas.

Posteriormente ocorreu o desenvolvimento do serviço que seria responsável por receber as informações da aplicação e persisti-las na base de dados. O desenvolvimento deste serviço ocorreu através da utilização do framework Spring, não só pelos recursos descritos no projeto, mas também por já possuir um conhecimento prévio sobre o funcionamento dele. Vale salientar que apesar do *endpoint* do serviço retornar o status correto do resultado da requisição, na aplicação não está sendo tratado diretamente este retorno, assim como as questões de segurança do serviço.

Para o desenvolvimento do aplicativo em si, foi realizado um estudo prévio sobre o Angular 5 para que fosse possível compreender o seu funcionamento, sua estrutura e analisar quais seriam as vantagens de aplicá-lo no projeto. Em seguida, foi realizado um estudo sobre o Ionic 3 que implementa uma estrutura de funcionamento integrada com a estrutura do Angular 5, o que facilitou muito na compreensão do funcionamento do framework Ionic, observando os exemplos e realizando a leitura das informações que estão presentes na documentação oficial do framework.

Durante o tratamento do envio de notificações que deveriam ser entregues ao cliente em tempo real ocorreram algumas adversidades, pois, por se tratar de um banco relacional e as informações do banco de dados não serem replicadas para o aplicativo em tempo real, não seria possível realizar o envio de notificações em tempo real. Como alternativa, optou-se por integrar o serviço de notificação do *OneSignal*, sendo necessário estudar o seu funcionamento, sua documentação e aplicá-lo no projeto para poder analisar se o serviço em si iria suprir esta necessidade, o que de fato aconteceu.

Outro ponto importante que deve ser abordado é que o processo de pagamento foi abstraído, pois para a integração com um serviço de pagamento real havia a necessidade de pagar pelo recurso, assim apenas a parte estética do procedimento foi desenvolvida, apresentando um *loader* que simula para o usuário que a realização do pagamento está sendo efetuada e uma notificação local informando que o pagamento foi realizado com sucesso.

Para fornecer ao usuário a possibilidade de realizar *Login* na aplicação através do Facebook e do Google, foi necessário realizar algumas configurações presentes na documentação oficial dos respectivos sites, que podem ser acessadas através de suas respectivas documentações: *Facebook Developers* (Facebook, 2018) e *Google Developers* (Google, 2018).

Um aspecto importante que vale ser abordado é que a aplicação foi desenvolvida utilizando como banco de dados o banco relacional PostgreSQL, entretanto, pelas características que foram observadas durante o desenvolvimento, a qualidade do projeto aumentaria caso ele tivesse sido desenvolvido utilizando um banco de dados NOSQL com armazenamento em nuvem, como o *Firebase*. O *Firebase* seria aplicado ao projeto por fornecer aos desenvolvedores a possibilidade de trabalhar com dados que são atualizados em tempo real, por facilitar a integração com serviços de autenticação através do e-mail ou mesmo de redes sociais e também por fornecer um serviço de envio de notificações, assim mantendo todas as funcionalidades da aplicação em um único serviço de hospedagem.

Por fim, apesar do aplicativo não ter sido aplicado em um estabelecimento real, o fluxo de funcionamento que foi desenvolvido tendo como base o estudo de caso supriu as necessidades básicas que foram identificadas durante a análise,

entretanto, caso a aplicação fosse comercializada, seria necessário realizar um estudo mais aprofundado sobre todo o contexto do funcionamento do Fast-Food.

8 REFERÊNCIAS

ANDROID. ANDROID. **ANDROID**. Disponível em: <<https://www.android.com>>. Acesso em: 5 Outubro 2017.

ANDROID. Desenvolvedores Android | Android Developers. **ANDROID**. Disponível em: <<https://developer.android.com/index.html?hl=pt-BR>>. Acesso em: 16 Setembro 2017.

APPLE. Apple. **Apple**. Disponível em: <<https://www.apple.com>>. Acesso em: 5 Outubro 2017.

BRAVI. Comparativo: as diferenças entre desenvolvimento nativo, híbrido e web app no contexto mobile. **Bravi - Tecnologia que projeta a sua empresa para o futuro**, 2015. Disponível em: <<http://blog.bravi.com.br/comparativo-as-diferencas-entre-desenvolvimento-nativo-hibrido-e-web-app/>>. Acesso em: 9 Novembro 2017.

APACHE CORDOVA, A. Architectural overview of Cordova platform – Apache Cordova. **APACHE CORDOVA**, 2015. ISSN <https://cordova.apache.org/docs/en/latest/guide/overview/>. Acesso em: 20 Junho 2017.

DEERING, S. Do you know what a REST API is? **sitepoint**. Disponível em: <<https://www.sitepoint.com/developers-rest-api/>>. Acesso em: 9 Novembro 2017.

DEITEL, H. M.; DEITEL, P. J. **Java: Como Programar**. [S.l.]: Pearson Education, 2009.

DRIFTY. Página Inicial. Ionic Framework. **Ionic Framework**. Disponível em: <<http://ionicframework.com>>. Acesso em: 20 Junho 2017.

EXAME. Estatísticas de uso de celular no Brasil | EXAME. **Exame**, 2016. Disponível em: <<https://exame.abril.com.br/negocios/dino/estatisticas-de-uso-de-celular-no-brasil-dino89091436131/>>. Acesso em: 30 Novembro 2017.

FACEBOOK DEVELOPERS – Facebook Developer Docs | Facebook APIs, SDKs & Guides. **FACEBOOK DEVELOPERS**. Disponível em: <<https://developers.facebook.com/docs>>. Acesso em: 10 Abril 2018

FARIA, T. **Java EE 7 com JSF, Primefaces e CDI**. [S.l.]: Novatec Editora, 2015.

GUEDES, G. T. A. **UML 2: uma abordagem prática**. 2ª. ed. São Paulo: Novatec Editora, 2011.

GOOGLE DEVELOPERS – Home Page. **GOOGLE DEVELOPERS**. Disponível em: <<https://developers.google.com/>>. Acesso em: 04 Abril 2018.

IONIC. Ionic Documentation Overview. **IONIC**. Disponível em: <<http://ionicframework.com/docs/v1/overview/>>. Acesso em: 26 Agosto 2017.

JAVA. **Java**. Disponível em: <https://www.java.com/pt_BR/>. Acesso em: 11 Novembro 2017.

JSON. JSON. **JSON**. Disponível em: <<http://www.json.org/json-pt.html>>. Acesso em: 26 Setembro 2017.

LECHETA, R. R. **Desenvolvendo para iPhone e iPad**. [S.l.]: Novatec Editora, 2012.

LECHETA, R. R. **Google Android: Aprenda a criar aplicações para dispositivos móveis com o Android SDK**. 3ª. ed. [S.l.]: Novatec Editora, 2013.

MICROSOFT. Microsoft – Official Home Page. **MICROSOFT**. Disponível em: <<https://www.microsoft.com>>. Acesso em: 5 Outubro 2017.

MILANI, A. **PostgreSQL – Guia do Programador**. 1ª. ed. São Paulo: Novatec Editora, 2008.

MILANI, A. **Programando para iPhone e iPad: Aprenda a construir aplicativos para o iOS**. [S.l.]: Novatec Editora, 2012.

MONACO, T.; DO CARMO, R. M. **Desenvolvendo aplicações para Windows Phone**. [S.l.]: Brasport, 2012.

MONTEIRO, J. B. **Google Android**: Crie aplicações para celulares e tablets. [S.l.]: Casa do Código, 2012.

NODEBR. O que é Node.js? - NodeBR. **NodeBR**. Disponível em: <<http://nodebr.com/o-que-e-node-js/>>. Acesso em: 6 Novembro 2017.

PHONEGAP. PHONEGAP. **PHONEGAP**. Disponível em: <<https://phonegap.com>>. Acesso em: 19 Outubro 2017.

POSTGRESQL. PostgreSQL: The world's most advanced open source database. **PostgreSQL**. Disponível em: <<https://www.postgresql.org>>. Acesso em: 9 Setembro 2017.

RELEVANT SOFTWARE – Top 6 reasons to use Angular 5 framework for your project | Relevant Software. **RELEVANT SOFTWARE**. Disponível em: <<https://relevant.software/blog/top-6-reasons-to-use-angular-5-framework-for-your-project/>>. Acesso em: 02 Julho 2018.

RESTAPITUTORIAL. What is REST? **Rest Api Tutorial**. Disponível em: <<http://www.restapitutorial.com/>>. Acesso em: 9 Novembro 2017.

SPRING. SPRING. **SPRING**. Disponível em: <<https://spring.io>>. Acesso em: 31 Outubro 2017.

TRUSTTHISPRODUCT – Home Page. **TRUSTTHISPRODUCT**. Disponível em: <<https://trustthisproduce.com>>. Acesso em: 15 Outubro 2017.

VASCONCELLOS, L. **Apps Híbridas com Cordova e Ionic**. Disponível em: <<https://medium.com/@lfv89/nessa-s%C3%A9rie-divida-em-3-partes-vou-falar-um-pouco-mais-a-fundo-sobre-desenvolvimento-h%C3%ADbrido-914f22453c83>>. Acesso em: 20 Junho 2017.

WALLS, C. **Spring in Action**. [S.l.]: Manning, 2014.

WARGO, J. M. **Apache Cordova 3 Programming**. [S.l.]: Addison-Wesley Professional, 2013.

XAMARIN. Mobile App Development & App Creation Software - Xamarin. **XAMARIN**. Disponível em: <<https://www.xamarin.com>>. Acesso em: 19 Outubro 2017.