LUCAS DE SOUZA FELÍCIO

KATARINA MOBILE: Projeto De Uma Aplicação Para O Gerenciamento De Lançamentos De Pedidos Do Sistema Katarina

JI-PARANÁ
2020

LUCAS DE SOUZA FELÍCIO

KATARINA MOBILE: Projeto De Uma Aplicação Para O Gerenciamento De Lançamentos De Pedidos Do Sistema Katarina

Monografia apresentada à Banca Examinadora do Curso de Sistemas de Informação do Centro Universitário São Lucas, como requisito de aprovação para obtenção do Título de Bacharel em Sistemas de Informação.

Orientador: Prof. Esp. Willian Fachetti

JI-PARANÁ
2020

Dados Internacionais de Catalogação na Publicação
Gerada automaticamente mediante informações fornecidas pelo(a) autor(a)

F311k Felício, Lucas de Souza.

Katarina Mobile: projeto de uma aplicação para o gerenciamento de lançamento de pedidos do sistema Katarina / Lucas de Souza Felício -- Ji-Paraná, RO, 2020.

129 p.

Orientador(a): Prof. Willian Fachetti

Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) - Centro Universitário São Lucas

1. Sistema de autoatendimento. 2. Atendimento ao consumidor. 3.Aplicativo de auto serviço. I.Fachetti, Willian. II. Título.

CDU 004.4

Bibliotecário(a) Alex Almeida CRB 11.8537

LUCAS DE SOUZA FELÍCIO

KATARINA MOBILE: Projeto De Uma Aplicação Para O Gerenciamento De Lançamentos De Pedidos Do Sistema Katarina

Monografia apresentada à Banca Examinadora do Curso de Sistemas de Informação do Centro Universitário São Lucas Ji-Paraná, como requisito de aprovação para obtenção do Título de Bacharel em Sistemas de Informação.

Orientador: Prof. Esp. Willian Fachetti

Ji-Paraná, 10 de julho de 2020

Avaliação/Nota: 9.7

BANCA EXAMINADORA

_____ Prof. Esp. Willian Alves de O. Fachetti	Centro Universitário São Lucas Ji-Paraná
_____ Prof. Esp. Hailton Cezar A. dos Reis	Centro Universitário São Lucas Ji-Paraná
_____ Prof. Esp. José Rodolfo M. Olivas	Centro Universitário São Lucas Ji-Paraná

A Erikson, meu namorado, amigo e companheiro. Que com seu amor me deu fôlego para seguir e completar minha jornada acadêmica.

Agradeço, primeiramente a toda minha família, em especial aos meus pais Enildo e Rosana, que me deram todo o suporte e apoio ao longo de toda a minha jornada acadêmica.

Agradeço ao meu companheiro, Erikson Rodrigues, que me apoiou e me incentivou a dar continuidade nos meus trabalhos.

Agradeço aos meus melhores amigos, Wesley, Guilherme, Karine, Sharon, Pamela e Poliana, que também me apoiaram e estão próximos em mais essa etapa em minha vida.

Agradeço aos meus amigos e colegas da faculdade que compartilharam esses anos de batalhas e conquistas juntos.

Agradeço ao Ewerton, dono e fundador da NorteSystem, amigo e companheiro de trabalho, que me incentivou e deu suporte para a realização desse projeto.

Enfim, agradeço a todos que de alguma forma tenha feito parte dessa jornada e de alguma forma tenha contribuído para a realização desse projeto.

RESUMO

Empresas de modo geral, mas especificamente aquelas que possuem atendimento diretamente com o consumidor final, estão em uma constante busca pela melhoria em seus serviços e processos de atendimento, a fim de garantir qualidade e satisfação de seus clientes. Contudo estabelecimentos que utilizam sistemas de gerenciamento instalados em terminais fixos, como por exemplo o sistema Katarina, podem ter essa evolução comprometida. Com isso essas empresas buscam alternativas de soluções que garantam flexibilidade. Diante disso, este trabalho, fazendo o uso de pesquisas bibliográficas, teve como objetivo principal desenvolver uma aplicação de software que provenha a capacidade de gerenciar pedidos em dispositivos móveis e que se integre ao ERP Katarina. Através dessa solução é possível gerenciar o atendimento de pedidos, dispensando o uso de terminais fixos, assim garantindo agilidade e simplicidade no processo de atendimento. Assim sendo este projeto resultou no desenvolvimento de uma api e de um aplicativo móvel para o gerenciamento de pedidos, do sistema Katarina, em dispositivos móveis Android utilizando as tecnologias do *framework* React Native e NodeJS.

Palavras-Chave: React Native. NodeJS. Aplicativo móvel. Android.

ABSTRACT

Companies in general, but those that have direct service to the final consumer, are in a constant search for improvement in their services and service processes, in addition to guaranteeing the quality and satisfaction of their customers. However, the management systems used in fixed terminals, such as the Katarina system, may have this evolution compromised. Thus, these companies seek alternative solutions that guarantee flexibility. Therefore, this work, using bibliographic research, had as main objective to develop a software application that provides the ability to manage orders on mobile devices and that integrate the ERP Katarina. Through this solution it is possible to manage order fulfillment, dispensing with the use of fixed terminals, as well as agility and simplicity in the service process. Therefore, this project resulted in the development of an API and a mobile application for order management, of the Katarina system, on Android mobile devices using the technologies of the React Native and NodeJS framework.

Keywords: React Native. NodeJS. Mobile Application. Android.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
DER	Diagrama de Entidade Relacionamento
DOM	*Document Object Model*
EDVAC	*Eletronic Discrete Variable Computer*
ENIAC	*Eletronic Numeral Integrator And Computer*
ER	Entidade Relacionamento
EUP	*Enterprise Unified Process*
GUI	*Graphical User Interface*
HTTP	*Hypertext Transfer Protocol*
IDC	*International Data Corporation*
JSON	*JavaScript Object Notation*
JSX	*JavaScript XML*
MER	Modelo de Entidade Relacionamento
NPM	*Node Package Manager*
PDV	Ponto de Venda
RUP	*Rational Unifield Process*
SGBD	Gerenciamento de Banco de Dados
SGBDOR	Sistema de Gerenciamento de Banco de Dados Objeto-Relacional
SQLS	*Tructured Query Language*
UC	*User Case*
UI	*User Interface*
UML	*Unified Modeling Language*
UP	*Unified Process*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas a humanidade tem vivido um intenso crescimento irreversível das tecnologias, comenta Godin (2017). Essas tecnologias ganham a cada dia mais espaço no dia a dia das pessoas. Segundo Godin (2017) seja qual for a tecnologia abraçada por uma cultura nunca será abandonada, exceto por uma mais avançada. Uma tecnologia adotada por uma sociedade pode ser substituída por uma alternativa melhor, porém jamais andará para trás. Tais tecnologias estão envolvidas em diversos setores e auxiliam em várias áreas, tais como educação, saúde, telecomunicações, politica, entretenimento e lazer, economia e negócios, dentre muitas outras, fornecendo soluções eficientes, seguras e, principalmente, automatizadas.

Dentre as tecnologias de informação que se popularizaram nos últimos anos, estão os dispositivos móveis, principalmente os *smartphones*. De acordo com dados da *International Data Corporation* (IDC, 2019a), a venda de *smartphones* superou a marca de 341,2 milhões de unidades enviadas pelos fornecedores no segundo trimestre de 2018. Contudo a previsão para o segundo trimestre de 2019 é uma queda de 2,7% na remessa dos fabricantes, representado um total de 332,2 milhões de unidades. Apesar da retração no mercado de *smatphones* no ano de 2019, o mercado mundial conta com mais de 5,1 bilhões de unidades, segundo rastreador da *GSMA Association* (GSMA, 2019).

Com essa nova realidade e com inovações constantes da tecnologia, a sociedade está cada vez mais emergida nesses avanços e impõe que as empresas também se integrem. Uma empresa nos dias atuais, dificilmente, se mantém por muito tempo no mercado sem usar a tecnologia para oferecer produtos, serviços e atendimento diferenciado aos seus clientes (GESTAOCLIK, 2018).

Porém uma pesquisa realizada pela IDC demonstrou que 42% das empresas brasileiras avaliadas possuem políticas de alinhamento de investimento em tecnologia. A pesquisa revela ainda que 34% das empresas preocupam-se em investir em tecnologias móveis, colocando o item em 4ª posição num ranking que avalia termos de importância estratégica para os negócios (IDC, 2017).

Segundo a pesquisa da IDC (2017), mostra que a adoção de tecnologias móveis para execução de funções de campo, como vendas e serviços, é de pouco mais de 52% das empresas pesquisadas. A pesquisa demonstra também que 78% dos colaboradores concordam ou concordam totalmente que o uso de dispositivos moveis aumentou a produtividade de suas atividades. E dentre os dispositivos mais utilizados pelos colaboradores estão os *smartphones*, que são utilizados por 75% deles (IDC, 2017).

Diante deste cenário, oferecer uma solução tecnológica e de preferência móvel, seja interessante aos usuários do sistema Katarina. Uma vez que, utilizando este tipo de tecnologia suas atividades sejam desempenhadas com mais eficiência e produtividade. Sendo assim, este trabalho tem por objetivo apresentar o desenvolvimento de uma aplicação para dispositivos móveis, mais precisamente para *smartphones*, que permitirá aos usuários lançarem os pedidos realizados pelos consumidores direto neste aplicativo, assim agilizando o processo de coleta de pedidos.

2 PROBLEMATIZAÇÃO

Um atendimento eficiente e com qualidade é capaz de levar uma empresa a um sucesso promissor, visto que leva a satisfação do cliente em consideração em virtude da solução de problemas ou na clareza quanto a informação ou serviços prestados (DUARTE, 2016).

Em qualquer estabelecimento, mas principalmente nos que há atendimento direto ao consumidor, como bares, restaurantes, pizzarias, e outros, um atendimento de qualidade é fundamental para garantir a satisfação do consumidor. Neste contexto, o garçom torna-se um agente influente na percepção final que o consumidor terá do estabelecimento, por estar diretamente relacionado com o cliente desde o pedido até o fechamento da conta (MAGALHÃES, 2018).

Com intuído de assegurar a qualidade no atendimento ao consumidor, investir em ferramentas e sistemas para os garçons é essencial. Pois com o uso de tecnologias o atendimento é completamente transformado, tornando-o mais eficiente, seguro, rápido e dinâmico, agilizando o trabalho dos garçons e garantindo um bom atendimento aos clientes (MAGALHÃES, 2018).

Diante deste contexto, através do uso de tecnologias inovadoras, como o React.js e o Node.js, é possível desenvolver uma aplicação mobile e uma API (*Application programming interface*) que se integrem ao sistema Katarina e possua conexão com seu banco de dados, PostgreSQL, para que permita garçons-usuários lançarem e acompanharem os pedidos dos consumidores?

3 OBJETIVOS

3.1 OBJETIVO GERAL

Desenvolver uma aplicação mobile e uma API (*Application Programming Interface*) que integre ao sistema Katarina, a fim de possibilitar aos garçons o gerenciamento dos lançamentos dos pedidos dos consumidores nas mesas e comandas.

3.2 OBJETIVOS ESPECÍFICOS

De forma conjunta ao objetivo geral, pode-se destacar alguns dos objetivos específicos, tais como:

- Desenvolver uma aplicação de fácil utilização;
- Permitir ao garçom autonomia para lançamentos e visualizações dos pedidos de forma prática;
- Simplificar as visualizações dos pedidos;

3.3 DELIMITAÇÃO DO ESTUDO

O projeto se propõe a elaborar e desenvolver um projeto de um aplicativo móvel e uma API que se integrarão ao sistema Katarina, já em uso, em todos os clientes da NorteSystem, no setor de atendimento.

Serão abordadas as funcionalidades de lançamento e visualização de pedido a nível de mesas e comandas.

O uso do aplicativo móvel será restrito aos usuários do sistema Katarina, que possuam credenciais no mesmo.

4 JUSTIFICATIVA

Segundo o ranking de reclamações do site ReclameAqui (2019), o mau atendimento está em segundo lugar nas categorias de bares e restaurantes. Reclamações deste tipo podem estar relacionados a diversos fatores, como cordialidade dos funcionários, atenção no momento do entendimento, agilidade e exatidão na preparação dos pedidos (BRACHOT, 2019).

Um mau atendimento pode causar grandes prejuízos financeiros a um estabelecimento, além de poder ter sua reputação manchada em redes sociais ou possuir reclamações compartilhadas por consumidores em sites de reclamação, como o ReclameAqui (ECOMANDA, 2019).

Segundo Brachot (2019), para evitar reclamações algumas medidas devem ser adotadas, como oferecer treinamento adequado aos funcionários, principalmente aos garçons, zelar pela qualidade dos produtos oferecidos, possuir uma boa estrutura física, zelar pela limpeza e oferecer experiências diferenciadas aos clientes. Além disso, é importante adotar ferramentas de automação eficientes que agilizem os processos de atendimento, principalmente na retirada dos pedidos (ECOMANDA, 2019).

Deste modo, fazendo o uso de um aplicativo móvel, por exemplo, o atendente poderá no momento de coletar o pedido agilizar o processo, tornando o atendimento mais eficiente e prático.

5 REFERENCIAL TEÓRICO

O início da era dos computadores eletrônicos digitais deu-se por volta dos anos de 1945, com o lançamento do ENIAC (*Eletronic Numeral Integrator And Computer)*. O ENIAC foi desenvolvido nas dependências da Universidade da Pensilvânia, nos Estados Unidos, com o objetivo de realizar cálculos de balística e utilizado durante a Segunda Guerra Mundial (CARVALHO; LORENA, 2017, p. 36). A programação do ENIAC era realizada de forma manual através de cabos que eram conectados e desconectados.

O primeiro computador totalmente eletrônico e digital programável foi o Colossus, desenvolvido pelo engenheiro Thomas Flowers e tinha como objetivo quebrar códigos utilizados pelo exército alemão durante a Segunda Guerra Mundial. Contudo foi o EDVAC (*Eletronic Discrete Variable Computer)*, desenvolvido por Jon Von Neumann, por volta dos anos de 1951, no Instituto de Estudos Avançados de Princeton, nos Estados Unidos, que introduziu, de forma pioneira, o uso de programas previamente armazenados em fitas magnéticas e utilizando internamente códigos binários (CARVALHO; LORENA, 2017, p. 37).

Ao longo dos anos seguintes muitos outros computadores foram projetados e construídos, contribuindo assim para o avanço tecnológico da época. Tais avanços levaram a uma grande redução do custo dos dispositivos e permitiram o desenvolvimento e construção de computadores menores e com mais recursos, principalmente, de processamento e de armazenagem. Resultando na criação de microcomputadores, chamados de computadores pessoais, e de supercomputadores com grande capacidade de processamento e memória de armazenagem (CARVALHO; LORENA, 2017, p. 38).

Segundo Sommerville (2007), a evolução dos *softwares* fora igualmente significativa ao longo das épocas, acompanhando a evolução surpreendente dos hardwares. A capacidade e a necessidade de criar grandes e complexos sistemas aumentou. Serviços e infraestrutura nacionais contam largamente com sistemas de computadores muito complexos e confiáveis.

Para Rezende (2002), os *softwares* se tornaram essenciais às nossas vidas, seja no dia a dia de pessoas comuns, que nem percebe o seu uso, seja em

atividades muito mais complexas. Hoje em dia é imprescindível o uso de *softwares* em setores como o das telecomunicações, comércio, setor financeiro, ensino e aprendizagem, no armazenamento e busca por informações, no meio cientifico e tecnológico e até mesmo nas guerras.

Contudo a construção de um sistema de informações é, sem dúvidas, uma tarefa muito complexa e à medida que o sistema cresce, a complexibilidade cresce junto (BEZERRA, 2007, p. 2).

Segundo Bezerra (2007), assim como na engenharia civil, que engenheiros e arquitetos constroem plantas, para a construção de sistemas de *software* também é necessário um planejamento inicial. E essa necessidade leva ao conceito de modelo. Os modelos, ainda segundo Rezende, podem ser vistos como uma ilustração elaborada de um sistema a ser construído.

5.1 RATINAL UNIFIED PROCESS (RUP)

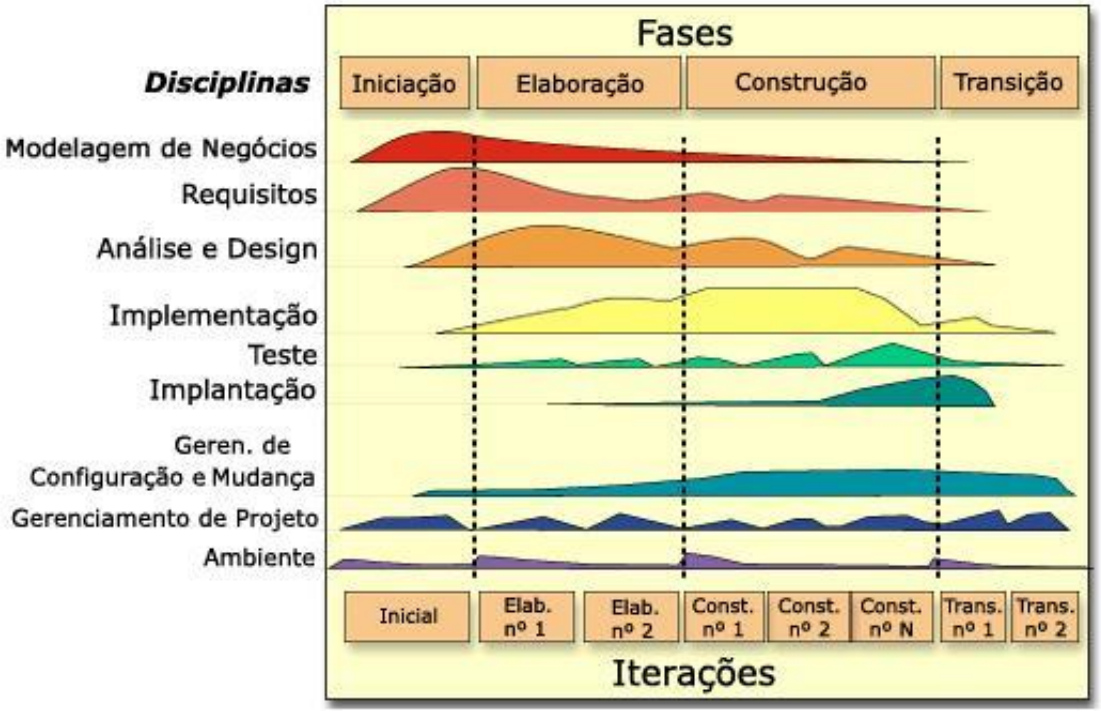
O *Rational Unifield Process* (RUP), é um processo dinâmico e interativo que proporciona a descrição sequencial de um problema, que organiza toda a solução, que desenvolve o *software* e que testa o produto final. O RUP possui flexibilidade para mudanças de objetivos, como a acomodação de novos requisitos e também concede ao projeto identificar e solucionar riscos no início (REZENDE, 2002, p.176).

Inicialmente desenvolvido pela Rational Corporation, e adquirido pela IBM, o RUP no início era conhecido como Processo Unificado, ou UP (do inglês *Unified Process)*. Seu objetivo, da mesma forma que ocorre em outros paradigmas da engenharia de *software*, é certificar que a produção de *software* tenha alta qualidade e que se mantenha de acordo com as necessidades dos usuários finais (SBROCCO, MACEDO, 2012, p. 67).

O RUP possui duas dimensões, conforme Figura 1. Uma dimensão no eixo vertical e uma no eixo horizontal (SBROCCO, MACEDO, 2012, p. 67), sendo:

- *Eixo vertical:* apresenta o núcleo do fluxo do trabalho (workflow) do processo, compreendendo grupos de atividades logicamente distribuídas.
- *Eixo horizontal:* representa o tempo e os vários aspectos do ciclo de vida do projeto.

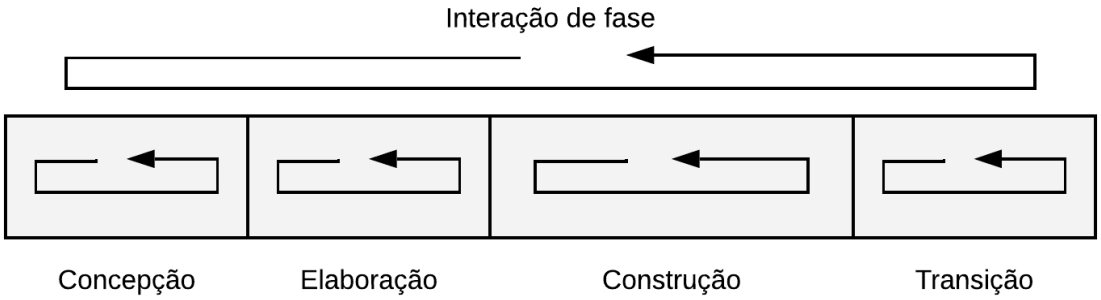
Figura 1 – Estrutura geral do RUP em duas dimensões



Fonte: Ratinal (1998)

No eixo horizontal do RUP encontra-se as interações e as fases do modelo. O RUP possui quatro fases em seu total, sendo elas: concepção, elaboração, construção e transição (SMMERVILLE, 2017, p. 55).

Figura 2 – Fases no Rational Unified Process



Fonte: Adaptação de Sommerville (2007), Figura 4.12

As duas primeiras fases do RUP, como demonstrado na Figura 2, estão relacionadas a modelagem do projeto. Na primeira fase, concepção, estabelece-se um *business case*, ou caso de negócio em português. Neste caso de negócio deve-se identificar todas as entidades externas (pessoas e sistemas), que irão relacionar-se com o sistema, e a atribuir essas relações. A Fase de elaboração tem por objetivo estabelecer um plano de projeto e compreender o domínio do problema. A conclusão desta fase resulta em um modelo de requisitos, apresentados

graficamente através de casos de uso, utilizando a conotação da *Unified Modeling Language (UML)*, detalhamento da arquitetura e um plano de desenvolvimento para o *software* (SOMMERVILLE, 2007, p. 54).

A terceira fase, de construção, está sobretudo relacionada a implementação, testes e integração. A conclusão desta fase resulta em um produto de *software* em funcionamento e a documentação associada e pronta para ser disponibilizada aos usuários (SOMMERVILLE, 2007, p. 55).

A última fase do RUP, transição, muitas vezes ignorada pela a maioria dos modelos de processo de *software*, está relacionada com a transferência do sistema da equipe de desenvolvimento para o cliente, bem como a entrada do sistema em funcionamento no ambiente real. Ao fim de todas as fases tem-se um sistema de *software* documentado e operando corretamente em seu ambiente operacional (SOMMERVILLE, 2007, p. 55).

A visão estática do RUP, eixo vertical, enfoca as atividades que ocorrem durante o processo de desenvolvimento. Elas são denominadas *workflows*[1]. Existem nove workflows, sendo seis atividades principais de engenharia de *software* e três principais de apoio. O RUP foi construído em conjunto com a UML e, por esse motivo, a descrição dos *workflows* são dirigidos em termos dos modelos da UML. Todos os workflows do RUP podem ativados em todas as fases do processo, contudo cada atividade será mais requisitada dependendo em que fase se encontra o projeto (SOMMERVILLE, 2007, p. 55). Os noves workflows do RUP são:

- Modelagem de negócio: Objetiva a compreensão comum entre os envolvidos sobre quais processos de negócio serão compreendidos no projeto;
- Requisitos: Visa entender os requisitos funcionais que serão contemplados pelo sistema. Descreve, baseando-se em casos de uso, tais requisitos;
- Análise e projeto: Essas atividades apresentam as várias visões da arquitetura do sistema. Objetivam detalhar e compreender os casos de usos. Resultam em apresentar os modelos do sistema focados na futura implementação;

1Termo em inglês que significa fluxo de trabalho, em português

- Implementação: Nesta atividade é de fato construído, testado e integrado o *software*. Visa a estruturação e produção dos códigos de programação para implementar as classes, os objetos e seus componentes. Nesta atividade elabora-se também testes dos componentes em termos de unidade;
- Teste: Apresenta os casos de teste, procedimentos e medidas de acompanhamento de erros. Objetiva examinar se os requisitos funcionais foram atendidos. Também avalia falhas que deverão ser removidas antes da implantação;
- Implantação: Abrange a configuração do sistema a ser entregue. Objetiva elaborar resultados, *releases*, do produto e entrega-los ao cliente final;
- Gerenciamento da configuração: Também chamado de gerenciamento de mudanças. Este workflow é de apoio e tem por objetivo controlar as modificações e manter a integridade dos artefatos e produtos do sistema. Controla e garante a integridade dos diversos releases produzidos, assegurando que os artefatos não conflitem entre si;
- Gerenciamento de projeto: Também é considerado um workflow de apoio e tem por objetivo descrever as estratégias adotadas para a gestão do projeto. Abrande um *framework* para a gestão total do projeto, com recomendações para organização das tarefas, alocação dos recursos e gerência de riscos;
- Ambiente: Nono e último workflow, e o terceiro de apoio. Esta atividade considera a infraestrutura necessária para o desenvolvimento do sistema. Abrange também o detalhamento de todas as ferramentas necessárias para a execução de todos os processos do *software* (REZENDE, 2002, p.178-180).

5.2 UNIFIED MODELING LANGUAGE (UML)

A *Unified Modeling Language* (UML) surgiu a partir de vários eventos históricos e principalmente com o intuito de padronizar o paradigma orientado a objetos. No início da década de 1980 a orientação a objetos já era bem reconhecida e contava com uma serie de diversidades de metodologias e notações que

representavam sistemas de *software* de diferentes maneiras (SBROCCO, 2011, p. 24).

Nessa época, toda vez que se iniciava um projeto, era necessário escolher um método de desenvolvimento específico e quase sempre era preciso treinar a equipe responsável pelo projeto. Com intuito de divulgar os benefícios do uso do paradigma orientado a objetos, observou-se que era fundamental a criação de uma padronização para o mesmo (SBROCCO, 2011, p. 24).

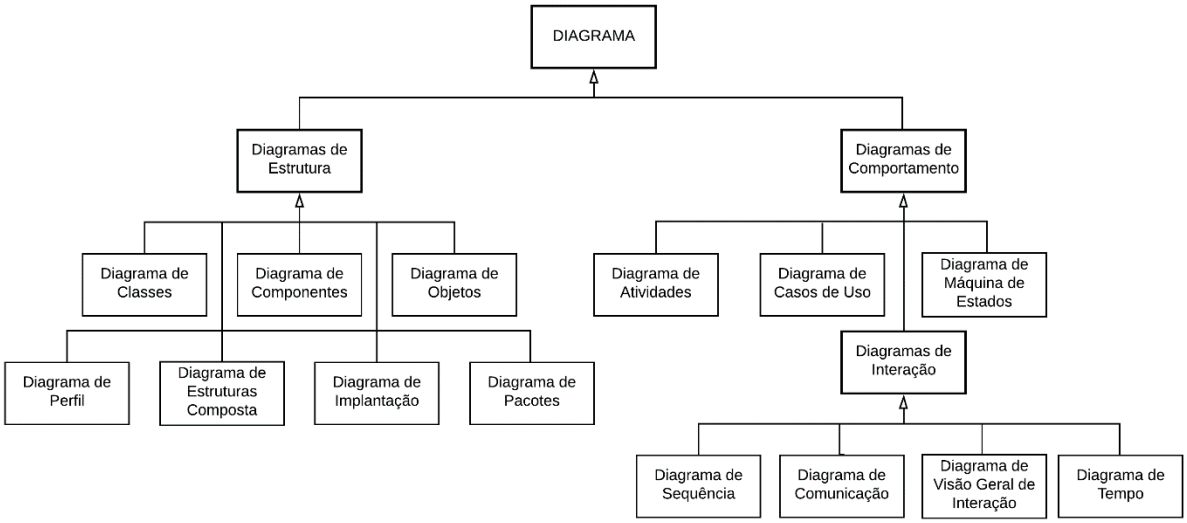
A construção da UML contou com várias contribuições de diversos pesquisadores, dentre os principais destacam-se Grandy Booch, James Rumbaugh e Ivar Jacason. No processo de elaboração da UML, esses pesquisadores procuraram aproveitar elementos que se destacavam em notações preexistentes, principalmente de técnicas propostas por eles mesmos (BEZERRA, 2007, p. 15).

Portanto a notação estabelecida para UML é uma junção dessas diversas notações preexistentes, com elementos removidos e outros adicionados com objetivo de torna-la mais relevante.

Segundo Bezerra (2007), “a UML é uma *linguagem visual* que define sistemas orientados a objetos”. Ou seja, a UML é uma notação que define elementos gráficos que são utilizados no processo de modelagem de sistemas. Assim através desses elementos gráficos pode-se representar os conceitos do paradigma orientado a objetos e construir diagramas que denotam diversas perspectivas de um sistema.

No processo de desenvolvimento que utilize a UML, são produzidos diversos tipos de documentos tanto textuais quanto gráficos. Segundo a notação da UML esses documentos são denominados artefatos de *software*, o qual compõem as visões do sistema (BEZERRA, 2007, p. 17). Atualmente a UML conta com quatorze diagramas, ilustrados na Figura 3, que individualmente representam uma perspectiva parcial do sistema.

Figura 3 – Taxonomia dos diagramas de estrutura e comportamento



Fonte: Adaptação de UML Specification 2.5.1, Figura A.5[2]

Os diagramas de estrutura demonstram estruturas estáticas dos objetos em um sistema, ou seja, descrevem esses elementos em uma especificação que independe do tempo. Os elementos em um diagrama de estrutura retratam os conceitos de um aplicativo e podem conter conceitos abstratos, do mundo real, ou de implementação. Já os comportamentos dinâmicos são representados nos diagramas de comportamento, que engloba os métodos, colaborações, atividade e histórico de estados (OMG, 2017, p. 685).

5.3 PROTOTIPAÇÃO

A prototipação tem por finalidade permitir que usuários consigam ter experiencias direta com as interfaces do *software*, pois descrições textuais e diagramas não são bons o suficiente para expressar os requisitos (SOMMERVILLE, 2007, p. 253).

Segundo Bezerra (2007), protótipos podem ser desenvolvidos para representar telas de entrada, telas de saída, subsistemas ou mesmo para o sistema como um todo. Qualquer particularidade que necessite ser mais bem entendido pode ser um alvo potencial de prototipagem.

Na disciplina de prototipagem, posteriormente ao levantamento de requisitos, um protótipo pode ser construído para ser usado no processo de validação, que é

2 Adaptação da Figura A.5 da UML Specification 2.5.1, disponível em: https://www.omg.org/spec/UML/2.5.1/PDF, feito com Lucidchart sobre a figura similar do verbete UML, em inglês.

quando o protótipo é submetido a avaliação de um ou mais usuários, que fazem críticas a respeito de uma ou outra característica. A pós este processo o protótipo é corrigido ou refinado e em seguida submetido ao processo de validação novamente. Esse processo de revisão e refinamento dura até que os usuários aceitem o protótipo. Por isso, a técnica de prototipagem tem por finalidade assegurar que os requisitos do sistema foram realmente bem entendidos e atentem as necessidades dos usuários (BEZERRA, 2007, p. 42).

5.4 DISPOSITIVOS MÓVEIS

Antes, os computadores eram máquinas gigantescas que ocupavam grandes espaços com fins de agilizar tarefas lógicas em instituições de pesquisas, grandes empresas, universidades e entidades governamentais (ALECRIM, 2015, p. 1). Contudo, através da evolução das tecnologias, essas máquinas evoluíram ao ponto de ficarem compactas, confiáveis, potentes, práticas, fáceis de usar, possibilitando, inclusive, serem levadas a qualquer lugar e ser utilizada por qualquer pessoa. Tais tecnologias flexíveis são conhecidas como dispositivos móveis.

Segundo Lee, Schneider e Schell (2005), um dispositivo móvel tem como principais características a portabilidade, a compactabilidade, ser altamente utilizável, funcional e possibilitar fácil conectividade e comunicação com outros dispositivos. Uma ou mais de suas características podem ser flexibilizadas pelos usuários, ou seja, um dispositivo com redução de portabilidade, mas com maior funcionalidade, pode ser aceito pelos usuários. Entretanto, o exagero de uma característica e a falta de outra, que prejudique o seu uso, não justifica seu aceite (LEE, SCHEIDER, SCHELL, 2005, p. 2).

A popularidade do uso de dispositivos móveis, devido seus benefícios, é perceptível através dos números que representam o mercado desses aparelhos. Segundo dados da GSMA Association[3], mostra que até o final do ano de 2018 haviam aproximadamente 9.1 bilhões de dispositivos mobile conectados, sendo que 60% são apenas smartphones, representado um total de 5.1 bilhões.

Uma pesquisa realizada pela FGV EAESP[4], demonstra que até o mês de maio de 2019, o Brasil, possuía cerca de 230 milhões de smartphones. E até o final

3 Entidade que representa dos interesses das operadoras moveis em todo o mundo (GSMA, 2019)

4 Escola de Administração de Empresas de São Paulo da Fundação Getúlio Vargas

do ano de 2019 terá cerca de 235 milhões de dispositivos smartphones circulando (MEIRELLES, 2019, p. 7).

Galvão (2019) comenta que cada vez mais somos dependentes dos smartphones, pois através deles, hoje, guardamos imagens, vídeos, nos comunicamos através de mensagens de texto instantâneas, isso tanto na vida particular quanto na profissional. E com o aumento do mercado mobile, oportunidades de crescimento e reconhecimento surgem para negócios locais de pequeno porte e empresas de médio ou grande porte (CASTRO, 2014, p. 1). Empresas que utilizam recursos mobiles ou possuem seus próprios aplicativos, aumentam a credibilidade e trabalham sua imagem no mercado, demonstrando que estão inseridas e atualizadas neste universo.

5.5 APLICATIVOS MÓVEIS

Segundo Zanini (2017), aplicativos mobiles são *softwares* desenvolvidos especialmente para dispositivos móveis.

A quantidade de pessoas acessando a Internet e ou utilizando aplicativos em dispositivos móveis, principalmente em smartphones, é tão grande, que quase alcança o acesso por um computador tradicional desktop ou notebook. E, neste cenário de bilhões de dispositivos móveis, quem domina o mercado de sistema operacional em uso é o Android (CORDEIRO, 2019, p. 1).

De acordo com dados do rastreador mundial de telefonia móvel, da *International Data Corporation* (IDC, 2019b), apresentados na Tabela 1, aponta que o sistema operacional Android, no ano de 2018, representava 85.1% de todo o mercado, seguido pelo sistema operacional da Apple, o iOS, que ocupava 13.0% do mercado mundial de dispositivos. A pesquisa ainda prevê que ao final do ano de 2019 a porcentagem de dispositivos que utilizam o sistema operacional Android salte para 87% e o iOS reduza sua participação para 13%. E a tendência é que o Android até o ano de 2023 atinja 87.3% do mercado mundial e o iOS reduza ainda mais sua participação, representado apenas 12.6% de todo o mercado de dispositivos móveis (IDC, 2019b, p. 1).

Tabela 1 – Previsão mundial de participação de SO mobile no mercado

Ano	2017	2018	2019	2020	2021	2022	2023
Android	85.1%	85.1%	87.0%	87.0%	87.2%	87.3%	87.4%

iOS	14.7%	14.9%	13.0%	13.0%	12.8%	12.7%	12.6%
Outras	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Total	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%

Fonte: Adaptação de IDC (2019b)

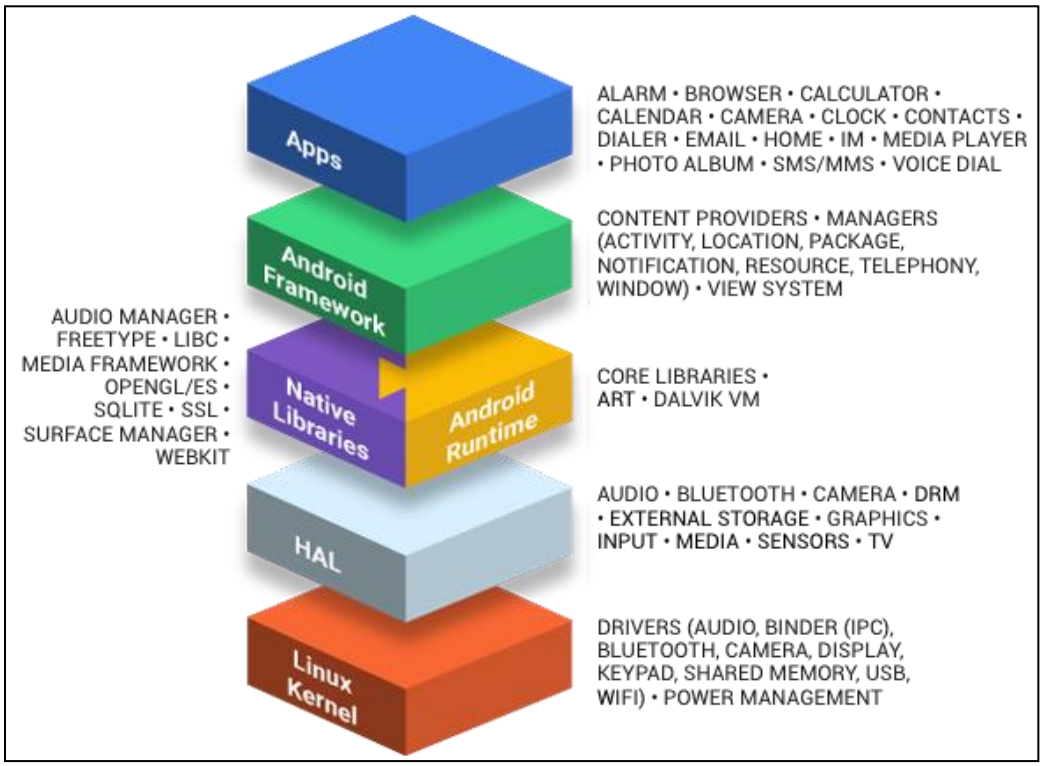
5.5.1 Android

O sistema operacional Android foi desenvolvido pela Android Inc. em Palo Alto, Califórnia. A Android Inc. foi fundada por Andy Rubin, Rich Miner, Nick Sears e Chris White, em outubro de 2003. O Objetivo inicial da Android Inc. era desenvolver um sistema operacional para as câmeras digitais. Contudo, quando perceberam que o mercado de câmeras não era grande o suficiente, mudaram o objetivo para desenvolver o sistema operacional voltado para os smartphones para competir com Symbian e Windows Mobile. Em agosto de 2005 o Google adquiriu a Android Inc. mantendo os funcionários chave, incluindo Rubin, Miner e White, e desde então o Android é mantido pelo Google e crescendo rapidamente (LOOPER, 2019, p. 1).

O Sistema Operacional Android é uma pilha de *software*, como demonstrado na Figura 4, de código aberto desenvolvida para uma grande variedade de dispositivos com diferentes formatos. O objetivo central do Android é criar uma plataforma de *software* aberta e pronta para operadoras, OEMs e desenvolvedores, a fim de converter as ideias inovadoras em realidade e apresentar um produto real e bem-sucedido que aprimore a experiência móvel dos usuários (SOURCE, 2019, p. 1).

Segundo o Source (2019), o Android não foi construído para ter um ponto central de falhas, em que um indivíduo do setor restrinja ou controle as inovações de outros. Como resultado disso o Android é um produto de consumo completo, com qualidade e que conta com um código-fonte aberto para personalização e portabilidade.

Figura 4 – Pilha do Android



Fonte: Source (2019)

A plataforma Android foi construída com base no sistema operacional Linux e é constituída por um conjunto de ferramentas que opera em todas as fases do desenvolvimento do projeto. Contudo, apesar de ter sido desenvolvido com base no Linux, o Android não é Linux, não dispõe de windowing system[5] nativo, não suporta glibc[6] e não possui algum dos conjuntos de padrões expostos em algumas distribuições Linux (PEREIRA, SILVA, 2009, p. 4).

Tradicionalmente, para o desenvolvimento na plataforma Android necessita-se conhecimento em linguagem de programação Java, utilizar o Kit de desenvolvimento Android SDK, e outras ferramentas pertinentes conforme a necessidade do desenvolvedor. Contudo há outros meios de desenvolvimento de aplicativos para esta plataforma, como através do uso de *frameworks,* ferramentas de desenvolvimento hibrido, e até mesmo com uso de tecnologias que desenvolve nativamente para Android, mas sem o uso das ferramentas tradicionais, que é caso da *stack* React e do Node.js.

5.6 REACT NATIVE

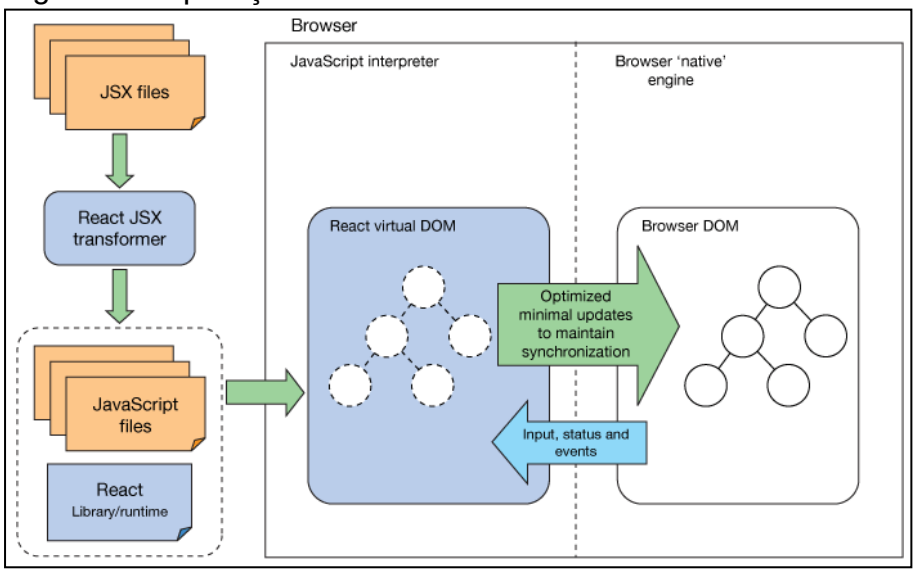
Segundo Cabal (2016), o React Native é um projeto de autoria dos engenheiros do Facebook, que integra uma série de ferramentas que possibilitam a

5Componentes de GUI (Graphical User Interface)
6Biblioteca padrão da linguagem de programação C

criação de aplicações móveis nativas para as plataformas Android e iOS, empregando a linguagem de programação JavaScript juntamente com o *framework* React.

React é uma biblioteca JavaScript XML (JSX) com ferramentas do desenvolvedor associadas. O React simplifica a criação de componentes de visualização de *User Interface* (UI) reutilizáveis, que podem ser construídos formando modernas UIs. Com intuito de potencializar o desempenho do tempo de execução, os componentes React são primeiramente renderizados em um DOM (*Document Object Model*) virtual (Li, 2016, p. 1). A Figura 5 demonstra as operações do React e interação do DOM virtual no processo.

Figura 5 – Operações do React



Fonte: Li (2016)

O DOM virtual do React é um conceito de programação, é mais um padrão do que uma tecnologia em si. Seu objetivo é manter em memória e sincronizar uma representação ideal, ou virtual, com o DOM real através de uma biblioteca, como o ReactDOM, tal processo é denominado reconciliação (FACEBOOK, 2019, p. 1).

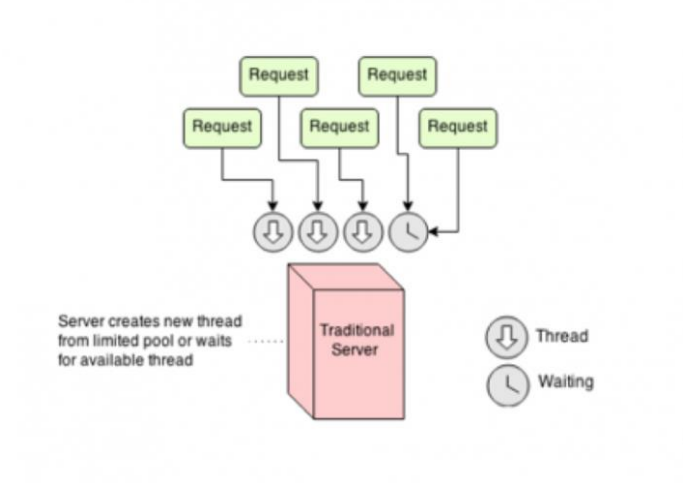
Segundo Li (2016), o desenvolvedor fornece de forma declarativa, no formato de um gráfico de cenário JSX, uma descrição de como a UI deverá aparecer e se comportar. A partir de então o mecanismo de tempo de execução estabelece a maneira mais eficiente de exibi-la em tela.

5.7 NODE.JS

No final de 2009, Ryan Dahl com auxílio inicial de quatorze colaboradores, criou o Node.js com o objetivo de solucionar um problema que o modelo blocking-thread causa no lado dos servidores web. O modelo bloqueante (blocking-thread),

construído sobre plataformas .NET, Java, PHP Ruby ou Python, tem por característica paralisar um processamento, uma thread[7], enquanto utilizam um I/O (em português, entrada e saída) no servidor (PEREIRA, 2013, p. 1). Ou seja, para cada requisição do usuário é criada uma thread para tratá-la. E conforme são criadas, as threads são enfileiradas e processadas uma a uma pelo servidor (LENON, 2018, p. 1). Neste modelo é gasto grande parte do tempo mantendo uma fila ociosa ao mesmo tempo que é executado um I/O.

Figura 6 – Cenário em um servidor tradicional



Fonte: Lenon (2018)

Ao longo do tempo, neste modelo, com o aumento de requisições, aumenta-se o consumo de recursos, principalmente de hardware forçando-o a um *upgrade*. Contudo *upgrades* dos equipamentos é algo muito custoso, o melhor seria procurar por novas tecnologias que façam um bom uso dos hardwares disponíveis (PEREIRA, 2013, p. 1-2).

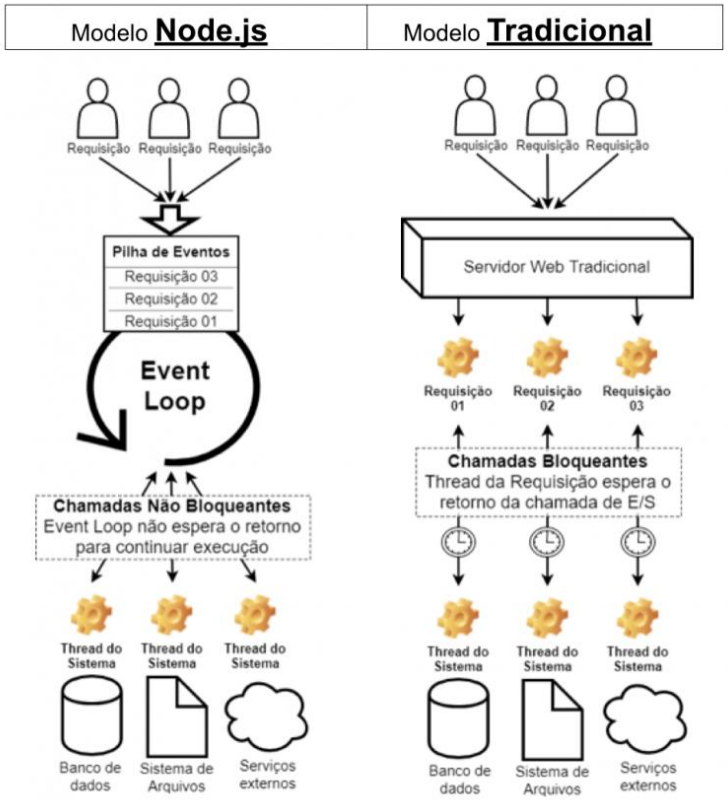
A arquitetura do Node.js é totalmente *non-blocking thread* (não bloqueante), entregando assim uma boa performance em consumo de memória e utilizando o máximo do poder de processamento dos servidores. Com isso, os usuários de sistema construídos utilizando Node.js, encontram-se livres de aguardarem por muito tempo os resultados de seus requisitos, ou processos (PEREIRA, 2013, p. 2).

No modelo do Node.js existe apenas uma *thread*, conforme apresentado na Figura 7, que é responsável por tratar as requisições dos usuários. Denominada *Event Loop,* essa *thread* fica em execução aguardando novas requisições para serem tratadas, e para cada requisição, um novo evento é criado, que por sua vez é

7 Termo em inglês que significa *Fio* de execução, também é conhecido como linha ou encadeamento de execução de um processo.

denominado de forma assíncrona a uma nova *thread* do sistema (LENON, 2018, p. 1).

Figura 7 – Funcionamento de um servidor Node.js



Fonte: Lenon (2018)

O Node.js é altamente escalável e de baixo nível, visto que se trabalha diretamente com inúmeros protocolos de rede e internet ou utiliza bibliotecas que acessam recursos do sistema operacional (PEREIRA, 2013, p. 2). A linguagem de programação utilizado pelo Node é o JavaScript, isso é possível em virtude de ter sido desenvolvido sob a *engine Chrome's V8 JavaScript* (NODE, 2019, p. 1).

Da mesma maneira que o *Gems* do Ruby ou o *Maven* do Java, o Node.js possui um gerenciador de pacotes, chamado de NPM (*Node Package Manager*). O NPM passou a ser instalado junto ao Node.js a partir da versão 0.6.0 devido sua popularidade na comunidade, tornando-se o gerenciador padrão (PEREIRA, 2013, p. 2). Apesar do NPM ser o gerenciador mais popular na comunidade, há outra alternativa de gerenciador dos pacotes Node.js, o Yarn.

5.7.1 YARN

Com intuído de tornar o processo de instalação de dependências não só mais rápido, mas também mais seguro, em outubro de 2016, o Facebook, em conjunto com o Google, Exponent e Tilde, lançou o Yarn (BOCK, 2017, p. 1).

O Yarn é um gerenciador de pacotes. Permitindo ao desenvolvedor compartilhar códigos com outros desenvolvedores em todo o mundo. Os códigos são compartilhados pelo Yarn através dos chamados pacotes, ou também módulos. Um pacote, ou módulo, possui todo o código-fonte compartilhado, juntamente com um arquivo package.json[8] que detalha o código (YARN, 2019, p. 1).

O método de instalação de pacotes através do Yarn é realizado em três etapas (BOCK, 2017, p. 1), sendo elas:

- Buscar recursivamente dependências no repositório do NPM;
- Procurar no cache global e, caso a dependência ainda não tenha sido baixada, salvar uma cópia no cache global;
- Conectar as dependências ao copia-las do cache global para a pasta node_modules[9] local.

Desta maneira, o Yarn pode maximizar o uso dos recursos disponíveis e reduzir o tempo de instalação de pacotes. Mostrando-se mais rápido que o NPM em diversos testes realizados (BOCK, 2017, p. 1).

5.8 APPLICATION PROGRAMING INTERFACE

Application programming interface ou API é um acrônimo da língua inglesa que em português significa interface de programação de aplicações (IPA). As API's são agrupamentos de protocolos e definições que são utilizadas no desenvolvimento e em integrações de *softwares* de aplicações (REDHAT, 2020, p.1). Uma API é uma forma de interação programaticamente com um ou diversos componentes e até recursos de um software separado. Seu conceito pode ser aplicado em qualquer ambiente, desde ferramentas de linha de código, linguagens de programação, aplicativos, sistemas web e outros (FREEMAN, 2019, p. 1).

Segundo Ventura (2015), as API's têm como o principal objetivo fornecer recursos de uma aplicação para serem consumidos por outras aplicações. As APIs

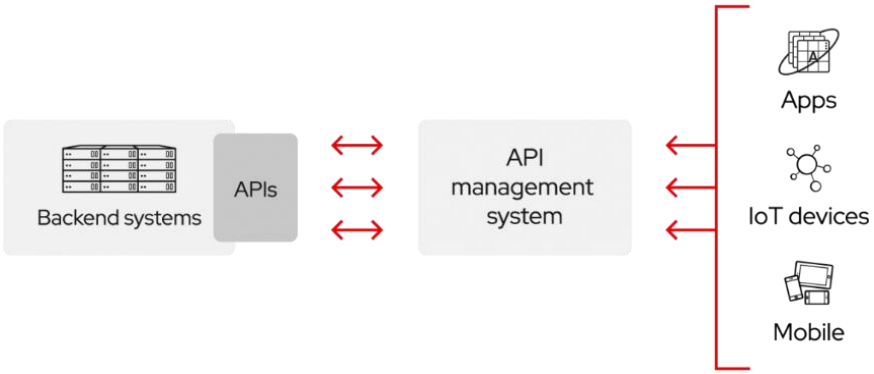
8 O arquivo package.json é o ponto de início para todo projeto Node.js. Ele é encarregado de detalhar o projeto, informar as engines (versão do node, npm ou yarn), url do repositório, versão do projeto, dependências de produção e de desenvolvimento dentre outros.

9 Pasta onde são salvas todas as dependências de produção e desenvolvimento do projeto.

abstraem os detalhes das implementações da aplicação de origem. Elas não necessitam demonstrar, a quem está consumindo o recurso, o que acontece por dentro dos mecanismos da aplicação fornecedora (FREEMAN, 2019, p. 1). A RedHat (2020) compara as APIs como “contratos”, ou seja, uma API simboliza um acordo entre as partes interessadas, na qual quando uma das partes envia uma solicitação remota, respeitando os acordos estabelecidos de uma forma especifica, com isso implicará em como o software da outra parte responderá.

A Figura 8 demonstra o funcionamento de uma API. Na esquerda da figura localiza-se um sistema qualquer que, através de APIs próprias, fornece interfaces de integração para consumo externo. Ao centro localiza-se uma API que faz requisições ao sistema backend e coleta as respostas fornecidas pelo mesmo. Por fim, na ponta direita da figura, encontram-se diversas aplicações e sistemas que consomem a interface e os recursos fornecidos pela API, que por sua vez se conecta com o sistema backend e entrega às aplicações os resultados das solicitações realizadas.

Figura 8 - Exemplo do funcionamento de uma API



Fonte: RedHat (2020)

O uso de APIs torna-se uma maneira simplificada de conectar a própria infraestrutura da aplicação mãe com dispositivos e aplicações externas bem como viabilizam o compartilhamento de dados com clientes e usuários. Outra grande vantagem do uso de API é a ampliação e a forma como aplicações conectam-se aos seus parceiros, com isso até monetizando a troca dessas informações e dados (HEDHAT, 2020, p. 1).

5.8.1 APIs remotas

As APIs surgiram nos primórdios da computação, nesta época seu uso era normalmente como bibliotecas para sistemas operacionais. Neste período, apesar de trocarem informações entre *mainframes* em determinados momentos, as APIs

quase sempre operavam em ambientes locais onde os sistemas encontravam-se (REDHAT, 2020, p. 1). Após um período de aproximadamente trinta anos, houve uma expansão das APIs e seu uso alcançou além dos ambientes locais. No início do século 21, as APIs tornaram-se uma tecnologia importante de integração remota de dados (REDHATE, 2020, p. 1)

APIs remotas foram projetas para interagir e comunicar-se através de uma rede de comunicações. Diz-se APIs remotas pelo fato de que os recursos manipulados por elas se localizam em algum lugar fora do ambiente em que foi realizado a solicitação. Visto que a maior rede de comunicações é a internet, grande parte das APIs são projetadas conforme padrões da WEB (REDHAT, 2020, p. 1).

Geralmente as APIs web utilizam o protocolo HTTP (*Hypertext Transfer Protocol*) para as mensagens de solicitação e disponibilizam uma definição da estrutura das mensagens de resposta. Tais mensagens geralmente possuem o formato de arquivo XML (*Extensible Markup Language*) ou JSON (*JavaScript Object Notation*) e ambos os formatos são comumente utilizados pois apresentam os dados de forma simplificada facilitando a manipulação por outras aplicações (REDHAT, 2020, p.1).

5.9 CLIENTE

A NorteSystem Software é uma *software house*[10] situada na rua Triangulo Mineiro, Nº 1485, no bairro Nova Brasília, neste município de Ji-Paraná/RO. Sua missão é proporcionar soluções tecnológicas ao mundo empresarial, disponibilizando equipamentos e *softwares* que auxiliam empreendedores de diversos seguimentos a gerenciar e tomar decisões de forma rápida e precisa, criando vantagens competitivas (NORTESYSTEM, 2019).

Dentre suas soluções tecnológicas, encontra-se o sistema Katarina. O Katarina é um sistema inteligente indicado para empresas do segmento alimentício, como restaurantes, pizzarias, padarias, bares e dentre outros. Possui diversos módulos para atender os diversos modelos de negócios. Dente os módulos principais do Katarina, estão: gerenciador de Mesas, Comandas, Delivery, vendas em Balcão, PDV (Ponto de Venda), emissão de documentos fiscais (NF-e e NFC-e), controle financeiro e de estoque (NORTESYSTEM, 2019).

10 Terno atribuído a empresas que se dedicam a construir softwares, geralmente com fins comerciais.

A NorteSystem conta com uma base de clientes consolidada e fiel, atendendo em todo o estado de Rondônia, Roraima e Amapá, conforme demonstrado na Figura 9. Utilizando uma de suas soluções, o sistema Katarina, os clientes da NorteSystem gerenciam seus negócios com eficiência, resultando em melhorias em seus processos administrativos quanto de gerenciamento de atendimento ao consumidor final. Contudo, a NorteSystem identificou a necessidade de entregar aos seus clientes uma solução mais ágil e prática, através de um aplicativo móvel. Para a NorteSystem este aplicativo visa melhorar o processo de atendimento, hoje feito através do sistema Katarina em terminais desktops e fixos.

Figura 9 – Atuação da NorteSystem nos estados brasileiros

Fonte: elaborado pelo autor

6 MATERIAL E MÉTODOS

O processo de elaboração do projeto foi realizado utilizando a metodologia do RUP (*Rational Unified Process*), que possui uma completa metodologia com quatro fases principais (SMMERVILLE, 2017), definidas abaixo:

a) **Concepção:** fase de entendimento e visão do problema e identificação das entidades que irão interagir entre si;
b) **Elaboração:** fase de construção da arquitetura do domínio através de diagramas utilizando a conotação da UML;
c) **Construção:** fase em que o projeto é implementado utilizando linguagens de programação, *frameworks*, bibliotecas e outras ferramentas. É nesta fase que o software de fato é construído;
d) **Transição:** fase em que o sistema se encontra em pleno estado de utilidade e pronto para ser transferido para o ambiente operacional;

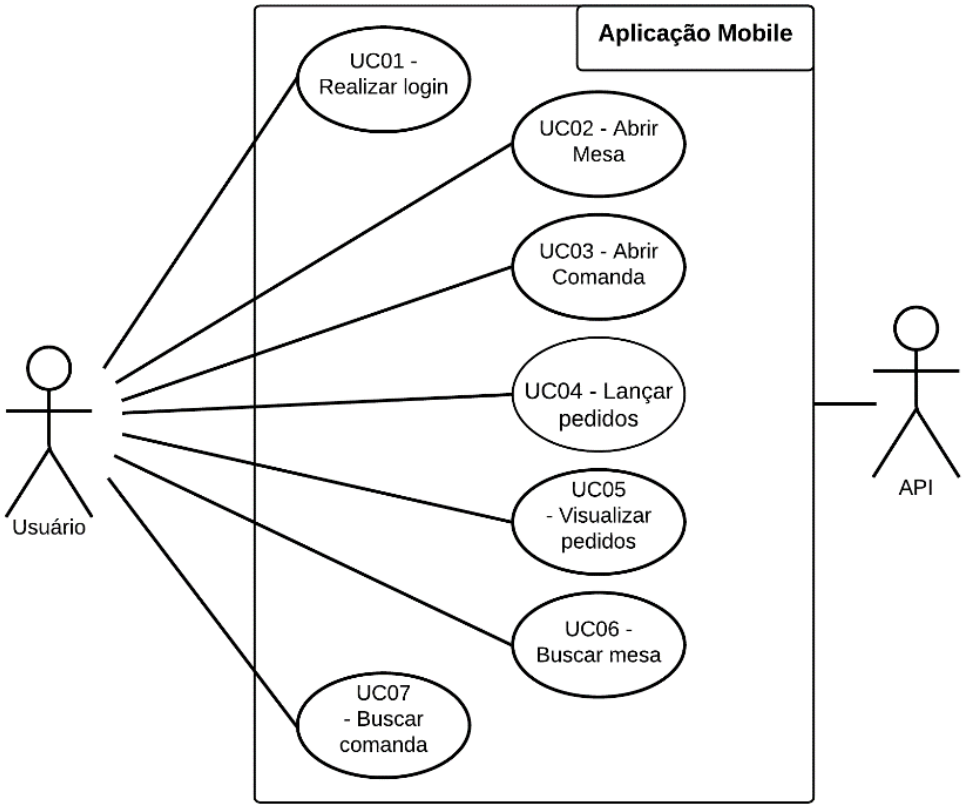
Abaixo são apresentadas as realizações das fases iniciais, de concepção e elaboração. Em um primeiro momento foi analisado o ambiente operacional e desenvolvido um diagrama de *business case* em que apresenta as entidades externas envolvidas nos processos. Em seguida, com base nesse *business case*, os requisitos funcionais foram levantados. Logo após foi construído o diagrama de classes, que demonstra o esqueleto da aplicação e os diagramas de atividades, que apresenta os fluxos das funcionalidades do aplicativo.

6.1 DIAGRAMA DE CASO DE USO

Segundo Bezerra (2007), “um caso de uso é a especificação de uma sequência completa de interações entre um sistema e um ou mais agentes externos a esse sistema”. O caso de uso é responsável por representar as funcionalidades do sistema sem demonstrar o comportamento interno dessas funções. Interessa aqui, para o agente externo principalmente, conhecer visualmente quais serão as funções entregues pelo sistema no seu estágio final.

Após uma análise realizada no ambiente operacional dos clientes da NorteSystem um diagrama de casos de uso, demonstrado na Figura 10, foi elaborado com intuído de apresentar as principais entidades externas que irão interagir com o aplicativo, neste caso usuários e banco de dados, bem como as principais funcionalidades que a aplicação mobile deve possuir.

Figura 10 – Diagrama de casos de uso



Fonte: elaborado pelo autor

Segundo Sbrocco (2011), após definirmos os casos de uso deve-se dividi-los em duas sequências: fluxo normal e fluxo alternativo. O fluxo normal demonstra a sequência de ações que necessitam ser executadas normalmente pelo usuário, sem considerar erros ou situações inesperadas. Já o fluxo alternativo tem esse objetivo, de representar sequências de ações alternativas, principalmente quando algo de inesperado ou erros correrem. O Apêndice A detalha a expansão dos casos de uso acima, juntamente com a descrição dos seus fluxos principais e alternativos.

6.2 REQUISITOS

Segundo Bezerra (2007), “o principal objetivo do levantamento de requisitos é que usuários e desenvolvedores tenham a mesma visão do problema a ser resolvido”. Nessa etapa são levantadas e definidas as necessidades dos futuros usuários, essas necessidades são designadas *requisitos*.

Os requisitos, formalmente, são condições ou capacidades que necessitam ser alcançadas ou possuídas por um sistema ou um componente para satisfazer um contrato, padrão, especificação ou outros documentos formais impostos no projeto.

O resultado do levantamento de requisitos é o *documento de requisitos*, que apresenta os diversos requisitos do sistema. Normalmente esse documento possui uma notação informal e três seções principais: requisitos funcionais, requisitos não-funcionais e requisitos normativos (Bezerra, 2007, p. 22-24).

Requisitos funcionais, segundo Bezerra (2007), determinam as funcionalidades do sistema. Já os requisitos não-funcionais especificam as características de qualidade que o sistema deve possuir e estão relacionadas às funcionalidades. Requisitos normativos declaram as restrições impostas sobre o desenvolvimento do sistema, ou seja, sobre adequação a custos e prazos, tecnologias, limitações sobre interfaces, componentes de hardware, comunicação com sistemas legados etc.

Os quadros 1, 2 e 3 demonstram os requisitos funcionais[11], não-funcionais e normativos, respectivamente, levantados no início do projeto conforme a Tabela 2 presente na seção cronograma. A forma utilizada para o levantamento de requisitos foi a entrevista, técnica essa que faz parte da maioria dos processos de engenharia de *software*.

Quadro 1 – Requisitos Funcionais

Identificação	Requisitos Funcionais (RF)	Caso de Uso (UC)
RF01	Permitir ao usuário realizar login na aplicação.	UC01
RF02	Permitir ao usuário abrir uma mesa e iniciar os lançamentos dos pedidos.	UC02
RF03	Permitir ao usuário abrir uma comanda e iniciar os lançamentos dos pedidos.	UC03
RF04	Permitir ao usuário lançar pedidos (produtos) tanto	UC04

11 O detalhamento dos requisitos funcionais encontra-se no documento de casos de uso presente no Apêndice A.

	em uma mesa pré-iniciada ou em uma comanda também pré-iniciada.	
RF05	Permitir ao usuário visualizar os pedidos de uma mesa ou comanda aberta.	UC05
RF06	Permitir ao usuário localizar uma mesa aberta no *grid* de visualização.	UC06
RF07	Permitir ao usuário localizar uma comanda aberta no *grid* de visualização.	UC07

Fonte: elaborado pelo autor

Quadro 2 – Requisitos Não-Funcionais

Identificação	**Requisitos Não-Funcionais (RNF)**
RNF01	Ser desenvolvido utilizando tecnologias que garantem a portabilidade da aplicação.
RNF02	Possuir interface interativa com intuído de facilitar o uso sem a necessidade de treinamento de usuários.

Fonte: elaborado pelo autor

Quadro 3 – Requisitos Normativos

Identificação	**Requisitos Normativos (RN)**
RN01	Rodar na plataforma Android.
RN02	Ser desenvolvido utilizando o Node.js para o back-end
RN03	Ser desenvolvido utilizando a Stack React.js e React Native para o front-end
RN04	Utilizar Visual Studio Code como editor do código fonte
RNF04	Possuir conexão com o banco de dados PostgreSQL.

Fonte: elaborado pelo autor

6.3 DIAGRAMA DE CLASSES

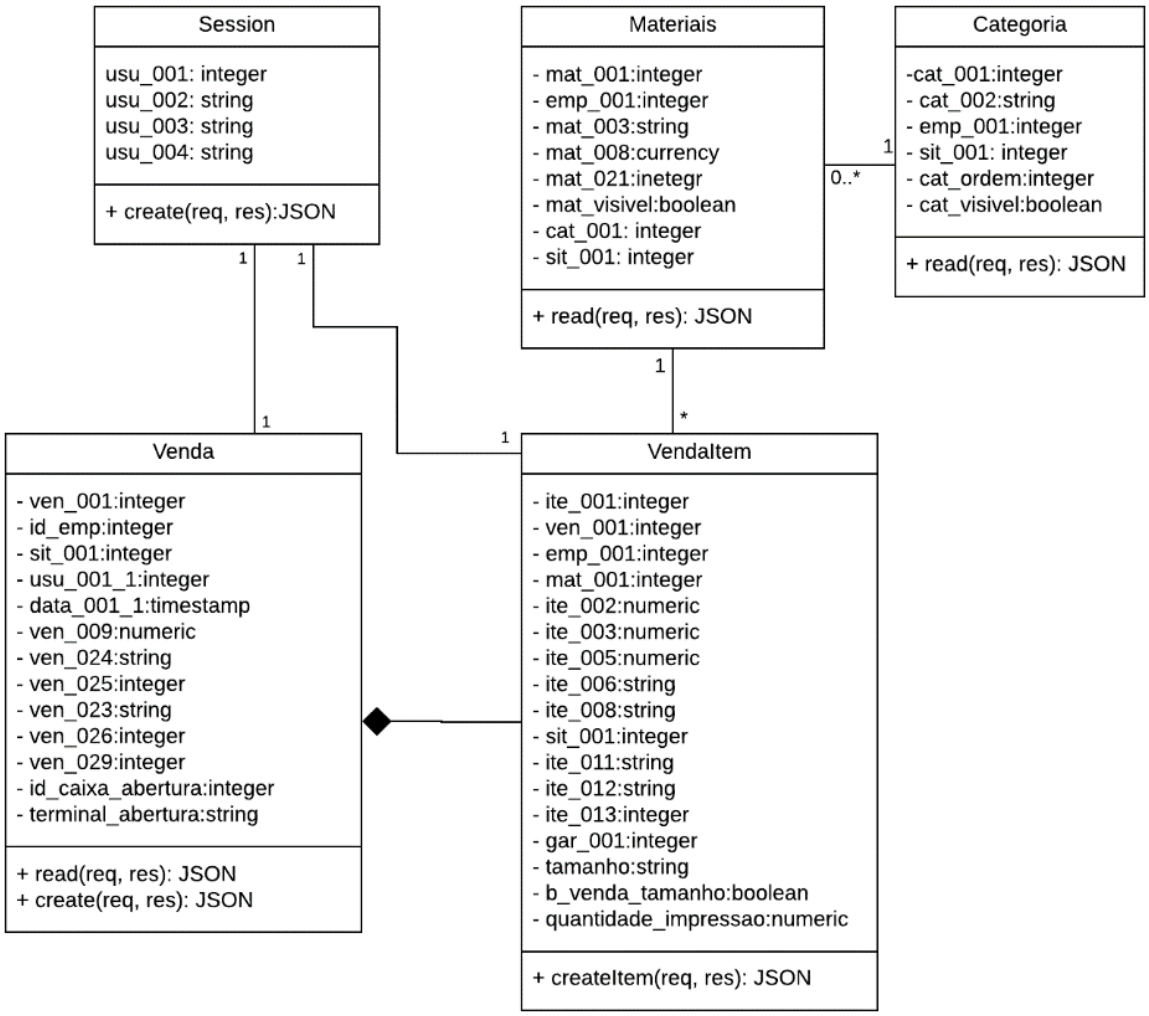
Os diagramas de classes representam os tipos de objetos presentes no sistema e os diversos tipos de relacionamentos estáticos existentes entre eles. Os diagramas de classes também apresentam as propriedades e as operações de uma classe bem como as restrições aplicadas às conexões entre os objetos (FOWLER, 2005, p. 52).

Para Sbrocco (2011), analogicamente comparado ao esqueleto dos animais, a estrutura dos sistemas de *software* é composta por uma armação, representadas

pelas classes, que dão forma ao sistema. Assim, o diagrama de classes necessita ser detalhado para poder representar com precisão o sistema a ser desenvolvido.

Tendo em vista essas definições, a Figura 11 demonstra o diagrama de classes[12] construído para representar o conjunto dos elementos necessários para compor a estrutura da aplicação proposta, detalhando seus atributos, operações e relacionamentos.

Figura 11 – Diagrama de classes



Fonte: elaborado pelo autor

6.4 DIAGRAMA DE ATIVIDADES

Diagramas de atividades servem para representar a lógica de procedimentos, processos de negócio e fluxo de trabalho. São muitos semelhantes aos fluxogramas,

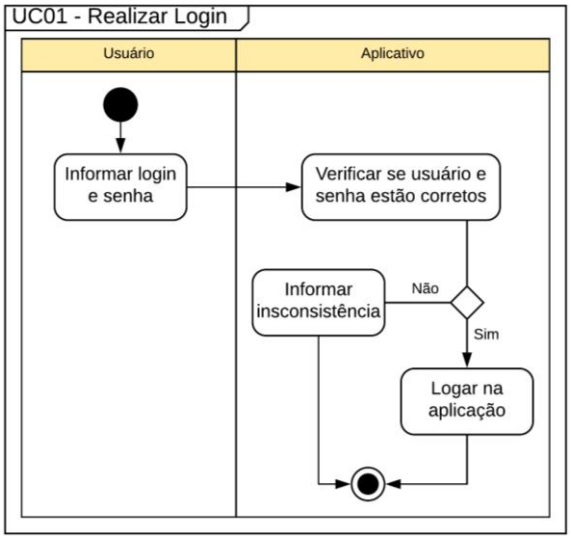
12 Diagrama de classe elaborado e exportado utilizando a ferramenta web Lucidchart disponível em: https://www.lucidchart.com/pages/pt

contudo diferenciam-se deles pelo fato de suportarem a representação de comportamentos paralelos (FOWLER, 2005, p. 118).

Segundo Bezerra (2007), os diagramas de atividades podem ser divididos em dois grupos: diagramas para controle de fluxo sequenciais e diagramas para controle de fluxos paralelos. Os de fluxo sequenciais iniciam e seguem um fluxo único, podendo possuir ramificações, contudo sempre seguindo um fluxo único, podendo ou não ter um estado final, o que significa que o processo ou procedimento modelado é circular. Os diagramas de fluxos paralelos, ou concorrentes, possuem as mesmas características do anterior, contudo, podem conter dois ou mais fluxos de controle sendo executados simultaneamente.

Abaixo são demonstrados os diagramas de atividade[13] baseando-se nos casos de uso apresentados na Figura 10 e detalhados no Apêndice A.

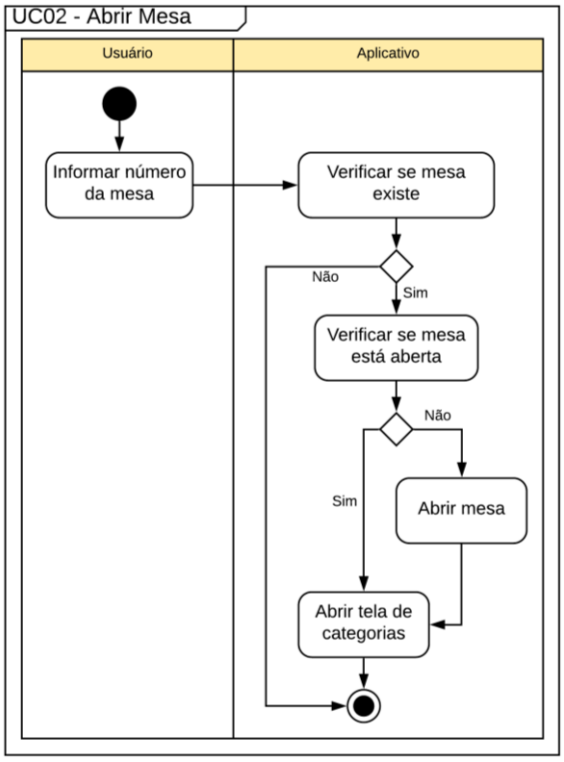
Figura 12 – Diagrama de atividade realizar login referente ao UC01



Fonte: elaborado pelo autor

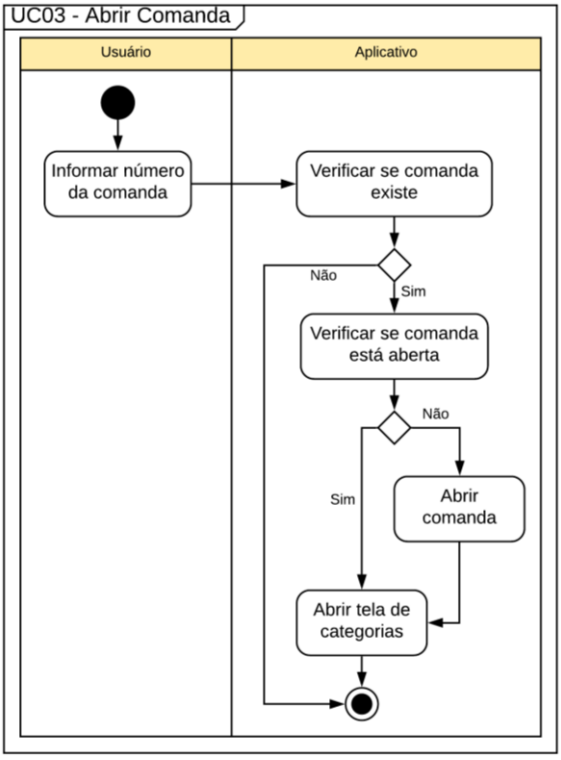
13 Os diagramas de atividades foram construídos e exportados utilizando a ferramenta web Lucidchart disponível em: https://www.lucidchart.com/pages/pt

Figura 13 – Diagrama de atividade abrir mesa referente ao UC02



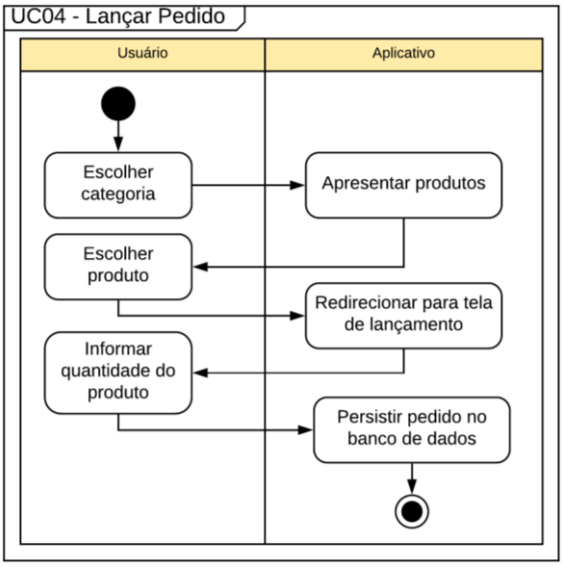
Fonte: elaborado pelo autor

Figura 14 – Diagrama de atividades abrir comanda referente ao UC03



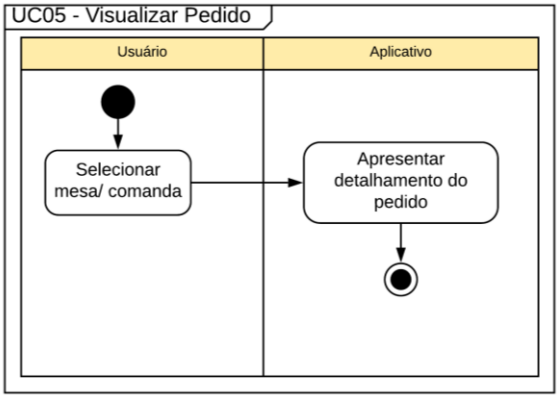
Fonte: elaborado pelo autor

Figura 15 – Diagrama de atividades lançar pedido referente ao UC04



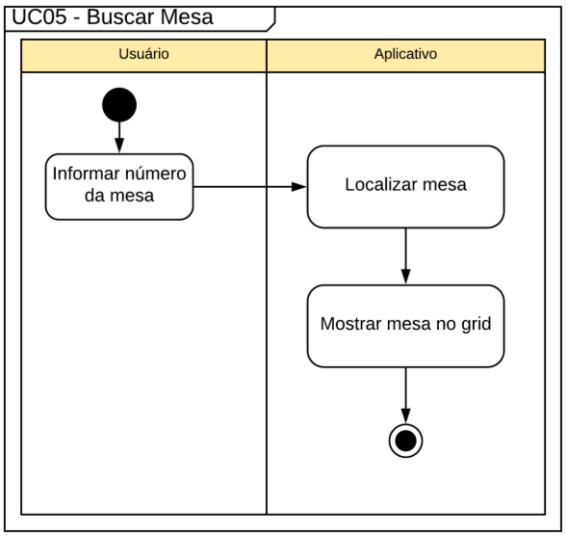
Fonte: elaborado pelo autor

Figura 16 – Diagrama de atividades visualizar pedido referente ao UC05



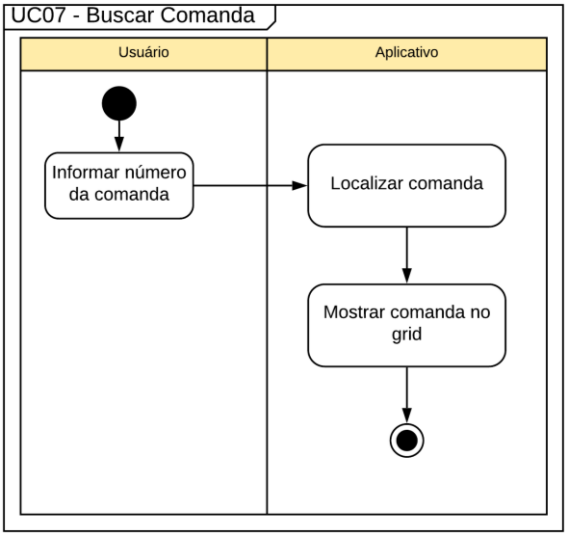
Fonte: elaborado pelo autor

Figura 17 – Diagrama de atividades buscar mesa referente ao UC06



Fonte: elaborado pelo autor

Figura 18 – Diagrama de atividades buscar comanda referente ao UC07



Fonte: elaborado pelo autor

6.5 BANCO DE DADOS

Segundo Heuser (2009), um banco de dados é o conjunto de arquivos integrados que atendem a um conjunto de sistemas. E para atender às necessidades dos diferentes sistemas, ou seja, o compartilhamento de dados, utiliza-se um Sistema de SGBD (Gerenciamento de Banco de Dados).

O sistema Katarina utiliza o SGBD PostgreSQL para o armazenamento e o gerenciamento dos dados geradas. Tendo em vista o requisito normativo proposto no Quadro 3, faz-se necessário conhecer este SGBD, bem como a estrutura das tabelas que a aplicação proposta neste trabalho irá utilizar.

6.5.1 Sistema de Gerenciamento de Banco de Dados PostgreSQL

O PostgreSQL foi construído pelo departamento de Ciência da Computação da Universidade da Califórnia, baseando-se no Postgres versão 4.2. O PostgreSQL é um Sistema de Gerenciamento de Banco de Dados Objeto-Relacional (SGBDOR), ou seja, é um sistema que possibilita aos desenvolvedores integrar ao banco de dados seus próprios tipos de dado e métodos personalizados (POSTGRESQL, 2019, p.1).

Descendente, de código aberto, do código de Berkeley, o PostgreSQL suporta grande parte do padrão da SLQ (*Structured Query Language*) e proporciona diversas funcionalidades modernas, como: comandos complexos, chaves

estrangeiras, triggers, views, integridade transacional, controle de simultaneidade multiversão dentre outros. É possível também, amplia-lo de diferentes maneiras pelo usuário, adicionando novos tipos de dados, funções, operadores, funções de agregação, métodos de índices e linguagens procedurais (POSTGRESQL, 2019, p. 1).

De acordo com a documentação do PostgreSQL (2019), em razão de sua licença ser liberal, este SGBD pode ser modificado e distribuído por qualquer pessoa para qualquer finalidade, seja particular, comercial ou acadêmica.

Por ser um SGBD Objeto-Relacional os dados são armazenados em relações. Relação é, basicamente, um termo matemático para tabela, é o conjunto de atributos relacionados a uma entidade do mundo real (SPRING, 2019, p.1).

O modelo relacional é útil para resolver problemas de banco de dados para aplicações administrativas e comerciais por ser a tecnologia mais difundida no segmento (SPRING, 2019, p.1).

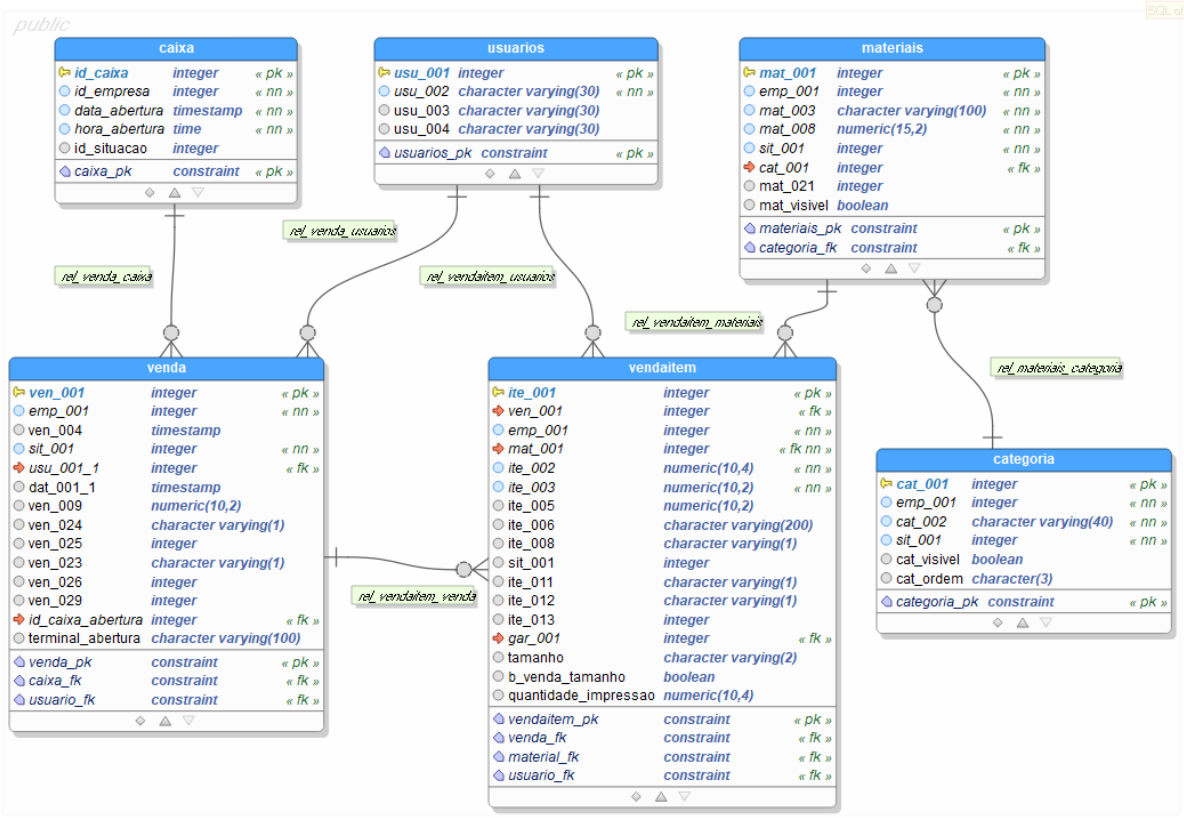
6.5.2 Diagrama entidade-relacionamento

Peter Chen, em 1976, elaborou a abordagem Entidade-Relacionamento (ER), podendo ser considerada como um padrão de fato para a modelagem conceitual.

Segundo Heuser (2009), a abordagem de ER é a técnica de modelagem mais difundida e utilizada. Nesta técnica, o modelo de dados é representado através de um Modelo Entidade-Relacionamento (MER), e geralmente, um MER é reproduzido graficamente através de um Diagrama Entidade-Relacionamento (DER).

A estrutura do banco de dados do sistema Katarina conta com 70 entidades, contudo a aplicação proposta neste trabalho irá consumir apenas 6 entidades. Portanto a Figura 19 apresenta a estrutura do diagrama de entidade-relacionamento baseando-se nas entidades necessárias para o pleno funcionamento da aplicação proposta.

Figura 19 – Diagrama de entidade-relacionamento da aplicação Katarina



Fonte: NorteSystem (2019)[14]

6.6 SISTEMA KATARINA E O APLICATIVO MÓVEL

O sistema Katarina é um *software* comercial especialista no comercio varejista do segmento alimentício. Foi desenvolvido utilizando a linguagem de programação Delphi[15] e possui como gerenciador de banco de dados o PostgreSQL (NORTESYSTEM, 2019).

O Katarina, até sua presente versão de número 2.27.4.2, conta com seis módulos administrativos, dois de apoio, um para emissão de notas fiscais eletrônicas (NF-e) e sete módulos para o gerenciamento de vendas. Os módulos administrativos disponibilizam recursos de cadastramento de produtos, clientes, fornecedores e outros, gerenciamento de movimento de vendas, de estoque, de financeiro e analises gerenciais através de inúmeros relatórios e gráficos. Os módulos de apoio possuem funções de configurações e ajustes do sistema, são utilizados pelos

14 Diagrama entidade-relacionamento adaptado da estrutura do banco de dados do sistema Katarina v.2.27.4.2. Elaborado e exportado utilizando a ferramenta pgModeler.

15 O Delphi é uma IDE (Ambiente de desenvolvimento integrado) para a criação de aplicativos multiplataforma compilados de forma nativa (EMBARCADEIRO, 2019)

funcionários da NorteSystem. O módulo de emissão de notas fiscais disponibiliza recursos para o cumprimento das obrigações legais vigentes no Brasil, através da emissão de NF-e. Os módulos de vendas estão subdivididos em tipos de atendimentos, são eles módulo de atendimento a mesa, atendimento na forma de comanda, delivery, balcão, PDV (Ponto de venda), PDV-Touch e orçamento (NORTESYSTEM, 2019).

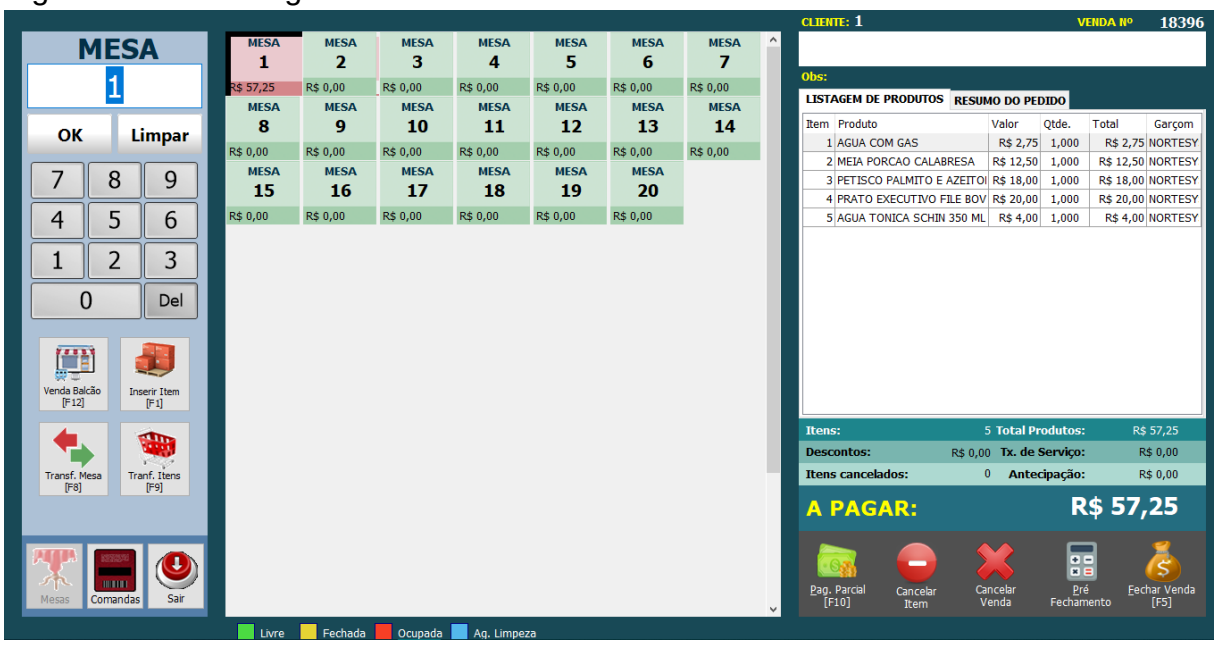
Figura 20 - Tela principal do sistema Katarina

Fonte: *print screen* da aplicação Katarina v. 2.27.4.2

Os módulos PDV, orçamento, delivery e balcão são, geralmente, operados por funcionários em terminais fixos localizados em um balcão de atendimento juntamente do caixa de recebimento. Já os módulos mesa, comanda e PDV-Touch são operados por garçons. Os termais destes módulos encontram-se instalado também em locais fixos, geralmente, espalhados pelo estabelecimento ou em balcões de operação (NORTESYSTEM, 2019).

A Figura 21 apresenta um dos módulos de vendas, o PDV-Touch. Através deste módulo o operador, no caso o garçom, possui uma vasta quantidade de funcionalidades que o auxilia no lançamento dos pedidos a mesa. Conta também com uma ampla visão de todas as mesas do estabelecimento, seu status e uma listagem de todos os itens pertencentes a cada mesa.

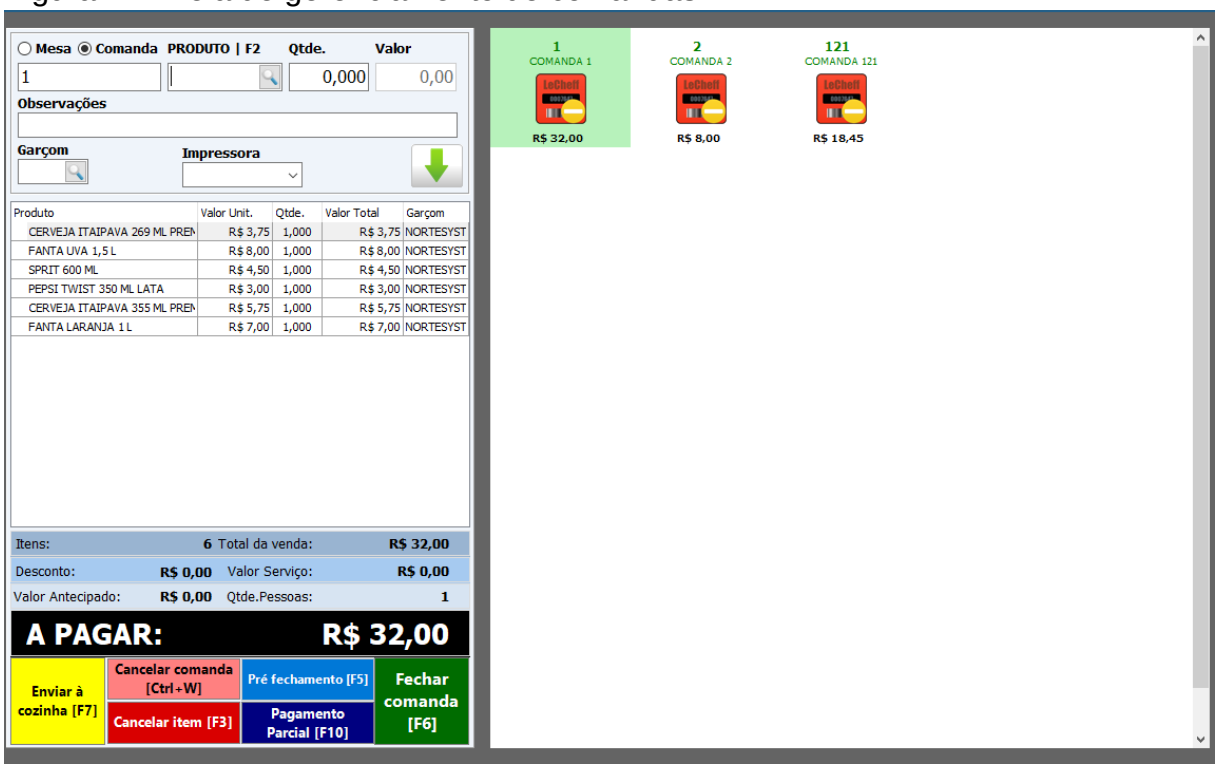
Figura 21 - Tela de gerenciamento de mesas



Fonte: *print screen* da aplicação Katarina v. 2.27.4.2

A Figura 22 demonstra outro módulo muito utilizado no sistema Katarina, o gerenciamento de comandas. Ele possui quase todas as mesmas funcionalidades que o módulo PDV-Touch, diferenciando-se na forma da visualização das comandas, que neste caso são apresentadas no *grid* de visualização apenas as comandas abertas e vigentes.

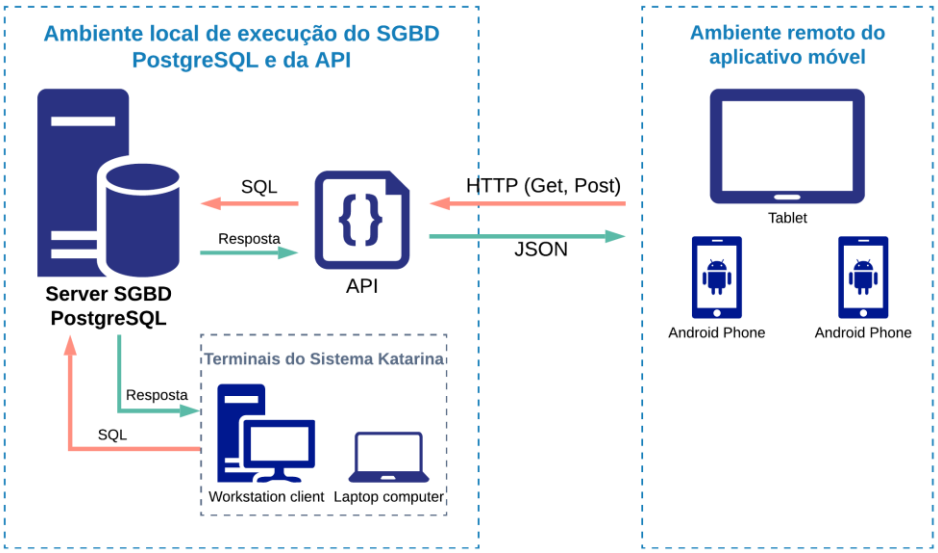
Figura 22 - Tela de gerenciamento de comandas



Fonte: *print screen* da aplicação Katarina v. 2.27.4.2

O aplicativo mobile tem por intenção reproduzir esses dois módulos, através de uma interface intuitiva construída utilizando as tecnologias React e React Native. Oferecendo os recursos de lançamento e visualização dos pedidos. A interação entre o aplicativo móvel e o sistema Katarina ocorrerá de forma indireta através de uma API (*Application programming interface*) como demonstrado na Figura 23.

Figura 23 – Arquitetura de execução dos ambientes do aplicativo móvel e da API



Fonte: elaborado pelo autor

A Figura 23 apresenta os ambientes de execução do sistema Katarina, do SGBD PostgreSQL e do aplicativo móvel. Geralmente o servidor do SGBD PostgreSQL e o sistema Katarina são instalados em máquinas separadas. Contudo encontram-se no mesmo ambiente local de execução, comunicando-se através da rede de internet local de utilizando a linguagem de consulta estruturada (SQL) (NORTESYSTEM, 2019).

A API, desenvolvida neste projeto, também será instalada no ambiente local do Sistema Katarina, preferencialmente no mesmo terminal onde encontra-se o servidor do SGBD PostgreSQL, e fornecerá uma porta de comunicação para que os seus recursos sejam acessados pelo aplicativo móvel utilizando o protocolo HTTP. A API se comunicará com o SGBD utilizando a biblioteca Knex e a linguagem de consulta estruturada (SQL). Os dados fornecidos serão armazenados em arquivos no formato JSON e os disponibilizará para consumo pelo aplicativo móvel.

O aplicativo móvel consumirá os recursos da API através da porta disponibilizada por ela e utilizando o protocolo HTTP. Para que seja possível essa

comunicação, o aplicativo móvel utilizará a biblioteca Axios, a qual fará as requisições no formato do protocolo HTTP e manipulará os arquivos JSON entregues pela API.

6.7 PROTÓTIPO

Para Sobral (2019), “o protótipo é o início do desenvolvimento concreto do design de uma interface”. Com base nele pode-se idealizar o dispositivo a ser desenvolvido, qual tipo de matéria-prima poderá ser utilizada (hardware), a organização dos botões, dos componentes e qualquer outro recurso necessário à interface.

Embora parcial, um protótipo é uma elaboração concreta de uma solução, consequentemente é possível: obter retorno rápido sobre o desenho, poupar tempo de desenvolvimento, experimentar alternativas de desenho e resolver problemas antes de escrever o código (SOBRAL, 2019, p. 72).

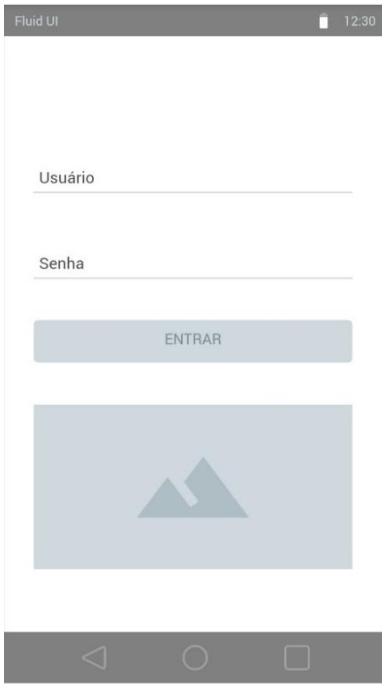
Segundo Sobral (2019), existem diversos tipos de protótipos, como: de cenários, Storyboards, de baixa e alta fidelidade e Wizard of OZ. Para o desenvolvimento deste trabalho foi utilizado a metodologia de prototipação de baixa fidelidade, a fim de, representar apenas a disposição dos elementos necessários ao funcionamento da aplicação proposta.

Segundo Sobral (2019), os protótipos de baixa fidelidade possuem uma representação pouco detalhada da interface, contudo uma de suas vantagens é o tempo, reduzido, para serem construídos e não custam caros.

As Figuras abaixo demonstram os protótipos, de baixa fidelidade, construídos de acordo com os requisitos levantados no início do projeto e validados pelo cliente usuário.

A tela de login, apresentada na Figura 24, permite ao usuário, informar um login e senha, fornecidos pelo administrador do sistema Katarina, assim sendo capaz ter acesso ao lançamento e o gerenciamento dos pedidos das comandas e das mesas.

Figura 24 – Tela login



Fonte: elaborado pelo autor

As Figuras 25 e 26 representam as telas de grid de comandas e mesas, respectivamente. Ambas as telas possuem as mesmas funções e layouts, contudo cada uma apresenta informações referente ao seu tipo de pedido. Na parte superior de ambas as telas, encontra-se um campo de busca que é responsável por filtrar pelo número da mesa ou da comanda desejada. No centro das telas é apresentado um grid de todas as mesas ou comandas abertas no sistema, contendo informações dos números das comandas ou mesas e o total dos pedidos. Na parte inferior das telas estão presentes o botão de incluir um novo pedido e os botões de navegação entre comandas e mesas.

Figura 25 – Tela grid de comandas

Fonte: elaborado pelo autor

Figura 26 – Tela grid de mesas

Fonte: elaborado pelo autor

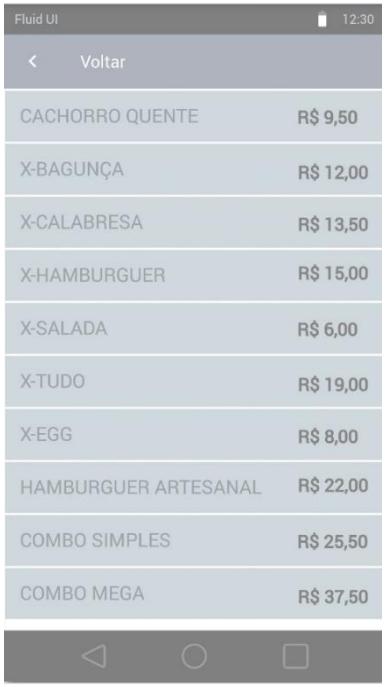
A tela de listagem de categorias, demonstrada na Figura 27, é apresentada ao iniciar um novo pedido. Através dela o usuário tem acesso a todas as categorias de produtos presentes no sistema Katarina. Também é possível voltar para tela anterior, grid de comandas ou mesas.

Figura 27 – Tela listagem categorias

Fonte: elaborado pelo autor

A tela de listagem de produtos, demonstrada na Figura 28, é apresentada ao usuário logo após ter escolhido uma categoria de produtos. Nesta tela são apresentados ao usuário as informações do nome e valor unitário de todos os produtos presentes na categoria escolhida. É possível voltar para tela anterior, listagem de produtos neste caso.

Figura 28 – Tela listagem produtos



Fonte: elaborado pelo autor

A tela de venda, demonstrada na Figura 39, apresenta ao usuário, na parte superior, as informações do pedido que está sendo feito, o produto escolhido, a quantidade a ser lançada, os totalizadores de valor unitário, de quantidade e do valor total do item. Logo abaixo está presente o botão “adicionar” o item no pedido. Na parte central da tela de lançamento de produtos, encontra-se a listagem dos produtos já lançados no pedido, contendo a descrição do produto, totalizadores de quantidade e valor total. Na parte inferior da tela é apresentado o total do pedido e três botões: novo item, cancelar e confirmar pedido. O botão novo item reinicia o processo de lançamento de um produto levando o usuário novamente para tela de listagem de categorias. O botão cancelar descarta a operação em percurso realizada pelo usuário, preservando os lançamentos anteriores confirmados pelo usuário. O botão confirmar realiza o lançamento do pedido no banco de dados, permitindo assim que a aplicação Katarina tenha acesso a nova venda e possa realizar as demais operações necessárias.

Figura 29 – Tela lançamento pedido

Fonte: elaorado pelo autor

6.8 PROJETO DE TESTE DE SISTEMA

Segundo Sommerville (2007), o teste de sistema deve focalizar a verificação de se o sistema atende aos requisitos funcionais e não funcionais e também se não se comporta de maneira inesperada. O teste de sistema abrange a integração de

dois ou mais componentes que implementam funções do sistema (SOMMERVILLE, 2007, p. 357).

Para Sommerville (2007) existem duas fases diferentes de teste do sistema, são elas:

- *Teste de Integração:* O qual deve-se acessar o código-fonte do sistema e integrar componentes para serem testados em conjunto.
- *Teste de Release:* Uma versão completa do sistema, que poderia ser liberada para os usuários, é testada.

O teste de integração abrange a identificação de clusters[16] de componentes que tem por finalidade executar uma funcionalidade da aplicação. O teste de integração afere se os componentes funcionam realmente em conjunto, após a integração, e se são chamados corretamente e se transferem dados em tempo hábil (SOMMERVILLE, 2007, p. 357).

O teste de release é o processo de testes do release a ser distribuído ao cliente final. O objetivo do teste de release é ampliar a confiança do fornecedor de que a aplicação atende aos requisitos. Geralmente, o teste de release, é um método de teste caixa-preta em que os testes são decorrentes das especificações do sistema. Outro nome para este método é *teste funcional*, pois os testes concentram-se na funcionalidade do sistema, e não implementação da aplicação (SOMMERVILLE, 2007, p. 357).

Como no processo de desenvolvimento de *software*, RUP, o teste é uma disciplina que decorre em todas as fases do processo, com ênfase na fase de construção (SOMMERVILLE, 2007, p. 55), para este projeto foi-se elaborado um projeto de testes de release. Tal projeto tem por objetivo testar o release de versão 1.0. O documento de projeto de teste encontra-se no Apêndice C, detalhando o objetivo do documento, os casos de teste e os requisitos para execução destes casos.

Os testes de integração de componentes serão realizados durante o processo de desenvolvimento da aplicação utilizando a ferramenta Insomnia para testar a API e o Expo para testar o aplicativo móvel.

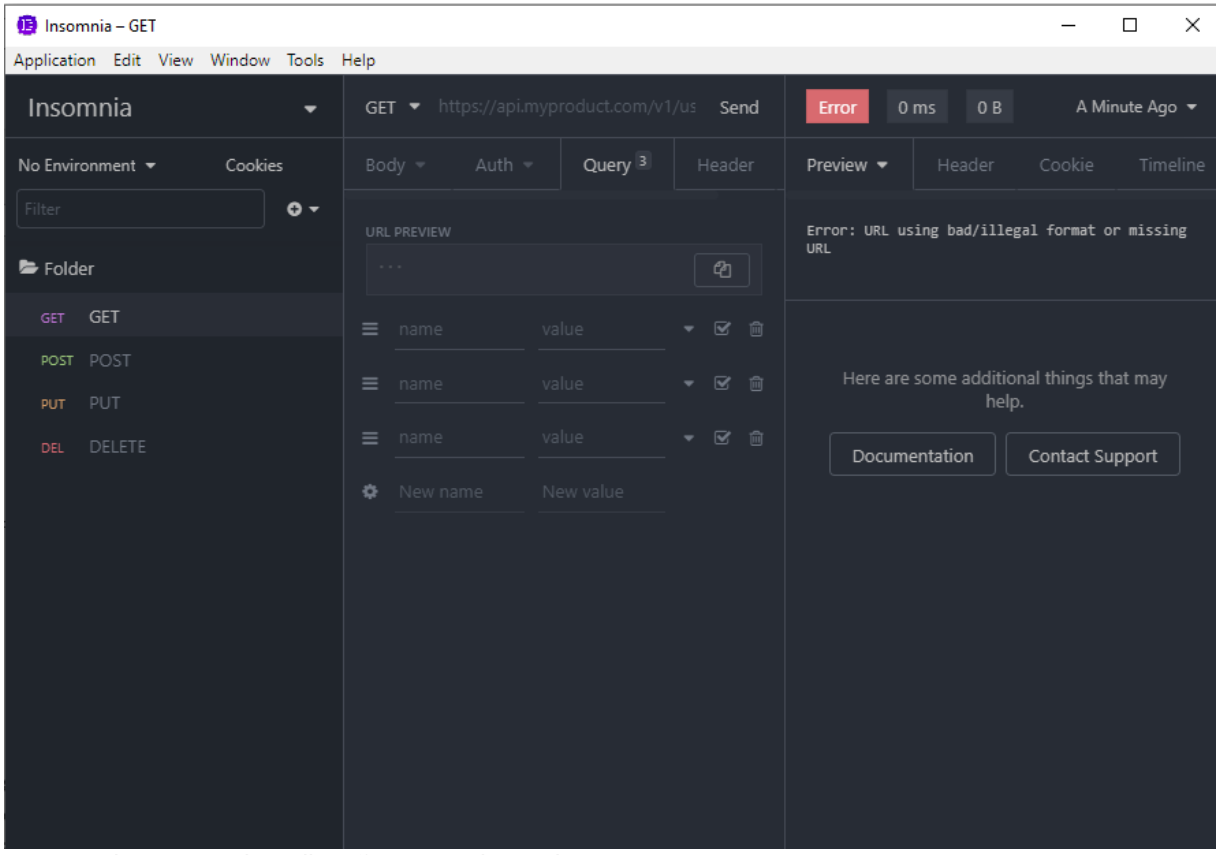
16Termo em inglês que significa *aglomerar* ou *aglomeração*

6.8.1 Insomnia Designer

O Insomnia é um aplicativo desktop de código-fonte aberto, que fornece um fluxo de trabalho moderno para o desenvolvimento de APIs. O objetivo do Insomnia é tornar o processo de desenvolvimento e interação com APIs mais simples e agradável. Atualmente o Insomnia é a principal ferramenta de código-fonte aberto para interagir com APIs REST e GranphQL (INSOMNIA, 2020).

Um dos principais recursos do Insomnia, apresentado na Figura 30, e utilizados nos testes de integração deste projeto, é a possibilidade de simular requisições no formato do protocolo HTTP. Através desde recurso monta-se uma requisição, podendo inserir no corpo da requisição os parâmetros exigidos pela API. Ao executar a requisição, através do botão "*Send*" localizado na parte superior do centro da interface, a ferramenta comunica-se com a API, através da URL e seus parâmetros, e apresenta o retorno obtivo. Assim sendo possível averiguar o funcionamento dos recursos da API.

Figura 30 - Visão geral da Interface do Insomnia Designer



Fonte: *print screen* da aplicação Insomnia Designer v2020.2.2

6.8.2 Expo

O Expo é um *framework* e uma plataforma para aplicações universais do React. Possui um conjunto de ferramentas e serviços desenvolvidos em torno do React e React Native que auxiliam a desenvolver, criar, implementar e iterar rapidamente com iOS, Android e aplicativos da Web a partir do mesmo código-fonte JavaScript/ TypeScript (EXPO, 2020).

Além de ser uma ferramenta para o desenvolvimento de aplicativos, o Expo possui uma interface gráfica do usuário (IUG) baseada na Web que inicializa no navegador da Web assim que o projeto é executado, desta forma o Expo simula a execução das aplicações. Além disso o Expo conta com uma ferramenta para Android, disponível na Play Store, que executa nativamente as aplicações de código-fonte JavaScript/TypeScript. Fazendo o uso dessa ferramenta é possível executar o aplicativo em qualquer dispositivo android. Desta forma é possível testar a interface e todas as funcionalidades do aplicativo de forma rápida e facilitada (EXPO, 2020).

7 RESULTADOS

O desenvolvimento do projeto foi realizado em duas etapas: a primeira deu-se pelo desenvolvimento de uma API para prover a comunicação do aplicativo com o sistema de gerenciamento de banco de dados (SGBD) do sistema Katarina e a segunda etapa deu-se no desenvolvimento do aplicativo móvel.

Neste capitulo é apresentado a analise realizada do processo de desenvolvimento de ambas as etapas.

Antes de realização do desenvolvimento do projeto, foi instalado e configurado o ambiente de desenvolvimento e de testes. O mesmo ambiente de desenvolvimento foi utilizado para o desenvolvimento da API quanto para o aplicativo móvel, pois ambos são praticamente similares no que diz respeito ao uso dos recursos necessários.

Primeiramente foi instalado e configurado o Visual Studio Code, seguindo as configurações básicas oferecidas pelo instalador da ferramenta. Em seguida foi instalado e também configurado a ferramenta Insomnia, a qual disponibilizou recursos para a realização dos testes de funcionamento da API.

Após instalar as ferramentas necessárias, foi instalado o Node.js utilizando o gerenciador de pacotes do Windows, *chocolatey*, através do comando *choco install -y nodejs-lts yarn*. Através deste comando foi instalado a versão LTS 12.16.1 do Node.JS, a versão 6.13.4 do NPM (*Node Package Manager*) e a versão 1.22.0 do Yarn.

Por fim, foi instalado e configurado o sistema Katarina seguindo as instruções baseadas nos procedimentos padrões de instalação fornecidas pela NorteSystem. Ao fim do procedimento obteve-se o sistema Katarina instalado bem como seu banco de dados, acessível através da ferramenta pgAdmin4 a qual foi instalada automaticamente junto do sistema Katarina.

7.1 DESENVOLVIMENTO DA API

Utilizando o ambiente instalado e configurado, inicialmente foi criado o diretório para o armazenamento dos arquivos da API. Em seguida, utilizando o

gerenciador de pacotes Yarn, foram instaladas todas as dependências necessárias para o desenvolvimento e o funcionamento da API. Abaixo são relacionadas as dependências utilizadas e uma descrição de suas principais funcionalidades:

- **Express:** *framework web* que fornece uma camada fina de recursos essenciais para aplicações, sem dispensar os recursos nativos do Node.js (STRONGLOOP, 2020).
- **Nodemon:** utilitário que auxilia o desenvolvimento de aplicativos baseados em Node.js. Monitorando e reiniciando automaticamente a execução do aplicativo quando são detectadas quaisquer alterações de arquivo no diretório do aplicativo (SHARP, 2020).
- **Knex:** construtor de consultas SQL para PostgreSQL, MySQL, Oracle e outros. Desenvolvido para ser flexível e portátil, apresenta retornos de chamadas tradicionais no estilo de nó, como também uma interface promissora para o controle assíncrono, interface de fluxo, construtores de consulta e esquemas, pool de conexões e respostas padronizadas entre diferentes clientes de consultas e dialetos (KNEX, 2020).
- **Cors:** O cors é um pacote Node.js que proporciona um *middleware* do *connect/express* e permite que recursos restritos em uma aplicação sejam requeridos de outro domínio fora do domínio no qual o primeiro recurso foi vinculado (CORS, 2018).
- **Celebrate:** Assim como o cors, o celebrate é um *middleware* do express que envolve a biblioteca de validação *joi*. Permitindo que as entradas, de uma rota única qualquer ou global, sejam validadas garantindo que todas as entradas estejam corretas antes que qualquer função do manipulador seja executada (COMEMORO, 2020).
- **Time-stamp:** Biblioteca que permite obter uma data e hora formatada (SCHLINKERT. 2018).

A relação dessas dependências e suas respectivas versões encontram-se no arquivo *package.json* no diretório do projeto da API.

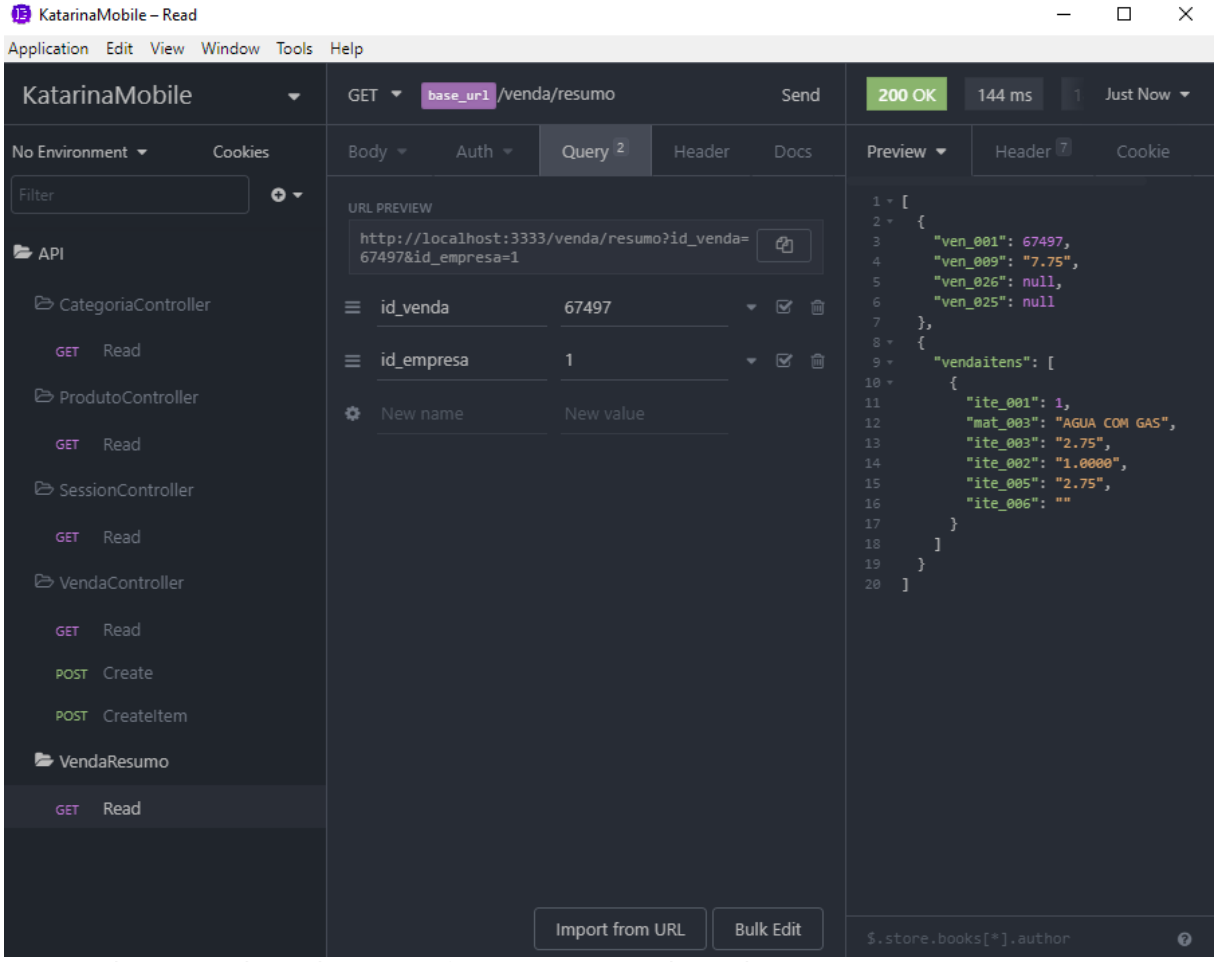
Após a instalação de todas as dependências necessárias foram criados todos os arquivos que compõem a estrutura da API, que definem a conexão com o SGBD PostgreSQL e todos os recursos oferecidos para consumo do aplicativo móvel. Os

códigos, destes arquivos, tidos como essenciais, são apresentados em listagens no Apêndice D.

Durante todo o processo de desenvolvimento da API foram realizados inúmeros testes de integração para garantir o funcionamento da mesma. Para isso foi utilizado os recursos da ferramenta Insomnia Designer, que possibilita a criação de requisições HTTP que se comunicam através da URL e da porta fornecida pela API. Para a realização dos testes, o servidor da API foi alocado localmente no endereço “http://localhost” e na porta “3333”. Portanto, no Insomnia, foi configurado uma *base_url* com o endereço http://localhost:3333. Assim sendo possível consumir todos os recursos fornecidos pela API bem como testar o seu funcionamento.

Utilizando o Insomnia foi criado um *workspace*, demonstrado na Figura 31, no qual foi criado, no canto esquerdo, a simulação dos arquivos principais da API e seus métodos. Para cada método de cada arquivo é configurado, no centro da Figura, o corpo da requisição, podendo atribuir os parâmetros necessários exigidos pela função em questão. Automaticamente o Insomnia cria uma URL para fazer a requisição para a API. Através do botão “Send” é possível executar essa requisição e visualizar seu retorno no canto direito da ferramenta.

Figura 31 - Workspace de testes da API na ferramenta Insomnia Designer



Fonte: *print screen* do *workspace* na ferramenta Insomnia Designer v.2020.2.2

A Figura 31 demonstra a execução do método GET/Read do arquivo VendaResumoController.JS. Podendo ser possível visualizar o retorno da requisição ao lado, com as informações toda venda no momento da chamada do método.

7.2 DESENVOLVIMENTO DO APLICATIVO MÓVEL

Para o desenvolvimento do aplicativo móvel, inicialmente foi criado um diretório para armazenar os arquivos necessários para o funcionamento do mesmo. Através da linha de comando “react-native init mobile” criou-se toda a estrutura React Native necessária para iniciar o desenvolvimento do aplicativo mobile. Este comando criou um diretório com nome *mobile* no qual armazena todos os arquivos base necessários para iniciar um novo projeto React Native.

Após a inicialização do projeto e com a estrutura base montada foi instalado todas as dependências, utilizando o gerenciador de pacotes Yarn, necessárias para o desenvolvimento e funcionamento do aplicativo móvel. Abaixo são listadas as

dependências auxiliares com uma breve descrição das suas principais funcionalidades:

- **Axios:** É um cliente HTTP baseado em *promise*[17] para navegadores e Node.js. Com Axios é possível realizar solicitações http. Transformações automáticas para dados JSON (AXIOS, 2020).
- **INTL:** É um namespace para a API de Internacionalização do ECMAScript que disponibiliza comparação de do tipo primitivo string, formatação de números, e formatação de data e hora (INTL, 2016).
- **Expo:** É um *framework* e uma plataforma para aplicações universais do React. O Expo possui ferramentas que auxiliam a desenvolver, criar, implementar e iterar rapidamente com iOS, Android e aplicativos da Web a partir do mesmo código-fonte JavaScript/ TypeScript (EXPO, 2020).

Todas as dependências e suas respectivas versões encontram-se listadas no arquivo *package.json* do projeto do aplicativo móvel, inclusive as dependências instaladas inicialmente na execução do comando para a criação da estrutura base do projeto.

Após toda a instalação das dependências necessárias para o desenvolvimento do aplicativo móvel foi criado um subdiretório, na pasta mobile, denominado “src” que tem por finalidade comportar todos os arquivos criados e necessários para a conexão com a API e a criação de todas as interfaces das telas do aplicativo. Bem como os arquivos de estilização e auxiliares.

Os arquivos tidos como essenciais e os que possuem toda a estrutura para a criação das interfaces gráficas e seus métodos, são apresentados e explicados em listagens no Apêndice E.

Assim como no desenvolvimento da API, durante todo o desenvolvimento do aplicativo móvel testes de integração foram realizados utilizando a ferramenta Expo, com a finalidade de validar a interface das telas desenvolvidas, bem como as funcionalidades destas telas. Para a realização dos testes foi utilizado dois smatphones Android, um com a versão do sistema operacional 9.0 e outro com aversão 8.0. Ambos com o aplicativo Expo instalado.

17 “Promise é um objeto usado para processamento assíncrono. Um Promise (de "promessa") representa um valor que pode estar disponível agora, no futuro ou nunca.” (MDN, 2019, p. 1).

Ao fim do desenvolvimento obteve-se um aplicativo móvel operável e com todas as funcionalidades validadas.

7.2.1.1 Apresentação do aplicativo na plataforma Android

Neste tópico são apresentadas as interfaces gráficas das telas do aplicativo móvel, assim como uma descrição de suas funcionalidades.

Ao ser iniciado o aplicativo móvel apresenta a tela de *splash*, demonstrada na Figura 32. Essa tela tem como finalidade apenas apresentar um visual amigável enquanto o aplicativo inicializa.

Figura 32 – Tela de Splash

Fonte: elaborado pelo autor

Após a tela de splash o usuário é redirecionado para a tela de autenticação, demonstrada na Figura 33. Nesta tela é possível informar os valores de um usuário e senha e pressionar o botão “entrar”. O aplicativo enviará uma requisição para a API, caso os dados informados estejam corretos o usuário será permitido acessar as demais telas e funcionalidades do aplicativo. Caso os dados estejam incorretos, uma mensagem será apresentada ao usuário: “Usuário inexistente ou com login/senha inválida!”. Antes de validar os dados do usuário é realizado uma verificação que valida se as configurações de localização do endereço da API e do código da

empresa foram configuradas pelo usuário. Através do botão de configuração, localizado abaixo do botão “entrar”, é possível acessar a tela de configurações.

Figura 33 – Tela de autenticação

Fonte: elaborado pelo autor

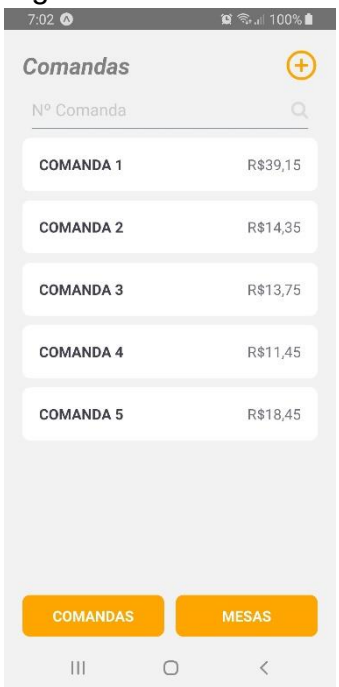
A Figura 34 demonstra a tela de configuração. Nesta tela é possível informar os parâmetros do endereço do servidor onde a API encontra-se e do código da empresa. Ao clicar em “salvar” o usuário será redirecionado de volta para tela de autenticação.

Figura 34 – Tela de configurações

Fonte: elaborado pelo autor

Após o usuário realizar a autenticação no aplicativo móvel, o mesmo é redirecionado para a tela de comandas, demonstrada na Figura 35. Ao ser carregada a tela de comandas lista todas as comandas abertas, caso aja alguma. Nesta tela pode-se buscar por apenas uma única comanda, utilizando o botão buscar localizado na caixa de consulta “Nº comanda”. Ao executar essa função apenas a comanda informada pelo usuário é apresentada na listagem. Na parte superior direita da tela, localiza-se o botão de abrir uma nova comanda. Ao clicar nesse botão é aberto uma caixa de diálogo que questiona ao usuário qual o número da comanda deverá ser aberta. Após o usuário informar o número da comanda e confirmar inicia-se uma nova venda e o usuário é redirecionado para tela de categorias. Caso o usuário clique sobre um registro listado no grid a tela de resumo de venda será apresentada. Por fim, no conto inferior da tela localiza-se os botões de navegação, que possibilita o usuário navegar entre as telas de comandas e mesas.

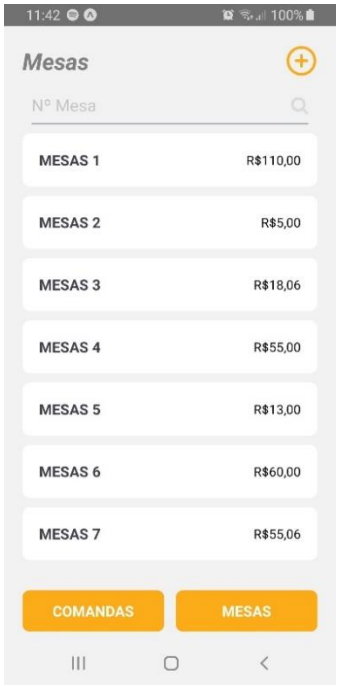
Figura 35 – Tela de listagem de comandas abertas



Fonte: elaborado pelo autor

A Figura 36 demonstra a tela de mesas. Esta tela tem exatamente os mesmos componentes e funções que a tela de comandas. Com exceção da apresentação dos dados, que neste caso é a listagem das mesas abertas.

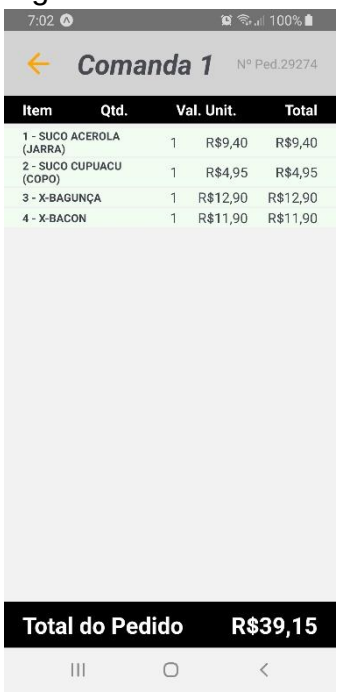
Figura 36 – Tela de listagem de mesas



Fonte: elaborado pelo autor

A Figura 37 apresenta a tela de resumo de venda. Esta tela tem apenas o intuito de apresentar as informações gerais do pedido referente a uma mesa ou comanda. Esta tela possui informações gerais de identificação da venda, no topo da tela, uma listagem dos itens da venda, ao centro, e informações do total da venda, na parte inferior da tela.

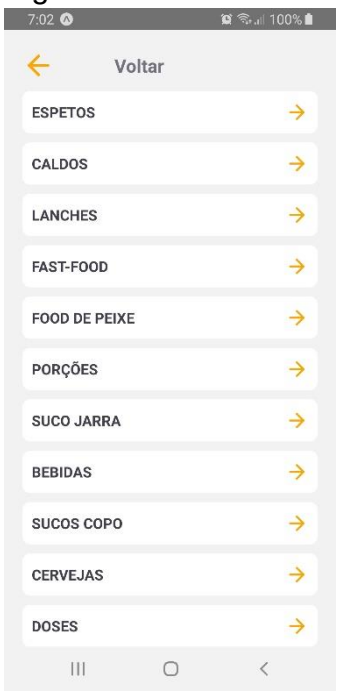
Figura 37 – Tela de resumo de venda



Fonte: elaborado pelo autor

A Figura 38 demonstra a tela de categorias. Esta tela tem por finalidade apresentar todas as categorias disponíveis e dar continuidade na nova venda aberta na tela de comandas ou mesas. Após o usuário clicar em uma das categorias listadas, ele será redirecionado para a tela de produtos. Nesta dela possui um botão voltar localizado no canto superior esquerdo, que ao ser clicado volta para a tela anterior, comandas ou mesas.

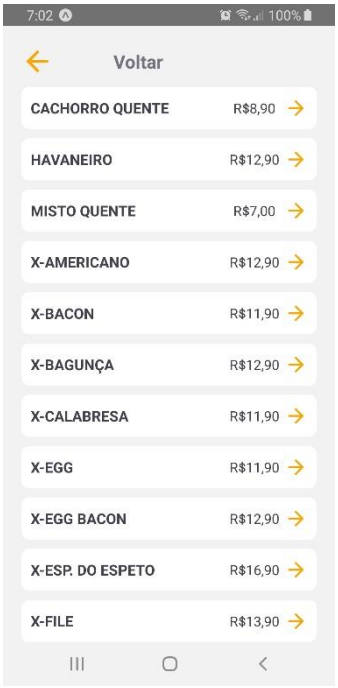
Figura 38 – Tela de listagem de categorias



Fonte: elaborado pelo autor

A Figura 39 demonstra a tela de listagem de produtos. Esta tela tem por finalidade apresentar todos os produtos, em um grid, de uma determinada categoria escolhida na tela de listagem de categorias. Nesta tela é apresentado os produtos juntamente com seu respectivo valor. Na parte superior da tela localiza-se um botão voltar, o qual retorna à navegação do usuário para a tela anterior, categorias. Ao clicar em um dos produtos o usuário é redirecionado para tela de venda.

Figura 39 - Tela de listagem de produto



Fonte: elaborado pelo autor

A Figura 40 demonstra a tela de venda. Esta é a última tela no processo de lançamento de um produto em um pedido já existente ou em um novo pedido. Esta tela conta, no topo, com a informação do pedido em edição, informações do produto que está sendo inserido, uma listagem dos itens já lançados, caso aja, e o total do pedido na parte inferior da tela. Na sessão das informações do item que está sendo inserido, existe dois botões: adicionar e um componente *stepper*, que incrementa e decrementa a quantidade do item além de recalcular o valor total do item. O botão adicionar tem a funcionalidade de inserir este novo produto ao pedido e atualizar a listagem e o total da venda. Na parte inferir da tela localiza-se três botões: novo item, cancelar e confirmar. O botão novo item tem a funcionalidade de reiniciar o processo de inserção de itens no pedido redirecionando o usuário novamente para tela de categorias. O botão confirmar tem a funcionalidade de concluir o processo de inserção de novos itens ao pedido e retornar para tela de comandas. Por fim, o botão cancelar encerra o processo de inserção de item abandonando o processo em andamento.

Figura 40 - Tela de venda

Fonte: elaborado pelo autor

7.3 VALIDAÇÃO DO APLICATIVO MÓVEL

As validações do aplicativo móvel ocorreram ao fim do desenvolvimento do projeto. Para a realização de testes foi disponibilizado, para possíveis usuários, que já utilizam o sistema Katarina, a versão de testes de número 1.0.0. Houve a participação de 7 colaboradores de um cliente da NorteSystem que já utiliza o sistema Katarina.

Após a realização dos testes, para fins avaliativos, os usuários foram submetidos a responderem um questionário, o qual possuía 10 perguntas, sendo 5 para validação das funcionalidades do aplicativo móvel e 5 perguntas para validar a experiencia de uso.

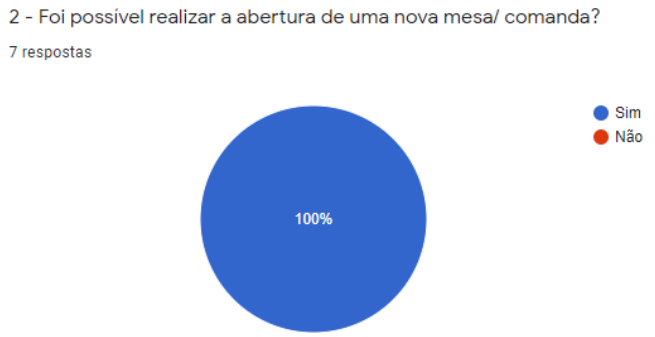
A primeira pergunta do questionário tinha como finalidade validar a execução da função de realizar login no aplicativo. 100% dos usuários afirmaram que foi possível realizar login com suas credenciais, como apresentado na Figura 41.

Figura 41 – Questionário 1

Fonte: elaborado pelo autor

O segundo questionamento era para validar a possibilidade da realização da abertura de mesas e comandas. 100% dos usuários afirmaram que conseguiram realizar esta função, como apresentado na Figura 42.

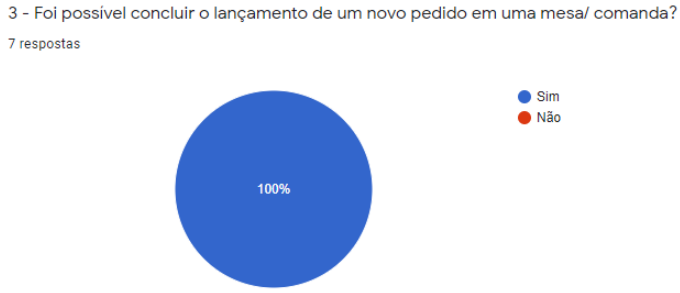
Figura 42 – Questionário 2



Fonte: elaborado pelo autor

O terceiro questionário validou se os usuários conseguiram concluir os lançamentos de novos pedidos nas mesas e comandas. 100% dos usuários responderem positivamente, como demonstrado na Figura 43.

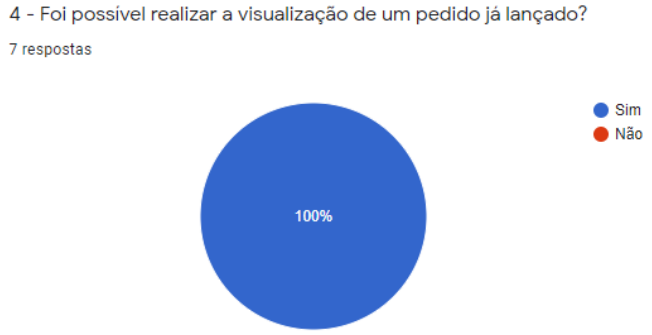
Figura 43 – Questionário 3



Fonte: elaborado pelo autor

O quarto questionário dedicou-se a saber se os usuários conseguiram visualizar os pedidos na tela de resumo no aplicativo. 100% dos usuários confirmaram positivamente, como demonstrado na Figura 44.

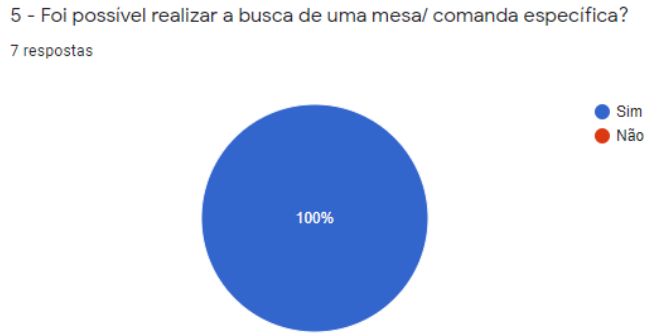
Figura 44 – Questionário 4



Fonte: elaborado pelo autor

O quinto questionário validou a funcionalidade de buscar por comandas e mesas especificas, presente nas telas de listagem de meses e comandas. 100% dos usuários responderam que sim. Como apresentado na Figura 45.

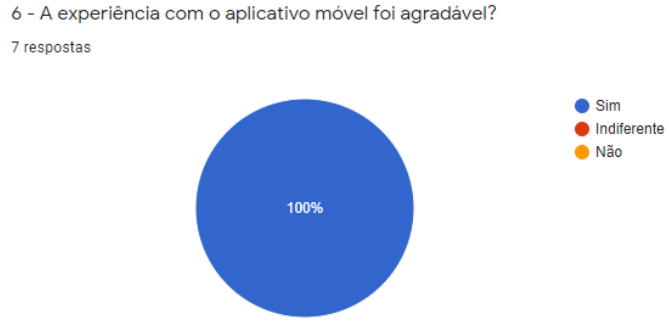
Figura 45 – Questionário 5



Fonte: elaborado pelo autor

A sexta pergunta voltou-se para questionar se a experiência de uso com o aplicativo móvel foi agradável. 100% dos usuários responderam positivamente, como apresentado na Figura 46.

Figura 46 – Questionário 6



Fonte: elaborado pelo autor

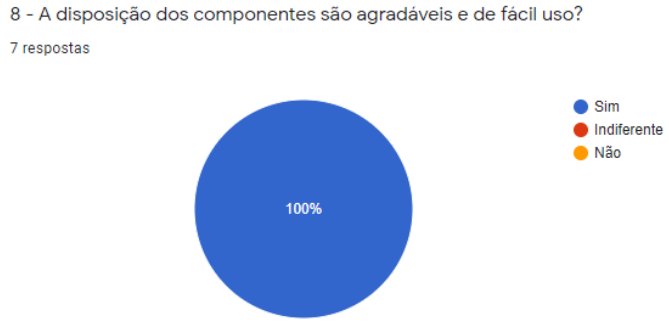
A pergunta 7 questionou aos usuários se a navegação entre as telas do aplicativo foi fácil e agradável. 100% dos usuários responderam que sim, como demonstrado na Figura 47.

Figura 47 – Questionário 7

Fonte: elaborado pelo autor

O oitavo questionário validou se a disposição dos componentes nas telas do aplicativo são agradáveis e de fácil uso. 100% dos usuários responderam que sim, como demonstrado na Figura 48.

Figura 48 – Questionário 8



Fonte: elaborado pelo autor

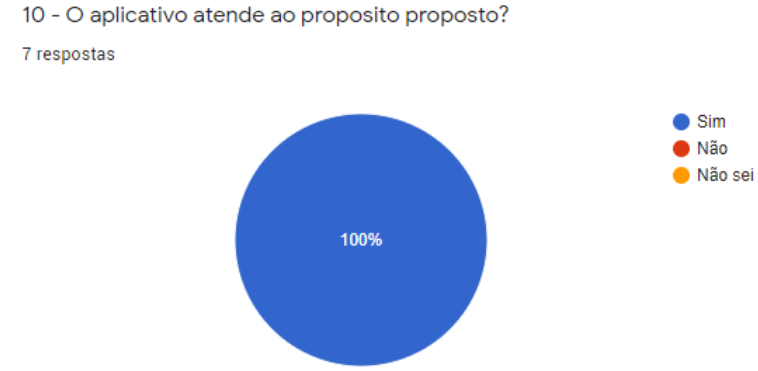
A nona pergunta teve como finalidade questionar aos usuários se as cores do aplicativo são agradáveis. 71,4% responderam positivamente e 28,6% responderam que são indiferentes quanto a cores do aplicativo, como demonstrado na Figura 49.

Figura 49 – Questionário 9

Fonte: elaborado pelo autor

Por fim, a pergunta de número 10 questionou aos usuários se o aplicativo móvel atente ao proposito proposto inicialmente. 100% dos usuários responderam que sim, como apresentado na Figura 50.

Figura 50 – Questionário 10



Fonte: elaborado pelo autor

8 CRONOGRAMA

Segundo Pressman e Maxim (2016), o cronograma de um projeto de *software* é uma ação que fragmenta o esforço estimado para todo o planejamento de desenvolvimento do projeto, destinando tal esforço para atividades específicas de engenharia de *software*.

Para Pressman e Maxim (2016), no início do estágio do planejamento do projeto, cria-se um cronograma macroscópico, o qual identifica as atividades principais, contudo, conforme o projeto avança, tais atividades são refinadas e cronogramas mais detalhados.

Considerando-se as definições de Pressman e Maxim, abaixo são demostradas as atividades principais deste projeto, reunidas em um macro cronograma e divididas em meses de desenvolvimento.

Tabela 2 – Cronograma de desenvolvimento do projeto

Atividade	**2019**			**2020**				
	Set.	Out.	Nov.	Fev.	Mar.	Abr.	Maio	Jun.
Levantamento de Requisitos								
Modelagem da aplicação								
Prototipação								
Desenvolvimento da aplicação								
Testes								
feedback dos usuários (v.1.0.0)								

Fonte: elaborado pelo autor

O cronograma deste projeto foi desenvolvido levando em consideração o tempo final para entrega do trabalho final de conclusão de curso (TCC) no ano de 2020. O tempo das atividades foram divididas em meses e os meses foram subdivididos entre os anos de 2019 e 2020, que representam os semestres letivos das disciplinas de TCC1 e TCC2.

9 CONCLUSÃO

Neste trabalho foram abordadas a importância do uso e do investimento em inovação, no que diz respeito a ferramentas para o atendimento ao público em estabelecimentos do segmento alimentício, como bares, restaurantes, pizzarias, lanchonetes e outros. Com base neste contexto e como solução proposta, foi desenvolvido um aplicativo móvel para dispositivos Android, que auxilia o gerenciamento do atendimento a mesas e comandas, a fim de facilitar o processo dos envolvidos.

Esse trabalho teve o objetivo principal o desenvolvimento de uma aplicação móvel, no qual foi possível empregar técnicas de programação para dispositivos móveis. E também no desenvolvimento de uma *Application Programing Interface* (API), a qual tem por objetivo prover recursos para o aplicativo móvel consumir e conseguir comunicar-se com o sistema de gerenciamento de banco de dados PostgreSQL, utilizado pelo sistema Katarina.

Com a utilização do React Native e bibliotecas auxiliares foi desenvolvido um produto que entrega as funcionalidades básicas necessárias para o gerenciamento de mesas e comandas além de possuir uma interface agradável e de fácil uso. Sendo também um produto com um bom desempenho devido a tecnologia do React Native.

E com a utilização do Node.js e também com bibliotecas auxiliares foi desenvolvido uma API local que, ao ser instalada no servidor do SGBD PostgreSQL, entrega diversos recursos ao aplicativo móvel.

Após a implantação do aplicativo móvel e a API em ambiente de produção os processos de atendimento e gerenciamento de mesas e comandas tornam-se rápidos e flexíveis, visto que o usuário pode acessar todas as informações de qualquer lugar dentro do estabelecimento, já que dispensará o uso de terminais desktop fixos e passará a usar dispositivos móveis Android com o aplicativo móvel instalado.

9.1 TRABALHOS FUTUROS

Após o desenvolvimento do projeto e analisando novos processos no ambiente de produção, observou-se que o aplicativo móvel possui potencial para entregar outros recursos que facilitarão o processo de atendimento ao consumidor. Portando como trabalhos futuros, pode-se apontar:

- Desenvolvimento de níveis de usuários: permitir o acesso limitado a determinados recursos do aplicativo dependendo do nível do usuário, ou seja, administrador, garçom, atendente e outros.
- Implementar recurso de junção de mesas e ou comandas: Permitir ao usuário a possibilidade de juntar os itens em diferentes mesas, visto que esse recurso está disponível no sistema Katarina e é frequentemente utilizado.
- Reimpressão de pedidos: possibilitar ao usuário requisitar a reimpressão dos pedidos em casos que haja algum tipo de falha de impressão dos pedidos por parte do sistema Katarina.
- Finalização de pedidos: recurso atualmente restrito a determinados níveis usuários e presente apenas na versão desktop do sistema Katarina. Através deste recuso o usuário poderá fazer o fechamento do pedido em necessitar utilizar um terminal fixo com o sistema Katarina.
- Possibilitar a inserção do cliente na abertura da mesa ou comanda: possibilitar ao usuário informar o nome ou o cadastro do cliente que consumirá em determinada mesa ou comanda.
- Lançamento de opcionais: Buscar os itens opcionais dos produtos para serem lançados no momento do lançamento do produto na venda.
- Lançamento de itens fracionados: dar suporte ao lançamento de itens fracionados, função muito utilizada em estabelecimentos de pizzaria.

10 REFERÊNCIAS

AXIOS. **Axios:** Documentation. *[S.l.].* NPM. 2020. Disponível em: https://www.npmjs.com/package/axios. Acesso em: 30 maio 2020.

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML:** modelagem e gerência de interfaces com o usuário. 2. ed. Rio de Janeiro: Elsevier, 2007. Rio de Janeiro: 369 p.

BOCK, Cássio. **NPM vs Yarn.** Umbler, 2017. Disponível em: https://blog.umbler.com/br/npm-vs-yarn. Acesso em: 05 nov. 2019

CABAL, Carlos. **React Native:** Construa aplicações móveis nativas com JavaScript. 2016. Disponível em: https://tableless.com.br/react-native-construa-aplicacoes-moveis-nativas-com-javascript/. Acesso em: 02 out. 2019.

CARVALHO, André C. P. L. F. de; LORENA, Ana Carolina. **Introdução à computação:** hardware, software e dados. Rio de Janeiro: LTC, 2017. 175 p.

CELEBRATE. **Celebrate:** Documentation. *[S.l.].* NPM. 2020. Disponível em: https://www.npmjs.com/package/celebrate. Acesso em: 01 jun 2020.

CORDEIRO, Filipe. **Motivos para desenvolver aplicativos Android.** *[S.l.].* AndoidPro. 2019. Disponível em: https://www.androidpro.com.br/blog/carreira /motivos-desenvolver-aplicativos-android/. Acesso em: 01 nov. 2018.

CORS. Cors: Documentation. *[S.l.].* NPM. 2018. Disponível em: https://www.npmjs.com/package/cors. Acesso em: 01 jun 2020.

DUARTE, Tomás. **Os erros mais comuns no atendimento ao cliente.** *[S.l.]*. 2016. Disponível em: https://satisfacaodeclientes.com/os-erros-mais-comuns-no-atendimento-ao-cliente/. Acesso em: 10 nov. 2019.

ECOMANDA. **Como evitar mau atendimento ao cliente em bares.** *[S.l.].* 2019. Disponível em: http://ecomanda.com.br/blog/bares-mau-atendimento-ao-cliente. Acesso em: 10 nov. 2019.

EXPO. **Introduction to Expo**. *[S.l.].* Expo. 2020. Disponível em: https://docs.expo.io/. Acesso em 30 maio 2020.

FACEBOOK. **React Native:** Documentation. *[S.l.]*. Facebook, Inc. 2019. Disponível em: https://pt-br.reactjs.org/docs/getting-started.html. Acesso em: 01 out. 2019.

FREEMAN, Jonathan. **What is an API? Application programming interfaces explained**. *[S.l]*. InfoWorld. 2019. Disponível em: https://www.infoworld.com/

article/3269878/what-is-an-api-application-programming-interfaces-explained.html. Acesso em 07 jun. 2020.

FOWLER, Martin. **UML essencial:** um breve guia para a linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2005. 166 p. Tradução João Tortello.

GALVÃO, Daniel. **Crescimento do mercado mobile altera comportamento sobre uso de apps.** *[S.l.].* DigitalTalks. 2019. Disponível em: https://digitalks.com.br /artigos/crescimento-do-mercado-mobile-altera-comportamento-sobre-uso-de-apps/#. Aceso em 02 nov. 2019.

GSMA. **The mobile economy 2019.** *[S.l.].* GSMA Association. 2019. Disponível em: https://www.gsmaintelligence.com/research/?file=b9a6e6202ee1d5f787cfebb95d3639c5&download. Acesso em 01 out. 2019.

GODIN, Seth. **O crescimento irreversível da tecnologia.** *[S.l.].* 2017. Disponível em: https://administradores.com.br/artigos/o-crescimento-irreversivel-da-tecnologia. Acesso em: 10 nov. 2019.

HEUSER, Carlos Alberto. **Projeto de banco de dados.** 6. ed. Porto Alegre: Bookman, 2009. 282 p.

IDC. **Market Share de smartphones.** *[S.l.].* IDC Corporate. 2019a. Disponível em: https://www.idc.com/promo/smartphone-market-share/vendor. Acesso em: 10 nov. 2019.

IDC. **Market Share de smartphones.** *[S.l.].* IDC Corporate. 2019b. Disponível em: https://www.idc.com/promo/smartphone-market-share/os. Acesso em: 10 nov. 2019.

INSOMNIA. **Introducing Insomnia Designer**. *[S.l.].* Kong, Inc. 2020. Disponível em: https://insomnia.rest/blog/introducing-designer. Acesso em: 30 maio 2020.

INLT. **Intl.js**. *[S.l.].* NPM. 2016. Disponível em: https://www.npmjs.com/package/intl. Acesso em: 30 maio 2020.

KNEX. Knex.js: Documentation. *[S.l.].* 2020. Disponível em: http://knexjs.org/. Acesso em: 01 jun 2020.

LEE, Valentino; SCHNEIDER, Heather; SCHELL, Robbie. **Aplicações móveis:** arquitetura, projeto e desenvolvimento. São Paulo: Pearson Education, 2005. 330 p. Tradução: Amaury Bentes, Deborah Rudiger.

LENON. **Node.js**: O que é, como funciona e quais as vantagens. *[S.l.].* 2018. Disponível em: https://www.opus-software.com.br/node-js/. Acesso em: 01 nov. 2019.

LOOPER, Christian. **From Android 1.0 to Android 10, here's how Google's OS evolved over a decade.** *[S.l.]*. Digital Trends. 2019. Disponível em:

https://www.digitaltrends.com/mobile/android-version-history/. Acesso em: 05 nov. 2019.

MAGALHÃES. **A importância de um bom atendimento ao cliente em restaurante.** *[S.l.].* 2018. Disponível em: https://www.casamagalhaes.com.br/blog /atendimento/atendimento-ao-cliente-em-restaurante/. Acesso em: 10 nov. 2019.

MDN. **Promise**. *[S.l.].* MDN Web docs. 2019. Disponível em: https://developer. mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise. Acesso em: 30 maio 2020.

MEIRELLES, Fernando S. **30ª pesquisa anual do uso de TI nas empresas.** *[S.l].* FGV EAESP. 2019. Disponível em: https://eaesp.fgv.br/ensinoeconhecimento /centros/cia/pesquisa. Acesso em: 03 nov. 2019.

NORTEYSTEM. **A empresa**. *[S.l].* 2016. Disponível em: http://nortesystem.com.br /a-empresa/. Acesso em 01 nov. 2019.

OMG. Specification. Versão 2.5.1. *[S.l.]*: Object Management Group, 2017. Disponível em: https://www.omg.org/spec/UML/2.5.1/PDF. Acesso em: 01 out. 2019.

PEREIRA, Caio Ribeiro. **Node.js:** aplicações web real-time com node.js. São Paulo: Caso do código, 2013. 131 p. Disponível em: https://www.academia.edu/13274789 /Aplica%C3%A7%C3%B5es_web_real_time_com_nodejs_-_tuanny_guinami_libre. Acesso em: 01 nov. 2019.

PEREIRA, Lúcio Camilo Oliva; SILVA, Michel Lourenço. **Android para desenvolvedores.** Rio de Janeiro: Brasport, 2009. 219 p.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software:** uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016. 940 p. Tradução: João Eduardo Nóbrega Tortello; Revisão técnica: Reginaldo Arakaki, Júlio Arakaki, Renato Manzan de Andrade.

POSTGRESQL. **Documentation.** *[S.l.]*: PostgreSQL Global Development Group, 2019. Tradução: PostgreSQL Development Group Brasil. Disponível em: https://www.postgresql.org/docs/. Acesso em: 01 out. 2019.

RATIONAL. **Rational Unified Process:** Best Practices for Software Development Teams. Cupertino: Rational Corporation, 1998. Disponível em: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251 _bestpractices_TP026B.pdf. Acesso em: 09 nov. 2019.

RECLAMEAQUI. **Tipos de problemas de Bares e Restaurantes.** *[S.l.].* 2019. Disponível em: https://www.reclameaqui.com.br/bares-e-restaurantes/. Acesso em: 01 nov. 2019.

REDHAT. **O que significa API e como ela funciona**. *[S.l.]*. RedHat, Inc. 2020. Disponível em: https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces. Acesso em: 07 jun. 2020.

REZENDE, Denis Alcides. **Engenharia de Software e Sistemas de Informação.** 2. ed. Rio de Janeiro: Brasport, 2002. 358 p.

SBROCCO, José Henrique Teixeira de Carvalho. **UML 2.3:** teoria e prática. São Paulo: Érica, 2011. 270 p.

SBROCCO, José Henrique Teixeira de Carvalho; MACHADO, Paulo Cesar de. **Métodos ágeis:** engenharia de software sob medida. São Paulo: Érica, 2012. 254 p.

SCHLINKERT, Jhon. **Time-Stamp:** Documentation. *[S.l.].* NPM. 2018. Disponível em: https://www.npmjs.com/package/time-stamp. Acesso em: 01 jun 2020.

SHARP, Remy. **Nodemon**: Documentation. *[S.l.].* GitHub, Inc. 2020. Disponível em: https://github.com/remy/nodemon. Acesso em: 01 jun 2020.

SOBRAL, Wilma Sirlange. **Design de Interfaces:** introdução. São Paulo: Érica, 2019. 152 p.

SOMMERVILLE, Ian. **Engenharia de Software.** 8. ed. São Paulo: Pearson Addison Wesley, 2007. 552 p. Tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edílson de Andrade Barbosa; Revisão técnica: Kechi Kirama.

SPRING. **O que é um banco de dados relacional.** [S.l.]: SPRING – DPI/INPE, 2019. Disponível em: http://www.dpi.inpe.br/spring/teoria/consulta/consulta.html. Aceso em: 01 out. 2019.

STRONGLOOP. **Express**: Documentation. *[S.l.]*. StrongLoop, Inc. 2020. Disponível em: http://expressjs.com/pt-br/. Acesso em: 01 jun 2020.

ZANINI. **Aplicativo:** o que é, como funciona e para que serve. [S.l.]. 2017. Disponível em: https://blog.za9.com.br/aplicativo-o-que-e-como-funciona-e-para-que-serve/. Acesso em: 11 nov. 2019.

APÊNDICE A – Detalhamento dos Casos de Uso

Neste apêndice são apresentados o detalhamento de todos os casos de uso conforme a Figura 10.

Quadro 4 – Detalhamento do caso de uso realizar login (UC01)

Número	001
Caso de Uso	UC01 – Realizar login
Ator	Usuário
Descrição	Descreve o fluxo para o usuário logar no aplicativo.
Pré-condições	Um usuário já deve estar cadastrado no sistema retaguarda Katarina.
Fluxo principal	01 – O usuário informa o login; 02 – O usuário informa a senha; 03 – O usuário clica no botão entrar; 04 – A aplicação redireciona o usuário para tela principal.

Fonte: elaborado pelo autor

Quadro 5 – Detalhamento do caso de uso abrir mesa (UC02)

Número	002
Caso de Uso	UC02 – Abrir mesa
Ator	Usuário
Descrição	Descreve o fluxo para o usuário abrir/iniciar uma mesa.
Pré-condições	Ter realizado o Login.
Fluxo principal	01 – O usuário clica no botão novo, representado pelo símbolo de mais (+), localizado no canto inferior direito; 02 – A aplicação solicita qual número da mesa o usuário deseja abrir; 03 – O usuário informa o número da mesa desejada e clica em confirmar; 04 – A aplicação redireciona o usuário para tela de categorias; 05 – Execução do UC04 – Lançar pedidos.
Fluxo Alternativo	A – Se o usuário clicar sobre uma mesa já iniciada; B – A aplicação redireciona o usuário para tela de categorias;

Fonte: elaborado pelo autor

Quadro 6 – Detalhamento do caso de uso abrir comanda (UC03)

Número	003
Caso de Uso	UC03 – Abrir comanda
Ator	Usuário
Descrição	Descreve o fluxo para o usuário abrir/iniciar uma comanda.
Pré-condições	Ter realizado o Login.
Fluxo principal	01 – O usuário clica no botão novo, representado pelo símbolo de mais (+), localizado no canto inferior direito; 02 – A aplicação solicita qual o número da comanda o usuário deseja abrir; 03 – O usuário informa o número da comanda desejada e clica em confirmar; 04 – A aplicação redireciona o usuário para tela de categorias; 05 – Execução do UC04 – Lançar pedidos.
Fluxo alternativo	A – Se o usuário clicar sobre uma comanda já iniciada; B – A aplicação redireciona o usuário para tela de resumo;

Fonte: elaborado pelo autor

Quadro 7 – Detalhamento do caso de uso lançar pedido (UC04)

Número	004
Caso de Uso	UC04 – Lançar pedido
Ator	Usuário
Descrição	Descreve o fluxo para o usuário lançar produtos em uma mesa ou comanda.
Pré-condições	Ter realizado o Login; Possuir mesas, ou comandas iniciadas anteriormente.
Fluxo principal	01 – O usuário seleciona a categoria desejada; 02 – A aplicação redireciona o usuário para tela de produtos; 03 – O usuário seleciona o produto a ser lançado; 04 – A aplicação redireciona o usuário para a tela de venda e aguarda o usuário informar a quantidade do produto; 05 – O usuário informa a quantidade desejada e clica em adicionar; 06 – A aplicação lança o produto na mesa ou comanda; 07 – A aplicação permanece na tela de venda aguardando o usuário efetuar uma das ações: novo item, cancelar, confirmar pedido;

Fonte: elaborado pelo autor

Quadro 8 – Detalhamento do caso de uso visualizar pedidos (UC05)

Número	005
Caso de Uso	UC05 – Visualizar pedido

Ator	Usuário
Descrição	Descreve o fluxo para o usuário visualizar os produtos em uma mesa ou comanda.
Pré-condições	Ter realizado o Login; Possuir mesas, ou comandas iniciadas anteriormente.
Fluxo principal	01 – O usuário clica em uma mesa ou comanda dentre as já abertas e listadas no *grid.* 02 – A aplicação redireciona o usuário para tela de resumo.
Fluxo alternativo	A – Se o usuário clicar no botão *voltar*; B – A aplicação redirecionará o usuário para a tela de listagem de mesa ou comanda.

Fonte: elaborado pelo autor

Quadro 9 – Detalhamento do caso de uso buscar mesa (UC06)

Número	006
Caso de Uso	UC06 – Buscar mesa
Ator	Usuário
Descrição	Descreve o fluxo para o buscar uma mesa específica dentre as listadas no *grid* de visualização.
Pré-condições	Ter realizado o Login; Possuir mesas, ou comandas iniciadas anteriormente.
Fluxo principal	01 – O usuário informa o número da mesa no campo de busca; 02 – O usuário clica no botão buscar; 03 – A aplicação localiza a mesa informada e apresenta apenas ela no *grid* de visualização;

Fonte: elaborado pelo autor

Quadro 10 – Detalhamento do caso de uso buscar comanda (UC07)

Número	007
Caso de Uso	UC07 – Buscar comanda
Ator	Usuário
Descrição	Descreve o fluxo para o buscar uma comanda específica dentre as listadas no *grid* de visualização.
Pré-condições	Ter realizado o Login; Possuir mesas, ou comandas iniciadas anteriormente.
Fluxo principal	01 – O usuário informa o número da comanda no campo de busca; 02 – O usuário clica no botão buscar; 03 – A aplicação localiza a comanda informada e apresenta apenas ela no *grid* de visualização;

Fonte: elaborado pelo autor

APÊNDICE B – Dicionário de Dados

Neste apêndice contém o dicionário de dados com base no diagrama de entidade relacionamento apresentado na Figura 19. Nos quadros abaixo são descritas as seguintes informações:

- Entidade: É o nome da entidade definida no DER (Diagrama entidade relacionamento);
- Atributo: São as características da entidade a serem guardadas no banco de dados;
- Descrição: Breve descrição do que o atributo representa na entidade;
- Tipo: Tipo primitivo que o atributo armazena no banco de dados;
- Tamanho: Quantidade de caracteres que são necessários para armazenar o conteúdo do atributo;
- Restrições: Descreve as possíveis restrições que o atributo possui para seu armazenamento.

Quadro 11 – Dicionário de dados da entidade categoria

Entidade: Categoria				
Atributo	**Descrição**	**Tipo**	**Tamanho**	**Restrições**
cat_001	Código de identificação da entidade	Integer		PK
emp_001	Código de identificação da empresa.	Integer		Not null
cat_002	Descrição da categoria	Character varying	40	Not null
sit_001	Situação	Integer		Not null
cat_visivel	Visibilidade da categoria	Boolean		
cat_ordem	Ordem de exibição da categoria	Character	3	

Fonte: elaborado pelo autor

Quadro 12 – Dicionário de dados da entidade materiais

Entidade: Materiais				
Atributo	**Descrição**	**Tipo**	**Tamanho**	**Restrições**
mat_001	Código de identificação da entidade	Integer		PK
emp_001	Código de identificação da empresa	Integer		Not null
mat_003	Descrição do produto	Character varying	100	Not null
mat_008	Valor unitário do produto	Numeric	15,2	Not null
sit_001	Situação	Integer		Not null
cat_001	Código de identificação da categoria	Integer		FK
mat_021	Código do setor da impressora do item	Integer		
mat_visivel	Visibilidade do material	Boolean		

Fonte: elaborado pelo autor

Quadro 13 – Dicionário de dados da entidade caixa

Entidade: caixa				
Atributo	**Descrição**	**Tipo**	**Tamanho**	**Restrições**
Id_caixa	Código de identificação da entidade	Integer		PK
Id_empresa	Código de identificação da empresa	Integer		Not null
data_abertura	Data da abertura do caixa	Character varying	40	Not null
hora_abertura	Hora da abertura do caixa	Integer		Not null
Id_situacao	Situação	Integer		Not null

Fonte: elaborado pelo autor

Quadro 14 – Dicionário de dados da entidade usuarios

Entidade: Usuarios				
Atributo	**Descrição**	**Tipo**	**Tamanho**	**Restrições**
usu_001	Código de identificação da	Integer		PK

	entidade			
usu_002	Nome do usuário	Character varying	30	Not null
usu_003	Login	Character varying	30	
usu_004	Senha	Character varying	30	

Fonte: elaborado pelo autor

Quadro 15 – Dicionário de dados da entidade venda

Entidade: Venda				
Atributo	**Descrição**	**Tipo**	**Tamanho**	**Restrições**
ven_001	Código de identificação da entidade	Integer		PK
emp_001	Código de identificação da empresa	Integer		Not null
ven_004	Data inclusão da venda na base de dados	Timestamp		
sit_001	Situação	Integer		Not null
usu_001_1	Usuário que iniciou a venda	Integer		FK
dat_001_1	Data e hora que a venda foi criada	Timestamp		
ven_009	Valor total da venda	Numeric	15,2	
ven_024	Tipo de venda	Character varying		
ven_025	Número da mesa	Integer		
ven_023		Character	1	
ven_026	Número da comanda	Integer		
ven_029	Número da venda	Integer		
id_caixa_abertura	Código de identificação do caixa aberto	Integer		FK
terminal_abertura	Terminal que iniciou a venda	Character varying	100	

Fonte: elaborado pelo autor

Quadro 16 – Dicionário de dados da entidade vendaItem

Entidade: VendaItem				
Atributo	**Descrição**	**Tipo**	**Tamanho**	**Restrições**
ite_001	Código da sequência do item na venda	Integer		PK
emp_001	Código de identificação da empresa	Integer		Not null
ven_001	Código de identificação da venda	Integer		FK

ite_002	Quantidade do item	Numeric	10,4	Not null
ite_003	Valor unitário do item	Numeric	10,2	Not null
ite_005	Valor total do item	Numeric	10,2	
ite_006	Observação	Character varying	200	
ite_008	Status da impressão para cozinha	Character varying		
sit_001	Situação	Integer		
gar_001	Código de identificação do usuário que lançou o item na venda	Integer		FK
ite_011	Status se o item deve ser impresso	Character varying	1	
data_hota_lacamento	Data e hora do lançamento do item	Timestamp		Not null
numero_pedido	Número do pedido	Integer		
mat_001	Código de identificação do produto	Integer		FK

Fonte: elaborado pelo autor

APÊNDICE C – Documento de Projeto de Testes

Neste apêndice é apresentado o documento de projeto de teste. Tal documento é de fundamental importância no desenvolvimento da aplicação, este pode garantir o sucesso do projeto, pois se todos os requisitos funcionais forem testados e a conclusão for positiva, provavelmente o cliente e ou usuário ficará satisfeito.

Histórico de Revisão

- Versão: 1.0
- Data: 09/11/2019
- Plano de Teste para o Release 1.0
- Autor: Lucas Felício

Sumário

1 INTRODUÇÃO

Este documento tem como objetivo documentar as informações, através do plano de testes, necessárias para planejar e controlar os testes de validação do projeto da aplicação Katarina Mobile v.1.0.

1.1 Escopo

Este documento descreve o Plano de Testes a ser usado pelo projeto mobile – Katarina v.1.0 para avaliar a qualidade funcional, confiabilidade e performance.

1.2 Referências

REZENDE, Denis Alcides. **Engenharia de Software e Sistemas de Informação.** 2. ed. Rio de Janeiro: Brasport, 2002. 358 p. 1ª Reimpressão: 2003.

SOMMERVILLE, Ian. **Engenharia de Software.** 8. ed. São Paulo: Pearson Addison Wesley, 2007. 552 p. Tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edílson de Andrade Barbosa; Revisão técnica: Kechi Kirama. 3ª Reimpressão: 2009.

2 REQUISITOS DE TESTES

A lista abaixo identifica os itens que foram identificados como alvos dos testes. Nesta lista são apresentados o que será testado e posteriormente será detalhado através dos Casos de Teste.

2.1 Teste de integridade dos dados e do banco de dados

- Verificar o acesso ao banco de dados
- Verificar recuperação correta de atualizações do banco de dados

2.2 TESTE DE SISTEMA

- Verificar Caso de Uso Realizar Login (UC01)
- Verificar Caso de Uso Abrir Mesa (UC02)
- Verificar Caso de Uso Abrir Comanda (UC03)
- Verificar Caso de Uso Lançar Pedido (UC04)
- Verificar Caso de Uso Visualizar Pedido (UC05)
- Verificar Caso de Uso Buscar Mesa (UC06)
- Verificar Caso de Uso Buscar Comanda (UC07)

3 PLANEJAMENTO PARA OS TESTES

As tabelas a seguir apresentam os recursos necessários para a execução dos Casos de Teste.

3.1 NECESSIDADE DE HARDWARE

Quadro 17 – Necessidades de hardware para o plano de testes

Tipo de Hardware	Detalhamento	Quantidade	Disponibilização
Smartphone	S.O Android v. 9.0	1	Próprio
Smartphone	S.O Android v. 8.0	1	NorteSystem

Fonte: elaborado pelo autor

3.2 NECESSIDADE DE *SOFTWARE*

Quadro 18 – Necessidade de software para o plano de testes

Tipo de *Software*	Detalhamento	Quantidade	Disponibilização
Visual Studio Code	v. 1.40 superior	1	Microsoft Disponível em: https://code.visualstudio.com/
Insomnia	v. 7.0 superior	1	Kong Inc. Disponível em: https://insomnia.rest/
pgAdmin 4	v. 4.13 superior	1	PostgreSQL Tools Disponível em: https://www.pgadmin.org/

Fonte: elaborado pelo autor

3.3 NECESSIDADE DE PESSOAS

Quadro 19 – Necessidade de pessoas para o plano de testes

Papel	Horas de envolvimento	Quantidade	Envolvimento
Testador	2 horas por semana	1	Março a junho de 2020

Fonte: elaborado pelo autor

4 CASOS DE TESTE

Nesta seção são apresentados os casos de teste definidos baseando-se nos requisitos de testes listados no tópico 2.

Quadro 20 – Caso de Teste 001 verificar acesso ao banco de dados

Caso Nº	CT001 – Verificar acesso ao banco de dados
Objetivo do Teste	Verificar se aplicação está se comunicando corretamente com o banco de dados.
Passos	1 – Utilizando a ferramenta Insomnia:

	Acionar um método do tipo *GET* 2 – Verificar no console se houve retorno de dados
Critérios de êxito	O Testador deve conseguir visualizar dados contidos no banco de dados.

Fonte: elaborado pelo autor

Quadro 21 – Caso de Teste 002 verificar recuperação de dados do BD

Caso Nº	CT002 – Verificar recuperação correta de atualizações do banco de dados
Objetivo do Teste	Verificar se aplicação está retornando corretamente os dados presentes nas tabelas do banco de dados.
Passos	1 – Utilizando a ferramenta Insomnia: Acionar um método do tipo *GET* 2 – Verificar no console se houve retorno de dados 3 – Utilizando o browser da ferramenta pgAdmin 4: Comparar dados do retorno do item 1 com os dados contidos nas tabelas no banco de dados
Critérios de êxito	O Testador deve conseguir visualizar dados e compara-los com o retorno do GET e no browser do pgAdmin 4

Fonte: elaborado pelo autor

Quadro 22 – Caso de Teste 004 Verificar caso de uso realizar login

Caso Nº	CT004 – Verificar caso de uso realizar login
Objetivo do Teste	Verificar se o usuário consegue realizar login na aplicação
Passos	1 – Acessar o aplicativo mobile Katarina 2 – Informar o usuário 3 – Informar a senha 4 – Clicar em entrar
Critérios de êxito	O usuário deve conseguir acessar a próxima ela, grid de mesas.

Fonte: elaborado pelo autor

Quadro 23 – Caso de Teste 005 verificar caso de uso abrir mesa

Caso Nº	CT005 – Verificar caso de uso abrir mesa
Objetivo do Teste	Verificar se o usuário consegue realizar a abertura de uma nova mesa
Passos	1 – Executar o CT004 2 – Clicar no botão novo (+) 3 – Informar o número da mesa desejada 4 – Clicar em confirmar
Critérios de êxito	O usuário deve conseguir abrir uma nova mesa e ser direcionado para próxima tela, categorias.

Fonte: elaborado pelo autor

Quadro 24 – Caso de Teste 006 verificar caso de uso abrir comanda

Caso Nº	CT006 – Verificar caso de uso abrir comanda
Objetivo do Teste	Verificar se o usuário consegue realizar a abertura de uma nova comanda

Passos	1 – Executar o CT004 2 – Clicar no botão novo (+) 3 – Informar o número da comanda desejada 4 – Clicar no botão confirmar
Critérios de êxito	O usuário deve conseguir abrir uma nova comanda e ser direcionado para próxima tela, categorias.

Fonte: elaborado pelo autor

Quadro 25 – Caso de Teste 007 verificar caso de uso lançar pedido

Caso Nº	CT007 – Verificar caso de uso lançar pedido
Objetivo do Teste	Verificar se o usuário consegue realizar o lançamento do pedido tanto na mesa quanto na comanda
Passos	1 – Executar o CT004 2 – Executar o CT005 ou CT006 3 – Selecionar uma categoria 4 – Selecionar um produto 5 – Informar a quantidade do produto 6 – Clicar no botão confirmar
Critérios de êxito	O usuário deve conseguir visualizar uma resposta da aplicação informando que o produto foi lançado no banco de dados

Fonte: elaborado pelo autor

Quadro 26 – Caso de Teste 008 verificar caso de uso visualizar pedido

Caso Nº	CT008 – Verificar caso de uso visualizar pedido
Objetivo do Teste	Verificar se o usuário consegue visualizar os pedidos de uma mesa ou uma comanda selecionada
Passos	1 – Executar o CT004 2 – Selecionar uma mesa ou comanda disponível no grid de comandas/ mesas
Critérios de êxito	O usuário deve conseguir visualizar os dados do pedido na tela de resumo

Fonte: elaborado pelo autor

Quadro 27 – Caso de Teste 009 verificar caso de uso buscar mesa

Caso Nº	CT009 – Verificar caso de uso buscar mesa
Objetivo do Teste	Verificar se o usuário consegue buscar uma mesa, podendo ela estar aberta ou não
Passos	1 – Executar o CT004 2 – Informar o número da mesa no campo de busca; 3 – Clicar no botão buscar;
Critérios de êxito	O usuário deve conseguir visualizar apenas a mesa informada no grid de visualização caso a mesma estiver aberta

Fonte: elaborado pelo autor

Quadro 28 – Caso de Teste 010 verificar caso de uso buscar comanda

Caso Nº	CT010 – Verificar caso de uso buscar comanda
Objetivo do Teste	Verificar se o usuário consegue buscar uma comanda, podendo ela estar aberta ou não
Passos	1 – Executar o CT004

	2 – Informar o número da comanda no campo de busca; 3 – Clicar no botão buscar;
Critérios de êxito	O usuário deve conseguir visualizar apenas a comanda informada no grid de visualização caso a mesma estiver aberta

Fonte: elaborado pelo autor

APÊNDICE D – Listas de códigos do desenvolvimento da api

Neste apêndice são apresentados os principais arquivos produzidos durante o processo do desenvolvimento da API, bem como listas dos principais códigos que descrevem as funcionalidades destes arquivos.

Knexfile.js: arquivo gerado pela biblioteca de consultas SQL Knex. Este arquivo tem por objetivo armazenar e prover configurações para a comunicação com o banco de dados. Nele encontram-se as informações de qual tipo de cliente de banco de dados será utilizado, informações de acesso como usuário, senha, porta de comunicação, nome do banco de dados e endereço do servidor do banco de dados.

Listagem 1 – Listagem de código JavaScript do arquivo knexfile.js

knexfile.js src ×

backend > src > knexfile.js > ...

```
1   module.exports = {
2     production: {
3       client: 'postgresql',
4       connection: {
5         database: 'nortesystem',// nome do banco de dados
6         user: 'nortesystem',//usuário do banco de dados
7         password: '[redacted]', //senha do usuário do banco de dados
8         port: 5432,//porta do serviço do banco de dados
9         host: 'localhost' //endereço do servidor do banco de dados
10      },
11      pool: {
12        min: 0,
13        max: 50
14      },
15      useNullAsDefault: true,
16    },
```

Fonte: elaborado pelo autor

Connection.js: arquivo responsável pela comunicação com o banco de dados PostgreSQL do sistema Katarina. Este arquivo requisita a biblioteca Knex,

importa as configurações do arquivo knexfile.js, cria e exporta uma variável *connection* responsável por instanciar uma conexão com o banco de dados.

Listagem 2 – Listagem de código JavaScript do arquivo Connection.js

connection.js database

backend > src > database > connection.js > ...

```
1  const knex = require('knex');
2  const configuration = require('../knexfile');
3
4  const connection = knex(configuration.production);
5
6  module.exports = connection;
7
```

Fonte: elaborado pelo autor

SessionController.js: arquivo responsável por buscar por um usuário no banco de dados do sistema Katarina. Este arquivo requere, através da variável *connection*, a conexão com o banco de dados e possui apenas uma função, *read*, que recebe dois parâmetros: login e senha. Utilizando a sintaxe da biblioteca Knex, através da variável *connection*, monta-se um SQL passando na seção do *where* os parâmetros de login e senha recebidos. Ao obter o retorno do banco de dados verifica-se há existência de dados retornados pela função. Caso a resposta seja negativa a função *read* tem como retorno status 401[18] e um JSON com uma descrição, caso seja positivo o retorno são os dados do usuário obtidos na consulta SQL.

18 Código padrão HTTP que especifica que o cliente não pode ser autenticado e ou autorizado.

Listagem 3 – Listagem de código JavaScript do arquivo SessionController.js

SessionController.js controllers

backend > src > controllers > SessionController.js > ...

```
1   const connection = require('../database/connection');
2   module.exports = {
3       async read(request, response) {
4           const { login, senha } = request.query;
5           const user = await connection('usuarios')
6               .where({ 'usu_003': login, 'usu_004': senha })
7               .select('usu_001 as id', 'usu_002 as user_name')
8               .first();
9           if (!user) {
10              return response.status(400)
11                  .json({
12                      descricao: 'Usuário inexistente ou login/senha inválidos'
13                  });
14          }
15          return response.status(200).json(user);
16      }
17  }
```

Fonte: elaborado pelo autor

CategoriaController.js: arquivo responsável por listar todas categorias de produtos disponíveis na base de dados do sistema Katarina. Este arquivo requere uma conexão com o banco de dados através da variável *connection*. Possui apenas uma função, *read*, que é responsável por buscar e retornar as categorias de produtos disponíveis para a empresa passada como parâmetro através da variável *id_empresa*. Após obter o retorno do banco de dados verifica-se a existência de dados. Caso não aja dados retornados pelo SQL, o retorno tem como status 404[19] e um JSON com uma descrição, caso aja dados retornados pelo SQL, o retorno da função são os próprios dados encontrados, através de um objeto JSON juntamente com um status 200[20].

19 Código padrão HTTP quando o servidor não pode encontrar os recursos solicitados
20 Código padrão HTTP quando o retorno da função for concluído com sucesso

Listagem 4 – Listagem de código JavaScript do arquivo CategoriaController.js

CategoriaController.js controllers

backend > src > controllers > CategoriaController.js > ...

```
1  const connection = require('../database/connection');
2  module.exports = {
3      async read(request, response) {
4          const { id_empresa } = request.query;
5          try {
6              const categorias = await connection('categoria')
7                  .where({
8                      'emp_001': id_empresa, 'sit_001': 4, 'cat_visivel': true
9                  })
10                 .select('cat_001', 'cat_002')
11                 .orderBy('cat_ordem', 'asc')
12             if (!categorias) {
13                 return response.status(404)
14                     .json({ descricao: 'Nenhuma categoria encontrada' })
15             } else {
16                 return response.status(200).json(categorias);
17             }
18         } catch (error) {
19             return response.status(500)
20                 .json({ descricao: 'Erro no servidor: ' + error })
21         }
22     },
23 };
```

Fonte: elaborado pelo autor

ProdutoController.js: arquivo responsável por listar todos produtos de uma determinada categoria, passada como parâmetro para a função *read*. Este arquivo requere uma conexão com o banco de dados através da variável *connection*. A função *read* recebe dois parâmetros *id_categoria e id_empresa*, que ao ser executada retorna os produtos associados a essa determinada categoria. Caso o retorno da consulta possua dados, os mesmos serão retornados através de um objeto JSON, caso não aja dados, será retornado um status 404 e uma descrição.

Listagem 5 – Listagem de código JavaScript do arquivo ProdutoController.js

ProdutoController.js controllers

backend > src > controllers > ProdutoController.js > ...

```
1   const connection = require('../database/connection');
2   module.exports = {
3       async read(request, response) {
4           const { id_categoria, id_empresa } = request.query;
5           try {
6               const produtos = await connection('materiais')
7                   .where({
8                       'emp_001': id_empresa, 'sit_001': 4,
9                       'mat_visivel': true,
10                      'cat_001': id_categoria
11                  })
12                  .select('mat_001', 'mat_003', 'mat_008', 'mat_021')
13                  .orderBy('mat_003', 'asc')
14              if (!produtos) {
15                  response.status(404)
16                      .json({ error: 'Nenhum produto encontrado.' })
17              } else {
18                  return response.status(200).json(produtos);
19              }
20          } catch (error) {
21              return response.status(500)
22                  .json({ descricao: 'Erro no servidor: ' + error })
23          }
24      },
```

Fonte: elaborado pelo autor

VendaController.js: arquivo responsável por prover a listagem e inserção das vendas e dos itens no banco de dados do sistema Katarina. Assim como os demais arquivos, ele requere uma conexão com o banco de dados através da variável *connection*. Este arquivo possui três funções, demonstradas na Listagem 6: *read, create, createItem*.

A função *read*, declarada entre as linhas 4 e 30, tem como objetivo buscar e listar todas vendas válidas e abertas, verificadas através do campo *'sit_001 = 8'*. Também é verificado qual tipo de venda deverá ser buscada pela consulta SQL, através do parâmetro *tipo*, que pode assumir valor '*C*' para comandas e '*M*' para mesas. Na montagem do SQL é verificado se o parâmetro *nro_com_mesa* possui algum valor informado, caso possua, a função buscará por apenas comandas ou mesas que sejam iguais ao do parâmetro informado, caso contrário, todas as vendas

abertas e válidas serão retornadas pela consulta ao banco de dados. Ao realizar a consulta SQL a função *read* verifica se ocorreu algum erro ou se o procedimento foi bem sucedido, caso a verificação seja negativa a função tem como retorno um status 401 e um JSON com uma descrição, caso seja positivo o retorno será um status 200 e um JSON com as informações das vendas localizadas.

A função *create*, declarada entre as linhas 31 e 74, tem por objetivo inserir uma nova venda no banco de dados do sistema Katarina. Esta função recebe uma série de parâmetros pertinentes para a criação de uma nova venda. Num primeiro momento a função *create* faz uma busca no banco de dados para retornar o ultimo *id* (identificador) da última venda inserida no banco de dados, a fim de dar continuidade na sequência numérica das vendas. Em seguida soma mais um digito ao *id* e atribui o resultado à variável *id_venda*. Abaixo monta-se um SQL de inserção, utilizando a sintaxe da biblioteca Knex, passando os parâmetros recebidos e declarados no escopo da função. Ao fim da execução da função *create* é retornado um status 200, de sucesso na execução, e um JSON com o *id_venda* da nova venda inserida no banco de dados. Caso aja algum problema na execução da função, um status 500[21] e uma descrição contendo o motivo do erro é retornado.

A função *ceateItem,* declarada entre as linhas 75 e 113, tem por objetivo inserir um novo registro de um novo item para uma venda em específico. Esta função recebe uma série de parâmetros pertinentes para a montagem do SQL de inserção de item. Antes de inserir o item na tabela no banco de dados, esta função faz uma consulta na tabela de itens no banco de dados, a fim de verificar se já existem itens registrados associados a venda passada como parâmetro. O retorno desta consulta é armazenado na variável *item*. Em seguida é acrescido um digito a esta variável, com proposito de iniciar ou manter uma sequência dos itens referente a uma determinada venda. Após esse procedimento, os parâmetros recebidos no início do escopo da função *createItem* são passados para a montagem do SQL, para finalmente serem inseridos no banco de dados. Após a execução da inserção dos dados no banco de dados, é executado uma *function*[22] que é responsável por atualizar o total da venda com base nos itens inseridos na tabela de itens. Por fim

21 Código padrão HTTP que representa um erro ocorrido na execução de algum procedimento do lado do servidor

22 Functions, em português funções, são instrumentos do gerenciador de banco de dados que são capazes de controlar e executar instruções SQL através da utilização de uma linguagem procedural (POSTGRESQL, 2019).

caso a execução da função seja bem sucedida haverá um retorno com status 200 e uma descrição. Caso ocorra algum problema, será retornado um JSON com uma descrição do erro e um status 500.

Listagem 6 – Listagem de código JavaScript do arquivo VendaController.JS

VendaController.js controllers

backend > src > controllers > VendaController.js > ...

```
1   const connection = require('../database/connection');
2   const timestamp = require('time-stamp');
3   module.exports = {
4       async read(request, response) {
5           const { tipo, nro_com_mesa, id_empresa } = request.query;
6           try {
7               let select = connection
8                   .select('ven_001', 'ven_009', 'ven_026', 'ven_025')
9                   .from('venda')
10                  .orderBy('venda.ven_001', 'asc')
11                  .where({ 'emp_001': id_empresa, 'sit_001': 8, 'ven_024': tipo })
12              if (nro_com_mesa != undefined && nro_com_mesa != 0) {
13                  select = select.andWhere(function () {
14                      this.where('ven_025', nro_com_mesa)
15                          .orWhere('ven_026', nro_com_mesa)
16                  })
17              }
18              const vendas = await select
19              if (!vendas) {
20                  return response.status(401)
21                      .json({ descricao: 'Erro ao consultar venda(s)!' })
22              }
23              else {
24                  return response.status(200).json(vendas);
25              }
26          } catch (error) {
27              return response.status(500)
28                  .json({ descricao: 'Erro no servidor: ' + error })
29          }
30      },
```

```
31      async create(request, response) {
32          const id_usuario = request.headers.authorization;
33          const { tipo, nro_comanda, nro_mesa, id_empresa } = request.body;
34          console.log(tipo, nro_comanda, nro_mesa, id_empresa)
35          try {
36              const [id] = await connection('venda')
37                  .where('emp_001', id_empresa)
38                  .max('ven_001')
39              id_venda = id['max'] + 1;
40              const [caixa] = await connection('caixa')
41                  .select('id_caixa')
42                  .where({
43                      'id_empresa': id_empresa, 'id_situacao': 4
44                  });
45              id_caixa = caixa['id_caixa'];
46              await connection('venda')
47                  .insert({
48                      'ven_001': id_venda,
49                      'ven_002': 0,
50                      'emp_001': id_empresa,
51                      'dat_001_1': timestamp('YYYY-MM-DD HH:mm:ss.ms'),
52                      'ven_025': nro_mesa,
53                      'sit_001': 8,
54                      'usu_001_1': id_usuario,
55                      'ven_024': tipo,
56                      'ven_029': id_venda,
57                      'ven_004': timestamp('YYYY-MM-DD HH:mm:ss.ms'),
58                      'ven_023': 'N',
59                      'id_caixa_abertura': id_caixa,
60                      'ven_026': nro_comanda,
61                      'terminal_abertura': 'Aplicativo mobile'
62                  });
```

```
46              await connection('venda')
47                  .insert({
48                      'ven_001': id_venda,
49                      'ven_002': 0,
50                      'emp_001': id_empresa,
51                      'dat_001_1': timestamp('YYYY-MM-DD HH:mm:ss.ms'),
52                      'ven_025': nro_mesa,
53                      'sit_001': 8,
54                      'usu_001_1': id_usuario,
55                      'ven_024': tipo,
56                      'ven_029': id_venda,
57                      'ven_004': timestamp('YYYY-MM-DD HH:mm:ss.ms'),
58                      'ven_023': 'N',
59                      'id_caixa_abertura': id_caixa,
60                      'ven_026': nro_comanda,
61                      'terminal_abertura': 'Aplicativo mobile'
62                  });
63              if (!id_venda) {
64                  return response.status(400)
65                      .json({ descricao: 'Erro ao inserir a venda' });
66              }
67              else {
68                  return response.status(200).json(id_venda);
69              }
70          } catch (error) {
71              return response.status(500)
72                  .json({ descricao: 'Erro no servidor: ' + error })
73          }
74      },
75      async createItem(request, response) {
76          const id_usuario = request.headers.authorization;
77          const { id_empresa, id_venda, id_produto, quantidade, valor_unit,
78              valor_total, observacao, id_impressora } = request.query;
79          try {
80              const [item] = await connection('vendaitem').max('ite_001')
81                  .where({
82                      "emp_001": id_empresa,
83                      "ven_001": id_venda
84                  });
85              nro_item = item['max'] + 1
```

```
            await connection('vendaitem')
                .insert({
                    'emp_001': id_empresa,
                    'ven_001': id_venda,
                    'mat_001': id_produto,
                    'sit_001': 4,
                    'ite_001': nro_item,
                    'ite_002': quantidade,
                    'ite_003': valor_unit,
                    'ite_005': valor_total,
                    'ite_006': observacao,
                    'ite_008': 'N',
                    'ite_011': 'S',
                    'ite_012': 'N',
                    'ite_013': id_impressora,
                    'gar_001': id_usuario,
                    'tamanho': 'M',
                    'b_venda_tamanho': false,
                    'quantidade_impressao': 1,
                });
            await connection.raw(`select fn_calcula_total_venda
                (${id_venda},${id_empresa})`)
            return response.status(200).send();
        } catch (error) {
            return response.status(500)
                .json({ descricao: 'Erro no servidor: ' + error })
        }
    },
};

```

APÊNDICE E – Listas de códigos do desenvolvimento do aplicativo móvel

Neste apêndice são apresentados os principais arquivos das telas de interface gráfica do aplicativo móvel, assim como as listagens dos principais códigos que descrevem as funcionalidades destes arquivos.

Api.JS: arquivo responsável por prover a comunicação com o *back-end* (api), demonstrado na Listagem 12. Na linha 1 é realizado a importação da biblioteca *Axios*. Da linha 3 a 5 são realizados a criação de uma instancia da interface do axios e armazenado na *property baseURL* a informação do endereço do servidor que a api está sendo executada. Por fim na linha 7 é exportado esta instancia do axios através da variável *api*.

Listagem 7 – Listagem de código JavaScript do arquivo Api.JS

```
api.js services ×
mobile > src > services > api.js > ...
1  import axios from 'axios';
2  const api = axios.create({
3      baseURL: `http://192.168.2.8:3333`
4  });
5
6  export default api;
7
8
```

Fonte: elaborado pelo autor

ButtonsBox.JS: arquivo que tem como objetivo exportar um componente contendo dois botões ‘Comandas’ e ‘Mesas’ com funções de navegação. Este componente é requerido pelas telas de Comandas e Mesas. Da linha 1 a linha 4, conforme demonstrado na Listagem 8, são importados os módulos necessários para a criação da interface do componente. Entre as linhas 9 e 22 são adicionados os componentes React Native que representam os botões ‘Comandas’ e ‘Mesas’. O botão ‘Comandas’ ao ser clicado aciona a função *navigationToComandas*, declarada entre as linhas 11 e 15, que tem por finalidade redirecionar o usuário para tela de Comandas. O botão ‘Mesas’ possui o mesmo comportamento, contudo aciona a

função *navigationToMesas*, declarado entre as linhas 16 e 20, e tem por finalidade navegar para tela Mesas.

Listagem 8 – Listagem de código JavaScript do arquivo buttonsBox.JS

buttonsBox.js components

mobile > src > components > buttonsBox.js > ...

```
1  import React from 'react';
2  import { View, Text, TouchableOpacity } from 'react-native';
3  import styles from './styles';
4  import { useNavigation } from '@react-navigation/native';
5  export default function ButtonsBox() {
6      const navigation = useNavigation();
7      function navigationToComandas() { navigation.navigate('Comandas') }
8      function navigationToMesas() { navigation.navigate('Mesas') }
9      return (
10         <View style={styles.botoesBox}>
11             <TouchableOpacity
12                 style={styles.button}
13                 onPress={() => navigationToComandas()}>
14                 <Text style={styles.buttonText}>COMANDAS</Text>
15             </TouchableOpacity>
16             <TouchableOpacity
17                 style={styles.button}
18                 onPress={() => navigationToMesas()}>
19                 <Text style={styles.buttonText}>MESAS</Text>
20             </TouchableOpacity>
21         </View>
22     );
23 }
```

Fonte: elaborado pelo autor

Index.JS da tela de Login**:** arquivo responsável pela interface gráfica da tela de autenticação e pela função de login no aplicativo, conforme demonstrado na Listagem 9. Entre as linhas 1 a 10 são importados os módulos necessários para a construção da interface e comunicação com o *back-end (api)*. Das linhas 35 a 69 são adicionados componentes visuais da biblioteca React Native. A função *SignIn*, declarada entre as linhas 15 a 28, tem como finalidade requisitar à api informações de autenticação utilizando os dados fornecidas pelo usuário. Caso a consulta ao banco de dados seja positiva, as informações do usuário serão salvas no sistema de armazenamento do aplicativo, *AsyncStorage*, e em seguida navegará para a tela de

comandas. Caso seja negativa, a resposta da execução da função será uma mensagem ao usuário com uma descrição fornecida pela api.

Listagem 9 – Listagem de código JavaScript da tela login

index.js Login ×

mobile > src > pages > Login > index.js > ...

```
1  import React, { useState } from 'react';
2  import {
3      Image, View, TouchableOpacity, TextInput,
4      Text, Alert, AsyncStorage
5  } from 'react-native';
6  import { Feather } from "@expo/vector-icons";
7  import { useNavigation } from '@react-navigation/native';
8  import styles from './styles';
9  import logoImg from '../../assets/logo.png';
10 import api from '../../services/api';
11 export default function fLogin() {
12     const navigation = useNavigation();
13     const [login, setlogin] = useState();
14     const [senha, setSenha] = useState();
15     async function signIn() {
16         await api.get('sessions', { params: { login, senha } })
17             .then(function (res) {
18                 user = res.data;
19                 navigation.navigate('Comandas');
20             })
21             .catch((error) => {
22                 Alert.alert('Atenção', error.response.data.descricao);
23             })
```

```
24          await AsyncStorage.multiSet([
25              ['@katarinaMobile:user_id', JSON.stringify(user.id)],
26              ['@katarinaMobile:user_name', JSON.stringify(user.user_name)],
27          ]);
28      }
29      function navigationToConfig() {
30          navigation.navigate('Configuracao');
31      };
32      return (
33          <View style={styles.container}>
34              <View style={styles.imageLogo}>
35                  <Image source={logoImg} />
36              </View>
37              <View style={styles.inputs}>
38                  <TextInput
39                      style={styles.input}
40                      keyboardType='visible-password'
41                      placeholder='Usuário'
42                      value={login}
43                      onChangeText={setlogin}
44                  />
45                  <TextInput
46                      style={styles.input}
47                      name={'edtsenha'}
48                      placeholder='Senha'
49                      value={senha}
50                      onChangeText={setSenha}
51                      secureTextEntry={true}
52                  />
53                  <TouchableOpacity
54                      onPress={() => signIn()}
55                      style={styles.buttom}
56                  >
57                      <Text style={styles.buttomText}>ENTRAR</Text>
58                  </TouchableOpacity>
59                  <TouchableOpacity
60                      onPress={() => navigationToConfig()} style={styles.settings}
61                  >
62                      <Feather name="settings" size={35} color={"#FFA500"} />
63                  </TouchableOpacity>
64              </View>
```

```
65              <View style={styles.rodape}>
66                  <Text style={styles.tituloButtom}>v.1.0.0</Text>
67              </View>
68          </View>
69      )
70  }
71
```

Fonte: elaborado pelo autor

Index.JS da tela Comandas: arquivo responsável pela interface gráfica da tela de listagem de comandas, conforme demonstrado na Listagem 10. Entre as linhas 1 e 13 são importados os módulos necessários para a construção da interface e comunicação com o *api*. Das linhas 71 a 130 são declarados os componentes presentes nesta tela. Entre as linhas 37 e 55 está declarada a função *abrirComanda* que é acionada pelo botão “abrir novo pedido”. Esta função além de criar uma nova venda aciona a função *navigationToCategorias*, declarada na linha 49. Função essa, que tem o objetivo de navegar para a tela de categorias passando o parâmetro *id_venda* adiante. Nas linhas 19 a 27 é declarado a função de *loadComandas*, que é responsável por buscar todas as comandas abertas e válidas no banco de dados. Os dados obtidos por essa função são consumidos pelo componente *FlatList*, declarado entre as linhas 102 e 127, que é responsável por apresentar os dados das comandas em um *grid* de listagem na tela. Entre as linhas 28 e 36 é declarado a função *loadComanda*, função responsável por localizar apenas uma única comanda informada pelo usuário. Por fim entre as linhas 56 e 60 é declarado a função *navigationToResumo* que tem por objetivo navegar até a tela de resumo de venda.

Listagem 10 – Listagem de código JavaScript da tela comandas

index.js Comandas

mobile > src > pages > Comandas > index.js > ...

```
1  import React, { useState, useEffect } from 'react';
2  import {
3      View, FlatList, Text,
4      TouchableOpacity, Alert, AsyncStorage
5  } from 'react-native';
6  import { Input } from 'react-native-elements';
7  import { Feather } from "@expo/vector-icons";
8  import { useNavigation } from '@react-navigation/native';
9  import Icon from 'react-native-vector-icons/FontAwesome';
10 import Dialog from 'react-native-dialog';
11 import ButtonsBox from '../../components/buttonsBox'
12 import styles from './styles';
13 import api from '../../services/api';
14 export default function Comandas() {
15     const navigation = useNavigation();
16     const [comandas, setComandas] = useState();
17     const [num_comanda, setNumComanda] = useState(0);
18     const [dialogVisible, setDialogVisible] = useState(false)
19     async function loadComandas() {
20         await api.get('venda', {
21             params: { tipo: 'C', id_empresa: 1 }
22         }).then(function (res) {
23             setComandas(res.data);
24         }).catch(function (error) {
25             Alert.alert('Atenção', error.response.data.descricao)
26         })
27     };
28     async function loadComanda() {
29         api.get('venda',
30             { params: { tipo: 'C', nro_com_mesa: num_comanda, id_empresa: 1 } })
31             .then(function (res) {
32                 setComandas(res.data)
33             }).catch(function (error) {
34                 Alert.alert('Atenção', error.response.data.descricao)
35             })
36     }
```

```
37      async function abrirComanda() {
38          let id_usuario = await AsyncStorage.getItem('@katarinaMobile:user_id')
39          await api.post('venda',
40              {
41                  tipo: 'C',
42                  nro_comanda: num_comanda,
43                  nro_mesa: 0,
44                  id_empresa: 1,
45                  id_usuario
46              })
47              .then(function (res) {
48                  console.log(res.data)
49                  navigation.navigate('Categorias', { id_venda: res.data });
50              }).catch(function (error) {
51                  if (error) {
52                      Alert.alert('Atenção', error.response.data.descricao)
53                  }
54              })
55      }
56      function navigateToResumo(id_venda, valor, num_comanda) {
57          navigation.navigate('Resumo',
58              { id_venda, valor, titulo: 'Comanda ' + num_comanda }
59          );
60      }
61      function showDialog() {
62          setDialogVisible(true);
63      };
64      function handleConfirmar() {
65          setDialogVisible(false);
66          abrirComanda();
67      }
68      useEffect(() => {
69          loadComandas();
70      }, [])
71      return (
72          <View style={styles.container}>
73              <View style={styles.header}>
74                  <Text style={styles.headerText}>Comandas</Text>
75                  <TouchableOpacity onPress={() => showDialog()}>
76                      <Feather name="plus-circle" size={35} color={"#FFA500"} />
77                  </TouchableOpacity>
```

```
78                </View>
79                <Dialog.Container visible={dialogVisible}>
80                    <Dialog.Title>Qual comanda deseja abrir?</Dialog.Title>
81                    <Dialog.Input
82                        placeholder='Nº comanda'
83                        keyboardType='numeric'
84                        onChangeText={text => setNumComanda(text)}
85                    >
86                    </Dialog.Input>
87                    <Dialog.Button label="confirmar"
88                        onPress={() => handleConfirmar()}
89                    />
90                </Dialog.Container>
91                <View style={styles.search}>
92                    <Input
93                        style={styles.searchText}
94                        placeholder="Nº Comanda"
95                        returnKeyType='search'
96                        keyboardType='numeric'
97                        onChangeText={text => setNumComanda(text)}
98                        onSubmitEditing={() => loadComanda()}
99                        rightIcon={<Icon name='search' size={20} color='#dcdcdc' />}
100                   />
101               </View>
102               <FlatList
103                   style={styles.comandaList}
104                   data={comandas}
105                   keyExtractor={comanda => String(comanda.ven_001)}
106                   showsVerticalScrollIndicator={false}
107                   renderItem={({ item: comanda }) => (
108                       <TouchableOpacity
109                           onPress={() => navigateToResumo(
110                               comanda.ven_001, comanda.ven_009, comanda.ven_026
111                           )}
112                       >
113                           <View style={styles.comanda}>
114                               <Text
115                                   style={styles.comandaTitle}>
116                                   COMANDA {comanda.ven_026}
117                               </Text>
118                               <Text style={styles.comandaValor}>
119                                   {Intl.NumberFormat('pt-BR', {
```

```
115                                 style={styles.comandaTitle}>
116                                 COMANDA {comanda.ven_026}
117                             </Text>
118                             <Text style={styles.comandaValor}>
119                                 {Intl.NumberFormat('pt-BR', {
120                                     style: 'currency',
121                                     currency: 'BRL'
122                                 }).format(comanda.ven_009)}
123                             </Text>
124                         </View>
125                     </TouchableOpacity>
126                 )}
127             />
128             <ButtonsBox />
129         </View>
130     );
131 }
132
```

Fonte: elaborado pelo autor

Index.JS da tela Mesas: arquivo responsável pela interface gráfica da tela de listagem de mesas, conforme demonstrado na Listagem 11. Entre as linhas 1 e 13 são importados os módulos necessários para a construção da interface e comunicação com o *api*. Das linhas 71 a 130 são declarados os componentes presentes nesta tela. Entre as linhas 37 e 55 está declarada a função *abrirMesa* que é acionada pelo botão “abrir novo pedido”. Esta função além de criar uma nova venda aciona a função *navigationToCategorias*, declarada na linha 49. Função essa, que tem o objetivo de navegar para a tela de categorias passando o parâmetro *id_venda* adiante. Nas linhas 19 a 27 é declarado a função de *loadMesas*, que é responsável por buscar todas as comandas abertas e válidas no banco de dados. Os dados obtidos por essa função são consumidos pelo componente *FlatList*, declarado entre as linhas 102 e 127, que é responsável por apresentar os dados das comandas em um *grid* de listagem na tela. Entre as linhas 28 e 36 é declarado a função *loadMesa*, função responsável por localizar apenas uma única comanda informada pelo usuário. Por fim entre as linhas 56 e 60 é declarado a função *navigationToResumo* que tem por objetivo navegar até a tela de resumo de venda.

Listagem 11 – Listagem de código JavaScript da tela Mesas

index.js Mesas

mobile > src > pages > Mesas > index.js > ...

```
import React, { useState, useEffect } from 'react';
import {
    View, FlatList, Text,
    TouchableOpacity, Alert, AsyncStorage
} from 'react-native';
import { Input } from 'react-native-elements';
import { Feather } from "@expo/vector-icons";
import { useNavigation } from '@react-navigation/native';
import Icon from 'react-native-vector-icons/FontAwesome';
import Dialog from 'react-native-dialog';
import ButtonsBox from '../../components/buttonsBox'
import styles from './styles';
import api from '../../services/api';
export default function Comandas() {
    const navigation = useNavigation();
    const [mesas, setMesas] = useState();
    const [num_mesa, setNumMesa] = useState(0);
    const [dialogVisible, setDialogVisible] = useState(false)
    async function loadMesas() {
        await api.get('venda', {
            params: { tipo: 'M', id_empresa: 1 }
        }).then(function (res) {
            setMesas(res.data);
        }).catch(function (error) {
            Alert.alert('Atenção', error.response.data.descricao)
        })
    };
    async function loadMesa() {
        api.get('venda',
        api.get('venda',
            { params: { tipo: 'M', nro_com_mesa: num_mesa, id_empresa: 1 } })
            .then(function (res) {
                setMesas(res.data)
            }).catch(function (error) {
                Alert.alert('Atenção', error.response.data.descricao)
            })
    }
    async function abrirMesa() {
        let id_usuario = await AsyncStorage.getItem('@katarinaMobile:user_id')
        await api.post('venda',
```

```
40              {
41                  tipo: 'M',
42                  nro_comanda: 0,
43                  nro_mesa: num_mesa,
44                  id_empresa: 1,
45                  id_usuario
46              })
47              .then(function (res) {
48                  console.log(res.data)
49                  navigation.navigate('Categorias', { id_venda: res.data });
50              }).catch(function (error) {
51                  if (error) {
52                      Alert.alert('Atenção', error.response.data.descricao)
53                  }
54              })
55      }
56      function navigateToResumo(id_venda, valor, num_mesa) {
56      function navigateToResumo(id_venda, valor, num_mesa) {
57          navigation.navigate('Resumo',
58              { id_venda, valor, titulo: 'Mesa ' + num_mesa }
59          );
60      }
61      function showDialog() {
62          setDialogVisible(true);
63      };
64      function handleConfirmar() {
65          setDialogVisible(false);
66          abrirMesa();
67      }
68      useEffect(() => {
69          loadMesas();
70      }, [])
71      return (
72          <View style={styles.container}>
73              <View style={styles.header}>
74                  <Text style={styles.headerText}>Mesas</Text>
75                  <TouchableOpacity onPress={() => showDialog()}>
76                      <Feather name="plus-circle" size={35} color={"#FFA500"} />
77                  </TouchableOpacity>
78              </View>
79              <Dialog.Container visible={dialogVisible}>
80                  <Dialog.Title>Qual mesa deseja abrir?</Dialog.Title>
81                  <Dialog.Input
82                      placeholder='Nº mesa'
83                      keyboardType='numeric'
```

```
84                  onChangeText={text => setNumMesa(text)}
85                  >
86                  </Dialog.Input>
87                  <Dialog.Button label="confirmar"
88                      onPress={() => handleConfirmar()}
89                  />
90              </Dialog.Container>
91              <View style={styles.search}>
92                  <Input
93                      style={styles.searchText}
94                      placeholder="Nº Mesa"
95                      returnKeyType='search'
96                      keyboardType='numeric'
97                      onChangeText={text => setNumMesa(text)}
98                      onSubmitEditing={() => loadMesa()}
99                      rightIcon={<Icon name='search' size={20} color='#dcdcdc' />}
100                 />
101             </View>
102             <FlatList
103                 style={styles.List}
104                 data={mesas}
105                 keyExtractor={mesa => String(mesa.ven_001)}
106                 showsVerticalScrollIndicator={false}
107                 renderItem={({ item: mesa }) => (
108                     <TouchableOpacity
109                         onPress={() => navigateToResumo(
110                             mesa.ven_001, mesa.ven_009, mesa.ven_026
111                         )}
```

```
112                     >
113                         <View style={styles.mesa}>
114                             <Text
115                                 style={styles.mesaTitle}>
116                                 MESA {mesa.ven_026}
117                             </Text>
118                             <Text style={styles.mesaValor}>
119                                 {Intl.NumberFormat('pt-BR', {
120                                     style: 'currency',
121                                     currency: 'BRL'
122                                 }).format(mesa.ven_009)}
123                             </Text>
124                         </View>
125                     </TouchableOpacity>
126                 )}
127             />
128             <ButtonsBox />
129         </View>
130     );
131 }
132
```

Fonte: elaborado pelo autor

Index.JS da tela Categorias: arquivo responsável por criar a interface da tela de listagem Categorias e armazenar as principais funções da interface, demonstrado na Listagem 12. Entre as linhas 1 e 6 são importados os módulos necessários para a criação da tela. Da linha 29 a linha 56 são adicionados os componentes da biblioteca React Native. Entre as linhas 12 a 14 é declarado a função *navigationBack* e tem como objetivo redirecionar o usuário para tela anterior a qual chamou a tela Categorias. Entre as linhas 35 e 54 é adicionado um *FlatList* que tem por finalidade listar, em um grid, as categorias carregadas pela função *loadCategorias* declara entre as linhas 28 e 23. Cada linha do grid de listagem das categorias, ao ser clicada, aciona a função *navigationToProduto* declarada nas linhas 15 a 17 e tem como objetivo redirecionar o usuário para tela de Produtos, passando os parâmetros *categoria e id_venda* a ela.

Listagem 12 – Listagem de código JavaScript da tela categorias

index.js Categorias

mobile > src > pages > Categorias > index.js > ...

```
1  import React, { useState, useEffect } from 'react';
2  import { View, Text, FlatList, TouchableOpacity } from 'react-native';
3  import { Feather } from "@expo/vector-icons";
4  import { useNavigation, useRoute } from '@react-navigation/native';
5  import styles from './styles';
6  import api from '../../services/api';
7  export default function Categorias() {
8      const navigation = useNavigation();
9      const route = useRoute();
10     const [categorias, setCategorias] = useState([]);
11     const id_venda = route.params.id_venda;
12     function navigationBack() {
13         navigation.goBack()
14     }
15     function navigationToProduto(categoria) {
16         navigation.navigate('Produtos', { categoria, id_venda })
17     }
18     async function loadCategorias() {
19         await api.get('categorias', { params: { id_empresa: 1 } })
20             .then(function (res) {
21                 setCategorias(res.data)
22             })
23     }
24     useEffect(() => {
25         loadCategorias();
26     }, [])
27     return (
28         <View style={styles.container}>
29             <View style={styles.header}>
30                 <TouchableOpacity onPress={navigationBack}>
31                     <Feather name="arrow-left" size={35} color={"#FFA500"} />
32                 </TouchableOpacity>
33                 <Text style={styles.headerText}>Voltar</Text>
34             </View>
```

```
35          <FlatList
36              data={categorias}
37              style={styles.categoriaList}
38              keyExtractor={categoria => String(categoria.cat_001)}
39              showsVerticalScrollIndicator={false}
40              renderItem={({ item: categoria }) => (
41                  <TouchableOpacity
42                      onPress={() => navigationToProduto(categoria)}
43                  >
44                      <View style={styles.categoria}>
45                          <Text style={styles.categoriaTitle}>
46                              {categoria.cat_002}
47                          </Text>
48                          <Feather
49                              name="arrow-right" size={25} color={"#FFA500"}
50                          />
51                      </View>
52                  </TouchableOpacity>
53              )}
54          />
55      </View>
56  );
```

Fonte: elaborado pelo autor

Index.JS da tela Produtos: arquivo responsável por criar a interface gráfica da tela de listagem de Produtos e armazenar as principais funções, demonstrado na Listagem 13. Entre as linhas 1 e 6 são importados os módulos necessários para a criação da tela. Das linhas 28 a 56 são adicionados os componentes da biblioteca React Native. O primeiro componente, linhas 31 a 33, representa o botão voltar da tela e é responsável por acionar a função *navigationBack* declarada entre as linhas 13 e 15, que tem por objetivo voltar à tela anterior, categorias. Entre as linhas 36 e 60 é adicionado o componente *FlatList*. Este componente cria um grid de listagem dos produtos localizados através da função *loadProdutos*. Esta função, por sua vez, tem a finalidade consultar no banco de dados todos os produtos associados a uma determinada categoria passada como parâmetro através da variável *cat_id*.

Listagem 13 – Listagem de código JavaScript da tela Produtos

index.js Produtos

mobile > src > pages > Produtos > index.js > ...

```
1  import React, { useState, useEffect } from 'react';
2  import { View, Text, FlatList, TouchableOpacity } from 'react-native';
3  import { Feather } from "@expo/vector-icons";
4  import { useNavigation, useRoute } from '@react-navigation/native';
5  import styles from './styles';
6  import api from '../../services/api';
7  export default function Produtos() {
8      const navigation = useNavigation();
9      const route = useRoute();
10     const [produtos, setProdutos] = useState();
11     const id_categoria = route.params.categoria.cat_001;
12     const id_venda = route.params.id_venda;
13     function navigationBack() {
14         navigation.goBack()
15     }
16     function navigationToVenda(produto) {
17         navigation.navigate('Venda', { produto, id_venda })
18     }
19     async function loadProdutos() {
20         await api.get('produtos', { params: { id_categoria, id_empresa: 1 } })
21             .then(function (res) {
22                 setProdutos(res.data)
23             });
24     }
25     useEffect(() => {
26         loadProdutos();
27     }, [])
28     return (
29         <View style={styles.container}>
30             <View style={styles.header}>
31                 <TouchableOpacity onPress={navigationBack}>
32                     <Feather name="arrow-left" size={35} color={"#FFA500"} />
33                 </TouchableOpacity>
34                 <Text style={styles.headerText}>Voltar</Text>
35             </View>
```

```
36          <FlatList
37              data={produtos}
38              style={styles.produtoList}
39              keyExtractor={produto => String(produto.mat_001)}
40              showsVerticalScrollIndicator={false}
41              renderItem={({ item: produto }) => (
42                  <TouchableOpacity
43                      onPress={() => navigationToVenda(produto)}>
44                      <View style={styles.produto}>
45                          <Text style={styles.produtoTitle}>
46                              {produto.mat_003}
47                          </Text>
48                          <Text style={styles.produtoValor}>
49                              {Intl.NumberFormat('pt-BR', {
50                                  style: 'currency',
51                                  currency: 'BRL'
52                              }).format(produto.mat_008)
53                              }</Text>
54                          <Feather
55                              name="arrow-right" size={25} color={"#FFA500"}
56                          />
57                      </View>
58                  </TouchableOpacity>
45              <TextInput
46                  style={styles.input}
47                  name={'edtsenha'}
48                  placeholder='Senha'
49                  value={senha}
50                  onChangeText={setSenha}
51                  secureTextEntry={true}
52              />
53              <TouchableOpacity
54                  onPress={() => signIn()}
55                  style={styles.buttom}
56              >
57                  <Text style={styles.buttomText}>ENTRAR</Text>
58              </TouchableOpacity>
59              <TouchableOpacity
60                  onPress={() => navigationToConfig()} style={styles.settings}
61              >
62                  <Feather name="settings" size={35} color={"#FFA500"} />
63              </TouchableOpacity>
64          </View>
```

```
59                )}
60              />
61            </View>
62        );
63    }
64
```

Fonte: elaborado pelo autor

Index.JS da tela de Venda: arquivo responsável por criar a interface gráfica da tela de venda, demonstrado na Listagem 14. Entre as linhas 1 e 8 são importados os módulos necessários para a criação da tela. Das linhas 38 a 140 são adicionados os componentes da biblioteca React Native. Entre as linhas 17 e 26 é declarado a função *loadItens* que é responsável por buscar as informações dos itens da venda, caso aja algum, bem como informações do resumo da venda. Este arquivo possui a função *inserirItem*, declarada entre as linhas 31 e 49, e é responsável por inserir um novo item na venda em questão. Entre as linhas 30 e 33 é declarado a função que calcula o total da venda e é acionado sempre que um novo item é adicionado a venda, garantindo assim que as informações de total sejam atualizadas em tela para o usuário visualizar.

Listagem 14 – Listagem de código JavaScript da tela venda

index.js Venda

mobile > src > pages > Venda > index.js > ...

```
1  import React, { useState, useEffect } from 'react';
2  import { View, Text, FlatList, TouchableOpacity, Alert } from 'react-native';
3  import { useRoute, useNavigation } from '@react-navigation/native';
4  import { Feather } from "@expo/vector-icons";
5  import { Input } from 'react-native-elements';
6  import UIStepper from 'react-native-ui-stepper';
7  import styles from './styles';
8  import api from '../../services/api';
9  import { AsyncStorage } from 'react-native';
10 export default function Produtos() {
11     const route = useRoute();
12     const navigation = useNavigation();
13     const id_venda = route.params.id_venda;
14     const produto = route.params.produto;
15     const [produtos, setProdutos] = useState();
16     const [obs, setObs] = useState('');
17     const [quantidade, setQuantidade] = useState(1);
18     const [val_totalitem, setValTotalItem] = useState(0.00);
19     const [total_venda, setTotal] = useState(0.00)
20     async function loadItens() {
21         await api.get('venda/resumo', { params: { id_venda, id_empresa: 1 } })
22             .then(function (res) {
23                 setTotal(res.data[0].ven_009);
24                 setProdutos(res.data[1].vendaitens);
25             })
26     }
27     function calculaTotalItem(quantidade) {
28         setQuantidade(quantidade)
29         setValTotalItem(produto.mat_008 * quantidade);
30     }
```

```
31    async function inserirItem() {
32        let id_usuario = await AsyncStorage.getItem('@katarinaMobile:user_id')
33        await api.post('venda/item', {
34            id_empresa: 1,
35            id_venda,
36            id_produto: produto.mat_001,
37            quantidade,
38            valor_unit: produto.mat_008,
39            valor_total: val_totalitem,
40            observacao: obs,
41            id_impressora: produto.mat_021,
42            id_usuario
43        }).then(function (res) {
44            loadItens();
45            Alert.alert('Atenção', 'Produto inserido com sucesso!');
46        }).catch(function (error) {
47            Alert.alert('Atenção', error.response.data.descricao)
48        })
49    }
50    useEffect(() => {
51        calculaTotalItem(1);
52        loadItens();
53    }, [])
54    return (
55        <View style={styles.container}>
56            <View style={styles.header}>
57                <Text style={styles.headerText}>Pedido Nº {id_venda}</Text>
58            </View>
59            <View style={styles.itemHeaderTitle}>
60                <Text style={styles.itemHeaderText}>
```

```
61                  {produto.mat_001} - {produto.mat_003}
62              </Text>
63          </View>
64          <View style={styles.containerItem}>
65              <View style={styles.infoItemHeader}>
66                  <Text style={styles.infoItemText}>Val. unit</Text>
67                  <Text style={styles.infoItemText}>Qtd.</Text>
68                  <Text style={styles.infoItemText}>Total Item</Text>
69              </View>
70              <View style={styles.infoItemValor}>
71                  <Text style={styles.infoItemText}>{
72                      Intl.NumberFormat('pt-BR', {
73                          style: 'currency',
74                          currency: 'BRL',
75                      }).format(produto.mat_008)}
76                  </Text>
77                  <Text style={styles.infoItemText}>{quantidade}</Text>
78                  <Text style={styles.infoItemText}>{
79                      Intl.NumberFormat('pt-BR', {
80                          style: 'currency',
81                          currency: 'BRL',
82                      }).format(val_totalitem)}</Text>
83              </View>
84          </View>
85          <View style={styles.containerAdd}>
86              <View style={styles.containerQtd}>
87                  <UIStepper
88                      onValueChange={(value) => { calculaTotalItem(value) }}
89                  />
90                  <TouchableOpacity style={styles.buttonAdd}
```

```
91                    onPress={() => inserirItem()}
92                >
93                    <Text style={styles.buttonText}>Adiconar</Text>
94                </TouchableOpacity>
95            </View>
96            <Input
97                style={styles.obsText}
98                placeholder="Observações"
99                onChangeText={text => setObs(text)}
100           />
101       </View>
102       <View style={styles.bannerLista}>
103           <View style={styles.headerPedList}>
104               <Text style={styles.listaHeader}>Itens do Pedido</Text>
105           </View>
106       </View>
107       <FlatList
108           data={produtos}
109           style={styles.lista}
110           keyExtractor={produtos => String(produtos.mat_001)}
111           showsVerticalScrollIndicator={false}
112           renderItem={({ item: produto }) => (
113               <View style={styles.itens}>
114                   <View style={styles.boxTitulo}>
115                       <Text style={styles.itemTitle}>
116                           {produto.ite_001} - {produto.mat_003}
117                       </Text>
118                   </View>
119                   <View style={styles.boxValor}>
120                       <Text style={styles.itemValor}>
```

```
121                                 {Intl.NumberFormat().format(produto.ite_002)}
122                             </Text>
123                             <Text style={styles.itemValor}>
124                                 {Intl.NumberFormat('pt-BR', {
125                                     style: 'currency',
126                                     currency: 'BRL',
127                                 }).format(produto.ite_003)}
128                             </Text>
129                             <Text style={styles.itemValor}>
130                                 {Intl.NumberFormat('pt-BR', {
131                                     style: 'currency',
132                                     currency: 'BRL'
133                                 }).format(produto.ite_005)}
134                             </Text>
135                         </View>
136                     </View>
137                 )}
138             />
139             < View style={styles.bannerTotal}>
140                 <Text style={styles.totalHeader}>Total do Pedido</Text>
141                 <Text style={styles.totalHeader}>
142                     {Intl.NumberFormat('pt-BR', {
143                         style: 'currency',
144                         currency: 'BRL'
145                     }).format(total_venda)}
146                 </Text>
147             </View>
148             <View style={styles.botoesBox}>
149                 <TouchableOpacity
```

```
150                     style={styles.button}
151                     onPress={() => {
152                         navigation.navigate('Categorias', { id_venda })
153                     }}>
154                     <Text style={styles.buttonText}>Novo item</Text>
155                     <Feather name="plus" size={25} color={"#FFF"} />
156                 </TouchableOpacity>
157                 <TouchableOpacity
158                     style={styles.button}
159                     onPress={() => { navigation.navigate('Comandas') }}>
160                     <Text style={styles.buttonText}>Cancelar</Text>
161                     <Feather name="x" size={25} color={"#FFF"} />
162                 </TouchableOpacity>
163                 <TouchableOpacity
164                     style={styles.button}
165                     onPress={() => { navigation.navigate('Comandas') }}>
166                     <Text style={styles.buttonText}>Confirmar</Text>
167                     <Feather name="check" size={25} color={"#FFF"} />
168                 </TouchableOpacity>
169             </View>
170         </View>
171     );
172 }
173
```

Fonte: elaborado pelo autor

Index.JS da tela de Resumo de Venda: arquivo responsável por criar a interface gráfica da tela de Resumo de Venda, demonstrado na Listagem 15. Entre as linhas 1 e 7 são importados os módulos necessários para a criação da tela. Das linhas 29 a 86 são adicionados os componentes da biblioteca React Native. Esta tela conta com um botão para voltar, declarado entre as linhas 32 a 34, que tem por finalidade acionar a função *navigation.goback*, função essa que retorna a navegação para tela anterior. Ao carregar essa tela, a função *loadVenda* é acionada e tem por finalidade buscar as informações da venda e dos itens do pedido. Entre as linhas 44 e 76 é declarado o componente *FlatList* responsável pela listagem dos itens. Na parte inferior da tela é apresentado o total do pedido, conforme declarado entre as linhas 77 e 84.

Listagem 15 – Listagem de código JavaScript da tela ResumoVenda

index.js ResumoVenda

mobile > src > pages > ResumoVenda > index.js > ...

```
1  import React, { useState, useEffect } from 'react';
2  import { useNavigation, useRoute } from '@react-navigation/native';
3  import { View, FlatList, Text, TouchableOpacity } from 'react-native';
4  import { Feather } from "@expo/vector-icons";
5  import styles from './styles';
6  import api from '../../services/api';
7  import { Alert } from 'react-native';
8  export default function resumo() {
9    const route = useRoute();
10   const navigation = useNavigation();
11   const id_venda = route.params.id_venda;
12   const titulo = route.params.titulo;
13   const valor_total = route.params.valor;
14   const [venda, setVenda] = useState();
15   const [vendaitens, setVendaitens] = useState();
16   async function loadVenda() {
17     await api.get('venda/resumo', { params: { id_venda, id_empresa: 1 } })
18       .then((res) => {
19         setVenda(res.data[0]);
20         setVendaitens(res.data[1].vendaitens);
21       })
22       .catch((error) => {
23         Alert.alert('Atenção', error.response.data.descricao)
24       });
25   };
26   useEffect(() => {
27     loadVenda();
28   }, [])
```

```
29    return (
30      <View style={styles.container}>
31        <View style={styles.header}>
32          <TouchableOpacity onPress={() => navigation.goBack()}>
33            <Feather name="arrow-left" size={35} color={"#FFA500"} />
34          </TouchableOpacity>
35          <Text style={styles.headerText}>{titulo}</Text>
36          <Text style={styles.headerTextPed}>Nº Ped.{id_venda}</Text>
37        </View>
38        <View style={styles.subHeader}>
39          <Text style={styles.subHeaderText}>Item</Text>
40          <Text style={styles.subHeaderText}>Qtd.</Text>
41          <Text style={styles.subHeaderText}>Val. Unit.</Text>
42          <Text style={styles.subHeaderText}>Total</Text>
43        </View>
44        <FlatList
45          style={styles.listItens}
46          data={vendaitens}
47          keyExtractor={item => String(item.ite_001)}
48          showsVerticalScrollIndicator={false}
49          renderItem={({ item: vitem }) => (
49          renderItem={({ item: vitem }) => (
50            <View style={styles.itens}>
51              <View style={styles.boxTitulo}>
52                <Text style={styles.itemTitle}>
53                  {vitem.ite_001} - {vitem.mat_003}
54                </Text>
55              </View>
56              <View style={styles.boxValor}>
57                <Text style={styles.itemValor}>
58                  {Intl.NumberFormat().format(vitem.ite_002)}
59                </Text>
60                <Text style={styles.itemValor}>
61                  {Intl.NumberFormat('pt-BR', {
62                    style: 'currency',
63                    currency: 'BRL',
64                  }).format(vitem.ite_003)}
65                </Text>
```

```
66                  <Text style={styles.itemValor}>
67                    {Intl.NumberFormat('pt-BR', {
68                      style: 'currency',
69                      currency: 'BRL'
70                    }).format(vitem.ite_005)}
71                  </Text>
72                </View>
73              </View>
74            )}
75          >
76          </FlatList>
77          <View style={styles.rodape}>
78            <Text style={styles.rodapeTex}>Total do Pedido</Text>
79            <Text style={styles.rodapeTex}>
80              {Intl.NumberFormat('pt-BR', {
81                style: 'currency',
82                currency: 'BRL'
83              }).format(valor_total)}</Text>
84          </View>
85        </View>
86      )
87    }
88
```

Fonte: elaborado pelo autor