UNIVERSIDADE FEDERAL DO CEARÁ

INSTITUTO UNIVERSIDADE VIRTUAL

CURSO DE GRADUAÇÃO EM SISTEMAS E MÍDIAS DIGITAIS

RAFAEL MAIA PINHEIRO

DESENVOLVIMENTO DO BACK-END DE COMPRA E VENDA DE ALIMENTOS EM CANTINAS UNIVERSITÁRIAS

FORTALEZA

2021

RAFAEL MAIA PINHEIRO

DESENVOLVIMENTO DO BACK-END DE COMPRA E VENDA DE ALIMENTOS EM CANTINAS UNIVERSITÁRIAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto Universidade Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Orientador: Prof. Me. Wellington Wagner Ferreira Sarmento

FORTALEZA

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P722d Pinheiro, Rafael Maia.
Desenvolvimento do Back-End de Compra e Venda de alimentos em cantinas universitárias / Rafael Maia Pinheiro. – 2021.
114 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Instituto UFC Virtual, Curso de Sistemas e Mídias Digitais, Fortaleza, 2021.
Orientação: Prof. Me. Wellington Wagner Ferreira Sarmento.

1. Desenvolvimento de software. 2. Compra e venda. 3. Lanchonetes. 4. Segurança de redes. 5. Java. I. Título.

CDD 302.23

RAFAEL MAIA PINHEIRO

DESENVOLVIMENTO DO BACK-END DE COMPRA E VENDA DE ALIMENTOS EM CANTINAS UNIVERSITÁRIAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto Universidade Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Aprovada em:

BANCA EXAMINADORA

Prof. Me. Wellington Wagner Ferreira Sarmento (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Leonardo Oliveira Moreira
Universidade Federal do Ceará (UFC)

Prof. Dr. Gabriel Antoine Louis Paillard
Universidade Federal do Ceará (UFC)

À minha mãe, fonte de força, amor e inspiração que não conhecem limites. Ao meu pai, por me acompanhar nas minhas jornadas. À minha avó Lenir, minha segunda mãe.

Aos meus finados avós Rita, Chico e Osvaldo. Aos parentes que sempre me apoiaram. Às tias Ritinha, Vilma e à dona Edite, falecidas pouco após eu iniciar o curso.

AGRADECIMENTOS

Ao Prof. Me. Wellington Ferreira Sarmento, pela valiosa orientação no TCC e o auxílio durante o curso.

Ao Prof. Dr. Leonardo Oliveira Moreira, por ter sido mais que professor, um companheiro no desenvolvimento deste trabalho desde a disciplina de Projeto Integrado II.

Ao Prof. Dr. Gabriel Antoine Louis Paillard, pela paciência e orientação na disciplina de Projeto de Trabalho Final.

Aos professores de Projeto Integrado II, por sugerirem tornar o projeto desenvolvido na disciplina algo maior, em especial ao Prof. Dr. Windson Viana de Carvalho, que sugeriu o tema da aplicação desenvolvida.

Ao Prof. Dr. Eduardo Santos Junqueira Rodrigues, pelo auxílio que me deu ao longo do curso.

A todos os meus professores do curso de Sistemas e Mídias Digitais, pelo conhecimento adquirido.

Ao meu colega e professor William Gois, que me ajudou nas vezes em que fiquei sem computador para realizar as atividades da universidade.

Aos membros da minha equipe da disciplina de Projeto Integrado II, pela cooperação no projeto que serviria de base para este trabalho.

RESUMO

Soluções *online* para a redução de filas em comércios de alimentos geram aumento de produtividade para vendedores e comodidade para consumidores. Microempresários desse ramo, como donos de lanchonetes em universidades, dispõem de poucos recursos para utilizar aplicativos populares no mercado. Nessa condição, este trabalho relata o desenvolvimento do *back-end* de um *software* de compra e venda de alimentos em cantinas universitárias, com o objetivo de suprir essa lacuna de mercado, prover aos donos de pequenos estabelecimentos uma solução viável para acelerar o processo de vendas e fornecer a seus clientes uma alternativa para evitar filas. A aplicação *web* denominada OnTime utiliza tecnologias *open source* para fornecer uma solução confiável para *smartphones*, integrada com um sistema de pagamento seguro por meio de API e código modular para melhor manutenção e escalabilidade. Este relatório apresenta o padrão de projeto *Model-View-Controller*, tecnologias Java para *web*, conceitos de comunicação segura e de bancos de dados, os procedimentos metodológicos utilizados para o desenvolvimento da solução proposta, sua descrição detalhada, os testes de *software* realizados, bem como os resultados e sugestões para futuros trabalhos.

Palavras-chave: Aplicação Web. Desenvolvimento. Compra e Venda. Alimentos. Pagamento Seguro. Java.

ABSTRACT

Online solutions to reduce queues in food businesses cause increased productivity for sellers and convenience for consumers. Microentrepreneurs in this area, such as university cafeteria owners, have few resources to use popular applications in the market. In this context, this paper reports the development of a software back-end for buying and selling food in university cafeterias, with the objective of filling this gap in the market, providing the owners of small establishments with a viable solution to accelerate the sales process and provide their customers with an alternative to avoid queues. The web application called OnTime uses open source technology to provide a reliable solution for smartphones, integrated with a secure payment system through API and modular code for better maintenance and scalability. This report presents the *Model-View-Controller* design pattern, Java technologies for web, concepts of secure communication and databases, the methodological procedures used to develop the proposed solution, its detailed description, the tests performed, as well as the results and suggestions for further work.

Keywords: Web Application. Development. Buying and Selling. Food. Secure Payment. Java.

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

AC	Autoridade Certificadora
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
AES	*Advanced Encryption Standard*
AJAX	*Asynchronous JavaScript and XML*
API	*Application Programming Interface*
BD	banco de dados
CNPJ	Cadastro Nacional de Pessoas Jurídicas
CRUD	*create, read, update, and delete*
CSS	*Cascading Style Sheets*
DAO	*Data Access Object*
DBA	*Database Administrator*
FK	*Foreign Key*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
JAR	*Java ARchive*
JCA	*Java Cryptography Architecture*
JDBC	*Java Database Connectivity*
JDK	*Java Development Kit*
JKS	*Java KeyStore*
JSP	*JavaServer Page*
JSSE	*Java Secure Socket Extension*
MER	Modelo Entidade-Relacionamento
MVC	*Model-View-Controller*
MVP	Mínimo Produto Viável
PK	*Primary Key*
PKCS	*Public Key Cryptography Standards*
REST	*REpresentational State Transfer*
RSA	Rivest–Shamir–Adleman
SBD	Sistema de Banco de Dados
SDK	*Software Development Kit*

SGBD	Sistema de Gerenciamento de Banco de Dados
SMD	Sistemas e Mídias Digitais
SMS	*short message service*
SQL	*Structured Query Language*
SSL	*Secure Sockets Layer*
TLS	*Transport Layer Security*
UFC	Universidade Federal do Ceará
XML	*eXtensive Markup Language*
XSS	*Cross-Site Scripting*

SUMÁRIO

1 INTRODUÇÃO

A formação de filas é uma ocorrência frequente no cotidiano de diversos estabelecimentos. Esse problema pode ser visto no Campus do PICI da Universidade Federal do Ceará (UFC). Como consequência da aglomeração de pessoas, seu tempo de espera para atendimento aumenta, o que gera uma situação desconfortável não só para os integrantes da fila como para os que trabalham nos estabelecimentos em questão. No caso de estabelecimentos comerciais que atuam no ramo alimentício, o problema mostra-se ainda mais crítico, pois demanda certa urgência, visto que se trata de uma necessidade básica do dia a dia pela qual muitas pessoas não têm disponibilidade de esperar por muito tempo.

Com o objetivo de mitigar o desperdício de tempo e produtividade desses estabelecimentos, surgiram diversos sistemas, tais como iFood e Uber Eats, que procuram fornecer-lhes plataformas *online* para organizar pedidos, atender seus consumidores à distância e fazer a entrega dos seus produtos, tornando mais cômodo para seus clientes a ação de realizar uma compra. Porém, para utilizar-se dos serviços dessas aplicações, é necessário não só que o estabelecimento em questão disponha de recursos, mas também que atue numa área maior, o que não é o caso de lanchonetes de pequeno porte locais.

No caso de cantinas localizadas em instituições de ensino superior, com base em pesquisas feitas pela equipe "Time5"[1] (Anexos A, B e C)– realizadas no segundo semestre de 2019 para a disciplina de Projeto Integrado II, do curso de bacharelado em Sistemas e Mídias Digitais (SMD) da UFC –, o capital nesse tipo de estabelecimento é pequeno e a produção, bastante controlada, com planejamento do estoque diário ou semanal. Nesse cenário, não há interesse no uso de alguma das aplicações existentes no mercado, devido ao custo-benefício. A possibilidade de entrega, também, é um fator limitante, visto a abrangência de sua atuação. A pesquisa também observou que já existem aplicações que fornecem uma boa solução no que concerne a filas, mas que não estão disponíveis no mercado brasileiro para o público em geral.

O atendimento presencial, portanto, continua sendo a única forma de esses estabelecimentos lidarem com as filas, cujos problemas são agravados pelo baixo número de funcionários, o que resulta na demora no atendimento e na perda de tempo dos seus clientes, em sua maioria estudantes e professores, que não raramente desistem ou mudam de estabelecimento, visto que muitos não dispõem de tempo para esperar em filas, gerando perda de lucratividade e, também,

[1] Integrantes da equipe Time5: Diêgo de Lima Barros, Erick de Oliveira Bessa Xavier, João Bosco Viana Ribeiro, João Paulo Bezerra Ribeiro e Rafael Maia Pinheiro.

má reputação, o que afasta mais clientes.

1.1 Pergunta Norteadora

Levando em consideração o contexto apresentado, como acelerar o processo de compra e venda nas cantinas universitárias com o uso de tecnologias móveis a um baixo custo?

1.2 Objetivos

Aqui são tratados o objetivo geral e os objetivos específicos deste trabalho de pesquisa.

1.2.1 Objetivo Geral

Tendo em vista o problema exposto, o principal objetivo do presente trabalho é implementar um sistema *open source* de compra e venda de produtos alimentícios baseado em tecnologia *web* e acessível por dispositivos móveis.

1.2.2 Objetivos Específicos

Seguem os objetivos específicos deste projeto de pesquisa:

1. Criar um sistema de compra e venda de produtos alimentícios baseado em tecnologias *open-source*;
2. Implementar a integração do sistema proposto com um serviço de pagamento seguro;
3. Criar uma estrutura de comunicação segura usando *Application Programming Interface* (API).

1.3 Visão Geral do Sistema

Em 2019 a equipe Time5 desenvolveu como solução para o problema das filas uma aplicação *web* responsiva denominada “OnTime”, focada no uso em *smartphones*, que permitia ao usuário dono do estabelecimento adicionar, alterar e excluir produtos disponíveis para venda, além de observar e gerenciar os pedidos dos seus clientes. Ao mesmo tempo, a aplicação permitia que clientes do estabelecimento se cadastrassem, alterassem suas informações,

consultassem os produtos disponíveis para venda e realizassem pedidos. O pagamento, porém, não foi implementado devido à indisponibilidade da API escolhida e à falta de tempo para sua apresentação em um Mínimo Produto Viável (MVP) na disciplina de Projeto Integrado II do curso de SMD.

O *back-end* foi desenvolvido seguindo o padrão *Model-View-Controller* com o uso da linguagem Java e um banco de dados feito em *Structured Query Language* (SQL). A comunicação entre as camadas de modelo e visão foi implementada através do uso de "Java Servlets" na camada de controle. Neste trabalho, o *back-end* foi expandido para permitir a realização de pagamentos com segurança que possa atender às necessidades das cantinas localizadas na universidade, bem como às de seus clientes, de forma que a utilização desse sistema seja livre, visto que será de código aberto.

1.4 Organização do Documento

O presente trabalho está organizado em seis capítulos, o que inclui este capítulo de introdução. No segundo capítulo, apresenta-se o referencial teórico utilizado para descrever padrões de projeto e tecnologias usados no desenvolvimento do produto, servindo como base para a tomada de decisões referentes ao projeto. O terceiro capítulo expõe os procedimentos metodológicos usados no desenvolvimento do trabalho. O quarto capítulo descreve como ocorreu o desenvolvimento do produto. O quinto capítulo relata os testes realizados durante o desenvolvimento e os resultados obtidos. Por fim, o sexto e último capítulo discute o cumprimento dos objetivos do trabalho, bem como sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo busca apresentar e explicar os principais conceitos e tecnologias utilizados ao longo do desenvolvimento deste trabalho, bem como embasar sua utilização e indicar as produções científicas que deram sustentação teórica ao projeto. Num primeiro momento, será definido o padrão MVC, suas vantagens e desvantagens no desenvolvimento do *back-end* de uma aplicação. Depois serão descritos os conceitos de Sistema de Gerenciamento de Banco de Dados (SGBD) e transações no contexto de banco de dados. Em seguida será detalhada a arquitetura cliente-servidor. Serão também explicadas algumas tecnologias Java comumente usadas no desenvolvimento de aplicações em rede. Além disso, serão apresentados alguns desafios relacionados à implementação de transações seguras, bem como tecnologias que auxiliam a aprimorar a segurança em aplicações e transações comerciais *online*. Após a definição desses conceitos, serão apresentadas as justificativas para o uso de algumas das tecnologias descritas. Por fim, serão mencionados alguns trabalhos relacionados.

2.1 Padrão Modelo-Visão-Controle

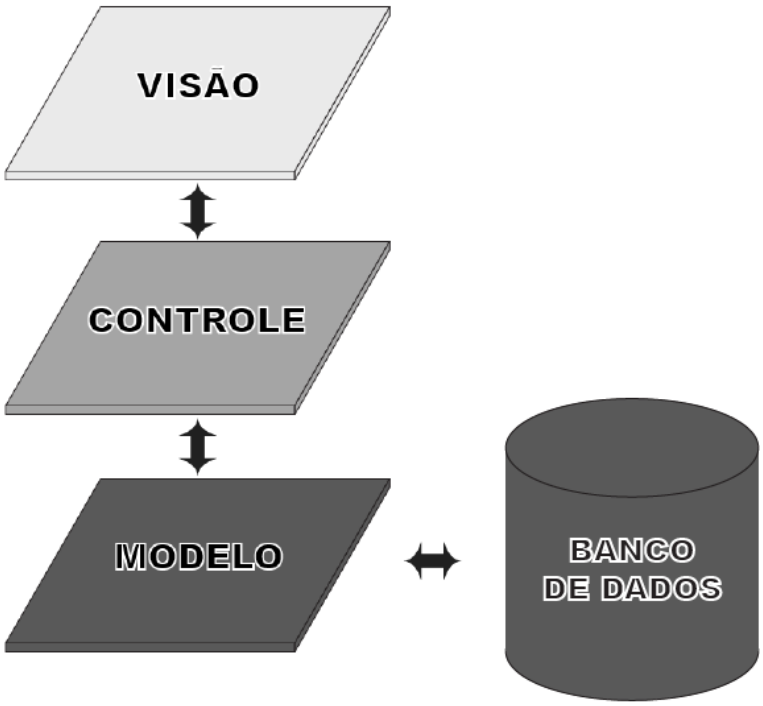
O padrão *Model-View-Controller* ou "Modelo-Visão-Controle" foi desenvolvido no final dos anos 1970 – mais especificamente em 1979 – por Trygve M. H. Reenskaug para uma linguagem de programação orientada a objetos chamada Smalltalk, desenvolvida no *Xerox Palo Alto Research Center* (LÓPEZ, 2009). Trata-se de um padrão de projeto que organiza o código de uma aplicação em três camadas bem definidas: modelo, visão e controle (PANTOJA, 2004). Seu objetivo é, através dessa organização, padronizar o projeto de modo a reduzir o esforço empreendido no desenvolvimento e implementação de vários sistemas que utilizam os mesmos dados (ROMERO; GONZÁLEZ, 2012).

A camada de modelo representa as informações com as quais se está trabalhando no momento de execução da aplicação. Nela se representam as variáveis, os objetos e os dados em geral que podem se modificar conforme o usuário interage, além de se conectar diretamente ao banco de dados (BD), se houver um, que armazena informações de forma persistente de acordo com os métodos contidos na camada de modelo (LÓPEZ, 2009).

A camada de visão é a camada encarregada do que o usuário final da aplicação vê, da interface gráfica com a qual ele interage. Nessa interface é possível para o usuário realizar requisições e observar os resultados de suas interações (LÓPEZ, 2009).

A camada de controle é responsável por modificar o modelo, bem como receber as requisições de modificação da interface, ou seja, ela trata de alterar variáveis, objetos e dados em geral de acordo com as requisições feitas pelo usuário na camada de visão (LÓPEZ, 2009). A camada de controle, portanto, é a única que tem acesso à camada de modelo e à camada de visão e tem o papel de intermediar os processos entre ambas (PANTOJA, 2004).

Figura 1 – Esquema ilustrativo das camadas do MVC e sua conexão com o BD



Fonte: Elaborado pelo autor (2020).

Como ilustra a Figura 1, o padrão MVC divide a aplicação numa estrutura de módulos com funções distintas.

2.1.1 Vantagens

O padrão MVC apresenta as seguintes vantagens:

- Uma aplicação desenvolvida em MVC apresenta uma separação clara entre os componentes do programa, sendo possível implementar cada um de forma separada (ROMERO; GONZÁLEZ, 2012);
- Um programador com acesso ao código pode modificar o modelo, a visão ou o controle sem grandes dificuldades (ROMERO; GONZÁLEZ, 2012);
- A conexão entre as camadas de modelo e visão é rápida, uma vez que ocorre em

tempo de execução, não de compilação (ROMERO; GONZÁLEZ, 2012);

- Mudanças feitas na camada de visão não interferem nas outras camadas (PANTOJA, 2004).

Por conta da organização do código no padrão MVC, ele comumente apresenta mais extensibilidade e manutenibilidade que códigos que não seguem esse padrão. O MVC é muito utilizado em *frameworks* orientados a objeto desenvolvidos para produzir aplicações grandes (PANTOJA, 2004). Alguns exemplos de *frameworks* que utilizam esse padrão são o Java Swing, o Apache Struts, o Microsoft ASP.NET MVC e o Ruby on Rails. Até mesmo documentos LaTeX seguem o padrão MVC (PANTOJA, 2004).

2.1.2 Desvantagens

Assim como apresenta vantagens, o MVC possui desvantagens que devem ser levadas em consideração na decisão quanto ao seu uso. São elas:

- O MVC é essencialmente um padrão de projeto voltado para o paradigma de orientação a objetos, por isso sua implementação em linguagens não orientadas a objetos é difícil e muitas vezes inviável (PANTOJA, 2004);
- Requer uma estrutura inicial, na qual as classes e interfaces que alteram e conectam as camadas da aplicação devem ser criadas (PANTOJA, 2004);
- O desenvolvimento de uma aplicação nesse padrão é mais demorado, pois ele exige que sejam implementadas classes a mais que seriam desnecessárias caso ele não fosse utilizado (PANTOJA, 2004).

É importante ressaltar que a última desvantagem é relativa, pois o esforço em organizar um código produzido no padrão MVC pode poupar tempo posteriormente, visto que é de fácil manutenção, o que o torna a longo prazo mais sustentável que um código não desenvolvido em MVC, podendo ser compreendido, estendido e modificado mais rapidamente (PANTOJA, 2004).

2.2 SGBD e Transações

A seguir são definidos os conceitos de SGBD e de transações no âmbito dos bancos de dados.

2.2.1 SGBD

Um SGBD é um conjunto de *softwares* de uso geral que permite criar e manter bancos de dados (ou bases de dados), que são conjuntos de dados persistentes e organizados que registram informações utilizadas numa aplicação. Um SGBD também facilita os processos realizados em bancos de dados entre diversos usuários e aplicações (ELMASRI; NAVATHE, 2011).

O Sistema de Banco de Dados (SBD) não contém somente o próprio banco de dados, mas uma descrição completa de sua estrutura e restrições, que funciona como uma espécie de catálogo da base de dados. Assim, o SBD consegue descrever através desse catálogo informações sobre os dados do BD, como tipos, estruturas, formatos e restrições. Essas informações registradas no catálogo que se referem à própria organização do BD chamam-se metadados (ELMASRI; NAVATHE, 2011).

Os metadados são fundamentais num SBD, pois um SGBD de uso geral não é desenvolvido para uma base de dados específica, por isso é preciso que ele acesse os metadados para ter informações sobre um BD. Também é necessário que o SGBD funcione com qualquer quantidade de aplicações de banco de dados (ELMASRI; NAVATHE, 2011). Para que o SGBD seja considerado satisfatório, ele deve fornecer:

- Controle de redundância, que é a capacidade de evitar problemas gerados pelo armazenamento dos mesmos dados diversas vezes (ELMASRI; NAVATHE, 2011);
- Restrição a acesso não autorizado, evitando, assim, alterações indesejadas de outros usuários no banco de dados (ELMASRI; NAVATHE, 2011);
- Armazenamento persistente para objetos e estruturas de dados, isto é, os dados devem persistir após o encerramento do software, podendo ser lidos posteriormente até mesmo por um *software* diferente (ELMASRI; NAVATHE, 2011);
- Estruturas de armazenamento e processamento eficiente de consulta, utilizando técnicas que tornem mais rápidas a leitura e a modificação dos dados (ELMASRI; NAVATHE, 2011);
- Possibilidade de realizar *backup* e recuperação de dados, isto é, a capacidade de restaurar o banco de dados a um estado anterior mesmo que o sistema falhe (ELMASRI; NAVATHE, 2011);
- Prover múltiplas interfaces para usuários, permitindo que usuários de diferentes níveis de conhecimento (de iniciantes a avançados) sejam capazes de interagir

com o sistema de forma satisfatória (ELMASRI; NAVATHE, 2011);

- Representar relacionamentos complexos entre dados, pois existem BDs com dados relacionados entre si dos mais variados modos, e o SGBD precisa ser capaz de reproduzir essas informações, alterá-las ou estabelecer novas relações entre os dados (ELMASRI; NAVATHE, 2011);
- Impor restrições de integridade aos dados, isto é, preservar a consistência (ou corretude) das informações armazenadas durante seu ciclo de vida – restrições essas que em grandes aplicações são normalmente chamadas "regras de negócio" (ELMASRI; NAVATHE, 2011).

2.2.2 Transações

Segundo Date (2004), no contexto de bancos de dados uma transação é uma unidade lógica de trabalho que normalmente envolve diversas operações nessa base de dados. Em geral, tratam-se de várias atualizações simultâneas.

O exemplo ilustrado por Date (2004) serve para esclarecer o conceito de transação. Considere-se uma simples transferência de um montante em dinheiro de uma conta para outra. Para isso, são necessárias duas atualizações: uma para retirar a quantia de uma conta e outra para adicionar essa mesma quantia na outra conta. Se o usuário do SGBD declarar que ambas as atualizações são partes de uma mesma transação, o sistema precisa garantir que ambas as atualizações sejam realizadas ou nenhuma, mesmo que ocorram falhas durante o processo. Ou seja, se houver falhas antes que a transação se complete, todas as atualizações serão desfeitas, de modo que o processo não seja realizado parcialmente. Essa capacidade do sistema consiste no que se convencionou chamar de atomicidade (DATE, 2004).

Segundo Haerder e Reuter (1983), existem quatro propriedades fundamentais de uma transação, e elas foram responsáveis por influenciar o desenvolvimento dos SBDs através dos anos. Tratam-se de Atomicidade, Consistência, Isolamento e Durabilidade (ACID):

- Atomicidade: as transações devem ser atômicas, ou seja, devem ocorrer por completo ou não ocorrer – em outras palavras, usando o exemplo já citado, não seria possível que fosse retirado o dinheiro de uma conta sem que ele fosse depositado na outra (DATE, 2004);
- Consistência: as transações devem transformar um estado correto do banco de dados em outro estado também correto – porém não é preciso que a correção seja

mantida durante a realização da transação, desde que no final o estado retorne a um estado correto (DATE, 2004);

- Isolamento: cada transação é isolada uma da outra, e mesmo que muitas transações sejam executadas ao mesmo tempo, as atualizações geradas por cada transação não só são ocultas das outras, como não interferem com nenhuma outra transação (DATE, 2004);
- Durabilidade: uma vez que a transação esteja completa, suas alterações são permanentes e persistem no banco de dados mesmo que ocorra uma queda do sistema (DATE, 2004).

2.3 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é um modelo lógico de distribuição de dados que consiste na divisão do processamento dos dados de uma aplicação em diferentes módulos: o cliente e o servidor. O cliente é um processo responsável pela requisição e obtenção dos dados, e o servidor é um processo responsável pela manutenção dos dados. Esses processos de manutenção implementados são denominados “serviços”, de onde vem a palavra “servidor”. Esse modelo lógico, porém, não é bem definido. Por exemplo, é possível que um sistema servidor às vezes se comporte como cliente, requisitando dados de outro sistema (TANENBAUM; STEEN, 2007).

Em geral, numa arquitetura cliente-servidor, um sistema servidor está sempre em funcionamento em uma máquina potente, com maior poder de processamento e armazenamento, uma vez que um servidor deve atender vários clientes ao mesmo tempo. Por conta disso, é possível que uma só máquina não seja suficiente para o funcionamento adequado de uma aplicação, sendo necessárias várias máquinas combinadas para criar um servidor melhor. Sistemas clientes, por sua vez, podem ser executados em máquinas mais simples, de capacidade mais modesta, uma vez que normalmente são responsáveis pelas interações com a interface da aplicação. Outra características dos clientes é que eles não podem se comunicar entre si diretamente, necessitando da intermediação de ao menos um servidor (KUROSE; ROSS, 2013).

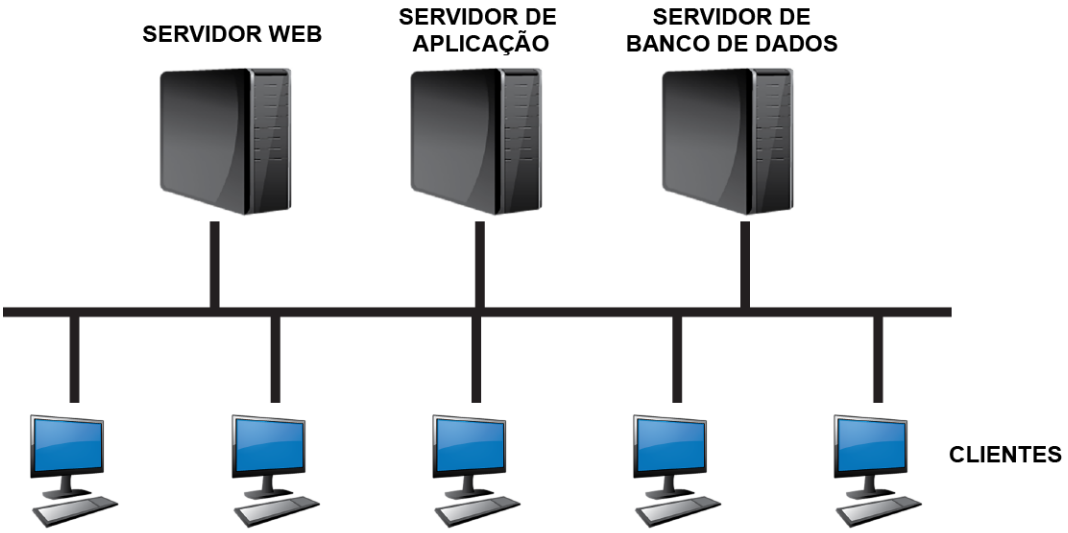
De forma a tornar a arquitetura cliente-servidor mais definida e facilitar o acesso a BDs, protegendo as aplicações de problemas de integridade dos dados e dificuldades de processamento, manutenção e escalabilidade, o processamento realizado pelo servidor com o tempo foi descentralizado em três camadas: a camada de interface (ou de apresentação), a camada de processamento (ou de lógica) e a camada de dados. O esquema de aplicações que utilizam

essa arquitetura cliente-servidor mais moderna, com o servidor com processos distribuídos em três camadas, é chamado de esquema em quatro camadas, muito usado em aplicações *web* (TANENBAUM; STEEN, 2007). Essas camadas estão descritas a seguir:

- Cliente: como já mencionado anteriormente, é a camada responsável por fazer as requisições e apresentar as respostas dessas requisições (ou seja, é onde são realizadas a entrada e a saída de dados) – no caso de aplicações *web*, o cliente interage com essa aplicação através do navegador, capaz de interagir com a camada de apresentação (TANENBAUM; STEEN, 2007);
- Apresentação (ou interface): é a camada que gerencia a exibição da interface da aplicação, ou seja, se a aplicação foi desenvolvida no padrão MVC, é na camada de apresentação do servidor que a camada de visão será apresentada – numa aplicação *web*, a camada de apresentação é chamada de servidor *web* (TANENBAUM; STEEN, 2007);
- Lógica (ou processamento): é a camada onde ocorre o processamento da aplicação, onde estão definidas suas regras de negócio, ou seja, onde ficam as camadas de modelo e controle de uma aplicação desenvolvida com MVC, recebendo as requisições da camada de apresentação e interagindo com a camada de dados – na *web*, essa camada é denominada servidor de aplicação (TANENBAUM; STEEN, 2007);
- Dados: é a camada onde são armazenados os dados, ou o SBD da aplicação – essa camada também pode ser referida como servidor de banco de dados (TANENBAUM; STEEN, 2007).

Na arquitetura cliente-servidor em quatro camadas, os dados da aplicação são organizados independentemente, de modo que é possível realizar mudanças em cada camada mais facilmente, visto que as camadas estão isoladas tal qual no padrão de projeto MVC. Essa arquitetura é particularmente útil para aplicações que utilizam SBDs, pois a organização independente dos dados na camada de dados torna possível mudanças nessa organização sem que elas afetem a aplicação, assim como alterações na aplicação também não afetam a organização do BD (TANENBAUM; STEEN, 2007).

Figura 2 – Esquema ilustrativo da arquitetura cliente-servidor em quatro camadas



Fonte: Elaborado pelo autor (2020).

2.4 Tecnologias Java

Java é uma linguagem de programação orientada a objetos, baseada em classes, que apresenta uma série de tecnologias que facilitam o desenvolvimento de aplicações em rede. Lançada em 1995 com o objetivo de ser facilmente implementável, seu código, uma vez compilado, é executável em qualquer sistema que suporte Java, não necessitando ser recompilado. Por conta disso, é uma linguagem bastante utilizada em aplicações *web* e para dispositivos móveis (ORACLE, 2013).

2.4.1 Servlet

Um "servlet" trata-se de um miniprograma escrito na linguagem Java. É uma classe que atua como um pequeno servidor *web* na camada de controle do MVC, recebendo e respondendo a requisições da interface. Numa aplicação Java *web* bem modularizada desenvolvida com Java servlets, cada atividade é realizada através de um servlet. Ele também é responsável por conectar a camada de visão com o BD operado pela camada de modelo na estrutura MVC da aplicação (ORACLE, 2015).

2.4.2 JavaBean

"*Bean*" é o nome pelo qual se convencionou chamar uma classe Java isolada – localizada na camada de modelo de uma aplicação em MVC – que representa uma entidade, com

seus atributos geralmente privados, acessíveis e modificáveis somente através de métodos de acesso ("*getters*") e métodos de modificação ("*setters*"), respectivamente, para evitar alterações indevidas. As classes da camada de modelo que se conectam ao banco de dados utilizam os métodos do *Bean* para ter acesso aos seus atributos e registrar dados na base de dados (ORACLE, 2019).

2.4.3 JavaServer Pages

Hypertext Markup Language (HTML) é uma linguagem de marcação padrão utilizada na interface de páginas *web*. Uma *JavaServer Page* (JSP) trata-se de uma tecnologia baseada em servlet, que combina HTML com código Java. É possível inserir diretamente código Java numa JSP através de elementos de *scripting*, como *scriptlets*, expressões e declarações. O código Java inserido nesses elementos é responsável por tornar a página dinâmica, mostrando os resultados das requisições dos usuários na interface, ou seja, na camada de visão de uma aplicação *web* desenvolvida seguindo o MVC (ORACLE, 2010).

2.5 Desafios e Tecnologias em Transações Online

Conquistar a confiança do cliente é o principal desafio a ser superado para o sucesso do *e-commerce*[2] (BELANGER *et al.*, 2002). O *e-commerce* só continuará ajudando no crescimento, desenvolvimento e aumento da produtividade de um empreendimento se for capaz de afastar as preocupações dos clientes com questões como roubo de identidade, vazamento de informações em operações bancárias, pagamentos e transações em geral (MANDLE; NAMDEO, 2019).

Segundo Belanger *et al.* (2002), ao tomar a decisão de fornecer seus dados pessoais, os clientes levam em consideração a confiabilidade passada pelo negócio. De acordo com sua pesquisa, os consumidores valorizavam recursos de segurança mais do que os outros índices de confiabilidade apresentados.

Conforme Moreira (2016), segurança e controle de acesso são alguns dos principais requisitos do comércio eletrônico. A realização de uma transação deve estabelecer uma confiança mútua entre as partes e garantir um acesso seguro. Nesse cenário, algumas tecnologias mostram-se fundamentais para a realização de transações de forma segura nos diversos equipamentos que

[2] Comércio eletrônico, isto é, qualquer transação comercial realizada através da Internet com o uso de algum equipamento eletrônico.

se conectam à Internet.

2.5.1 Criptografia

A criptografia foi criada para proteger informações sensíveis do acesso não autorizado de terceiros, isto é, de pessoas que não são o remetente nem o destinatário dessas informações (ABREU, 2017). Segundo Abreu (2017), essa forma de proteção de informações já era utilizada na Antiguidade. Sua essência, porém, permanece ainda hoje na segurança da informação, embora muito mais complexa.

De acordo com Mandle e Namdeo (2019), criptografar consiste no processo de transformar dados comuns em dados cifrados. Em formas mais simples de encriptação, por exemplo, de uma mensagem, rearranja-se o alfabeto de forma a torná-la incompreensível para terceiros, e somente quem tem o conhecimento sobre a regra de reorganização consegue decodificar a mensagem para compreendê-la. Essa lógica de organização chama-se comumente de "chave". Com os avanços das tecnologias computacionais, a criptografia evoluiu ao ponto em que uma chave é gerada não por simples reordenação alfabética, mas por fórmulas matemáticas complexas, usadas para cifrar e decifrar dados (ABREU, 2017).

À medida que o *e-commerce* continua a crescer, a necessidade de se utilizar criptografia para proteger informações aumenta, uma vez que a Internet é conhecida por não ser um meio de comunicação seguro (MANDLE; NAMDEO, 2019). De acordo com Yu (2015), a criptografia fornece soluções para as seguintes necessidades da segurança da informação:

- Confidencialidade: trata-se da capacidade de manter as informações em segredo (MURPHY; MURPHY, 2001);
- Integridade: refere-se à proteção da informação contra mudanças e à detecção de mudanças que possam ter acontecido (MURPHY; MURPHY, 2001);
- Autenticabilidade: significa assegurar as identidades de todas as partes envolvidas, ou seja, garantir que elas sejam quem realmente dizem ser (MURPHY; MURPHY, 2001);
- Não repudiabilidade: garantia de que o remetente e o destinatário não podem negar a comunicação, isto é, um remetente não pode, por exemplo, enviar uma mensagem e posteriormente negar que foi ele quem a enviou (MURPHY; MURPHY, 2001).

Cabe ainda destacar dois tipos de criptografia: a simétrica e a assimétrica (TANEN-

BAUM; STEEN, 2007).

Na criptografia simétrica (ou de chave privada ou de chave compartilhada), o conceito de simetria é uma alusão ao fato de que a mesma chave privada é utilizada por remetente e destinatário, que devem mantê-la em sigilo. Um algoritmo computacional produz a chave, que serve tanto para cifrar como para decifrar uma mensagem. Esse sistema de criptografia, contudo, não tem como verificar a identidade de ambos nem como garantir seu armazenamento confiável (TANENBAUM; STEEN, 2007).

Na criptografia assimétrica (ou de chave pública), são gerados dois tipos de chaves, uma pública para cada parte envolvida, visível a qualquer usuário (inclusive terceiros) e uma privada para cada parte, conhecida somente pelo usuário dono dessa chave. Como exemplo, se A deseja se comunicar com B, A deve consultar a chave pública de B e usá-la para criptografar a mensagem que deseja enviar, usando um algoritmo padrão. Ao receber a mensagem, B utiliza o mesmo algoritmo padrão junto com sua própria chave privada para descriptografar a mensagem. Nesse caso, não é necessário o compartilhamento de chaves privadas, o que por si só aumenta a segurança (KUROSE; ROSS, 2013). Além disso, a criptografia assimétrica é mais segura por ser capaz de autenticar a identidade dos usuários, bastando trocar os papéis das chaves: se A utiliza sua própria chave privada para criptografar mensagens, B pode ter certeza de que a mensagem é mesmo de A se conseguir descriptografá-la usando a chave pública de A, disponível a todos (TANENBAUM; STEEN, 2007).

A aplicação desenvolvida neste trabalho usa criptografia assimétrica para garantir a comunicação fim-a-fim entre cliente e servidor, através do *Hypertext Transfer Protocol Secure*. Também utiliza criptografia simétrica para armazenamento de informações, do lado do cliente, na forma de *cookies*. Os referidos protocolo e recurso de armazenamento local serão descritos a seguir.

2.5.2 Hypertext Transfer Protocol Secure

O *Hypertext Transfer Protocol* (HTTP) é um protocolo que determina como as mensagens são transmitidas entre cliente e servidor e é a base de toda a comunicação na *web*. Ele define a estrutura e o modo de troca de mensagens entre ambas as partes, de forma a garantir que uma requisição do cliente chegue ao servidor e que este envie uma resposta que seja entregue com sucesso ao cliente, procurando nesse processo evitar o corrompimento de mensagens, ocultar informações sobre o cliente e acelerar a troca dessas mensagens (KUROSE; ROSS, 2013).

Uma transação, como explicado anteriormente na Subseção 2.2.2, é, simplificadamente, uma troca de mensagens requisitando atualizações simultâneas em bancos de dados. Ainda assim, as funcionalidades do HTTP não são suficientes para tornar essa troca segura. Por isso, foi criado o HTTPS, uma extensão do HTTP utilizada para comunicação segura na *web* através de mensagens criptografadas (KUROSE; ROSS, 2013).

No HTTPS, o protocolo de comunicação é criptografado geralmente utilizando o *Transport Layer Security* (TLS), uma versão melhorada do *Secure Sockets Layer* (SSL), protocolo criptográfico utilizado desde que o HTTPS foi concebido. Apesar de o TLS fornecer uma criptografia mais forte, sua semelhança com o SSL e o fato de ele ainda ser usado em aplicações fazem com que o HTTPS seja frequentemente referido como um protocolo de comunicação criptografado por TLS/SSL (TANENBAUM, 2003).

O protocolo TLS/SSL aplicado ao HTTPS utiliza criptografia assimétrica e certificados digitais, o que torna possível realizar transações mais seguras, além de aprimorar a privacidade e a autenticação (YASIN *et al.*, 2012).

2.5.3 Distribuição de Dados no Contexto Móvel e Cookies

Segundo Coelho e Prado (2014), os SBDs oferecem ferramentas de confiança, segurança, disponibilidade, integridade e acesso eficiente aos dados. Num contexto de utilização de uma aplicação que faz uso de um SBD através de um dispositivo móvel, o acesso ao banco de dados ocorre por meio de uma conexão sem fio. Portanto, existe a necessidade de registrar dados de uma forma que não comprometa os processos realizados pelos usuários.

Ainda de acordo com Coelho e Prado (2014), para a persistência dos dados nesse contexto, são necessárias mudanças no modo de registro de dados de maneira a tratar dificuldades inerentes à comunicação móvel, como as limitações da rede, a mudança de localização constante, as desconexões frequentes dos dispositivos móveis, bem como as limitações de processamento, memória, bateria e largura de banda desses dispositivos.

Portanto, é necessário que aplicações que se destinam também ao uso em dispositivos móveis utilizem-se de dados distribuídos entre cliente e servidor. Assim, existe uma certa divisão da responsabilidade pela gestão dos dados entre as unidades móveis – que por vezes atuam não só como clientes móveis, mas como servidores móveis – e os servidores fixos, de maneira que as unidades móveis também armazenam alguns dados de forma *offline*, a fim de certificar que as propriedades fundamentais ACID das transações entre as partes sejam cumpridas, e as transações,

realizadas com sucesso (COELHO; PRADO, 2014).

Nesse cenário, a utilização de *cookies*, uma tecnologia do protocolo HTTP que consiste no armazenamento de pequenas quantidades de dados no lado do cliente por meio do seu navegador, pode mostrar-se conveniente no ambiente de desenvolvimento de aplicações *web* para dispositivos móveis, na medida que permite que a aplicação *web* leia informações armazenadas no lado do cliente e seja capaz de facilmente recuperar dados mesmo após sucessivas desconexões.

2.5.4 Modelo de Dados Relacional

Como já mencionado na Subseção 2.2.1, bases de dados possuem informações relacionadas entre si de diversas maneiras. Ao gerenciar BDs, um *Database Administrator* (DBA), ou administrador de banco de dados, precisa lidar com grandes quantidades de informações compartilhadas por muitos usuários, de diferentes perfis. Por essa razão, a possibilidade de danificar esses dados é grande, visto que é improvável que todos esses usuários obedeçam a regras de modificação de dados de forma espontânea. Foi nessa circunstância que foi desenvolvido o paradigma de banco de dados relacional, capaz de fornecer um gerenciamento de dados bastante disciplinado (CODD, 1986).

Na abordagem relacional, a estruturação do BD é feita através de um conjunto de relações. Todos os processos realizados na base de dados são baseados em relações e retornam relações como resultados. Um SBD relacional é atômico, isto é, nele cada dado é indivisível. Por isso sua representação é simples, ao contrário dos modelos não relacionais, que comumente apresentam muitos dados compostos ao invés de atômicos, cuja representação e manuseio tornaram-se cada vez mais complicados até o advento do paradigma relacional. Além disso, as diversas restrições impostas por um SGBD relacional facilitam o trabalho do DBA, impedindo que usuários comprometam a consistência dos dados (CODD, 1986).

No caso de uma aplicação *e-commerce*, a utilização de bancos de dados relacionais em conjunto com uma arquitetura cliente-servidor em quatro camadas auxilia na separação do nível de processamento do nível de dados, já que dados e processamento são considerados independentes. Isso fornece maior segurança aos dados, que ficam isolados no servidor de banco de dados, separados de qualquer falha ou brecha no funcionamento da aplicação. Ademais, os dados armazenados nesse servidor são facilmente replicáveis para outro servidor, o que torna mais simples a recuperação em caso de perda de dados ou de comprometimento do servidor original (TANENBAUM; STEEN, 2007).

2.6 Tecnologias Utilizadas

O projeto proposto é um sistema que utiliza um SGBD relacional, desenvolvido considerando o padrão MVC e a arquitetura cliente-servidor, implementando tecnologias Java e os conceitos de segurança expostos na Seção 2.5.

O padrão MVC foi escolhido devido à facilidade de manutenção, organização e extensibilidade do código, de forma que cada camada pode ser alterada separadamente. Como o objetivo do projeto é implementar um sistema *back-end* com suporte a pagamentos seguros, as camadas de modelo e controle serão as mais trabalhadas.

A arquitetura cliente-servidor descrita também auxilia na manutenção da estrutura do padrão MVC e na comunicação com outros servidores, de forma que o servidor de aplicação por vezes se comporta como cliente. Como pagamentos tratam-se de comunicações, o papel dos servidores na descentralização dos dados é essencial nessas transações, visto que eles são responsáveis por manter cada processo funcionando de forma independente, fornecendo maior segurança.

Nessa circunstância, a linguagem Java e o paradigma de dados relacional, com o uso de SQL[3], foram escolhidos pela maior familiaridade do autor. Java, em particular, apresenta diversas tecnologias, como as já descritas na Seção 2.4, que facilitam o desenvolvimento de aplicações que utilizam comunicações em rede entre clientes e servidores, o que inclui a implementação de tecnologias de segurança da informação citadas. Um SGBD relacional, por sua vez, permite o gerenciamento de grandes quantidades de dados de maneira mais simples e intuitiva, visto que os representa em forma de tabelas, enquanto o SQL é um padrão conhecido, o que facilita a manutenção do banco de dados por outrem. O HTTP *cookie* (também conhecido como *web cookie*), por sua vez, será utilizado no lado do cliente para sanar algumas necessidades descritas na Subseção 2.5.3.

2.7 Trabalhos Relacionados

Conforme mencionado no Capítulo 1 e apresentado nos Anexos A, B e C, a equipe Time5 realizou diversas pesquisas para a disciplina de Projeto Integrado II, com o objetivo de coletar dados sobre o problema da formação de filas de espera em lanchonetes. Tais dados auxiliaram também na definição dos requisitos da aplicação OnTime, que se propunha a ser uma

[3] SQL é a linguagem padrão utilizada em SGBDs relacionais.

solução *web* responsiva para reduzir esse problema, em particular em cantinas universitárias.

A coleta de dados contou com uma entrevista presencial com um cliente de teste para a aplicação a ser desenvolvida (Anexo A), encontros presenciais com proprietários e funcionários de cantinas localizadas na UFC para o preenchimento de um formulário para avaliá-los como potenciais clientes da aplicação (Anexo B), pesquisa remota com clientes de lanchonetes, público-alvo do sistema que seria desenvolvido (Anexo C), além de uma pesquisa de mercado sobre aplicações com funcionalidades similares já existentes no mercado.

Através dos estudos realizados nas cantinas da UFC, percebeu-se resistência em reconhecer o problema das filas, uma vez que se observou sua formação nos estabelecimentos que negaram ter problemas com isso. Além disso, juntamente com a pesquisa junto ao cliente de teste, foi possível compreender como é feito o gerenciamento de estoque e vendas nesses estabelecimentos, que, por serem de pequeno porte, ou não realizam qualquer forma de gerenciamento, ou o fazem de forma empírica, conforme necessidades diárias ou semanais. Nesse contexto, embora alguns dos entrevistados tenham demonstrado relutância em utilizar um sistema *online* que os auxiliasse nesse gerenciamento, o recurso de agendamento de pedidos foi considerado interessante para o auxílio no atendimento aos clientes, conforme as respostas dos questionários.

Na entrevista com o cliente de teste, constatou-se que, além do agendamento de pedidos, seria importante a disponibilização de uma forma de pagamento à distância, visto que o tempo de preparo de certos produtos foi apontado como um dos maiores fatores de formação de filas. Como certos pedidos demoram a ser preparados, o pagamento realizado previamente daria a segurança ao proprietário do estabelecimento de que o pedido agendado pode ser preparado sem prejuízo, visto que já foi pago.

Na pesquisa realizada com o público-alvo em potencial da aplicação, usuários frequentes de cantinas universitárias, constatou-se que muitos reconhecem as filas de espera como um problema recorrente na compra de produtos em lanchonetes e estariam dispostos a utilizar uma aplicação para dispositivos móveis para diminuir essa inconveniência. Uma quantidade significativa prefere o pagamento por meio de cartão de crédito. Observou-se também que todos que responderam ao formulário utilizam *smartphones* no seu dia a dia, e a maior parte deles dispõe de conexão móvel com a Internet em seus dispositivos.

Na pesquisa de mercado, juntamente com dados das pesquisas com clientes em potencial da aplicação, verificou-se que aplicativos populares, como iFood, Uber Eats e Rappi, concentram-se na entrega de pedidos, o que seria inviável para cantinas universitárias, que são de

pequeno porte e de atuação local. O foco da pesquisa, então, foi a procura por soluções diferentes das já citadas, resumidas a seguir:

- "James Delivery"[4] – do grupo GPA, dona das redes Pão de Açúcar e Extra, trata-se de um aplicativo para dispositivos móveis de encomenda e entrega de produtos de supermercado, restaurantes ou farmácias;
- "Tem Fila?"[5] – aplicação disponível para *smartphones* e também para *web* que permite consultar o tempo de espera e compartilhar informações sobre filas;
- "Hurikat"[6] – um site colaborativo em que usuários compartilham informações com o objetivo de indicar o melhor horário para se evitar grandes filas em estabelecimentos;
- "Merconnect"[7] – aplicativo para dispositivos móveis que se destina aos donos de estabelecimentos com aplicativos móveis desenvolvidos pela Mercadapp para realizar o gerenciamento de pedidos e receber notificações;
- "u.Sit"[8] – sistema que se propõe a servir de interface de gerenciamento de filas, avisando seu cliente por *short message service* (SMS) quando for sua vez de ser atendido em algum estabelecimento, não necessitando do uso de um aplicativo;
- "Pede Pronto"[9] – aplicativo móvel cuja proposta é reduzir o aborrecimento gerado pelas filas, tendo como público-alvo os consumidores que se alimentam diariamente fora de casa.

O aplicativo "James Delivery" apresenta a mesma desvantagem mencionada de aplicativos como o iFood, focando-se na entrega de produtos. Outra desvantagem é que está disponível para apenas algumas localidades.

O "Tem Fila?" e o "Hurikat" são soluções colaborativas voltadas para que clientes de estabelecimentos evitem filas, não oferecendo uma solução para pagamentos, além de também estarem disponíveis para apenas alguns lugares do Brasil.

O "Merconnect" e o "u.Sit" são voltados para o gerenciamento de pedidos, enquanto o "Pede Pronto" apresenta uma solução para evitar filas e para realizar pagamentos. O "Merconnect" é para clientes donos de supermercados com aplicativos já desenvolvidos pela

[4] James Delivery – https://play.google.com/store/apps/details?id=br.com.jamesdelivery.android.james. Acesso em: 8 abr. 2021.
[5] Tem Fila? – http://www.temfila.com.br/. Acesso em: 8 abr. 2021.
[6] Hurikat – https://www.naoficonafila.com.br/. Acesso em: 8 abr. 2021.
[7] Merconnect – https://play.google.com/store/apps/details?id=com.merconnect. Acesso em: 8 abr. 2021.
[8] u.Sit – https://giinger.com.br/usit/. Acesso em: 8 abr. 2021.
[9] Pede Pronto – https://pedepronto.com.br/. Acesso em: 8 abr. 2021.

“Mercadapp”, o que exclui microempreendedores como os donos de lanchonetes locais. Já o “u.Sit” não constitui uma aplicação móvel propriamente dita, baseando-se no envio de notificações via SMS. O “Pede Pronto”, por sua vez, apresenta uma proposta mais interessante, mas seu alcance está limitado somente às cidades de São Paulo e região metropolitana, Rio de Janeiro e Brasília.

3 PROCEDIMENTOS METODOLÓGICOS

Como já referido no Capítulo 1 deste trabalho, o problema das filas ocorre em muitos estabelecimentos comerciais, o que causa diversos transtornos, como a perda de tempo na espera de atendimento e o consequente aborrecimento para clientes, além da perda de produtividade e até de clientes para vendedores, cujos comércios podem adquirir má reputação ou serem evitados em certos horários.

No ramo do comércio de lanches, já existem diversas soluções no mercado que consistem em sistemas que visam fornecer ao cliente a possibilidade de realização de seus pedidos à distância, dispensando sua presença em filas tanto para realizar um pedido como para recebê-lo. Tais sistemas também permitem aos vendedores expandir sua capacidade de atendimento. A conveniência de tais sistemas aumenta por estar ao alcance dos usuários em dispositivos móveis como *smartphones*, o que acelera o processo de compra de alimentos para consumo imediato via rede, oferecendo opção de pagamento por cartão de crédito ou débito, o que antes era realizado de forma local ou à distância por interação com atendentes por chamadas telefônicas, com a impossibilidade de se usar cartões. Aplicações móveis como iFood, Uber Eats e Rappi são alguns exemplos dessas soluções encontradas no mercado.

No contexto específico de cantinas universitárias ou escolares de pequeno porte e atuação local, porém, tais soluções não se mostram adequadas, uma vez que muitas delas focam na entrega de pedidos, além de cobrar taxas que não são condizentes com a realidade financeira desses estabelecimentos. A oferta de produtos nesses comércios, também, é bastante limitada, e seu estoque é planejado através de observação empírica ou a curto prazo. As filas nesses tipos de lanchonete, contudo, também podem ser grandes em certos horários e contam com o fator agravante de os clientes disporem de pouco tempo para serem atendidos, como em intervalos entre aulas, enquanto o número de funcionários é pequeno, e alguns pedidos demandam tempo para serem preparados, o que torna a perda de vendas mais provável, uma vez que os clientes procuram evitar locais com maior tempo de espera. Muitas dessas cantinas também contam com opção de pagamento por cartão de crédito, mesmo que ao custo de uma taxa sobre as vendas, uma vez que aceitar somente pagamento em dinheiro por vezes causa a perda de clientes que só dispõem de cartão para realizar a compra.

Observando o mérito de preencher essa lacuna de mercado, realizou-se o desenvolvimento do protótipo de uma aplicação *web* responsiva denominada “OnTime” na disciplina de Projeto Integrado II, que visou apresentar uma solução viável para cantinas universitárias

realizarem suas vendas à distância mediante a realização de um pedido prévio pelo cliente através de seu *smartphone*, possibilitando seu preparo e sua posterior retirada no local, evitando as filas.

O presente trabalho utiliza o método experimental com análise qualitativa de dados para refatorar e desenvolver o *back-end* dessa aplicação, prototipado pelo autor, procurando estudar e experimentar alternativas que auxiliem na construção de um *software* que cumpra com o objetivo de fornecer um sistema com transações seguras e de código aberto para dispositivos móveis que possa atender a cantinas universitárias. O processo de desenvolvimento usou o modelo incremental, com ciclos de desenvolvimento iterativos, em que a cada etapa do desenvolvimento do *software* os processos já realizados foram analisados e testados novamente, visando aperfeiçoar o sistema de forma gradativa.

No contexto deste projeto, o método de pagamento por meio de cartão crédito foi definido como forma de garantir ao pequeno empreendedor que o preparo do pedido seja feito sem prejuízos, uma vez que o pagamento já terá sido realizado quando o pedido for feito. O uso de API de terceiros já conhecidos no mercado busca fornecer credibilidade, confiança e segurança para a realização de transações entre clientes e vendedores, sem que dados sensíveis sejam armazenados no sistema desenvolvido no projeto.

Houve um levantamento de requisitos para adequar o sistema às necessidades do trabalho, assim como a elaboração de um Modelo Entidade-Relacionamento (MER) para definir a estrutura do banco de dados. Com base nisso, foram feitos testes unitários não automatizados da parte básica da aplicação, que contém o "CRUD"[10] dos dados necessários para o funcionamento do *e-commerce*.

Foi realizado um estudo de algumas APIs de pagamento seguro disponíveis no mercado, tais como PayPal, PagSeguro e Mercado Pago, para sua inclusão no código, bem como a melhoria do código do *back-end* de forma a aumentar a segurança dos dados da aplicação e estudar sua adaptação ao ambiente móvel. Nessa fase os experimentos com o código definiram qual tipo de criptografia, como e quais outras tecnologias de segurança seriam utilizadas e sua adequação aos propósitos do *software*.

Por fim, a integração do sistema foi realizada através da utilização da API de pagamento seguro. Foi realizado um teste com o sistema e com o serviço de pagamento seguro para a verificação da correta integração de ambos.

[10] As operações *create, read, update, and delete* (CRUD) – conhecidas como criar, ler, alterar e apagar, respectivamente, em português – são as funções básicas do armazenamento persistente de dados. Criar é a ação de cadastrar informações no sistema. Ler significa ser capaz de recuperar informações sobre o que foi cadastrado. Alterar é a ação de modificar esses dados. Apagar é a ação de apagar um dado do sistema.

4 PRODUTO

Neste capítulo será apresentada uma visão geral do sistema, com os requisitos levantados de acordo com os objetivos do trabalho, as restrições do sistema e a modelagem do banco de dados conforme os requisitos elicitados. Em seguida serão descritas as principais funcionalidades da aplicação. Por fim será detalhada a arquitetura do sistema, com o modo como as classes estão organizadas, as medidas de segurança adotadas para atender aos requisitos e a forma como foi realizada a integração da aplicação com o sistema de pagamento do PagSeguro, que, cabe destacar, foi escolhido para este trabalho de acordo com critérios explicados no Capítulo 5 (Seção 5.2.1).

4.1 Visão Geral do Sistema

A aplicação OnTime, conforme exposto, foi criada para auxiliar donos de cantinas em *campi* universitários que busquem agilizar seus processos de vendas. Ela foi dividida em um conjunto de requisitos funcionais e não funcionais a fim de ser codificada em consonância com os interesses dos clientes. A seguir, serão descritos tais requisitos, as restrições do sistema e a sua codificação.

4.1.1 Requisitos

Encontram-se a seguir os requisitos a serem cumpridos pelo sistema, levantados com base nas necessidades apresentadas no Capítulo 1 e detalhadas no Capítulo 3.

Lista de requisitos funcionais:

- RF01 – Autenticação: o usuário deve ser capaz de realizar o *login* no sistema;
- RF02 – Diferenciação de Perfis de Usuário: perfil de “cliente” para consumidores e perfil de “vendedor” para proprietários ou atendentes da lanchonete, de forma a possibilitar diferentes funcionalidades a cada perfil;
- RF03 – Realização de Pedido: o cliente pode selecionar entre os itens disponíveis e realizar o seu pedido;
- RF04 – Integração para Pagamento: o sistema deve estar integrado a um serviço de pagamento seguro por meio de API de forma que, ao finalizar do pedido, o cliente pode realizar o pagamento com cartão de crédito de forma segura;
- RF05 – Máquina de Estados do Pedido: o pedido deve ter um estado, modificável

ao longo do processo de venda, consultável pelo cliente para saber se o pagamento foi finalizado com sucesso ou se o pedido já está pronto para entrega, enquanto o vendedor pode consultar e alterar o estado para pronto para entrega após o preparo;

- RF06 – Identificação do Pedido: o sistema deverá fornecer uma senha para cada pedido realizado, para possibilitar sua identificação junto ao estabelecimento;
- RF07 – CRUD[11] de Produto: o vendedor pode cadastrar, consultar, alterar ou remover produtos para venda na aplicação.

Lista de requisitos não funcionais:

- RNF01 – Segurança: o sistema deve utilizar criptografia para proteger os dados dos usuários e de suas transações;
- RNF02 – Otimização para *Smartphones*: a aplicação deverá ser desenvolvida de modo a permitir seu uso adequado por meio de *smartphones*;
- RNF03 – Arquitetura: o sistema deve ser desenvolvido de forma a atender à arquitetura cliente-servidor, bem como utilizar o padrão de projeto MVC para facilitar a organização do código e sua manutenção;
- RNF04 – Validação: o sistema deve realizar a validação dos dados fornecidos pelo usuário.

o sistema de pagamento do PagSeguro foi escolhido para a aplicação de acordo com critérios explicados no Capítulo 5 (Seção 5.2.1).

4.1.2 Restrições

A seguir encontram-se listadas as restrições do sistema:

- Plataforma de desenvolvimento: Oracle Java 8 *Software Development Kit* (SDK), Java EE 7 Web, Servlet 3.1, JSP 2.3, Apache NetBeans 8.2 e pgAdmin 4 (versão 3.2);
- Plataforma de serviço *web*: Apache Tomcat 8.0.27.0;
- Banco de dados: PostgreSQL 9.6.10;
- Servidor: Microsoft Windows Server 2019.

O *back-end* foi desenvolvido com o uso do Apache NetBeans por ser um IDE *open source* que oferece diversos recursos para desenvolvimento de aplicações em rede, entre eles a

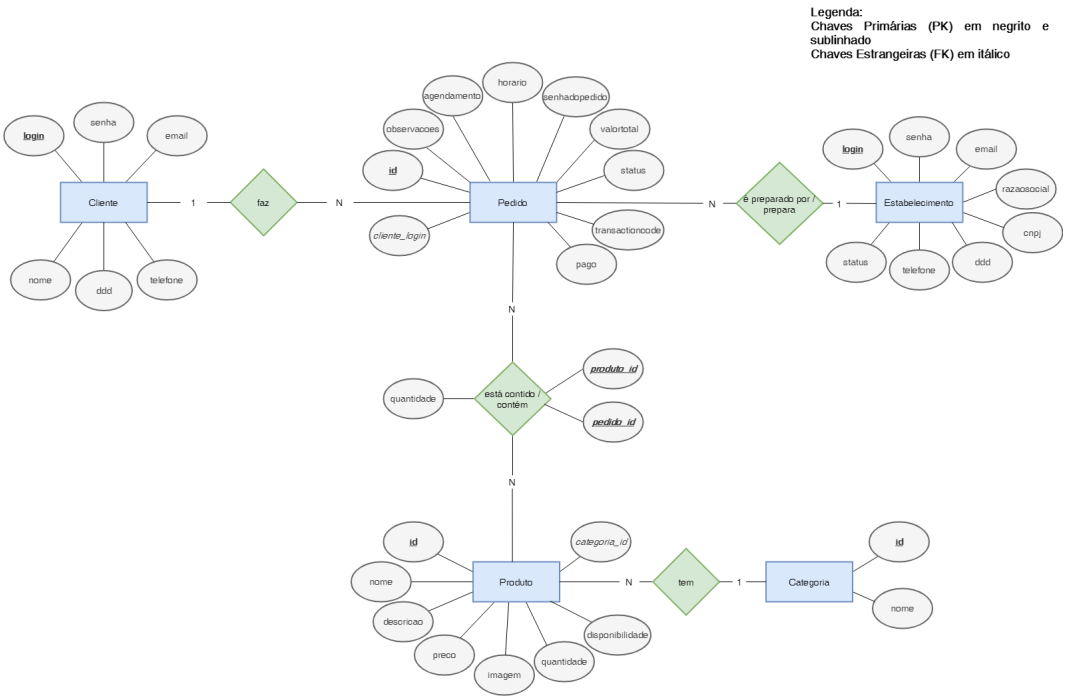
[11] *Create, read, update, and delete*.

fácil codificação e refatoração do código e o grande suporte a tecnologias Java, tendo sido ele próprio desenvolvido nessa linguagem. O PostgreSQL, por sua vez, é um servidor de banco de dados *open source* que permite o gerenciamento simples e seguro do bancos de dados através do pgAdmin 4, uma plataforma também de código aberto. Além disso, tais ferramentas, juntamente com o servidor de aplicação Apache Tomcat, com suas respectivas versões, foram as mesmas utilizadas na prototipação. A escolha por sua utilização deve-se também à familiaridade do autor, bem como à limitação do *hardware* utilizado para o desenvolvimento.

4.1.3 Modelagem dos Dados

A Figura 3 apresenta o Modelo Entidade-Relacionamento que representa a modelagem dos dados utilizada para a elaboração do banco de dados relacional da aplicação.

Figura 3 – Modelo Entidade-Relacionamento



Fonte: Elaborado pelo autor (2020).

A Figura 4 mostra as tabelas criadas no banco de dados, com os nomes das entidades em negrito precedendo suas respectivas tabelas, compostas por colunas que representam seus atributos. A notação "PK"[12] indica um atributo que é chave primária da tabela, enquanto a notação "FK"[13] indica um atributo que é chave estrangeira. A observação "*NOT NULL*"[14] indica

[12] *Primary Key* (PK): chave primária, em português.
[13] *Foreign Key* (FK): chave estrangeira, em português.
[14] *NOT NULL*: não nulo, em português.

um atributo obrigatório. Tendo como base o MER da Figura 3, foram criadas seis entidades:

- “cliente”: representa o usuário consumidor;
- “estabelecimento”: representa o usuário vendedor;
- “categoria”: representa o tipo de um produto à venda;
- “produto”: representa cada produto cadastrado para venda no estabelecimento;
- “pedido”: representa a solicitação de compra feita por um cliente;
- “pedido_produto”: representa os produtos que pertencem a um determinado pedido, o que é decorrente do relacionamento binário N:N entre pedido e produto mostrado no MER.

Figura 4 – Lista de Tabelas do Banco de Dados

cliente									
login (PK)	senha (NOT NULL)	email (NOT NULL)	nome (NOT NULL)	ddd (NOT NULL)	telefone (NOT NULL)				
estabelecimento									
login (PK)	senha (NOT NULL)	email (NOT NULL)	razaosocial (NOT NULL)	cnpj (NOT NULL)	status (NOT NULL)	ddd (NOT NULL)	telefone (NOT NULL)		
categoria									
id (PK)	nome (NOT NULL)								
produto									
id (PK)	nome (NOT NULL)	descricao	preco (NOT NULL)	imagem	quantidade (NOT NULL)	disponibilidade (NOT NULL)	categoria_id (FK) (NOT NULL)		
pedido									
id (PK)	observacoes	agendamento	horario (NOT NULL)	senhadopedido (NOT NULL)	status (NOT NULL)	valortotal (NOT NULL)	transactioncode (NOT NULL)	pago (NOT NULL)	cliente_login (FK) (NOT NULL)
pedido_produto									
pedido_id (PK) (FK)	produto_id (PK) (FK)	quantidade (NOT NULL)							

Fonte: Elaborado pelo autor (2020).

O detalhamento de cada campo utilizado no banco de dados pode ser visto no Apêndice A.

4.2 Principais Funcionalidades

A seguir são descritas as funcionalidades mais importantes da aplicação.

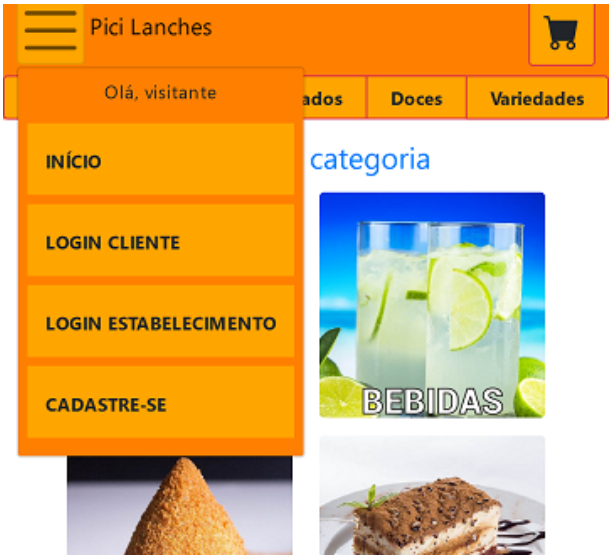
4.2.1 Diferenciação de Perfis de Usuário

Os usuários da aplicação diferenciam-se em três perfis: visitante, cliente e estabelecimento. Cada perfil possui acesso a ações diferentes no sistema.

O usuário visitante (Figura 5) trata-se de um usuário que não realizou o *login* e possui acesso limitado às funcionalidades da aplicação. Ele pode visualizar os produtos, adicioná-los ao seu carrinho de compras, mas não pode realizar o seu pedido. É possível para o visitante fazer

o *login* como cliente ou como estabelecimento, além de poder cadastrar-se como cliente.

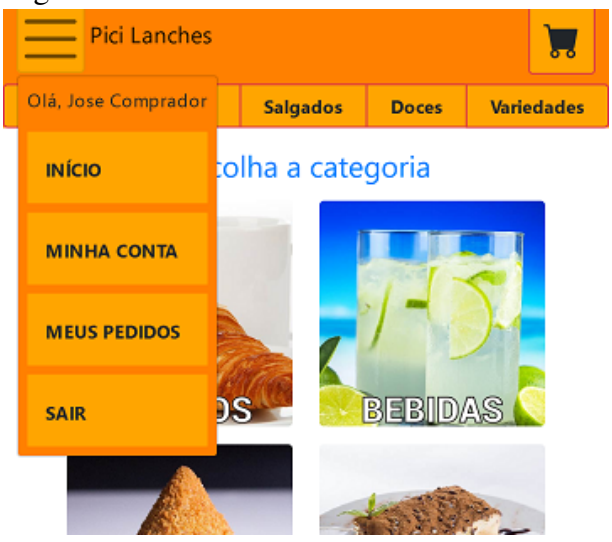
Figura 5 – Tela de usuário visitante



Fonte: Captura de tela realizada pelo autor (2021).

O usuário cliente (Figura 6) trata-se do usuário que realizou o *login* como cliente. Ele pode visualizar os produtos, adicioná-los ao carrinho de compras, bem como fazer o seu pedido, podendo realizar o pagamento via redirecionamento no sistema do PagSeguro. O cliente pode visualizar seu histórico de pedidos, alterar os dados cadastrados na sua conta (com exceção do login) ou, ainda, excluir a própria conta. Também pode optar por fazer o *logout* da aplicação, voltando à condição de visitante.

Figura 6 – Tela de usuário cliente



Fonte: Captura de tela realizada pelo autor (2021).

O usuário estabelecimento (Figura 7) trata-se do perfil do vendedor no sistema. Não

é possível realizar o cadastro do estabelecimento através da aplicação, sendo tarefa do DBA adicionar seus dados ao BD. O vendedor, porém, pode alterar seus dados, com exceção do *login*, da sua razão social e do número de Cadastro Nacional de Pessoas Jurídicas (CNPJ). Como vendedor, ele pode visualizar os pedidos pendentes do dia e alterar seu status conforme o pedido estiver pronto para entrega ou já foi entregue. Também pode alterar o status do estabelecimento, definindo-o como aberto ou fechado. Além disso, pode realizar operações CRUD de produtos na aplicação, visualizar o histórico de pedidos ou realizar o *logout*.

Figura 7 – Tela de usuário estabelecimento

Fonte: Captura de tela realizada pelo autor (2021).

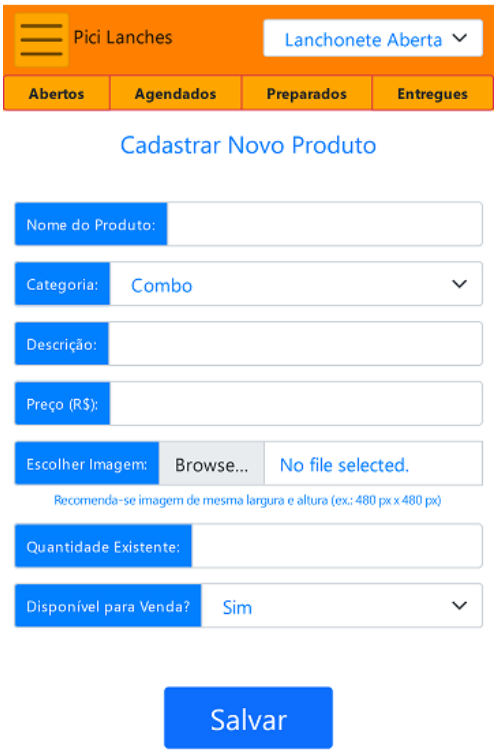
4.2.2 Pagamento Seguro

O cliente, ao finalizar seu pedido, é direcionado à página do PagSeguro para realizar seu pagamento. Os dados da compra, bem como os dados do cliente, são passados para o servidor do PagSeguro via API, com a utilização do protocolo HTTPS para comunicação mais segura entre clientes e servidores. Os dados do seu cartão de crédito não são compartilhados com a aplicação.

4.2.3 Gerenciamento de Produtos

O usuário com perfil do estabelecimento pode realizar o cadastro de um produto para venda na aplicação (Figura 8). Ao realizar o cadastro, ele insere o nome do produto, pode inserir uma descrição e uma imagem ilustrativa do produto de forma opcional, estabelecer o preço e a que categoria pertence o produto, a quantidade de unidades em estoque para venda na aplicação e se deseja que o produto fique disponível ou não para venda, o que determina se clientes e visitantes verão o produto no sistema. A aplicação também não colocará à venda na interface produtos cujo estoque for 0. O vendedor pode, ainda, consultar a lista de produtos cadastrados, modificar seus dados (exceto o ‘’id”) ou excluir um produto (Figura 9).

Figura 8 – Cadastro de novo produto



Fonte: Captura de tela realizada pelo autor (2021).

Figura 9 – Lista de produtos

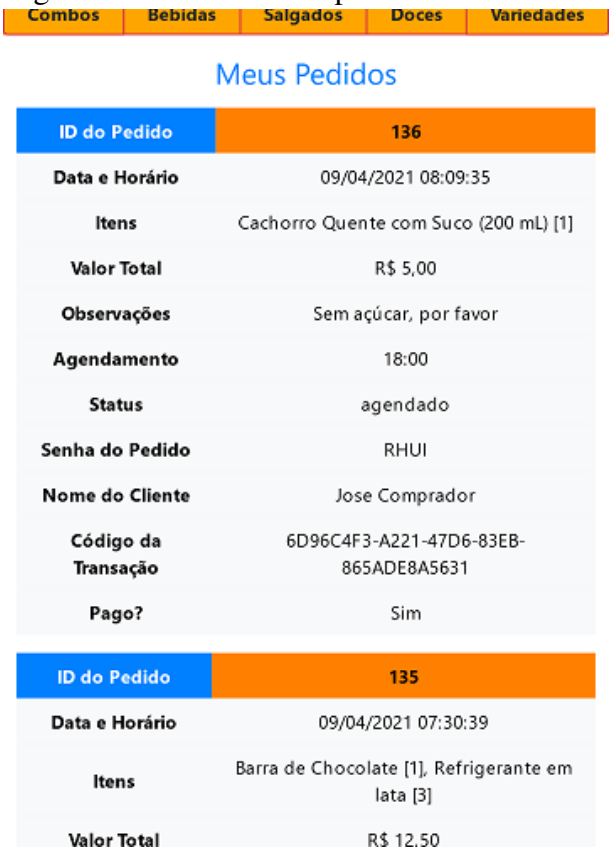
Fonte: Captura de tela realizada pelo autor (2021).

4.2.4 Gerenciamento de Pedidos

O cliente, ao realizar seu pedido, pode incluir observações sobre como quer que sejam preparados e se deseja agendar para um horário específico disponível. Ao terminar de realizar o *checkout*. Caso seu pagamento não seja aprovado, o pedido tem seu status definido

como “cancelado”. Caso seja aprovado, o status é definido como “em preparo”, se não houve agendamento, ou como “agendado”, se houve. Após a aprovação do pagamento, o pedido passa a ficar visível na interface principal do vendedor. O cliente recebe a senha do seu pedido e pode acompanhar seu status na página “Meus Pedidos”, junto com todos os seus detalhes (Figura 10).

Figura 10 – Histórico de pedidos do cliente



Fonte: Captura de tela realizada pelo autor (2021).

O vendedor tem em sua interface principal os pedidos feitos no dia separados em quatro abas: “Abertos”, “Agendados”, “Preparados” e “Entregues”. Quando a aplicação recebe a confirmação de pagamento, o usuário com perfil estabelecimento recebe o pedido pendente nessa interface e, assim como o cliente, pode ver seus detalhes: produtos, quantidade, senha, data e horário, código da transação do pagamento realizado no PagSeguro, valor total e nome do cliente que fez o pedido, além de agendamento e observações, caso o cliente tenha inserido esses campos opcionais.

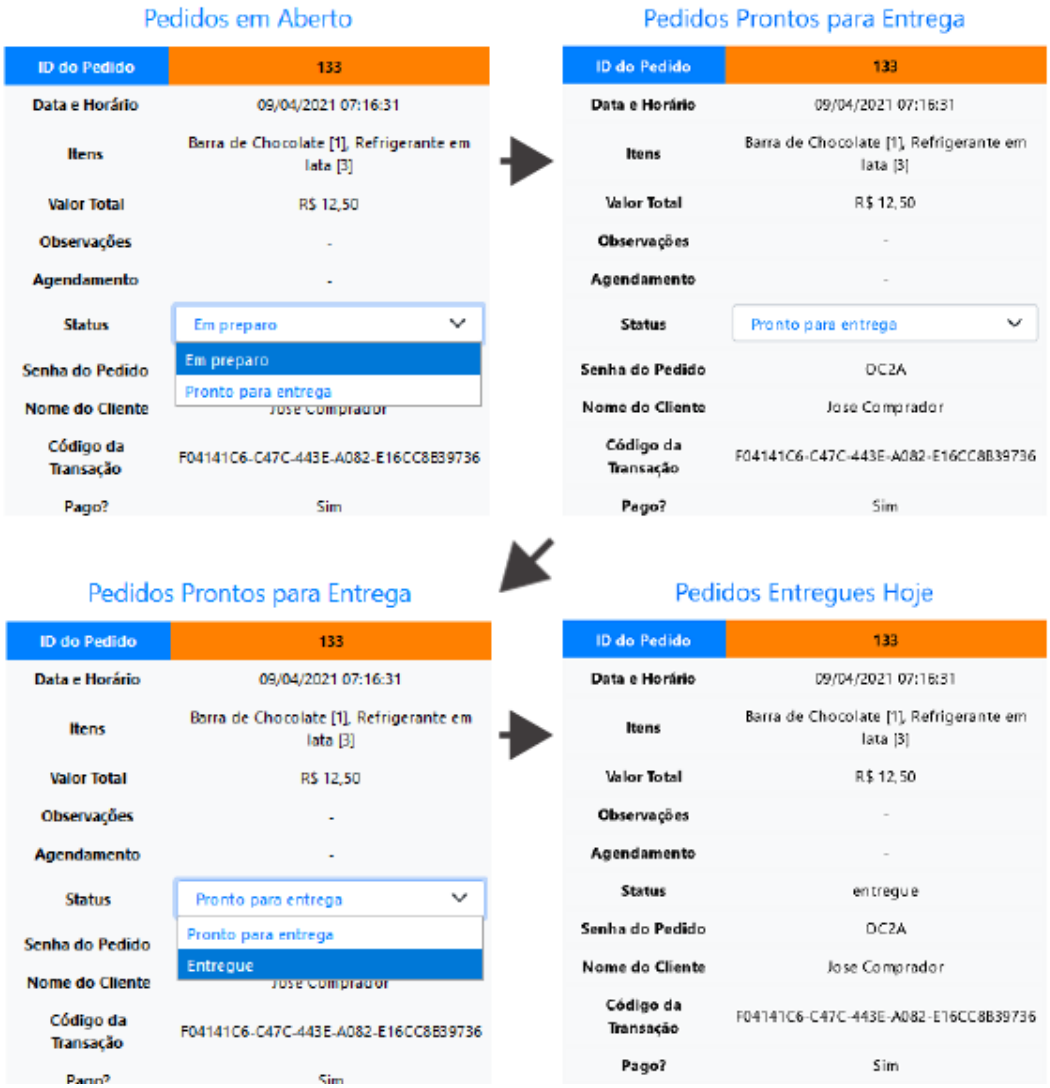
Os pedidos pendentes sem agendamento ficam separados na página “Abertos”, enquanto os pedidos pendentes ficam na página “Agendados”, como mostra a Figura 11), o que possibilita priorizar certos pedidos.

Figura 11 – Pedidos pendentes separados por realização de agendamento

Fonte: Capturas de tela realizadas pelo autor (2021).

Após o preparo, o vendedor pode modificar o status do pedido de “em preparo” ou “agendado” para “pronto para entrega”. O pedido pronto para entrega, então, é transferido para a página “Preparados”. Após entregar o pedido, ele pode modificar o status do pedido de “pronto para entrega” para “entregue”, assim o pedido é transferido para a página de “entregues”. Esse processo é exposto na Figura 12.

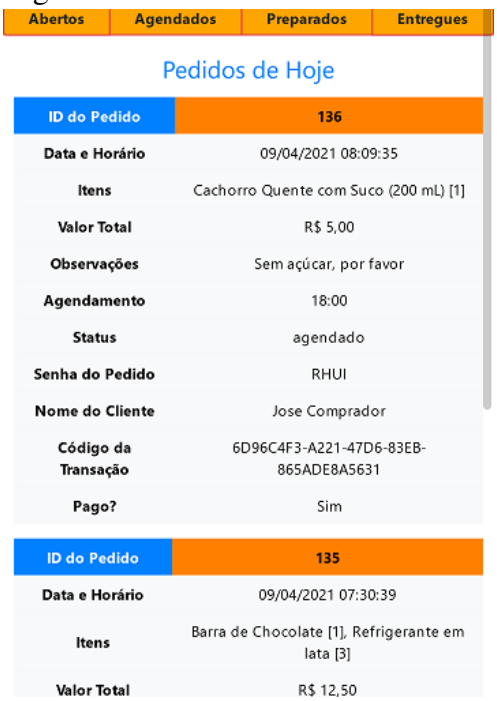
Figura 12 – Mudanças de estado do pedido



Fonte: Capturas de tela realizadas pelo autor (2021).

O usuário vendedor pode ainda ver o histórico dos seus pedidos do dia (Figura 13) e de todos os pedidos já feitos na aplicação (Figura 14).

Figura 13 – Pedidos do dia



Fonte: Captura de tela realizada pelo autor (2021).

Figura 14 – Histórico de pedidos

Fonte: Captura de tela realizada pelo autor (2021).

4.2.5 Notificações

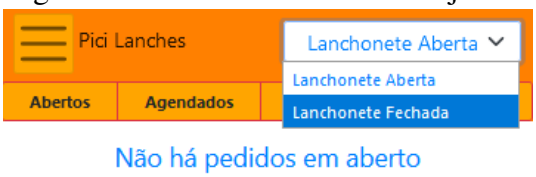
Depois de realizar um pagamento com sucesso, o cliente é notificado por *e-mail*, através do seu endereço de *e-mail* cadastrado no sistema, sobre a senha do seu pedido. Ao mesmo tempo, o vendedor também recebe em seu *e-mail* cadastrado na conta uma mensagem de que há um novo pedido pendente.

De forma semelhante, quando o vendedor altera o status para “pronto para entrega”, o sistema envia para o cliente um *e-mail* informando que seu pedido está pronto.

4.2.6 Controle de Funcionamento da Loja

Quando o vendedor desejar, pode fechar a sua loja através do seletor que se encontra no canto superior direito da sua interface (Figura 15). Ao fazer isso, os demais perfis de usuário perdem acesso às funcionalidades do sistema, sendo redirecionados para uma página de aviso de que o estabelecimento está fechado através de um filtro aplicado aos servlets relacionados a esses perfis.

Figura 15 – Seletor de status da loja



Fonte: Captura de tela realizada pelo autor (2021).

Neste contexto, um filtro é uma classe Java que realiza filtragem de requisições e respostas de recursos do sistema, como servlets, e atua na camada de controle especificamente entre a interface e o controlador (ORACLE, 2011)

Quaisquer usuários que tentem acessar o sistema serão redirecionados pelo filtro de aviso, em que somente está disponível a opção de realizar o *login* como vendedor. Apenas o usuário com esse perfil pode autenticar-se no sistema e alterar o status da loja para aberta.

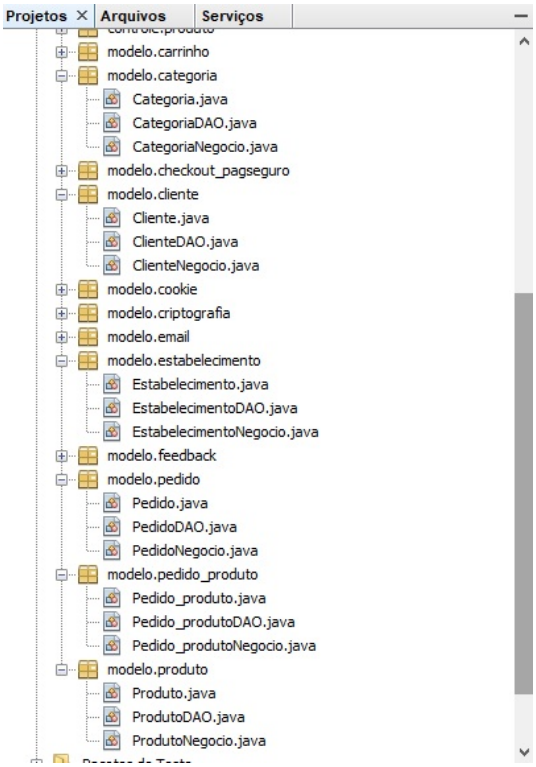
4.3 Arquitetura

O *software* produzido segue o padrão de projeto MVC, já descrito na Seção 2.1. No *back-end* desenvolvido, foi utilizado o modelo de dados descrito na Subseção 4.1.3 para definir a estrutura básica da camada de modelo. Abaixo encontra-se detalhada a organização do código que se integra com a base de dados.

4.3.1 Modelo

Na camada de modelo, encontra-se a lógica de negócio do sistema. No caso das entidades descritas na modelagem de dados, cada entidade foi separada num pacote Java e modularizada em três classes (Figura 16): uma classe do tipo "*Bean*", que representa a própria entidade com seus atributos; uma classe do tipo "*Data Access Object* (DAO)", que representa os acessos aos dados dessa entidade persistidos no banco de dados; uma classe do tipo "Negócio", que encapsula as regras de negócio da entidade.

Figura 16 – Estrutura de Pacotes da Camada de Modelo



Fonte: Captura de tela realizda pelo autor (2021).

No *Bean* existem as representações dos dados das colunas mostradas nas tabelas da Figura 4 mostrada na Seção 4.1.3 em forma de atributos, junto aos seus métodos acessores e modificadores, como já explicado na Seção 2.4.2. O DAO contém métodos específicos que utilizam a tecnologia *Java Database Connectivity* (JDBC) e comandos SQL para acesso ao banco de dados, realizando o CRUD das informações conforme as necessidades da aplicação através do uso dos métodos do *Bean*, retornando um resultado para cada operação. A classe de Negócio possui métodos de mesmo nome correspondentes aos métodos da classe DAO, que instanciam objetos do tipo DAO, utilizam seus respectivos métodos e os resultados retornados por eles. Assim, a classe do tipo "Negócio" funciona como uma interface de acesso ao DAO, desacoplando a lógica de negócio do código que realiza o acesso ao armazenamento persistente de informações, aumentando a flexibilidade e a portabilidade do código, uma vez que as tecnologias de acesso ao SGBD podem ser modificadas sem prejuízo às classes de lógica de negócio, das quais outras camadas da aplicação utilizam métodos para realizar operações na base de dados. Segundo Bezerra (2007), essa divisão entre classes de acesso ao BD e classes de lógica de negócio é chamada de "padrão DAO".

Utilizando a entidade "categoria" como exemplo ilustrativo, na aplicação ela possui

o pacote “modelo.categoria”, contendo as classes “Categoria” (*Bean*), “CategoriaDAO” (DAO) e “CategoriaNegocio” (Negócio).

A classe “Categoria” possui os atributos privados “id”, do tipo *Integer* – representando a coluna “id” da entidade “categoria” –, e “nome”, do tipo String – representando a coluna “nome” da mesma entidade. Na mesma classe encontram-se seus respectivos *getters* e *setters*.

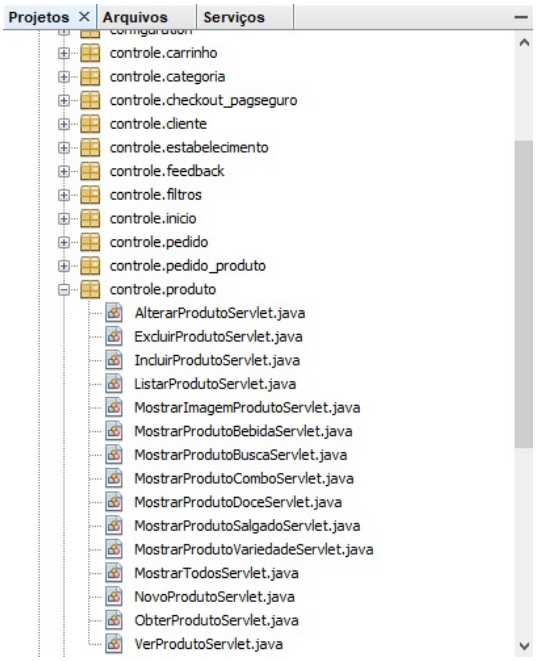
A classe “CategoriaDAO” possui diversos métodos que utilizam a linguagem SQL para realizar procedimentos no BD: “obterNovoId” retorna um novo “id” gerado por autoincremento; “inserir” utiliza o parâmetro “nome” para inserir uma nova categoria no BD com um “id” gerado automaticamente pelo método anterior; “obterTodos” obtém uma lista com todas as categorias cadastradas; “obterCategoria”, por sua vez, utiliza um “id” como parâmetro para consultar as informações de uma categoria específica (no caso, o “nome”); “alterar” realiza a alteração de uma categoria utilizando como parâmetros o “id” da categoria que se deseja modificar e o novo "nome" que se deseja dar a essa categoria; “excluir” apaga uma determina categoria usando como parâmetro o “id” da categoria que se deseja apagar do BD.

A classe “CategoriaNegocio” possui métodos com os mesmos nomes e os mesmos parâmetros, porém utilizando um objeto da classe “CategoriaDAO” internamente para realizar a atividade desejada. Dessa forma, caso um diferente programador, por exemplo, precise realizar uma operação na entidade “categoria” do banco de dados numa camada diferente, basta criar um objeto da classe “CategoriaNegocio” e utilizar o método desejado contido nessa classe, sem a necessidade de maiores conhecimentos sobre JDBC ou SQL. Caso deseje mudar de tecnologia de acesso ao SGBD, também poderá fazê-lo sem modificar a lógica de negócio.

4.3.2 Controle e Visão

A camada de controle da aplicação foi composta essencialmente por servlets, classes Java descritas na Subseção 2.4.1. Cada servlet encapsula o código que representa cada atividade do sistema de receber requisições e gerar respostas, sejam internas ou externas. Os servlets foram modularizados em pacotes de acordo com cada entidade ou funcionalidade (Figura 17), inicializada através de interações do usuário com a camada de visão, composta por JSPs. O servlet também faz a intermediação entre as camadas de visão e modelo para realizar operações no banco de dados a partir da entrada de dados pelo usuário.

Figura 17 – Estrutura de Pacotes da Camada de Controle



Fonte: Captura de tela realizada pelo autor (2021).

Dessa maneira, por exemplo, as atividades de acessar a página de cadastrar um novo produto, cadastrar um produto para venda e alterar um produto já existente na camada de visão são controladas, respectivamente, pelas classes “NovoProdutoServlet”, “IncluirProdutoServlet” e “AlterarProdutoServlet”, localizadas no pacote “controle.produto”.

Considere-se o exemplo prático de um usuário vendedor adicionar um novo produto para venda. Para exibir a página do formulário para que ele possa realizar o cadastro de um produto, é acessada através do método “GET” do HTTP a classe Java “NovoProdutoServlet”, um servlet que gera uma lista de todas as categorias de produtos já cadastradas no BD através do método “obterTodos” de um objeto instanciado da classe “CategoriaNegocio”. A classe “NovoProdutoServlet” é um servlet que redireciona para a página “novoProduto.jsp”, que contém o formulário de cadastro de produto, enviando também a lista de categorias obtida, que é recuperada na camada de visão por meio de código Java na JSP, para que o usuário possa escolher a que categoria pertence o produto que deseja cadastrar, como mostra a Figura 18. Ao preencher o formulário corretamente e clicar no botão “Salvar”, a página “novoProduto.jsp” envia ao servlet os dados do produto através de um método “POST” para o servlet “IncluirProdutoServlet”, que realiza a requisição da operação de cadastrar o produto no BD a partir da entrada de dados do usuário na camada de visão, utilizando o método “inserir” de um objeto instanciado da classe “ProdutoNegocio”, localizada no pacote “modelo.produto”. Os dados são validados no servidor

pelo SGBD, que retorna uma resposta à JSP sobre o sucesso ou falha na inclusão do novo produto.

Figura 18 – Seleção de categorias existentes no cadastro de um produto

Fonte: Captura de tela realizada pelo autor (2021).

4.3.3 Segurança no Lado do Cliente

Como mencionado na Subseção 2.5.3, aplicações *web* por vezes precisam distribuir o armazenamento de dados entre cliente e servidor, especialmente em aplicações voltadas para dispositivos móveis. Nesse contexto, o *cookie* é a tecnologia utilizada para armazenar pequenas quantidades de dados persistentes em arquivos contidos no navegador do cliente a serem recuperadas e interpretadas pelo servidor de aplicação.

No *software* desenvolvido, a criação de um *cookie* foi utilizada para armazenar informações do carrinho de compras do cliente em seu navegador, exemplificadas na Figura 19, de forma a recuperar rapidamente seus dados mesmo que o cliente se desconecte do servidor. Ao utilizar a aplicação, tais informações são constantemente criadas, consultadas, atualizadas ou removidas conforme as necessidades do sistema por meio de servlets. No entanto, dados

armazenados no lado do cliente são vulneráveis, visto que podem ser facilmente interceptados por terceiros. Portanto, fez-se necessário o uso de criptografia para a cifragem das informações contidas no *cookie*, de forma a torná-las ilegíveis por outrem que não sejam o servidor de aplicação.

Para tal foi escolhido o padrão *Advanced Encryption Standard* (AES), o algoritmo de criptografia simétrica mais utilizado (ARAI; OKAZAKI, 2013). O AES de 128 bits, conhecido como AES-128, utiliza uma chave privada de 16 caracteres, ou seja, 128 bits. Tal chave possui 2^{128} combinações possíveis, tornando inviável sua descoberta por força bruta, o que expressa a alta qualidade criptográfica do algoritmo (PADATE; PATEL, 2014).

A cada vez que o servidor de aplicação realiza uma operação no *cookie* do navegador do cliente – como quando um cliente adiciona um produto ao carrinho (Figura 20) –, um servlet descriptografa o conteúdo do *cookie* para ler suas informações, realiza a operação e novamente criptografa os dados para registrá-los de forma cifrada no *cookie*.

Figura 19 – Carrinho de compras

Fonte: Captura de tela realizada pelo autor (2021).

Figura 20 – Adição de produto ao carrinho

Fonte: Captura de tela realizada pelo autor (2021).

Após registrar as novas informações no *cookie*, ainda, adicionam-se a ele as *flags* "secure" e "HttpOnly". A primeira indica que o navegador somente enviará o *cookie* para o servidor se estiver sendo usado um protocolo de comunicação seguro, como, por exemplo, o HTTPS. A segunda indica que o *cookie* não pode ser acessado por *scripts* no lado do cliente, se houver suporte do navegador, o que pode auxiliar a reduzir o risco de ataques do tipo *Cross-Site Scripting* (XSS).

4.3.4 Segurança na Comunicação entre Cliente e Servidor

Para conferir maior segurança na comunicação entre cliente e servidor e possibilitar que o *cookie* descrito na subseção anterior seja lido pelo servidor, fez-se necessário configurar o servidor de aplicação utilizado pela aplicação, o Apache Tomcat, para usar o protocolo HTTPS, que, como explicado na subseção 2.5.2, utiliza criptografia assimétrica para realizar a comunicação entre cliente e servidor.

A implementação do HTTPS usa o protocolo TLS/SSL, que usa certificados digitais com o objetivo de conferir autenticidade ao servidor *web*. *Websites* confiáveis contam com certificados fornecidos por uma Autoridade Certificadora (AC) conhecida e assinados através de chaves criptográficas muito difíceis de se forjar. Tais certificados só são conferidos após uma série de análises que confirmam a autenticidade do solicitante, que deve ter sua aplicação *web* implantada em ambiente de produção. Para efeito de testes, porém, o *Java Development Kit* (JDK) conta com uma ferramenta utilitária denominada "*keytool*", capaz de gerar certificados autoassinados.

O padrão de criptografia utilizado para geração de chaves foi o RSA[15], um algoritmo de criptografia assimétrica. A implementação do protocolo TLS/SSL configurada no servidor de aplicação deu-se através da API *Java Secure Socket Extension* (JSSE). Por fim, foi necessário adicionar uma restrição de segurança no descritor de implantação da aplicação.

4.3.5 Integração com a API do Sistema de Pagamento

A API do PagSeguro utiliza a arquitetura *REpresentational State Transfer* (REST) para se comunicar com sua plataforma de pagamentos. A plataforma conta, ainda, com um *sandbox*, um ambiente de testes que simula o ambiente de produção sem realizar operações financeiras de fato, o que auxilia na realização de testes de integração com uma aplicação *web* (PAGSEGURO, 2019).

No *software* desenvolvido, foi realizada a integração com a API do PagSeguro por meio de *checkout* com redirecionamento, utilizando sua biblioteca de integração para Java e seu ambiente *sandbox*. Assim, ao realizar o *checkout*, a API redireciona a aplicação para a página do PagSeguro, cujo sistema realiza a solicitação de cobrança do cartão de crédito sem que a aplicação ou o vendedor tenham acesso aos dados do cartão do cliente. A API deve passar

[15] Rivest–Shamir–Adleman (RSA), sigla que contém os sobrenomes de Ron Rivest, Adi Shamir e Leonard Adleman, que projetaram o algoritmo.

credenciais da conta PagSeguro do vendedor para que seja possível realizar o pagamento.

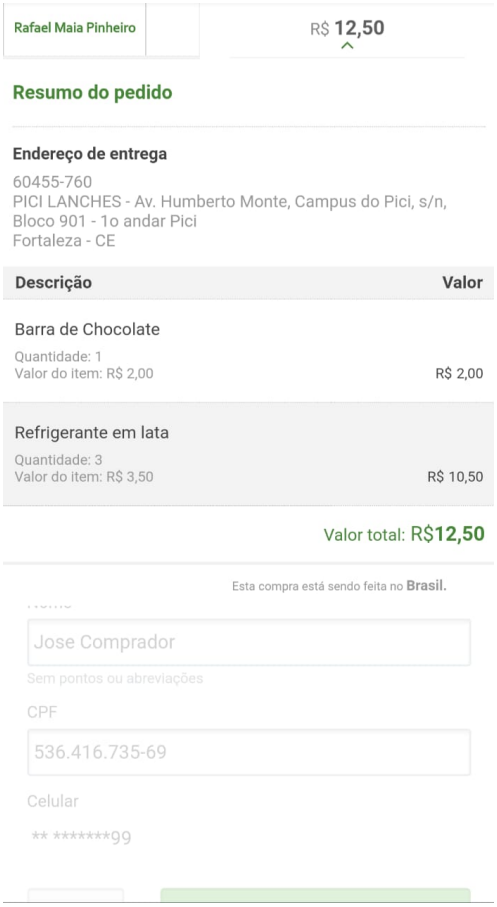
A API do PagSeguro fornece meios de autenticação, utilizando as credenciais do vendedor e o tipo de ambiente a ser usado (produção ou *sandbox*) para realizar o *checkout*, além de uma classe que monta uma instância com os dados da compra e do cliente para serem passados para o sistema do PagSeguro no momento do redirecionamento.

Na aplicação, ao clicar no botão “Finalizar” na página do carrinho de compras, os dados da compra são enviados por meio de um método “POST” para o servlet “PrepararPedidoServlet”, do pacote "controle.pedido", que contém as classes relacionadas à manipulação dos dados do pedido no ambiente interno do sistema. Esse servlet armazena os dados da compra como atributos da sessão do usuário cliente e em seguida direciona para o servlet “CheckoutPagSeguroServlet”, do pacote “controle.checkout_pagseguro”, que contém classes relacionadas com a comunicação do sistema da aplicação com o sistema do PagSeguro. Esse último servlet utiliza a API do PagSeguro para fazer o *checkout*. Nele, as credenciais do vendedor e o tipo de ambiente são passados para autenticação, enquanto os dados da compra e do cliente são passados como parâmetros que serão lidos na plataforma do PagSeguro através de um “POST” para a realização do *checkout*.

Na página do PagSeguro, o cliente pode verificar para quem está sendo feito o pagamento, os dados da compra e o local de retirada do pedido, que deve ser a própria lanchonete (Figura 21). Nessa página, o cliente pode inserir seus dados de cartão de crédito ou, caso tenha uma conta no PagSeguro, utilizar um cartão de crédito já cadastrado. Após o processamento da compra, o PagSeguro exibe uma mensagem de confirmação com os dados da compra e redireciona o usuário de volta à aplicação após um período ou, caso deseje, ele pode clicar no *link* de “voltar para a loja” para voltar imediatamente. O endereço da página à qual o sistema de pagamento do PagSeguro deve retornar é configurada diretamente na conta do vendedor no PagSeguro.

Ao retornar para a página da aplicação, o PagSeguro envia um código de transação identificador único, que é registrado através de um *scriptlet* contendo código Java na JSP através de um “GET”. Essa JSP informa ao cliente para aguardar a aprovação do pagamento, ficando nesse estado até que haja uma definição (Figura 22). Somente após essa definição, a interface sai do estado de espera e exibe uma mensagem que informa ao usuário o resultado do pagamento.

Figura 21 – Dados da compra (nome do autor usado como vendedor)



Fonte: Captura de tela realizada pelo autor (2021).

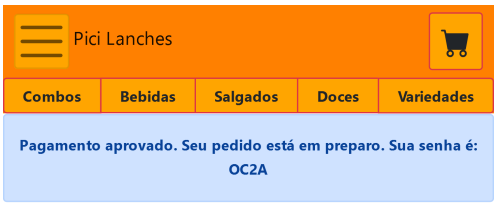
Figura 22 – Tela de espera pela aprovação do pagamento

Fonte: Captura de tela realizada pelo autor (2021).

Para realizar a consulta sobre a aprovação do pagamento, enquanto a JSP permanece mostrando a mensagem de aguardo, o código da transação é capturado e enviado por um "POST" através de AJAX[16] para o servlet "ProcessarPedidoServlet", do pacote "controle.pedido", que recupera os dados da compra através da sessão do cliente e registra o pedido no banco de dados junto com seu código de transação. Após o registro no BD, esse servlet envia o código da transação para o servlet "VerificadorPagamentoAsyncServlet", contido no pacote "controle.checkout_pagseguro". Esse servlet é assíncrono e utiliza o código da transação para fazer consultas assíncronas sobre o estado da transação no servidor do PagSeguro através da sua API por um "GET". As consultas são realizadas a cada 10 segundos, estabelecido um máximo de 6 consultas, ou seja, a espera pela aprovação do pagamento dura no máximo cerca de 60 segundos. Caso retorne que o pagamento foi aprovado antes que o número máximo de consultas seja atingido, o pedido é atualizado como pago e passa a ser visto como pendente pelo usuário

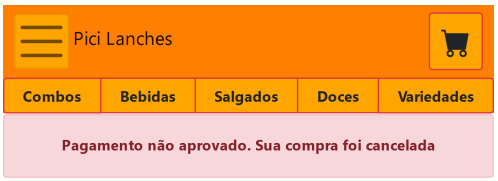
vendedor, que já pode cuidar do seu preparo (Figura 23). Caso contrário, o pedido tem seu status atualizado para “cancelado” (Figura 24).

Figura 23 – Tela de pagamento aprovado



Fonte: Captura de tela realizada pelo autor (2021).

Figura 24 – Tela de pagamento não aprovado



Fonte: Captura de tela realizada pelo autor (2021).

4.3.6 Detalhamento do Carrinho de Compras

O *cookie* do carrinho de compras desempenha um papel fundamental na aplicação, visto que suas informações são lidas e alteradas em diversas classes, inclusive sendo responsável por armazenar informações sobre o pedido que serão recuperadas para envio para o sistema do PagSeguro. Seu código encontra-se detalhado no Apêndice B.

A lógica de negócio resumida do código do carrinho está disposta a seguir, no Código-fonte 1. O Código-fonte 7 no Apêndice B mostra mais detalhes.

Código-fonte 1 – Lógica de negócio do carrinho de compras

```
1 public final class CarrinhoNegocio {
2     ...
3     public static final List<CarrinhoItem> obterCarrinho(
          String valor) {
4         ...
5     }
6
7     public static final String adicionarItem(int produtoId,
          int quantidade, String valor) {
8         ...
```

[16] *Asynchronous JavaScript and XML* (AJAX) é uma técnica para realizar requisições assíncronas no lado do cliente.

```
9       }
10
11      public static final String removerItem(int produtoId,
           String valor) {
12          ...
13      }
14      ...
15  }
```

O Código-fonte 1 acima apresenta a classe estática que encapsula as regras de negócio de manipulação das informações a serem escritas no *cookie* do carrinho em formato de texto. As informações do texto gravado no *cookie* são compostas pelo id e a quantidade de cada produto.

O método “obterCarrinho” (linha 3) retorna uma lista dos itens contidos no carrinho de compras com suas respectivas quantidades, montada ao filtrar as informações de texto registradas no *cookie* divididas por separadores.

O método “adicionarItem” (linha 7) adiciona um item ao carrinho, retornando o novo texto a ser escrito no *cookie* do carrinho, reescrevendo o que havia anteriormente.

O método “removerItem” (linha 11) remove um item completamente do carrinho, ou seja, remove o produto junto com toda a sua quantidade. Esse método também retorna um novo texto a ser adicionado ao *cookie*, que é similar ao anterior, mas sem o item removido.

O pacote “controle.carrinho” contém os servlets responsáveis por acessar e modificar o *cookie* do carrinho de compras. Para tal, é necessário utilizar uma classe estática denominada “AES”, localizada no pacote “modelo.criptografia” e contém os métodos para criptografar e descriptografar o conteúdo do carrinho.

O servlet “MostrarProdutoCarrinhoServlet” é responsável por exibir o carrinho de compras na página do carrinho, enquanto o servlet “AdicionarProdutoCarrinhoServlet” é responsável por adicionar um produto ao carrinho de compras. A classe “RemoverProdutoCarrinhoServlet”, por sua vez, remove um item do carrinho de compras.

Sempre ao fazer uma gravação de dados no carrinho de compras, o valor do *cookie* é recuperado, descriptografado, sobrescrito com o novo valor, recriptografado e, então, gravado no *cookie*. Para realizar uma leitura, o processo consiste apenas em recuperar e descriptografar para

que a informação seja recuperada por algum método.

Para exibir a página do carrinho de compras, por exemplo, o valor do *cookie* é recuperado por meio de *scriptlets* e transformado em uma lista de itens conforme o id e a quantidade gravados de cada produto no valor do *cookie*.

Outras classes não diretamente relacionadas com a exibição das informações do *cookie* também realizam a leitura do seu valor.

A página inicial da aplicação e as páginas que exibem os produtos sempre realizam a leitura do *cookie* para, em caso de não haver um *cookie* do carrinho de compras armazenado no navegador do cliente, armazenar um *cookie* vazio para que seja possível realizar operações nele.

O servlet “PrepararPedidoServlet”, por exemplo, descriptografa esse valor e monta uma lista com os itens do carrinho de compras para enviá-la à API do PagSeguro. Já o servlet “PedidoSucessoServlet”, responsável por encaminhar para a página que mostra ao cliente que seu pagamento foi aprovado, esvazia o carrinho de compras ao ler e apagar os dados do *cookie* antes de criptografá-lo.

5 TESTES E RESULTADOS

Neste capítulo serão apresentados como foram conduzidos os estudos sobre a tomada de decisões sobre o que seria implementado na expansão do trabalho realizado na disciplina de Projeto Integrado II, os testes de *software*, as principais dificuldades encontradas, os resultados e análise dos resultados.

Foram efetuados testes unitários de criptografia para os dados contidos no *cookie* do carrinho de compras e para o protocolo de comunicação cliente-servidor. Em seguida, foram feitos testes de integração com a API de pagamento. Por fim, realizaram-se testes de integração com o *front-end* e um *framework* para criação de um *layout* responsivo.

Devido à atual condição de isolamento social imposta pela pandemia, não foi possível a realização de testes de sistema com potenciais usuários da aplicação.

5.1 Testes Unitários de Segurança

Após a implementação da estrutura básica do sistema em torno das entidades definidas na Subseção 4.1.3, foram realizados testes unitários para incrementar a segurança de alguns elementos do programa antes da integração com o sistema de pagamento.

5.1.1 Proteção de Dados do Cookie

Como já relatado na Subseção 4.3.3, para adicionar segurança ao *cookie* do carrinho de compras, foi utilizado o AES, um criptossistema simétrico.

Como o *cookie* é uma informação que deve ser escrita e consultada somente pelo servidor, o uso de criptografia de chave privada é suficiente para a cifragem de dados. Além disso, por contar com apenas uma chave que não é compartilhada, o processo de encriptação e decriptação acontece de forma mais rápida.

Como mencionado na Subseção 2.5.1, a cifragem de dados computacionais utiliza modelos matemáticos complexos aplicados a algoritmos. Portanto, para realizar a criptografia dos dados do *cookie*, foi utilizado o *framework Java Cryptography Architecture* (JCA), um conjunto de APIs da plataforma Java que fornece uma estrutura para a cifragem de dados, com classes que fornecem métodos para a implementação de algoritmos criptográficos em diferentes modos.

A lógica de negócio de criptografia AES foi encapsulada na camada de modelo

utilizando uma chave privada de 128 bits. A cifragem dos dados foi, então, implementada na camada de controle nos servlets.

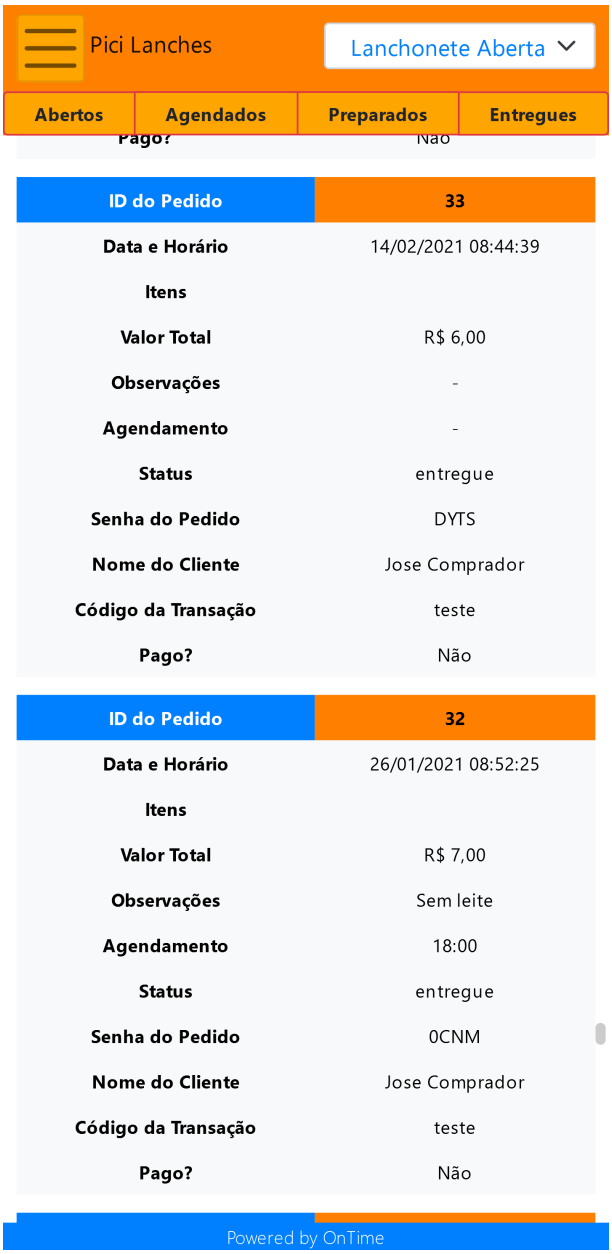
Para a realização dos testes foram utilizados: o NetBeans, para edição do código; o navegador Mozilla Firefox, para visualização da saída de dados na interface da aplicação e verificação de valores registrados no *cookie* através do seu inspetor de armazenamento de *cookies*; o pgAdmin, para análise dos dados inseridos no BD; um repositório no BitBucket[17] para controle de versão.

Mesmo com a constatação através do navegador de que os dados do carrinho de compras registrados no *cookie* haviam sido cifrados com sucesso, levou algum tempo para a compreensão do padrão a ser seguido no algoritmo de gravação e recuperação dos dados do *cookie* no servlet, resultando em erros no registro de alguns dados no BD e sua consequente ausência na interface.

Após sucessivos testes, observou-se que os dados estavam sendo cifrados, mas não estavam sendo decifrados pelo servidor em algumas partes do código, o que resultava na leitura de dados defeituosa (Figura 25). Assim, o código foi corrigido de modo que a cada vez que o valor do *cookie* for recuperado, seu conteúdo é descriptografado, e que antes de gravar um valor no *cookie*, ele é primeiro criptografado e, então, gravado.

[17] BitBucket é uma plataforma que permite a criação de repositórios privados gratuitamente através da tecnologia *open source* Git.

Figura 25 – Dados de itens ausentes no BD devido à falha na leitura dos *cookies*



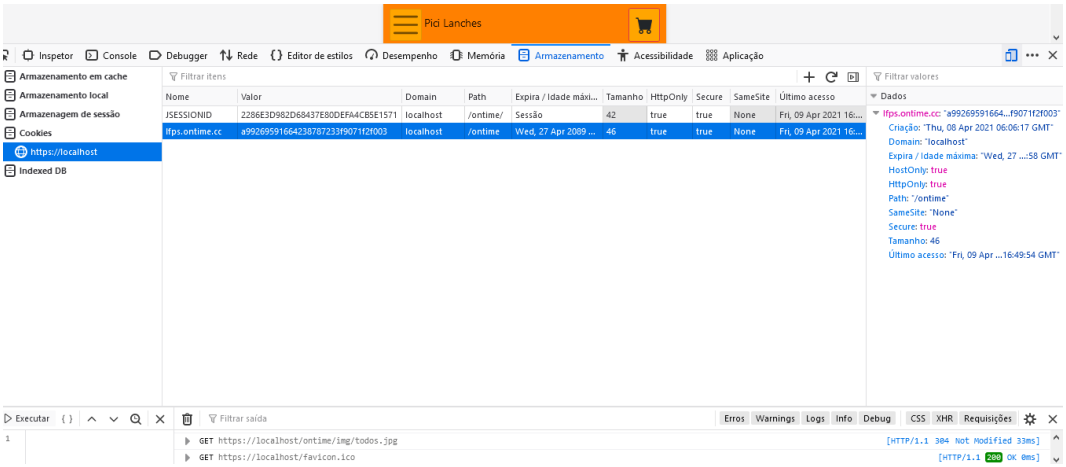
Fonte: Captura de tela realizada pelo autor (2021).

Durante os testes anteriores, observou-se que as *flags* "secure" e "httpOnly" mostravam ambas o valor "*false*" para o *cookie* do carrinho de compras, ao contrário do *cookie* de sessão, de forma que foram feitos testes com métodos da classe "Cookie" para a modificação do valor dessas *flags*, de forma a fornecer mais segurança, conforme citado na Subseção 4.3.3.

De início, foi usado o método "setSecure" da classe "Cookie" para alterar a *flag* para "*true*". Durante os testes, através do navegador percebeu-se, porém, que havia alternância do valor da *flag* conforme o uso da aplicação, de modo que, dependendo da página, o valor da *flag* era diferente.

Ao analisar o código e os resultados dos testes, percebeu-se que o valor da *flag* precisava ser estabelecido como "*true*" a cada vez que o *cookie* fosse manipulado, não bastando sua configuração uma só vez. Para que funcionasse de forma adequada, o método "setSecure" foi utilizado imediatamente antes da gravação do *cookie* no navegador. Alcançado o sucesso em manter a *flag* "secure" constante, o mesmo procedimento foi aplicado ao método "setHttpOnly" para manter sempre ativada a *flag* "httpOnly", logrando, assim, sucesso na inserção das *flags* e na cifragem do *cookie* (Figura 26).

Figura 26 – Valor do *cookie* e seus atributos



Fonte: Captura de tela realizada pelo autor (2021).

5.1.2 Implementação de Protocolo de Comunicação Segura

Para a implementação do HTTPS na aplicação, foram seguidos os passos descritos no manual de configuração do protocolo TLS/SSL do Apache Tomcat 8[18].

Durante os testes foram utilizados: a *keytool* do JDK para manipulação da *keystore*; o *prompt* de comando para a utilização da *keytool*; o NetBeans para edição do código, verificação do console e do uso do servidor de aplicação; o navegador Firefox para verificação da interface e de informações do certificado; os arquivos no formato *eXtensive Markup Language* (XML) dos diretórios do Tomcat para sua configuração; um repositório no BitBucket para controle de versão.

A *keytool*, já citada na Subseção 4.3.3, foi utilizada para gerar um certificado auto-assinado com uma chave privada do servidor usando o criptossistema RSA, recomendado pela Apache, desenvolvedora do Tomcat, por assegurar maior compatibilidade com outros servidores

[18] Apache Tomcat 8: SSL/TLS Configuration HOW-TO – https://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html. Acesso em: 11 jan. 2021.

(APACHE, 2018).

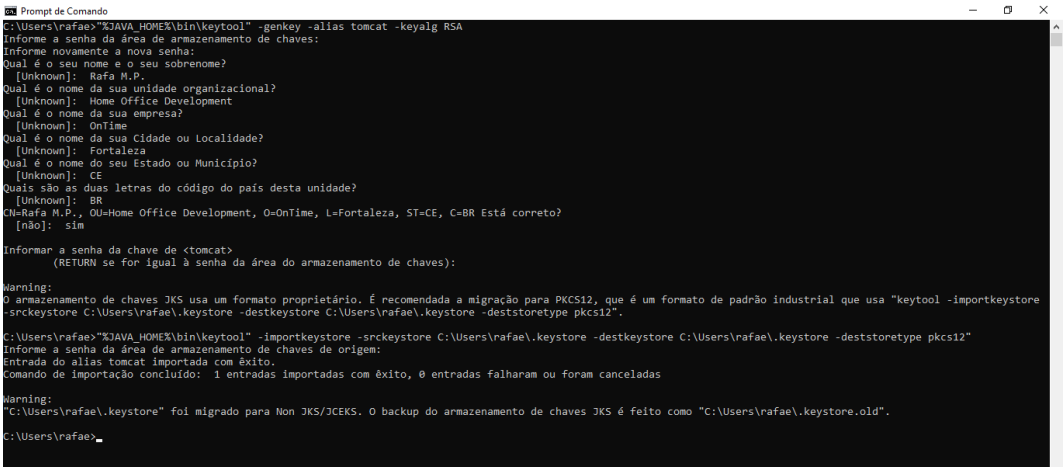
A *keytool* trata-se de uma ferramenta do JDK, localizada no seu diretório de instalação e usada via *prompt* de comando, com o objetivo de gerenciar uma *keystore*. Uma *keystore* é um repositório protegido por senha que armazena dados criptográficos, como chaves e certificados (ORACLE, 2018).

Na geração de um certificado, é necessária a inserção de informações para identificá-lo, como nome e sobrenome, unidade organizacional, nome da empresa e sua localização. Esse certificado terá uma senha individual, configurada no momento de sua criação, que pode ser igual ou diferente da senha da *keystore*.

A *keystore* gerada pela *keytool* do Java 8 utiliza o formato *Java KeyStore* (JKS). Ao criar o certificado, a *keytool* informa ao usuário que a *keystore* utiliza esse formato proprietário e recomenda sua migração para o formato *Public Key Cryptography Standards* (PKCS) #12, um formato padrão da Internet compatível com outras ferramentas de manipulação de *keystores*. A *keytool* também fornece o comando para realizar a migração para o formato PKCS #12.

Sem realizar a migração supracitada, foi conduzido um teste de acesso à *keystore* para atestar que o certificado foi criado. Ao inserir a senha, as informações inseridas no certificado apareceram para alteração, o que indica que o certificado foi criado com sucesso. Foi, então, realizada a migração, e as informações do certificado continuaram legíveis. A Figura 27 mostra todo o processo de criação do certificado autoassinado e a migração da *keystore* para o formato PKCS #12.

Figura 27 – Criação de certificado e migração da *keystore*



Fonte: Captura de tela realizada pelo autor (2021).

Tendo sido bem sucedida a criação do certificado, foi realizada a configuração do

Tomcat para usá-lo. Conforme os passos indicados no manual referido anteriormente, o arquivo de configuração do Tomcat “server.xml” localizado no seu diretório de instalação foi editado, de forma que a implementação do TLS/SSL escolhida foi através da JSSE, por ser de mais simples configuração. passando, entre outros comandos e parâmetros, o caminho do diretório da *keystore* gerada e sua senha. Os efeitos não apareceram, visto que o endereço da página *web* da aplicação continuou a indicar a utilização do HTTP e da porta 8084, padrão do Tomcat 8 para o HTTP, em vez da porta 8443, configurada no Tomcat para o HTTPS.

O manual do Apache Tomcat 8 para implementação do HTTPS, no entanto, não menciona um passo essencial para o uso desse protocolo pela aplicação, que é a adição de uma restrição de segurança ao seu descritor de implantação (“web.xml”), o que pode ser resolvido com a adição do trecho de código abaixo.

Código-fonte 2 – web.xml: trecho de código adicionado ao descritor de implantação da aplicação para o uso do protocolo HTTPS

```
1  ...
2  <security-constraint>
3      <web-resource-collection>
4          <web-resource-name>securedapp</web-resource-name>
5          <url-pattern>/*</url-pattern>
6      </web-resource-collection>
7      <user-data-constraint>
8          <transport-guarantee>CONFIDENTIAL</transport-
               guarantee>
9      </user-data-constraint>
10 </security-constraint>
11 ...
```

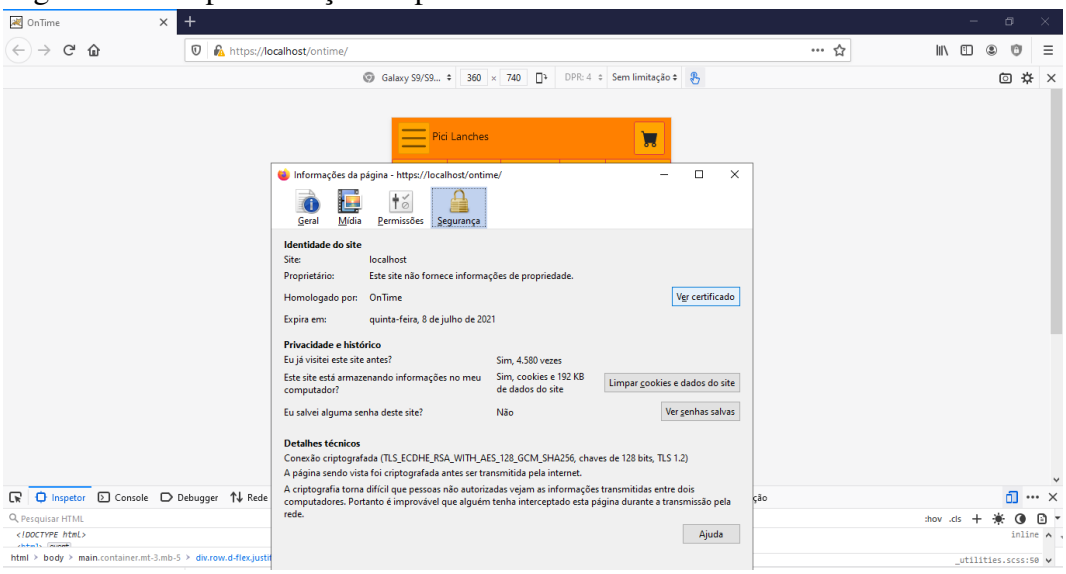
O trecho acima restringe a aplicação a utilizar somente uma conexão segura. Sua adição, porém, fez a aplicação parar de funcionar, o que significava que o Tomcat não estava corretamente configurado para utilizar TLS/SSL.

O manual de configuração de TLS/SSL do Tomcat 8 especificava que as configurações deveriam ser feitas no arquivo “server.xml” do diretório “CATALINA_BASE”. “Catalina” é um *servlet container*, componente que constitui o núcleo do Tomcat. Quando o Tomcat

é instalado numa máquina, a localização da sua instalação é geralmente referida através da variável de sistema “CATALINA_HOME” e contém a configuração geral do Tomcat. Através do CATALINA_HOME, só é possível executar uma instância do Tomcat. Já a variável “CATALINA_BASE” aponta para um diretório onde se localizam configurações específicas das aplicações que utilizam o Tomcat. Através do CATALINA_BASE podem ser executadas múltiplas instâncias do Tomcat. Caso um diretório CATALINA_BASE não seja configurado, sua localização é usada pelo Tomcat como a mesma do diretório CATALINA_HOME. Em outras palavras, caso seja configurado um diretório CATALINA_BASE diferente do CATALINA_HOME, quaisquer aplicações que utilizam o Tomcat usarão prioritariamente as configurações especificadas no CATALINA_BASE, não as do CATALINA_HOME (APACHE, 2007).

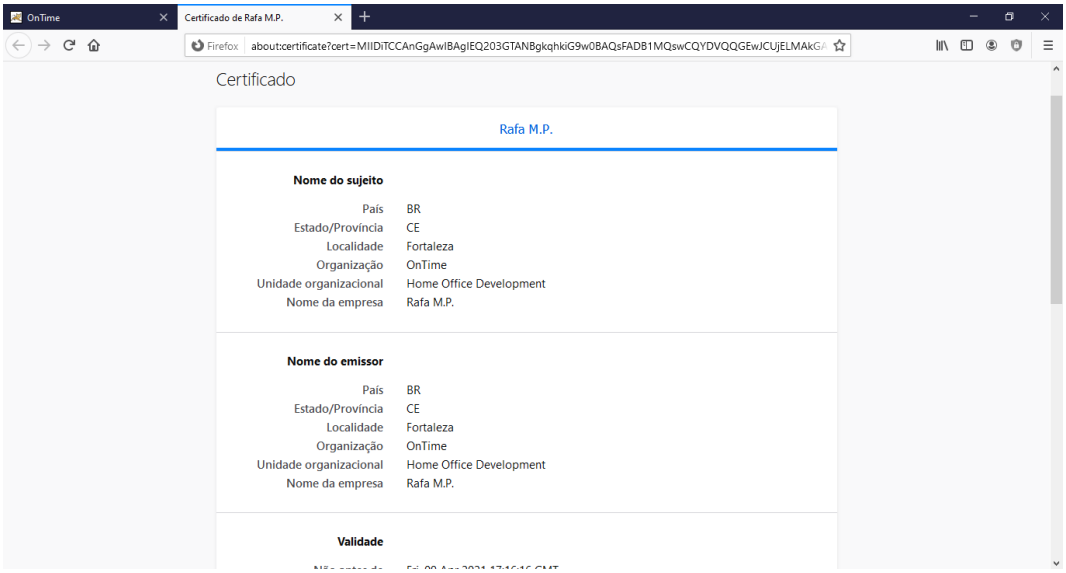
Ao procurar o servidor da aplicação usado no NetBeans, encontrado na opção “Servidores” do menu “Ferramentas”, na aba “Conexão”, descobriu-se que, aparentemente, ao instalar o Tomcat junto ao NetBeans, ele configura automaticamente um diretório para o CATALINA_BASE diferente do diretório de instalação do Tomcat (CATALINA_HOME). No caso do sistema operacional utilizado, o diretório CATALINA_BASE estava numa pasta oculta pelo sistema. Foi importante notar esse detalhe, visto que as configurações de TLS/SSL estavam sendo aplicadas ao CATALINA_HOME, por isso a aplicação permanecia sem mudanças no protocolo HTTP. Ao aplicar as configurações descritas no arquivo “server.xml” do CATALINA_BASE, a aplicação foi executada com o uso do protocolo HTTPS, e as informações inseridas no certificado estavam disponíveis para leitura no navegador, comprovando o sucesso da implementação (Figuras 28 e 29).

Figura 28 – Implementação do protocolo HTTPS



Fonte: Captura de tela realizada pelo autor (2021).

Figura 29 – Certificado autoassinado



Fonte: Captura de tela realizada pelo autor (2021).

5.2 Testes de Integração com a API de Pagamento

Durante o desenvolvimento deste trabalho, foi estudada a viabilidade de uso de APIs de diferentes sistemas que intermediam transações entre consumidor, instituições bancárias e operadoras de cartão de crédito para atender aos objetivos da aplicação. Devido ao tempo limitado para a finalização deste projeto de pesquisa, estabeleceu-se que seria escolhida uma API para integração com o sistema.

5.2.1 Escolha da API

Depois de análises utilizando como critérios popularidade e credibilidade no mercado, disponibilidade de API para aplicações Java, de ambiente *sandbox* e de documentação, três serviços foram selecionados para análise: PayPal[19], PagSeguro [20] e Mercado Pago[21].

O PayPal é um intermediador de pagamento muito usado internacionalmente. Seu uso em estabelecimentos brasileiros, porém, é limitado. Possui um ambiente *sandbox* para testes por desenvolvedores bem completo, possibilitando inclusive a realização de *login* no sistema do PayPal usando nome de usuário e senha fictícios gerados pelo *sandbox*. Sua documentação, porém, é vaga. Sua API faz uso de Maven, uma ferramenta com a qual o autor não tem familiaridade. O PagSeguro é um serviço muito usado para pagamentos *online* no Brasil e conta com máquinas de cartão de crédito para lojas físicas. Possui um ambiente *sandbox* simples se comparado ao do PayPal. Sua documentação é vaga, porém fornece um tutorial de como utilizar a ferramenta *open source* Gradle, sem necessidade de grandes conhecimentos sobre ela, para gerar uma biblioteca no formato *Java ARchive* (JAR) com a API do PagSeguro compilada, junto com outro arquivo JAR com seus códigos-fonte.

O Mercado Pago é outro serviço de intermediação muito usado no Brasil, contando também com máquinas de cartão de crédito para lojas físicas. Possui ambiente *sandbox*, mas seu acesso depende do fornecimento de dados sensíveis que demandam tempo para análise, mesmo já tendo cadastro na plataforma. O mesmo ocorre para sua documentação. Sua API, assim como a do PayPal, faz uso de Maven.

O uso da API do PayPal foi descartado devido à sua baixa adesão entre microempreendedores com estabelecimentos físicos no Brasil. O desconhecimento do autor sobre a tecnologia Maven também foi levado em consideração. Entre PagSeguro e Mercado Pago, pesou também o uso do Maven na API do Mercado Pago, mas pesou principalmente a burocracia para o acesso de desenvolvedores ao ambiente de desenvolvimento e à documentação, uma vez que no PagSeguro, tendo uma conta, é possível ter acesso imediato à documentação e ao ambiente *sandbox*, que é muito importante para a realização de testes de integração, visto que não só testa diferentes cenários de pagamento sem realizar operações financeiras reais, mas porque o servidor do ambiente de produção de serviços de pagamento costuma por segurança bloquear requisições

[19] PayPal Checkout Java SDK v2 – https://github.com/paypal/Checkout-Java-SDK. Acesso em: 5 fev. 2021.
[20] Biblioteca de Integração PagSeguro para Java – https://github.com/pagseguro/pagseguro-sdk-java. Acesso em: 27 jan. 2021.
[21] Mercado Pago SDK for Java – https://github.com/mercadopago/sdk-java. Acesso em: 25 jan. 2021.

similares demasiadas de uma mesma origem, inviabilizando os testes.

A escolha pela API do PagSeguro é reforçada pelo fato de o cliente de teste utilizado na disciplina de Projeto Integrado II contar com uma máquina de cartão de crédito do PagSeguro em seu estabelecimento, o que torna mais simples a junção desse, assim como outros que também a utilizam, com o *software* desenvolvido, uma vez que usuários da máquina precisam possuir conta no PagSeguro e estão mais familiarizados com a plataforma.

5.2.2 Testes de Integração com o Sistema de Pagamento

Além da documentação, o estudo sobre a integração do sistema com a API do PagSeguro deu-se também através da análise do código constante nos arquivos de extensão JAR gerados e das classes de exemplo contidas no SDK e do uso de seu ambiente *sandbox* para testes.

O PagSeguro fornece, entre outras, três modalidades principais de integração[22]: o Checkout PagSeguro, o Checkout Transparente e o Split de Pagamentos. O primeiro utiliza redirecionamento para a página do PagSeguro no momento do *checkout*, com *front-end* pré-definido, possibilitando ao cliente a inserção dos dados do cartão diretamente na página do PagSeguro, externa à aplicação, com redirecionamento de volta à página do vendedor após o processamento do pagamento. O segundo permite ao desenvolvedor realizar o *checkout* no *front-end* da própria aplicação, inserindo nela os dados do cartão de crédito, criptografados pela API do PagSeguro. O terceiro é uma variação do segundo, porém com opção de dividir o pagamento entre mais de um vendedor.

O método escolhido foi o Checkout PagSeguro, por deixar explícito ao cliente que a transação está sendo realizada num ambiente diferente do ambiente da aplicação, com a credibilidade da segurança do serviço do PagSeguro, não inserindo nenhum dado sensível no *front-end* da aplicação.

Durante os testes de integração foram utilizados: o NetBeans para edição do código, verificação do console e estudo do código-fonte da API; o navegador Firefox, para observação das interfaces *front-end* da aplicação e do PagSeguro, além do uso do seu console *web* para verificação de mensagens de sucesso e de erro; o ambiente *sandbox* do PagSeguro para uso de dados de testes, configuração da página de redirecionamento, análise de requisições ao servidor do *sandbox* e modificação de status das transações; um gerador de números de CPF válidos para

[22] PagSeguro Developers: Guia Rápido – https://dev.pagseguro.uol.com.br/page/guiarápido-pagseguro. Acesso em: Acesso: em 27 jan. 2021.

desenvolvedores[23]; repositório no BitBucket para controle de versão.

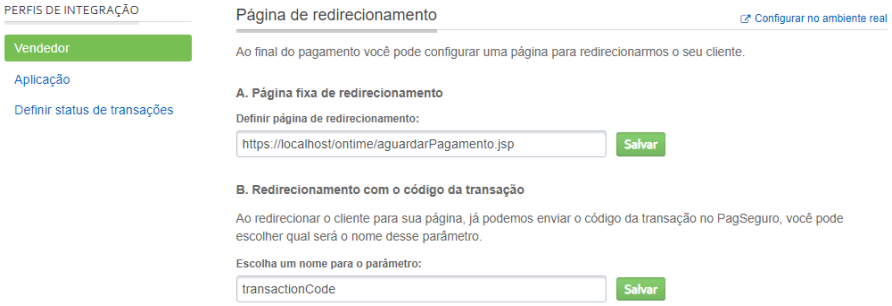
Após análise dos exemplos e do código-fonte da API, o código para *checkout* foi implementado num servlet, recuperando informações do pedido, do vendedor e do cliente para serem passadas para o PagSeguro, como já explicado na Subseção 4.3.5 e detalhado no Apêndice C.

O maior desafio do método de *checkout* por redirecionamento escolhido é a recuperação dos dados inseridos pelo usuário após a finalização do *checkout* num servidor externo. Para tanto, foram utilizados atributos de sessão de usuário para armazenar essas informações e recuperá-las após o retorno do redirecionamento, registrando-as no banco de dados junto com informações enviadas pelo servidor.

Após a finalização do *checkout*, o cliente é redirecionado de volta à aplicação por meio de um endereço configurado na própria página do *sandbox*[24].

É através do *sandbox* que se configura não apenas o endereço de redirecionamento, mas também o código da transação que a identifica no PagSeguro para que seja recebido pela aplicação *web* (Figura 30). Também é possível utilizar um comprador de teste, com *e-mail*, senha e cartão de crédito fictício para testar a compra no redirecionamento da aplicação.

Figura 30 – Configurações no *sandbox*

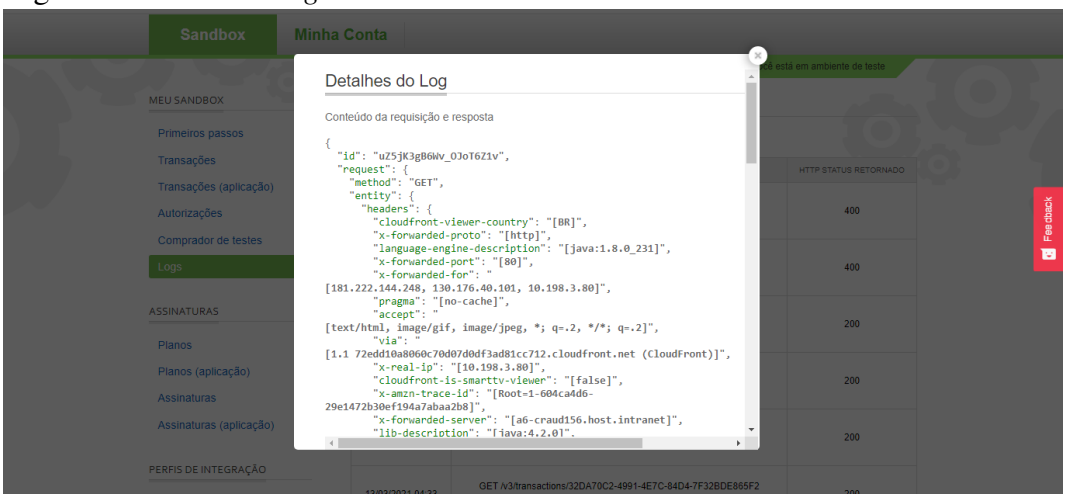


Fonte: Captura de tela realizada pelo autor (2021).

Outra funcionalidade útil é a lista de *logs* (Figura 31) das requisições, em que se pode visualizar o conteúdo das requisições enviadas pela aplicação e a resposta gerada, o que auxilia na resolução de problemas.

[23] Gerador de CPF – https://www.geradorcpf.com/. Acesso: em 27 jan. 2021.
[24] PagSeguro Sandbox – https://acesso.pagseguro.uol.com.br/sandbox. Acesso em: 27 jan. 2021.

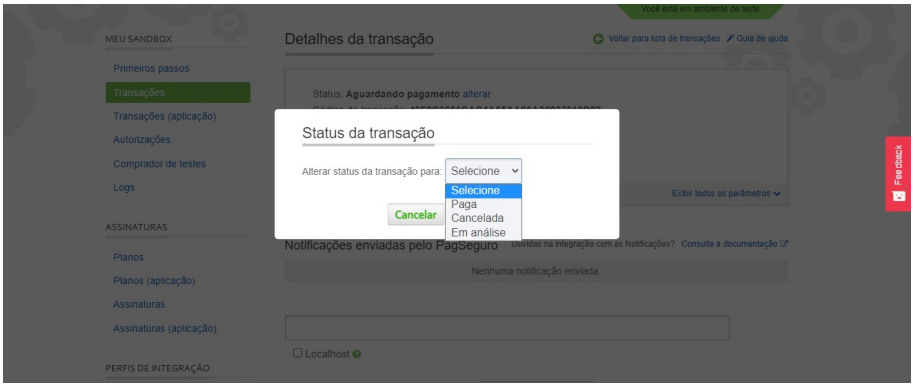
Figura 31 – Lista de *logs*



Fonte: Captura de tela realizada pelo autor (2021).

Também é através da plataforma *sandbox* que se pode ver a lista de transações, visualizar seus detalhes e modificar seu status para “paga” ou “cancelada”, por exemplo (Figura 32).

Figura 32 – Modificando o status da transação no ambiente *sandbox*



Fonte: Captura de tela realizada pelo autor (2021).

Para saber se uma compra foi paga, é preciso que a aplicação tenha um modo de fazer consultas sobre o status da transação no servidor. Isso pode ser feito de duas formas: através da configuração do *sandbox* para o envio de notificações para uma página da aplicação quando o status é alterado ou por meio de consultas enviadas pela aplicação para o servidor do PagSeguro.

A forma escolhida foi o envio de consultas, por conta das falhas do servidor do *sandbox*, que apresenta lentidão ou cai por alguns momentos, o que pode comprometer o envio de notificações. Se uma notificação não for recebida pela aplicação, ela é enviada de novo apenas após 2 horas, com um número máximo de 5 notificações. Como 2 horas para a confirmação de um pedido não são um tempo razoável para testes ou mesmo para os objetivos da

aplicação, o envio de consultas regulares pela aplicação mostrou-se uma opção mais adequada. Na sua documentação[25], o próprio PagSeguro recomenda que, caso seja utilizada a consulta por notificação, sejam implementadas também consultas realizadas pela aplicação.

Para realizar consultas foi utilizado um laço de repetição num servlet com o seguinte trecho de código.

Código-fonte 3 – Trecho de código que faz a consulta se a transação foi paga em intervalos regulares

```
...
boolean pagamentoAprovado = false;
int statusId = 0;
int maxReq = 6;
int sleeptime = 10 * 1000;
int i = 0;
Object obj = new Object();
try {
    synchronized (obj) {
        while (statusId != 3 && i < maxReq) {
            final PagSeguro pagSeguro = PagSeguro.instance(
               new SimpleLoggerFactory(), new JSEHttpClient
               (),
                            Credential.sellerCredential(
                               SELLER_EMAIL, SELLER_TOKEN),
                                ENV);
            TransactionDetail transaction = pagSeguro.
               transactions().search().byCode(
               transactionCode);
            statusId = transaction.getStatus().getStatusId
               ();
            i++;
            obj.wait(sleeptime);
```

[25] PagSeguro Developers: Notificações – https://dev.pagseguro.uol.com.br/docs/api-notificacao-v1. Acesso em: 6 abr. 2021.

```
17          }
18      }
19  } catch (InterruptedException ex) {
20      ex.printStackTrace();
21  }
22  PedidoNegocio pedidoNegocio = new PedidoNegocio();
23  if (statusId == 3) {
24      pagamentoAprovado = true;
25  ...
```

O trecho mostrado no Código-fonte 3 declara uma *flag* que indica se o pagamento foi aprovado (linha 2), um atributo com o id do status da transação (linha 3), o número máximo de requisições a serem feitas antes de registrar a transação como não paga (linha 4) e o intervalo em milissegundos entre as consultas (linha 5).

Das linhas 8 a 17 o laço de repetição estabelece que serão feitas consultas ao servidor do PagSeguro (linha 11) em intervalos regulares de 10 segundos (linha 16) através do código da transação (linha 13) enquanto o id do status da transação não for igual a 3, que é o número inteiro retornado pela API do PagSeguro para indicar que a transação foi paga, ou até que o número máximo de requisições (6) seja atingido.

Se o id do status da transação for igual a 3 ao término do laço (linha 23), significa que a transação foi paga (linha 24), e a *flag* será registrada no BD. Caso contrário, o pedido tem seu status alterado para cancelado.

O problema da utilização desse código em um servlet comum é que o usuário fica preso à página do PagSeguro, e nenhuma resposta sobre o que está acontecendo é mostrada. Para resolver, foi utilizada uma JSP que exibe uma página de espera para o usuário, captura o código da transação recebido pelo servidor do PagSeguro por meio de um *scriptlet* e o envia através de uma chamada AJAX com o uso de jQuery[26] para um servlet que registra o pedido no BD e envia seus dados para um servlet assíncrono para consulta, que utiliza uma instância da classe "VerificadorPagamentoAsyncRequestProcessor", localizada na camada de modelo, no pacote "modelo.checkout_pagseguro", para encapsular a lógica de negócio para consultas, contendo o trecho mostrado no Código-fonte 3.

[26] jQuery é uma biblioteca JavaScript *open source* para a manipulação de eventos e chamadas AJAX em documentos HTML.

O número máximo de requisições foi estabelecido para 6 e o intervalo entre elas para 10 segundos porque, ao enviar muitas requisições, o servidor do *sandbox* pode bloquear o endereço IP do usuário por um dia – mais especificamente, após 1000 requisições. Como o status do pagamento por cartão de crédito pode ser modificado quase instantaneamente, 6 requisições são suficientes para captar a mudança de status da transação.

Para que o Tomcat não apresente problemas ao utilizar requisições assíncronas, é precisa configurar o filtro *HTTP Server-Side Monitor* utilizado por padrão na sua configuração para aceitá-las ou desativá-lo. Como tal filtro auxilia a analisar problemas no fluxo de dados nos servlets e JSPs, escolheu-se o primeiro método, através do seguinte trecho de código do arquivo de configuração "web.xml" localizado no diretório do CATALINA_BASE.

Código-fonte 4 – Trecho do código de configuração do arquivo web.xml do diretório CATALINA_BASE

```
1  ...
2  <filter>
3      <filter-name>HTTPMonitorFilter</filter-name>
4      <filter-class>org.netbeans.modules.web.monitor.server.
           MonitorFilter</filter-class>
5      <async-supported>true</async-supported>
6      <init-param>
7        <param-name>netbeans.monitor.ide</param-name>
8        <param-value>127.0.0.1:8082</param-value>
9      </init-param>
10 </filter>
11 ...
```

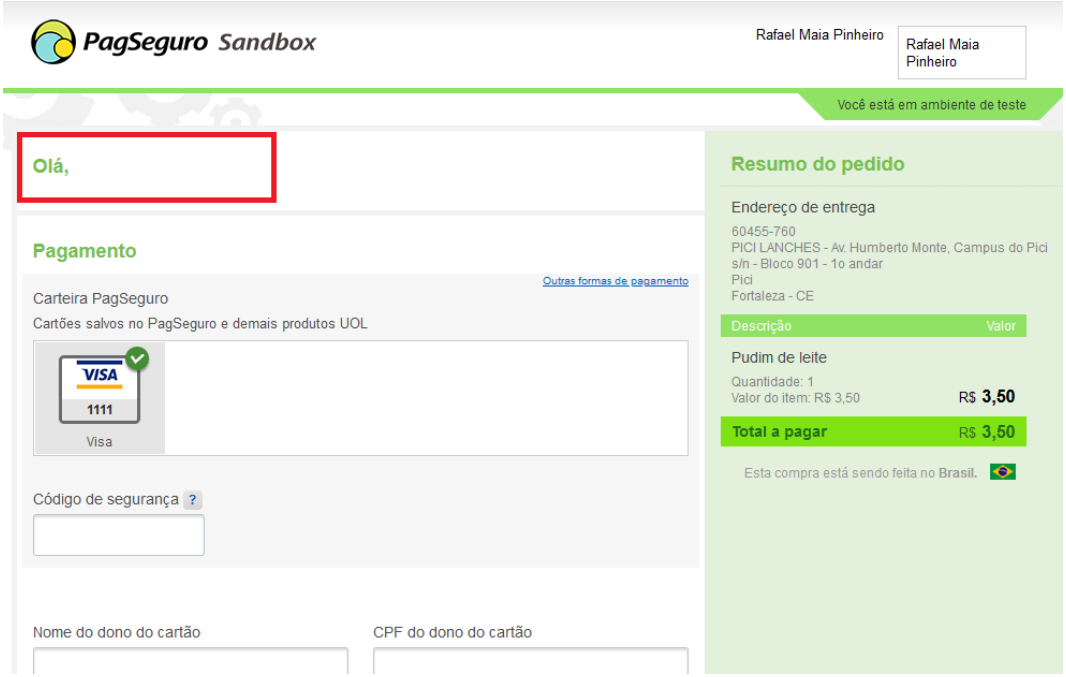
No Código-fonte 4, a linha 5 foi acrescentada para habilitar requisições assíncronas.

Apesar de ser uma funcionalidade muito útil, a lista de *logs* de requisições do *sandbox* tem um limite de 200, não registrando mais novas requisições depois disso, o que depois de algumas semanas testando a aplicação causou problemas para identificar o motivo de algumas falhas.

Outra limitação do ambiente *sandbox* do PagSeguro é que, apesar de permitir a criação de vários vendedores de teste, para o caso de a aplicação servir a mais de um vendedor,

não é possível criar mais de um comprador de teste como ambiente *sandbox* do PayPal, e as informações disponíveis da conta desse comprador de testes são limitadas. Ele não possui nome cadastrado (Figura 33) nem é possível ver seu número de telefone ou CPF, embora seja possível descobri-los através dos *logs* das transações realizadas, diferentes dos dados inseridos na página do PagSeguro (Figura 34). Também não é possível sobrescrever suas informações com as informações do cliente da aplicação no pagamento via *sandbox*, o que impediu de verificar se as informações citadas no Apêndice C além do *e-mail* foram enviadas corretamente.

Figura 33 – Nome inexistente da conta de teste



Fonte: Captura de tela com marcação realizada pelo autor (2021).

Figura 34 – Telefone e CPF da conta de testes

Fonte: Captura de tela com marcação realizada pelo autor (2021).

Através do *sandbox* o fluxo de *checkout* para um usuário não cadastrado no PagSeguro apresenta uma mensagem de falha após o processamento do pagamento, não seguindo para a página que redireciona o usuário de volta à aplicação. Entretanto, é possível para um usuário sem conta no PagSeguro realizar um pagamento[27]. Além disso, através do extrato de transações, é possível verificar que as compras de teste realizadas sem cadastro foram realizadas com sucesso, pois não mostram como nome o *e-mail* da conta de testes, que é utilizado para autenticação, como revelado na Figura 35.

Figura 35 – Extrato de transações com usuários sem conta

DATA	TIPO	DE / PARA	NOME	STATUS	VALOR (R$)	FRETE E ENVIO
07/04/2021 11:10	Pagamento	De	c828730991572058168l5@sandbox.pagseguro.com.br	Aprovada	3,50	Envio Fácil indisponível
07/04/2021 11:03	**Pagamento**	**De**	**Jose Comprador**	**Aguardando Pgto.**	**3,50**	
04/04/2021 12:02	**Pagamento**	**De**	**Joao Sem Conta**	**Aguardando Pgto.**	**38,00**	
04/04/2021 11:57	**Pagamento**	**De**	**Joao Novo Comprador**	**Aguardando Pgto.**	**38,00**	
04/04/2021 11:55	**Pagamento**	**De**	**Jose Comprador**	**Aguardando Pgto.**	**38,00**	
04/04/2021 11:43	**Pagamento**	**De**	**Jose Comprador**	**Aguardando Pgto.**	**38,00**	
30/03/2021 23:48	Pagamento	De	c828730991572058168l5@sandbox.pagseguro.com.br	Aprovada	15,50	Envio Fácil indisponível
29/03/2021 14:18	Pagamento	De	c828730991572058168l5@sandbox.pagseguro.com.br	Aprovada	3,50	Envio Fácil indisponível

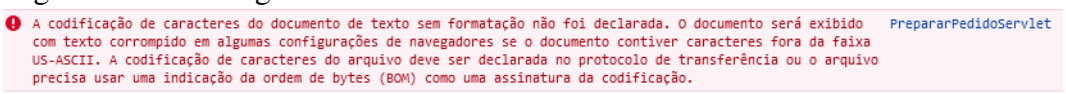
Fonte: Captura de tela com marcação realizada pelo autor (2021).

O extrato de transações, apesar de ser um recurso útil do *sandbox*, raramente esteve disponível. Durante todo o período de testes, poucas vezes foi possível visualizá-lo, visto que o servidor ou realizava o *logout* do sistema ao tentar acessá-lo ou apresentava uma demora muito grande em carregar a página. Portanto, na prática, não foi utilizado em testes, servindo somente para constatar o que foi afirmado no parágrafo anterior.

Por fim, um problema que persiste sem solução é o fato de a página do PagSeguro não exibir caracteres especiais corretamente. Algumas vezes seu sistema nem mesmo inicia o processo de *checkout*, mostrando a mensagem de erro mostrada na Figura 36 no console do Firefox.

[27] PagSeguro FAQ – https://faq.pagseguro.uol.com.br/duvida/para-usar-os-servicos-preciso-me-cadastrar-no-pagseguro/80#rmcl. Acesso em: 5 abr. 2021.

Figura 36 – Mensagem de erro



Fonte: Captura de tela realizada pelo autor (2021).

Apesar de ser a solução mais comum para esse problema encontrada nas pesquisas, todas as JSPs já possuem a *tag* "meta" com a codificação definida para "UTF-8". Todas as requisições e respostas dos servlets também foram configuradas para o uso do formato UTF-8. A página do PagSeguro utiliza o formato "ISO-8859-1". Porém modificar a codificação na aplicação não resolve o erro, cuja causa permanece desconhecida.

5.3 Testes de Integração com o *Front-End*

Para o cumprimento dos requisitos RNF02 e RNF04, apresentados na Subseção 4.1.1, com o objetivo de realizar uma adequada apresentação dos dados do *back-end* em telefones celulares, foi desenvolvido um *front-end* básico utilizando JSPs e o Bootstrap, um *framework open source* para o desenvolvimento de *layouts* responsivos, além de um código JavaScript básico para o auxílio na validação dos dados de entrada.

Para estes testes foram utilizados: o NetBeans para edição de códigos HTML, CSS[28] e JavaScript; o navegador Firefox para verificação dos resultados das mudanças no *layout* através do modo de *design* responsivo, da validação da entrada de dados e análise do *Cascading Style Sheets* (CSS) por meio do seu inspetor de estilos; um *smartphone* para a verificação da correta adaptação do *layout*; repositório no BitBucket para controle de versão.

A importação do Bootstrap fornece às páginas CSS e JavaScript pré-compilados, de forma que o uso de certos valores de classes e identificadores fornecem *layouts* e comportamentos que dispensam maiores preocupações em formatações de estilo e codificação. Ainda assim, para customização conforme as necessidades da aplicação, foi necessária a realização de testes e o acréscimo de código CSS e JavaScript em arquivos separados, além de incremento.

A maior dificuldade encontrada foi a adaptação de tamanhos para diferentes dispositivos móveis. Através da codificação CSS acrescentada e de testes com o modo de *design* responsivo e com o *smartphone*, foi possível otimizar as páginas usadas pela aplicação para um *layout* de *smartphone* vertical. Não houve tempo para ajustes para uso horizontal ou para *tablets*.

O HTML, por sua vez, foi modificado para melhor utilizar as propriedades do

[28] CSS é uma linguagem comumente utilizada para estilizar páginas em HTML.

Bootstrap e, ao mesmo tempo, complementar o JavaScript na validação da entrada de dados. O JavaScript utilizado contou com simples funções de validação de tipos de entrada de dados em campos de formulários em que não foi possível a realização de validação por meio do HTML. Foi utilizado, ainda, o jQuery maskMoney, um *plugin* de código aberto que utiliza a biblioteca jQuery para formatar a entrada de dados monetários.

5.4 Análise de Resultados

Dos requisitos listados na Subseção 4.1.1, este capítulo cobriu:

- RNF01 na Seção 5.1;
- RF04 na Seção 5.2;
- RNF02 e RNF04 na Seção 5.3.

Na Subseção 5.1.1, foi descrito o processo de encriptação dos dados armazenados no *cookie* do carrinho de compras, no qual se logrou sucesso ao aplicar criptografia simétrica AES. Embora as informações armazenadas nesse *cookie* não sejam tão sensíveis na atual versão do sistema, a implementação desse tipo de segurança é importante para a escalabilidade do sistema, uma vez que o a informações sobre produtos comprados podem futuramente auxiliar a aplicação a traçar um perfil de preferências dos consumidores para auxiliá-los na procura de produtos que sejam do seu interesse. A utilização dessas informações por terceiros é potencialmente danosa à privacidade dos usuários.

Na Subseção 5.1.2, apresentou-se o processo de aplicação do protocolo TLS/glsSSL através de criptografia assimétrica RSA e geração de certificado autoassinado. Através da configuração do servidor de aplicação e do descritor de implantação do sistema, foi possível implementar o protocolo HTTPS. É relevante salientar que num ambiente de produção esse protocolo depende da assinatura digital de uma autoridade certificadora confiável para autenticidade da segurança do *website*.

Na Subseção 5.2.2, relatou-se a integração do sistema com a API, realizada de forma satisfatória, visto que o *checkout* no ambiente *sandbox* foi finalizado na maior parte dos testes. Conforme o relatado, o *sandbox* não realiza o fluxo completo de pagamento de um usuário não cadastrado, mas a transação é registrada, o que atesta o êxito na comunicação. A consulta sobre a aprovação do pagamento também é realizada sem problemas. Em algumas tentativas, no entanto, a comunicação com o servidor do PagSeguro apresenta um *bug* que exibe uma mensagem de erro de codificação. Após tentativas sem sucesso de solução, o problema persiste. Algumas medidas

preventivas para esse tipo de falha já haviam sido aplicadas ao código antes da identificação do *bug*. Sua causa é desconhecida. A possibilidade de a falha estar na API não é descartada. Futuras integrações com outras APIs de pagamento podem auxiliar na identificação da origem do problema.

Por último, na Seção 5.3 descreveram-se os testes de integração do *front-end* com o Bootstrap para otimização para uso em *smartphones*, além da aplicação de validação de dados através de HTML e JavaScript. Devido à limitação de tempo, foi possível otimizar o sistema somente para uso em celulares na vertical, o que se mostra suficiente para a maior parte dos usos pretendidos para a aplicação. São necessários, porém, posteriores trabalhos para otimização em uso horizontal e em outros dispositivos de forma a ampliar as formas de acesso à aplicação. A validação dos dados aplicada, por sua vez, não teve maiores problemas e contou com funcionalidades do Bootstrap para apresentação de *feedback* sobre a entrada de dados.

6 CONCLUSÃO

Neste capítulo serão apresentadas as considerações finais com uma síntese dos resultados, analisando o cumprimento dos objetivos do projeto, e sugestões para trabalhos futuros.

6.1 Considerações Finais

Este trabalho relatou o desenvolvimento do *back-end* de uma aplicação *web* para a compra e venda de alimentos em dispositivos móveis. O sistema utiliza tecnologias de código aberto para o cumprimento desse objetivo, sendo capaz de realizar o CRUD de produtos e o gerenciamento de pedidos através de um *front-end* otimizado para uso vertical em *smartphones*.

A arquitetura do sistema faz uso de modularização de código através do padrão de *software* MVC e das tecnologias JavaBeans, servlet e JSP. Tal característica facilita a manutenção, implementação de melhorias, inserção de novas funcionalidades e "departamentalização" de *bugs*.

O *software* possui suporte ao protocolo HTTPS para comunicação segura entre cliente e servidor, além de utilizar a tecnologia *web cookie* com dados criptografados para serem lidos somente por meio de um protocolo de comunicação seguro.

O sistema conta também com integração à API de pagamento do PagSeguro, um intermediador conhecido no mercado. Por meio dela, foi possível completar o processo de pagamento através de *checkout* com redirecionamento em ambiente de testes na maior parte das tentativas.

Destaca-se que um *bug* foi identificado em algumas tentativas de comunicação com o servidor do sistema de pagamento. A falha aponta para um erro na codificação da página, porém a informação não se mostrou suficiente para a identificação da causa do problema, que pode ter origem na API, uma vez que diferentes medidas para sua solução foram tomadas sem mudança nos resultados.

6.2 Trabalhos Futuros

Apresentam-se a seguir sugestões de trabalhos posteriores:

- A realização de testes de sistema com usuários em *smartphones*, bem como a aplicação de entrevistas e pesquisas de satisfação para a validação dos requisitos,

da qualidade do sistema e da aceitação entre potenciais utilizadores;

- Ajuste de responsividade para demais dispositivos móveis;
- Maior utilização de AJAX nas páginas para diminuir o volume de transferência de dados entre cliente e servidor;
- Integração do sistema com outras APIs de pagamento de serviços intermediadores confiáveis;
- Implantação da aplicação num ambiente de produção.

Para escalabilidade do sistema, sugere-se, ademais, a sua expansão para atender a mais de um estabelecimento, no formato *marketplace*, de forma a inserir novas funcionalidades e ampliar sua utilidade.

REFERÊNCIAS

ABREU, J. d. S. Passado, presente e futuro da criptografia forte: desenvolvimento tecnológico e regulação. **Revista Brasileira de Políticas Públicas**, Brasília, v. 7, n. 3, p. 24–42, dez. 2017. Disponível em: <https://www.publicacoes.uniceub.br/RBPP/article/view/4869/3658>. Acesso em: 27 ago. 2020.

APACHE. **Running The Tomcat 4.1 Servlet/JSP Container**. Wilmington, 2007. Disponível em: <http://tomcat.apache.org/tomcat-4.1-doc/RUNNING.txt>. Acesso em: 7 abr. 2021.

APACHE. **SSL/TLS Configuration HOW-TO**. Wilmington, 2018. Disponível em: <https://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html>. Acesso em: 11 jan. 2021.

ARAI, K.; OKAZAKI, H. Formalization of the advanced encryption standard. part i. **Formalized Mathematics**, Warsaw, v. 21, n. 3, p. 171–184, set. 2013. Disponível em: <https://sciendo.com/issue/forma/21/3>. Acesso em: 2 abr. 2021.

BELANGER, F.; HILLER, J. S.; SMITH, W. J. Trustworthiness in electronic commerce: the role of privacy, security, and site attributes. **The Journal of Strategic Information Systems**, Amsterdam, v. 11, n. 3-4, p. 245–270, dez. 2002. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.6616&rep=rep1&type=pdf>. Acesso em: 27 ago. 2020.

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2. ed. Rio de Janeiro: Campus/Elsevier, 2007.

CODD, E. F. **The Relational Model for Database Management**: version 2. Boston: Addison-Wesley, 1986.

COELHO, T.; PRADO, G. Análise comparativa para avaliação de tecnologias de banco de dados para dispositivos móveis. **Caderno de Estudos em Sistemas de Informação**, Juiz de Fora, v. 1, n. 1, p. 1–22, 2014. Disponível em: <https://seer.cesjf.br/index.php/cesi/article/view/128/48>. Acesso em: 27 ago. 2020.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. 8. ed. Rio de Janeiro: Campus/Elsevier, 2004.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de Banco de Dados**. 6. ed. São Paulo: Pearson, 2011.

HAERDER, T.; REUTER, A. Principles of transaction-oriented database recovery. **ACM Computing Surveys**, New York, v. 15, n. 4, p. 287–317, dez. 1983.

KUROSE, J. F.; ROSS, K. W. **Redes de Computadores e a Internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson, 2013.

LÓPEZ, C. A. Cómo mantener el patrón modelo vista controlador en una aplicación orientada a la web. **Inventum**, Bogotá, v. 4, n. 7, p. 72–78, jul./dez. 2009.

MANDLE, A. K.; NAMDEO, V. Role of encryption in e-commerce. **International Journal of Scientific & Technology Research**, New Delhi, v. 8, n. 10, p. 2881–2883, out. 2019. Disponível em: <http://www.ijstr.org/final-print/oct2019/Role-Of-Encryption-In-E-commerce.pdf>. Acesso em: 27 ago. 2020.

MOREIRA, R. A. O comércio eletrônico, os métodos de pagamento e os mecanismos de segurança. **Revista FATEC Zona Sul**, São Paulo, v. 3, n. 1, p. 16–30, out. 2016. Disponível em: <http://www.revistarefas.com.br/index.php/RevFATECZS/article/view/67/93>. Acesso em: 27 ago. 2020.

MURPHY, A.; MURPHY, D. The role of cryptography in security for electronic commerce. **The ITB Journal**, Dublin, v. 2, n. 1, p. 21–50, mai. 2001.

ORACLE. **The Java EE 5 Tutorial: JavaServer Pages technology**. Redwood City, 2010. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html>. Acesso em: 28 ago. 2020.

ORACLE. **Interface Filter**. Redwood City, 2011. Disponível em: <https://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html>. Acesso em: 8 abr. 2021.

ORACLE. **Obtenha Informações sobre a Tecnologia Java**. Redwood City, 2013. Disponível em: <https://www.java.com/pt_BR/about/>. Acesso em: 30 ago. 2020.

ORACLE. **Servlet (Java(TM) EE 7 Specification APIs)**. Redwood City, 2015. Disponível em: <https://docs.oracle.com/javaee/7/api/javax/servlet/Servlet.html>. Acesso em: 28 ago. 2020.

ORACLE. **keytool - Key and Certificate Management Tool**. Redwood City, 2018. Disponível em: <https://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>. Acesso em: 6 abr. 2021.

ORACLE. **The Java Tutorials: writing JavaBeans components**. Redwood City, 2019. Disponível em: <https://docs.oracle.com/javase/tutorial/javabeans/writing/index.html>. Acesso em: 28 ago. 2020.

PADATE, R.; PATEL, A. Encryption and decryption of text using aes algorithm. **International Journal of Emerging Technology and Advanced Engineering**, New Delhi, v. 4, n. 5, p. 883–886, mai. 2014. Disponível em: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.639.7316&rep=rep1&type=pdf>. Acesso em: 02 abr. 2021.

PAGSEGURO. **PagSeguro Developers**. São Paulo, 2019. Disponível em: <https://dev.pagseguro.uol.com.br/>. Acesso em: 26 jan. 2021.

PANTOJA, E. B. El patrón de diseño modelo-vista-controlador (mvc) y su implementación en java swing. **Acta Nova**, Cochabamba, v. 2, n. 4, p. 493–507, dez. 2004.

ROMERO, Y. F.; GONZÁLEZ, Y. D. Patrón modelo-vista-controlador. **Telem@tica**, Havana, v. 11, n. 1, p. 47–57, jan./abr. 2012. Disponível em: <http://revistatelematica.cujae.edu.cu/index.php/tele/article/download/15/10/0>. Acesso em: 27 ago. 2020.

TANENBAUM, A. S. **Redes de Computadores**. 4. ed. Rio de Janeiro: Campus/Elsevier, 2003.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. São Paulo: Pearson, 2007.

YASIN, S.; HASEEB, K.; QURESHI, R. J. Cryptography based e-commerce security: A review. **International Journal of Computers Science Issues**, [*s. l.*], v. 9, n. 2, t. 1, p. 132–137, mar. 2012.

YU, M.-C. **A Secure Mobile Agent E-Commerce Protocol.** 2015. Dissertação (Mestrado em Science in Electrical and Computer Engineering) — Purdue University, Indianapolis, 2015.

APÊNDICE A – DETALHAMENTO DOS CAMPOS UTILIZADOS NO BANCO DE DADOS

Entidade “cliente”:

- “login” (tipo “*character varying*”): nome de usuário usado pelo cliente para realizar sua autenticação ("*login*”) no sistema;
- “senha” (tipo “*character varying*”): senha utilizada junto ao “login" para realizar a autenticação no sistema;
- “email” (tipo “*character varying*”): endereço de *e-mail* que serve como contato para o sistema realizar envio automático de mensagens com relação aos seus pedidos e que possa ser passado como identificador ao direcioná-lo para a plataforma de pagamento do PagSeguro através de sua API integrada ao sistema;
- “nome” (tipo “*character varying*”): nome do cliente para sua identificação junto ao vendedor e passado para a plafatorma do PagSeguro através de sua API;
- “ddd” (tipo “*integer*”): código de área do telefone do cliente, utilizado pela API do PagSeguro para passá-lo para sua plataforma;
- “telefone” (tipo “*bigint*”): número do telefone do cliente, passado para a plataforma do PagSeguro através de sua API.

Entidade “estabelecimento”:

- “login” (tipo “*character varying*”): nome de usuário usado pelo vendedor para realizar sua autenticação (“*login*”) no sistema;
- “senha” (tipo “*character varying*"): senha utilizada junto ao “login” para realizar a autenticação no sistema;
- “email” (tipo “*character varying*”): endereço de *e-mail* que serve como contato para o sistema realizar envio automático de mensagens com relação aos pedidos pendentes e usado como credencial para o recebimento de pagamentos na plataforma do PagSeguro;
- “razaosocial” (tipo “*character varying*”): razão social do estabelecimento, ou seja, o nome do estabelecimento, para sua identificação no sistema;
- “cnpj” (tipo “*bigint*”): número do CNPJ[29] para identificação do estabelecimento no sistema;
- “status” (tipo “*boolean*”): indica se o estabelecimento encontra-se aberto ou fechado, o que determina se o cliente pode ou não fazer um pedido;

- “ddd” (tipo “*integer*”): código de área do telefone do vendedor;
- “telefone” (tipo “*bigint*”): número do telefone do vendedor.

Entidade “categoria”:

- “id” (tipo “*integer*”): número identificador da categoria do produto, autogerado pelo sistema;
- “nome” (tipo “*character varying*”): nome da categoria do produto.

Entidade “produto”:

- “id” (tipo “*integer*”): número identificador do produto, autogerado pelo sistema, passado para a plataforma do PagSeguro através de sua API no momento da compra;
- “nome” (tipo “*character varying*”): nome do produto, passado para a plataforma do PagSeguro através de sua API;
- “descricao” (tipo “*character varying*”): detalhes opcionais sobre o produto;
- “preco” (tipo “*double precision*”): preço do produto, passado para a plataforma do PagSeguro através de sua API;
- “imagem” (tipo “*character varying*”): imagem opcional do produto especificada através de um nome e inclusa num diretório especificado no sistema;
- “quantidade” (tipo “*integer*”): número de unidades do produto em estoque reservado para venda na aplicação;
- “disponibilidade” (tipo "*boolean*"): se o produto encontra-se ou não pronto para ser vendido no sistema;
- “categoria_id” (tipo "*integer*"): número identificador da categoria à qual pertence o produto.

Entidade “pedido”:

- “id” (tipo “*bigint*”): número identificador do pedido, autogerado pelo sistema;
- “observacoes” (tipo “*character varying*”): observações opcionais que o cliente pode passar ao vendedor no momento da realização do pedido;
- “agendamento” (tipo “*character varying*”): escolha opcional de um horário para o qual o cliente deseja receber o pedido no estabelecimento;
- “horario” (tipo “*timestamp without time zone*”): data e horário em que o pedido foi realizado, registrados automaticamente pelo sistema;

[29] CNPJ.

- “senhadopedido” (tipo “*character varying*”): senha gerada para identificação do pedido no momento do seu recebimento e passada como referência do pedido para a plataforma do PagSeguro através de sua API;
- “status” (tipo “*character varying*”): estado em que o pedido se encontra, que pode ser “em preparo” (se não tiver sido agendado), “agendado” (se tiver sido agendado), “pronto para entrega” (se o pedido já foi preparado), “entregue” (se o pedido já foi recebido) ou “cancelado” (se o pagamento não tiver sido aprovado);
- “valortotal” (tipo “*double precision*”): valor total a ser pago pelo cliente com base no cálculo do preço e da quantidade escolhida de cada produto;
- “transactioncode” (tipo “*character varying*”): código da transação gerado e enviado para o sistema pela plataforma do PagSeguro, que pode ser usado para identificação da transação e consulta sobre a aprovação do pagamento no seu servidor;
- “pago” (tipo “*boolean*”): se o pedido foi pago ou não;
- “cliente_login” (tipo “*character varying*”): “login” do cliente usado pelo sistema como identificador do autor do pedido.

Entidade “pedido_produto”:

- “pedido_id” (tipo “*bigint*”): número identificador para cada pedido no sistema;
- “produto_id” (tipo “*integer*”): número identificador para cada produto contido num pedido de mesmo id no sistema;
- “quantidade” (tipo “*integer*”): número de unidades para cada produto contidas no pedido, passado para a plataforma do PagSeguro através de sua API.

APÊNDICE B – DETALHAMENTO DO CÓDIGO-FONTE DE IMPLEMENTAÇÃO DO COOKIE DO CARRINHO DE COMPRAS

Para implementar o *cookie*, criou-se uma classe estática na camada de modelo da aplicação, dentro do pacote “modelo.cookie”, mostrada no Código-fonte 5.

Código-fonte 5 – CookieUtils.java: classe estática para manipulação do *cookie*

```
1  public final class CookieUtils {
2      public static final String COOKIE_KEY = "lfps.ontime.cc
           ";
3
4      private CookieUtils() {
5      }
6
7      public static Cookie obterCookie(HttpServletRequest
           request) {
8          Cookie[] cookies = request.getCookies();
9          Cookie c = null;
10         for (int i = 0; cookies != null && i < cookies.
               length; i++) {
11             if (cookies[i].getName().equals(COOKIE_KEY)) {
12                 c = cookies[i];
13                 break;
14             }
15         }
16         return c;
17     }
18 }
```

O método estático “obterCookie” cria uma instância “c” de Cookie de valor nulo na linha 9 e em seguida faz uma busca pelo *cookie* da aplicação. Após encontrá-lo, atribui-o como valor da referência “c” e retorna essa referência.

Em seguida, também na camada de modelo, no pacote “modelo.carrinho”, foram

criadas duas classes: “CarrinhoItem” e “CarrinhoNegocio”, mostradas nos Códigos-fonte 6 e 7, respectivamente.

Código-fonte 6 – CarrinhoItem.java: classe *Bean* que representa um item do carrinho de compras

```
1  public class CarrinhoItem {
2      private Produto produto;
3      private int quantidade;
4      ...
5      public Produto getProduto() {
6          return produto;
7      }
8
9      public void setProduto(Produto produto) {
10         this.produto = produto;
11     }
12
13     public int getQuantidade() {
14         return quantidade;
15     }
16     ...
17 }
```

Nessa classe, para um item do carrinho de compras é identificado o produto através de um atributo da classe Produto e a quantidade desse produto adicionada ao carrinho através de um atributo do tipo inteiro.

Código-fonte 7 – CarrinhoNegocio.java: classe com a lógica de negócio do carrinho de compras

```
1  public final class CarrinhoNegocio {
2      private static final String SEPARADOR_DE_ITENS = "
           SEP_ITENS";
3      private static final String SEPARADOR_NO_ITEM = "
           SEP_REGISTRO";
4
```

```
5       private CarrinhoNegocio() {
6
7       }
8
9       public static final List<CarrinhoItem> obterCarrinho(
           String valor) {
10          List<CarrinhoItem> carrinhoItens = new ArrayList<
               CarrinhoItem>();
11          if (valor == null || valor.trim().length() == 0 ||
               !valor.contains(SEPARADOR_NO_ITEM)) {
12              return carrinhoItens;
13          }
14          ProdutoNegocio produtoNegocio = new ProdutoNegocio
               ();
15          if (valor.contains(SEPARADOR_DE_ITENS)) {
16              String[] itens = valor.split(SEPARADOR_DE_ITENS
                   );
17              for (int i = 0; itens != null && i < itens.
                   length; i++) {
18                  String[] item = itens[i].split(
                       SEPARADOR_NO_ITEM);
19                  CarrinhoItem carrinhoItem = new
                       CarrinhoItem();
20                  Produto produto = produtoNegocio.
                       obterProduto(Integer.parseInt(item[0]));
21                  carrinhoItem.setProduto(produto);
22                  carrinhoItem.setQuantidade(Integer.parseInt
                       (item[1]));
23                  carrinhoItens.add(carrinhoItem);
24              }
25          } else {
26              String[] item = valor.split(SEPARADOR_NO_ITEM);
```

```
27              CarrinhoItem carrinhoItem = new CarrinhoItem();
28              Produto produto = produtoNegocio.obterProduto(
                    Integer.parseInt(item[0]));
29              carrinhoItem.setProduto(produto);
30              carrinhoItem.setQuantidade(Integer.parseInt(
                    item[1]));
31              carrinhoItens.add(carrinhoItem);
32          }
33          return carrinhoItens;
34      }
35
36      public static final String adicionarItem(int produtoId,
            int quantidade, String valor) {
37          List<CarrinhoItem> carrinhoItens = obterCarrinho(
                valor);
38          if (carrinhoItens.isEmpty()) {
39              return produtoId + SEPARADOR_NO_ITEM +
                    quantidade;
40          }
41          boolean adicionou = false;
42          String resultado = "";
43          for (CarrinhoItem carrinhoItem : carrinhoItens) {
44              if (carrinhoItem.getProduto().getId() ==
                    produtoId) {
45                  carrinhoItem.setQuantidade(carrinhoItem.
                        getQuantidade() + quantidade);
46                  adicionou = true;
47              }
48              if (!resultado.isEmpty()) {
49                  resultado += SEPARADOR_DE_ITENS;
50              }
51              resultado += carrinhoItem.getProduto().getId()
```

```
                + SEPARADOR_NO_ITEM + carrinhoItem.
                getQuantidade();
52          }
53          if (!adicionou) {
54              resultado += SEPARADOR_DE_ITENS + produtoId +
                    SEPARADOR_NO_ITEM + quantidade;
55          }
56          return resultado;
57      }
58
59      public static final String removerItem(int produtoId,
            String valor) {
60          List<CarrinhoItem> carrinhoItens = obterCarrinho(
                valor);
61          if (carrinhoItens.isEmpty()) {
62              return "";
63          }
64          String resultado = "";
65          for (CarrinhoItem carrinhoItem : carrinhoItens) {
66              if (carrinhoItem.getProduto().getId() ==
                    produtoId) {
67                  continue;
68              }
69              if (!resultado.isEmpty()) {
70                  resultado += SEPARADOR_DE_ITENS;
71              }
72              resultado += carrinhoItem.getProduto().getId()
                    + SEPARADOR_NO_ITEM + carrinhoItem.
                    getQuantidade();
73          }
74          return resultado;
75      }
```

```
76          . . .
77  }
```

O Código-fonte 7 acima apresenta a classe estática que encapsula as regras de negócio de manipulação das informações a serem escritas no *cookie* do carrinho em formato de texto, contendo os números identificadores dos produtos, suas quantidades adicionadas ao carrinho e separadores.

A variável estática “SEPARADOR_DE_ITENS” (linha 2) consiste numa String que separa os itens do carrinho, enquanto a variável “SEPARADOR_NO_ITEM” (linha 3) é uma String separa as informações de um mesmo item.

Para exemplificar a estrutura do valor escrito no *cookie* do carrinho, considere-se que, por exemplo, o usuário adicionou a seu carrinho 1 unidade do produto cujo identificador é 5 e, depois, 3 unidades do produto cujo identificador é 6. O valor apresentado no *cookie* seria “5SEP_REGISTRO1SEP_ITENS6SEP_REGISTRO3”.

O método “obterCarrinho” (linha 9) retorna uma lista dos itens contidos no carrinho de compras com suas respectivas quantidades, montada ao filtrar as informações de texto registradas no *cookie* divididas por separadores.

O método “adicionarItem” (linha 36) adiciona um item ao carrinho, retornando o novo texto a ser escrito no *cookie* do carrinho, reescrevendo o que havia anteriormente. Primeiro ele obtém a lista de itens do carrinho (linha 37) e verifica se o carrinho está vazio (linha 38). Se estiver vazio, adiciona o novo valor à String que contém as informações a serem adicionadas ao *cookie* (linha 38 a 40). Caso contrário, é criada uma String vazia que receberá o novo valor a ser escrito no *cookie* (linha 42). Em seguida, percorre-se a lista através de um laço de repetição (linha 43). Ao percorrer a lista, averigua se o item adicionado já está no carrinho (linha 44). Se estiver, apenas atualiza sua quantidade na lista (linha 45). O laço segue preenchendo a String criada vazia com os valores da lista (linhas 48 a 52). Caso a lista tenha sido percorrida, e o item adicionado não tenha sido encontrado no carrinho (linha 53), ele é adicionado ao final da String junto com sua quantidade (linha 54).

O método “removerItem” (linha 59) remove completamente um item do carrinho, também retornado um novo texto a ser adicionado ao *cookie*, que é similar ao anterior, mas sem o item removido.

Inicialmente obtém-se a lista de itens do carrinho de compras (linha 60) e verifica-se

se ela está vazia (linha 61). Se sim, retorna apenas uma String vazia. Se a lista não estiver vazia, uma String vazia é criada para ser preenchida com o novo valor que terá o *cookie* (linha 64). Um laço de repetição percorre a lista (linha 65), preenchendo a String vazia com os valores da lista (linhas 69 a 72), verificando a cada iteração se foi encontrado o item que se deseja remover da lista (linha 66). Se foi, o laço passa para a próxima iteração (linha 67), sem escrever seus dados na String, produzindo, assim, uma nova String sem o item que se desejava remover.

Com as classes de modelo do carrinho e do *cookie* já definidas, os servlets utilizam seus métodos para alterar o *cookie*.

O pacote "controle.carrinho" contém os servlets responsáveis por acessar e modificar o *cookie* do carrinho de compras. Para tal, é necessário utilizar uma classe estática denominada "AES", localizada no pacote "modelo.criptografia" e contém os métodos para criptografar e descriptografar o conteúdo do carrinho.

O servlet "MostrarProdutoCarrinhoServlet" é responsável por exibir o carrinho de compras na página do carrinho, como mostra o Código-fonte 8.

Código-fonte 8 – MostrarProdutoCarrinhoServlet.java: classe que recupera o carrinho de compras do usuário

```
1  public class MostrarProdutoCarrinhoServlet extends
       HttpServlet {
2    protected void service(HttpServletRequest request,
       HttpServletResponse response)
3          throws ServletException, IOException {
4      request.setCharacterEncoding("UTF-8");
5      response.setCharacterEncoding("UTF-8");
6      ...
7      Cookie c = CookieUtils.obterCookie(request);
8      if (c == null) {
9          c = new Cookie(CookieUtils.COOKIE_KEY, null);
10         c.setValue("");
11         String valorVazioCriptografado = AES.
             criptografar(c.getValue());
12         c.setValue(valorVazioCriptografado);
13     } else {
```

```
14              String valorDescriptografado = AES.
                   descriptografar(c.getValue());
15              List<CarrinhoItem> carrinho = CarrinhoNegocio.
                   obterCarrinho(valorDescriptografado);
16              request.setAttribute("carrinho", carrinho);
17          }
18          c.setMaxAge(Integer.MAX_VALUE);
19          c.setSecure(true);
20          c.setHttpOnly(true);
21          response.addCookie(c);
22
23          request.getRequestDispatcher("carrinho.jsp").
               forward(request, response);
24      }
25  }
```

Na linha 7, é recuperado o *cookie* da aplicação, caso exista. Se não existir (linha 8), cria o *cookie* com valor vazio (linhas 9 e 10). Na linha 11, esse valor é criptografado. Na linha 12, o valor criptografado é gravado no *cookie*.

Caso já exista o *cookie*, seu valor é descriptografado (linha 14) para que a lista de itens do carrinho possa ser obtida (linha 15) e configurada como atributo (linha 16) que será despachado para a página "carrinho.jsp" (linha 23).

Na linha 18, a validade do *cookie* em segundos é atualizada para o máximo valor inteiro. Na linha 19, é adicionada a *flag* para que o *cookie* só seja lido caso a comunicação seja segura, conforme mencionado na Subseção 4.3.3. Segundo o que foi citado na mesma subseção, na linha 20, adiciona-se a *flag* que protege o *cookie* de *scripts* executados no lado do cliente. Por fim, armazena-se o *cookie* com os valores atualizados no navegador do cliente (linha 21 e 23).

No Código-fonte 9, encontra-se o código do servlet "AdicionarProdutoCarrinhoServlet", usado a cada vez que se adiciona um produto ao carrinho de compras.

Código-fonte 9 – AdicionarProdutoCarrinhoServlet.java: classe que adiciona um item ao carrinho

```
1  public class AdicionarProdutoCarrinhoServlet extends
       HttpServlet {
2      protected void service(HttpServletRequest request,
           HttpServletResponse response)
3              throws ServletException, IOException {
4          request.setCharacterEncoding("UTF-8");
5          response.setCharacterEncoding("UTF-8");
6          String origem = request.getParameter("origem");
7          ...
8          int produtoId = Integer.parseInt(request.
               getParameter("produtoId"));
9          int quantidade = Integer.parseInt(request.
               getParameter("quantidade"));
10
11         Cookie c = CookieUtils.obterCookie(request);
12         String valorAnterior = AES.descriptografar(c.
               getValue());
13         String novoValor = CarrinhoNegocio.adicionarItem(
               produtoId, quantidade, valorAnterior);
14         ...
15         String novoValorCriptografado = AES.criptografar(
               novoValor);
16         c.setValue(novoValorCriptografado);
17         c.setSecure(true);
18         c.setHttpOnly(true);
19         ...
20         request.getRequestDispatcher(origem).forward(
               request, response);
21     }
22 }
```

Nas linhas 8 e 9, recuperam-se os parâmetros do id do produto e sua quantidade,

respectivamente, adicionados ao carrinho via JSP. Na linha 11, é obtido o *cookie* do *website*. Na linha 12, seu valor é descriptografado para que possa ser atualizado com o novo valor (linha 13), que é criptografado (linha 15) e gravado no *cookie* (linha 16). Em seguida, nas linhas 17 e 18, respectivamente, são inseridas as *flags* "secure" e "HttpOnly".

A classe "RemoverProdutoCarrinhoServlet", responsável por remover um item do carrinho de compras, possui código similar.

Código-fonte 10 – RemoverProdutoCarrinhoServlet.java: classe que remove um produto do carrinho

```
1  public class RemoverProdutoCarrinhoServlet extends
       HttpServlet {
2    protected void service(HttpServletRequest request,
       HttpServletResponse response)
3          throws ServletException, IOException {
4      request.setCharacterEncoding("UTF-8");
5      response.setCharacterEncoding("UTF-8");
6      int produtoId = Integer.parseInt(request.
         getParameter("produtoId"));
7      ...
8      Cookie c = CookieUtils.obterCookie(request);
9      String valorAnterior = AES.descriptografar(c.
         getValue());
10     String novoValor = CarrinhoNegocio.removerItem(
         produtoId, valorAnterior);
11     String novoValorCriptografado = AES.criptografar(
         novoValor);
12     ...
13     c.setValue(novoValorCriptografado);
14     c.setSecure(true);
15     c.setHttpOnly(true);
16
17     request.getRequestDispatcher("
         MostrarProdutoCarrinhoServlet").forward(request,
```

```
                response);
18      }
19  }
```

A diferença entre os Códigos-fonte 9 e 10 é que o último recupera somente o id do produto e remove esse produto completamente do carrinho de compras, qualquer que seja a quantidade que havia. Além disso, a classe “RemoverProdutoCarrinhoServlet” não necessita recuperar a página de origem (linha 6 do Código-fonte 9 para que não mude de página após utilizar o método “getRequestDispatcher”, pois a ação de remover um produto do carrinho de compras só pode ser realizada na página do carrinho de compras.

Na página do carrinho de compras, a lista de produtos com suas respectivas quantidades é recuperada através de *scriptlets*.

Código-fonte 11 – carrinho.jsp: trecho de código que recupera o carrinho de compras

```
1  ...
2  <%
3      List<CarrinhoItem> carrinhoItens = (List<CarrinhoItem>)
4      request.getAttribute("carrinho");
5      if (carrinhoItens != null && carrinhoItens.size() > 0)
           {
6  %>
7  ...
8  <%
9      double total = 0;
10     for (CarrinhoItem c : carrinhoItens) {
11     total += c.getQuantidade() * c.getProduto().getPreco();
12 %>
13 ...
14 <%
15     }
16 %>
17 ...
```

```
18 <%
19     } else {
20 %>
21 ...
22 <%
23     }
24 %>
```

A linha 3 recupera a lista de itens do carrinho de compras do atributo “carrinho”. Se a lista não for nula ou vazia (linha 5), são mostrados respectivos nomes dos produtos, preços e quantidades adicionados no código HTML da página, omitido por não fazer parte do escopo desta subseção. Através do carrinho também é obtido o valor total a ser pago pelo cliente ao multiplicar o preço de cada produto com sua quantidade no carrinho (linhas 10 e 11).

Outros servlets que recebem requisições e enviam respostas para JSPs da página inicial do *website* e das páginas de escolha de produtos à venda também utilizam o trecho de código mostrado a seguir.

Código-fonte 12 – Trecho do código que procura o *cookie* no navegador

```
1  Cookie c = CookieUtils.obterCookie(request);
2  if (c == null) {
3      c = new Cookie(CookieUtils.COOKIE_KEY, null);
4      c.setValue("");
5      String valorVazioCriptografado = AES.criptografar(c.
           getValue());
6      c.setValue(valorVazioCriptografado);
7  } else {
8      String valorDescriptografado = AES.descriptografar(c.
           getValue());
9      List<CarrinhoItem> carrinho = CarrinhoNegocio.
           obterCarrinho(valorDescriptografado);
10     request.setAttribute("carrinho", carrinho);
11 }
```

```
12 c.setMaxAge(Integer.MAX_VALUE);
13 c.setSecure(true);
14 c.setHttpOnly(true);
15 response.addCookie(c);
```

Como já mostrado no Código-fonte 8, o código acima procura o *cookie* do *website* no navegador. Se não houver, cria-o vazio e criptografa-o. Se houver, descriptografa seu valor para que seja montada uma lista com os itens do carrinho de compras do usuário, passada como atributo da requisição feita ao servlet, para que ele utilize essa informação para realizar alguma operação. Em seguida atualiza o tempo de validade do *cookie*, adiciona as *flags* de segurança já mencionadas na Subseção 4.3.3 e grava o *cookie* no navegador do cliente. A criação do *cookie* vazio (linha 3) é especialmente útil quando o usuário acessa o endereço *web* pela primeira vez.

O servlet “PrepararPedidoServlet” utiliza o seguinte trecho de código para montar o carrinho de compras para enviá-lo à API do PagSeguro.

Código-fonte 13 – Trecho de código que monta o carrinho de compras e o envia ao servlet que utiliza a API de Pagamento

```
1  ...
2  Cookie c = CookieUtils.obterCookie(request);
3  String valorDescriptografado = AES.descriptografar(c.
      getValue());
4  List<CarrinhoItem> carrinhoItens = CarrinhoNegocio.
      obterCarrinho(valorDescriptografado);
5  ...
6  session.setAttribute("carrinhoItens", carrinhoItens);
7  ...
8  RequestDispatcher rd = request.getRequestDispatcher("
      CheckoutPagSeguroServlet");
9  rd.forward(request, response);
10 ...
```

A linha 2 recupera o *cookie* do carrinho de compras, e a linha 3 descriptografa o

valor contido nele. A linha 4 monta uma lista de objetos da classe “CarrinhoItem” utilizando o método para obter o carrinho através do valor decifrado do *cookie*. A linha 6 armazena a lista como um atributo de sessão, enquanto as linhas 8 e 9 encaminham para o servlet que utiliza a API do PagSeguro para fazer a compra, utilizando, entre outros, os dados do carrinho recuperados nessa classe.

No caso do servlet “PedidoSucessoServlet”, responsável por encaminhar para a página que mostra ao cliente que seu pagamento foi aprovado, é necessário também esvaziar o carrinho.

Código-fonte 14 – Trecho de código que esvazia o carrinho de compras automaticamente

```
1  boolean zerarCarrinho = false;
2  if (request.getParameter("carrinhoVazio") != null &&
       Boolean.parseBoolean(request.getParameter("carrinhoVazio
       "))) {
3      zerarCarrinho = true;
4  }
5  Cookie c = CookieUtils.obterCookie(request);
6  if (!zerarCarrinho) {
7      if (c == null) {
8          c = new Cookie(CookieUtils.COOKIE_KEY, null);
9          c.setValue("");
10         String valorVazioCriptografado = AES.criptografar(c
               .getValue());
11         c.setValue(valorVazioCriptografado);
12     } else {
13         String valorDescriptografado = AES.descriptografar(
               c.getValue());
14         List<CarrinhoItem> carrinho = CarrinhoNegocio.
               obterCarrinho(valorDescriptografado);
15         request.setAttribute("carrinho", carrinho);
16     }
17 } else {
```

```
18      c = new Cookie(CookieUtils.COOKIE_KEY, null);
19      c.setValue("");
20      String valorVazioCriptografado = AES.criptografar(c.
           getValue());
21      c.setValue(valorVazioCriptografado);
22  }
23  c.setMaxAge(Integer.MAX_VALUE)
24  c.setSecure(true);
25  c.setHttpOnly(true);
26  response.addCookie(c);
```

As linhas 5 a 16 e 23 a 26 estavam presentes no Código-fonte 12 e já foram explicadas anteriormente. A linha 1 inicializa como falsa a variável lógica “zerarCarrinho”, que representa se o servlet recebeu alguma *flag* para esvaziar o carrinho de compras. Nas linhas 2 e 3 e 17 a 22, é definido que se o carrinho não estiver vazio e for recebida uma *flag* para esvaziar o carrinho, o *cookie* deve ser criado vazio e criptografado, para depois sobrescrever o *cookie* existente (linha 26).

APÊNDICE C – DETALHAMENTO DO CÓDIGO-FONTE DO SERVLET DE CHECKOUT

O código a seguir é responsável por realizar a construção da cobrança e redirecionamento para o sistema do PagSeguro.

Código-fonte 15 – CheckoutPagSeguroServlet.java

```
1  public class CheckoutPagSeguroServlet extends HttpServlet {
2
3      protected void service(HttpServletRequest request,
           HttpServletResponse response)
4              throws ServletException, IOException {
5          request.setCharacterEncoding("UTF-8");
6          response.setCharacterEncoding("UTF-8");
7
8          HttpSession session = request.getSession();
9          String cliente_login = (String) session.
               getAttribute("login");
10         List<CarrinhoItem> carrinhoItens = (List<
               CarrinhoItem>) session.getAttribute("
               carrinhoItens");
11         String senhadopedido = (String) session.
               getAttribute("senhadopedido");
12
13         ClienteNegocio clienteNegocio = new ClienteNegocio
               ();
14         Cliente cliente = clienteNegocio.obterCliente(
               cliente_login);
15         String cliente_nome = cliente.getNome();
16         String cliente_ddd = String.valueOf(cliente.getDdd
               ());
17         String cliente_telefone = String.valueOf(cliente.
               getTelefone());
```

```
18          String cliente_email = cliente.getEmail();
19
20          try {
21              final PagSeguro pagSeguro = PagSeguro
22                      .instance(new SimpleLoggerFactory(),
                            new JSEHttpClient(),
23                              Credential.sellerCredential(
                                    SELLER_EMAIL, SELLER_TOKEN),
                                     ENV);
24              CheckoutRegistrationBuilder builder = new
                    CheckoutRegistrationBuilder();
25              builder.withCurrency(Currency.BRL);
26              builder.withReference(senhadopedido);
27              builder.withSender(new SenderBuilder()
28                      .withEmail(cliente_email)
29                      .withName(cliente_nome)
30                      .withPhone(new PhoneBuilder()
31                              .withAreaCode(cliente_ddd)
32                              .withNumber(cliente_telefone)))
                                ;
33              builder.withShipping(new ShippingBuilder()
34                      .withType(ShippingType.Type.USER_CHOISE
                            )
35                      .withCost(BigDecimal.ZERO)
36                      .withAddress(new AddressBuilder()
37                              .withPostalCode("60455760")
38                              .withCountry("BRA")
39                              .withState(State.CE)
40                              .withCity("Fortaleza")
41                              .withComplement("Bloco 901 - 1o
                                    andar")
42                              .withDistrict("Pici")
```

```
43                          .withNumber("s/n")
44                          .withStreet("PICI LANCHES - Av.
                                Humberto Monte, Campus do
                               Pici")));
45              for (int i = 0; i < carrinhoItens.size(); i++)
                   {
46                  builder.addItem(new PaymentItemBuilder()
47                          .withId((String.valueOf(
                               carrinhoItens.get(i).getProduto
                               ().getId())))
48                          .withDescription(carrinhoItens.get(
                               i).getProduto().getNome())
49                          .withAmount(BigDecimal.valueOf(
                               carrinhoItens.get(i).getProduto
                               ().getPreco()))
50                          .withQuantity(carrinhoItens.get(i).
                               getQuantidade())
51                          .withShippingCost(BigDecimal.ZERO)
52                          .withWeight(0)
53                  );
54              }
55              builder.withAcceptedPaymentMethods(new
                   AcceptedPaymentMethodsBuilder()
56                         .addInclude(new PaymentMethodBuilder()
57                                 .withGroup(PaymentMethodGroup.
                                    CREDIT_CARD)
58                         )
59              );
60              builder.addPaymentMethodConfig(new
                   PaymentMethodConfigBuilder()
61                         .withPaymentMethod(new
                            PaymentMethodBuilder()
```

```
62                                  .withGroup(PaymentMethodGroup.
                                        CREDIT_CARD)
63                          )
64                          .withConfig(new ConfigBuilder()
65                                  .withKey(ConfigKey.
                                        MAX_INSTALLMENTS_LIMIT)
66                                  .withValue(new BigDecimal(1))
67                          )
68                  );
69                  RegisteredCheckout registeredCheckout =
                        pagSeguro.checkouts().register(builder);
70                  response.sendRedirect(registeredCheckout.
                        getRedirectURL());
71              } catch (IOException e) {
72                  e.printStackTrace();
73              }
74      }
75  }
```

A linha 9 recupera o login do cliente através de um atributo de sessão para posteriormente utilizá-lo como parâmetro para obter uma instância da classe Cliente (linha 14) com todos os dados do cliente que está fazendo a compra.

As linhas 10 e 11 recuperam o carrinho de compras e a senha do pedido através de atributos de sessão. É importante ressaltar que esse servlet recebe informações de um servlet anterior (“PrepararPedidoServlet”), como descrito na Subseção 4.3.5, que realiza a recuperação do valor do *cookie* para montar a lista do carrinho de compras como descrito no Código-fonte 13.

As linhas 15 a 18 acessam informações do cliente através da instância da classe Cliente criada na linha 14. Note-se que o código DDD e o número de telefone foram convertidos para Strings, porque a API do PagSeguro usa Strings como parâmetros para o registro de quaisquer informações do cliente.

A seguir, as linhas 21 a 70 utilizam classes e métodos da API do PagSeguro para criar a cobrança e realizar o acesso ao seu servidor.

A linha 23 passa as credenciais do vendedor e o ambiente de desenvolvimento como parâmetros para autenticação no PagSeguro.

As credenciais do vendedor tratam-se de seu *e-mail* de cadastro no PagSeguro e um *token* de segurança gerado nas configurações da sua conta no PagSeguro especialmente para utilização de serviços de integração via API. No ambiente *sandbox*, um *token* diferente é gerado para testes.

Os atributos com as credenciais de vendedor e o ambiente de desenvolvimento são importados por meio de uma classe estática denominada "Configuracao", que contém atributos estáticos utilizados em toda a aplicação. O atributo que define qual o ambiente de desenvolvimento ("ENV", no caso do código acima) precisa ser da classe "PagSeguroEnv", definida na API. Seu valor deve ser "PagSeguroEnv.PRODUCTION" caso o ambiente de produção esteja sendo usado. Para utilizar o ambiente de testes, o valor atribuído deve ser "PagSeguroEnv.SANDBOX".

Na linha 24 é instanciado um objeto da classe CheckoutRegistrationBuilder, responsável por passar todos os parâmetros da cobrança para o objeto de *checkout* que será instanciado depois (linha 69).

A linha 25 passa a moeda em que será realizado o pagamento, ou seja, o Real. Apesar de esse parâmetro ser obrigatório, a API do PagSeguro não aceita outras moedas.

A linha 26 passa como parâmetro uma String para servir como valor de referência para servir como identificador personalizado para facilitar a identificação da cobrança no sistema do PagSeguro e é opcional. Pode ser o número identificador do pedido ("id") na aplicação, mas aqui foi passada a senha do pedido, uma vez que a senha já foi gerada no servlet anterior, mas o id não, já que ele é gerado automaticamente por autoincremento somente no momento do registro do pedido no BD, após a finalização do *checkout* na página do PagSeguro.

As linhas 27 a 32 enviam informações do comprador no formato String (*e-mail*, nome, código DDD e telefone) e são opcionais, porém são importantes caso o comprador já tenha conta no PagSeguro, especialmente o *e-mail*, que possibilita o autopreenchimento do campo de nome de usuário (que é o endereço de *e-mail*) usado para fazer o *login* no sistema do PagSeguro. Enviá-lo trará comodidade ao usuário quando for redirecionado, só precisando inserir sua senha.

As linhas 33 a 44 passam Strings com informações de frente e também são opcionais. Como a aplicação foi feita para retirada de pedidos no local, não precisaria dessas opções utilizar a classe ShippingBuilder, mas julgou-se interessante inserir o custo do frente como igual a 0 e o endereço da lanchonete com o seu nome (fictício, no exemplo desenvolvido) onde o pedido

será retirado, para que o usuário veja essas informações no momento do *checkout* na página do PagSeguro, conferindo mais autenticidade ao processo. Na linha 34, podem ser passados os parâmetros “PAC”, “SEDEX” e “USER_CHOISE” (com essa grafia). Para personalizar o frete tal como foi feito, é necessário usar a opção “USER_CHOISE”.

Nas linhas 45 a 56 foi realizado um laço de repetição para a adição de informações de cada item constante no pedido. Na linha 47, o id do produto deve ser uma String. Na linha 48, a descrição também é uma String e deve conter o nome do produto. Na linha 49 é passado o preço do produto, que deve ser do tipo BigDecimal. Na linha 50, a quantidade de unidades do produto comprada, do tipo Integer. A linha 51 é opcional: passar o valor do frete do tipo BigDecimal como 0 ou retirar essa linha do código têm o mesmo efeito. Na linha 52, deve ser passada a massa em gramas do item, no tipo Integer, para fins de cálculo de frete. Esse parâmetro é obrigatório, mesmo com o frete gratuito, seja omitindo-o do código ou passando 0 como parâmetro do custo. Como não é relevante para a aplicação, a massa foi atribuída com valor 0.

As linhas 55 a 59 configuram os métodos de pagamento aceito. É possível incluir mais de um método, porém para o objetivo da aplicação, foi incluso somente o método por cartão de crédito (linha 57).

As linhas 60 a 68 fazem referência a configurações específicas para certos métodos de pagamento. É possível estabelecer um desconto para um pagamento em boleto, por exemplo, mas aqui nenhum método de desconto foi acrescentado. Esse trecho de código estabelece apenas que para o método de pagamento por cartão de crédito (linha 62), pode-se realizar o pagamento em no máximo 1 parcela (linhas 65 e 66). Caso o valor inserido fosse 3, por exemplo, seria possível realizar o parcelamento em 1, 2 ou 3 parcelas.

O objeto instanciado na linha 69, do tipo RegisteredCheckout, recebe todas as informações acima armazenadas na instância da classe CheckoutRegistrationBuilder. A linha 70 realiza o redirecionamento com essas informações para o sistema do PagSeguro, para a continuação do processo de *checkout*.

ANEXO A – ENTREVISTA COM O CLIENTE DE TESTE DA APLICAÇÃO DESENVOLVIDA NA DISCIPLINA DE PROJETO INTEGRADO II

Figura 37 – Questionário aplicado em entrevista presencial com Adriano Queiroz, dono da cantina "Virtual Lanches", localizada no bloco didático do curso de Sistemas e Mídias Digitais da UFC

Questões	Respostas
1 Que produtos você vende, em uma visão geral/resumida do seu negócio?	Bebidas, salgados, sucos.
2 Quais os produtos mais vendidos diariamente?	O kit de promoção: um salgado com bebida.
3 Você faz kits/combos de produtos (por exemplo: refrigerante com salgado, sanduíche com suco) para vender em um valor mais baixo que a compra isolada desses produtos? Se sim, quais são os kits/combos que você faz?	Salgado com suco, combo que diminui o valor em relação à compra separada.
4 Você conhece os horários de pico do seu negócio, horários nos quais, todo dia ou quase todo dia, o movimento da cantina é intenso? Se sim, quais são esses horários?	16 h, 18 h e 20 h. O movimento varia bastante durante a semana.
5 Nesses horários, a fila de atendimento fica realmente grande?	Sim, a fila fica grande.
6 Nos horários de pico de sua cantina, geralmente ocorre demora no atendimento aos clientes?	Sim, por conta da demanda entre os pedidos que demoram a ser preparados e os que não precisam de preparo na hora.
7 O que você considera ser a causa, na maioria dos casos, da demora no atendimento aos clientes quando sua cantina está em horário de pico? Seria o pagamento em cartão, a demora do cliente em decidir o que pedir, a demora no preparo dos produtos ou outro (qual)?	Principalmente o preparo de alguns produtos. O mistão, a vitamina e o prato feito são os que consomem mais tempo.
8 Se pudermos desenvolver um aplicativo para melhorar o seu negócio agilizando o atendimento aos clientes (mesmo quando estiver nos horários de pico), você utilizaria? O que você considera particularmente importante que esse aplicativo permita realizar: pedidos online, agendamento de pedidos, pagamento online, outras funções #quais)?	Com certeza. O agendamento dos pedidos seria de grande ajuda.
9 Se desenvolvermos um aplicativo por meio do qual os clientes possam fazer seus pedidos (agendando-os ou não), como você faria para atendê-los devidamente e ainda atender os clientes que fariam seus pedidos presencialmente na sua cantina? Você teria algum problema nesse contexto? Se sim, quais seriam?	Colocaria pelo menos uma pessoa para realizar o gerenciamento dos pedidos online no celular (não houve comentários sobre possíveis problemas).
10 Como você gerencia seus produtos atualmente e como gerenciaria os mesmos no caso de ter esse aplicativo que estamos projetando?	Cardápio. O controle de estoque e venda é realizado diariamente ou semanalmente (não houve comentários sobre alguma mudança na forma de gerenciamento).
11 Seria importante o aplicativo vender apenas "combos/kits" ou isso poderia ser feito ao menos para a realidade do projeto que estamos fazendo para a disciplina que estamos cursando?	Focar nos produtos de maior demanda de tempo seria o melhor a ser feito.
12 Seria importante o aplicativo ter uma "cota" separada somente para os usuários do aplicativo de cada produto ou apenas para combos/kits?	(O entrevistado não soube responder ao certo qual seria a melhor solução, mas, por razões logísticas, foi definido que uma cota de venda pelo aplicativo seria uma solução inicial)
13 Como funciona, atualmente, o pagamento de compras por cartão (crédito/débito) na sua cantina? Há taxas envolvidas? Se sim, quanto?	Aceitamos crédito e débito. Há uma diferença na taxa de cobrança entre os dois mecanismos de pagamento. O pagamento via débito é mais vantajoso para o estabelecimento, por possuir menores taxas.
14 Caso a aplicação venha a ser realmente desenvolvida para o mercado, como você preferiria pagar pelo uso e pela manutenção dos serviços do aplicativo (registro oficial de sua lanchonete na aplicação): por taxa sobre as vendas ou por um valor fixo mensal (como uma assinatura do serviço)?	Uma taxa mensal seria mais viável.

Fonte: Entrevista realizada pela equipe Time5 (2019).

ANEXO B – PESQUISA COM POSSÍVEIS CLIENTES DA APLICAÇÃO DESENVOLVIDA NA DISCIPLINA DE PROJETO INTEGRADO II

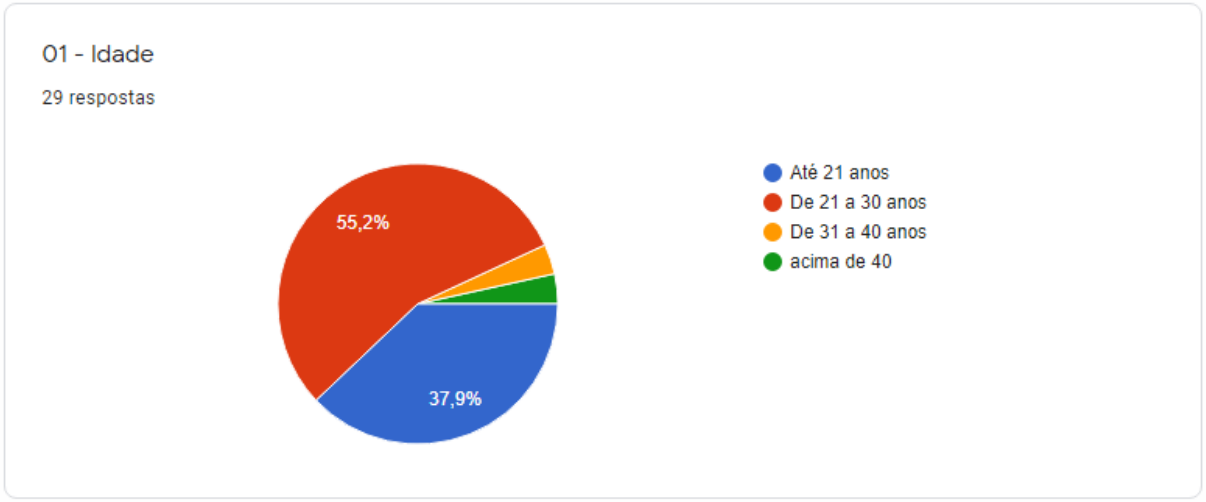
Figura 38 – Formulário aplicado presencialmente com proprietários/funcionários de cantinas da UFC

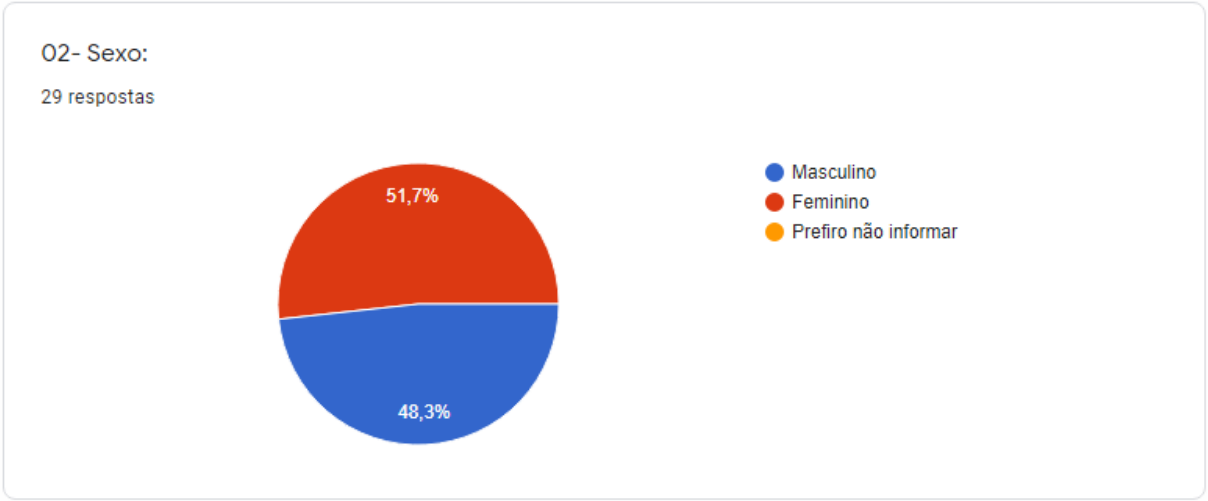
Questões	Opções	Resposta do entrevistado A	Resposta do entrevistado B	Resposta do entrevistado C	Resposta do entrevistado D	Resposta do entrevistado E
1 Você tem problemas com a formação de filas de clientes em atendimento na sua lanchonete?	• **"Sim"** • **"Não"** **(resposta única)**	Sim.	Não.	Não.	Não.	Não.
2 Qual o horário de maior movimento e formação de filas na sua lanchonete?	• **"Início da manhã"** • **"Meio da manhã"** • **"Final da manhã"** • **"Início da tarde"** • **"Meio da tarde"** • **"Final da tarde"** • **"Outros..."** **(múltiplas respostas possíveis)**	Meio da manhã; início da tarde.	Meio da tarde.	Meio da manhã; início da tarde.	Meio da tarde.	Início da manhã; meio da manhã; início da tarde; meio da tarde.
3 Você estaria disposto a utilizar uma aplicação digital para reduzir as filas e/ou o tempo de atendimento aos seus clientes?	• **"Sim, com certeza!" "Provavelmente sim"** • **"Não"** **(resposta única)**	Sim, com certeza!	Não.	Provavelmente sim.	Provavelmente sim.	Não.
4 Você possui alguma forma de gerenciamento de estoque e vendas da lanchonete? Se sim, como funciona?	**(Questão aberta)**	"Faço anotações e reponho o estoque na sexta."	"Sim, observação diária por se tratar de poucos itens."	"Manual caderno e posterior passagem para planilha eletrônica. Pedidos agendados via WhatsApp da proprietária da lanchonete." (sic)	"De estoque > não. De vendas > não." (sic)	"Estoque > planilhas eletrônicas. Vendas > sistema de vendas." (sic)
5 Cite 5 dos principais itens vendidos nos horários de grande movimento na sua lanchonete.	**(Questão aberta)**	Salgado, café, bolo, almoço e refrigerante.	Vitaminas de guaraná e açaí.	Almoço, salgados.	Tapioca, pão de queijo, vitaminas, suco de laranja, salgados, café.	Salgado, tapioca, cuscuz, sucos e vitaminas, bolos, almoços.
6 Você costuma vender kits/combos de produtos?	• **"Sim"** • **"Não"** **(resposta única)**	Não.	Não.	Sim.	Não.	Sim.
7 Quando você vende kits/combos de produtos, você dá ao cliente um desconto em relação ao preço de cada produto comprado isoladamente?	• **"Sim"** • **"Não"** **(resposta única; não se aplica a quem marcou a opção "Não" na questão anterior)**	(Não se aplica)	(Não se aplica)	Sim.	(Não se aplica)	Sim.
8 Você gostaria de utilizar um sistema/aplicativo para facilitar a venda de lanches nos horários de grande movimento?	• **"Sim, com certeza!" "Provavelmente sim"** • **"Não"** **(resposta única)**	Provavelmente sim.	Não.	Sim, com certeza!	Provavelmente sim.	Não.
9 Você estaria disposto a pagar um valor (taxa sobre as vendas) para cada venda realizada através desse aplicativo?	• **"Sim, sem problemas"** • **"Não, preferiria pagar uma taxa mensal pelo uso da aplicação"** • **"Não, de forma alguma"** **(resposta única)**	Não, preferiria pagar uma taxa mensal pelo uso da aplicação.	Não, de forma alguma.	Não, de forma alguma.	Não, de forma alguma.	Não, de forma alguma.
10 Você aprova a ideia de realização de pedidos para sua lanchonete, sendo eles agendados (com horário marcado) ou não, via aplicativo?	• **"Sim, apenas com agendamento de pedido"** • **"Sim, sendo os pedidos com horário marcado ou não"** • **"Não aprovo as ideias propostas"** **(resposta única)**	Sim, apenas com agendamento de pedido.	Não aprovo as ideias propostas.	Sim, apenas com agendamento de pedido.	Sim, sendo os pedidos com horário marcado ou não.	Não aprovo as ideias propostas.

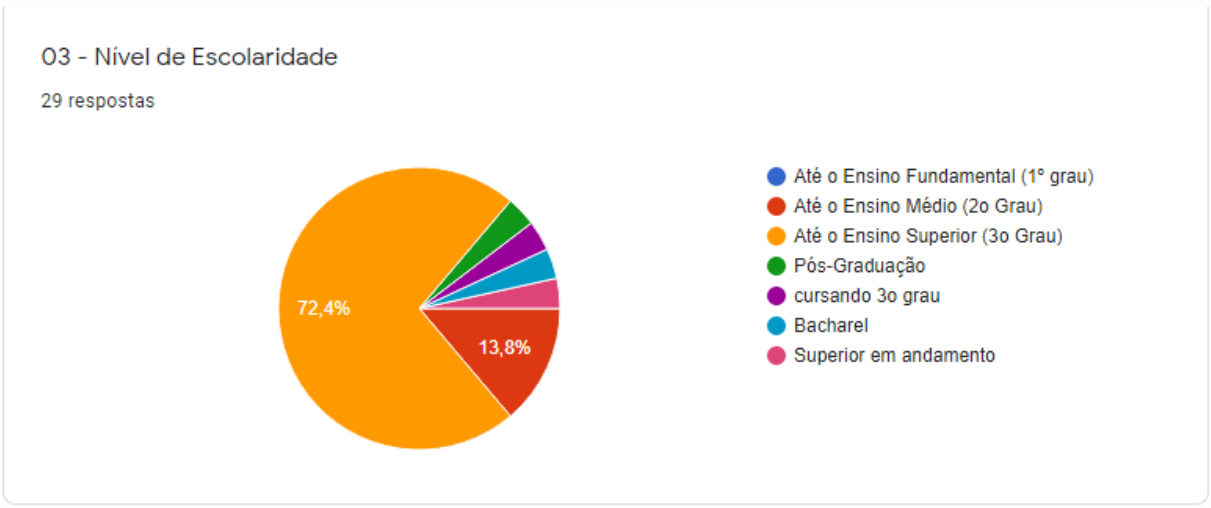
Fonte: Pesquisa realizada pela equipe Time5 (2019).

ANEXO C – PESQUISA COM O PÚBLICO-ALVO DA APLICAÇÃO DESENVOLVIDA NA DISCIPLINA DE PROJETO INTEGRADO II

Figura 39 – Formulário *online* aplicado remotamente com clientes de lanchonetes

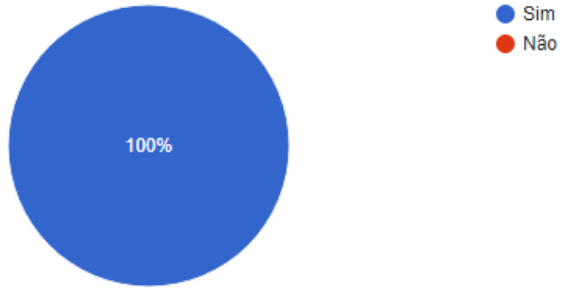






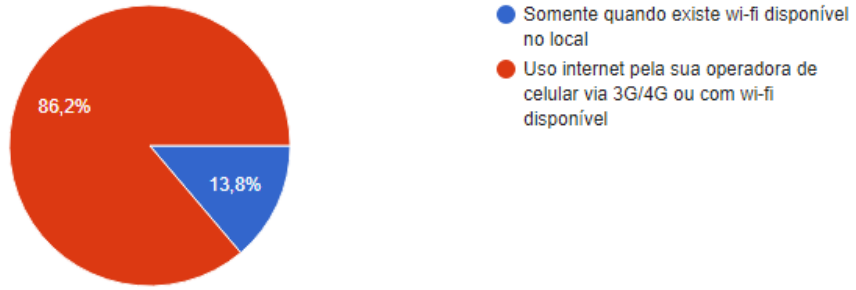
04 - Você possui e utiliza Smartphone?

29 respostas



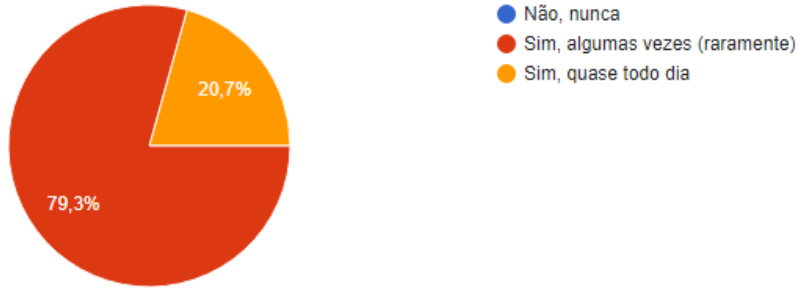
05 - Quando Você utiliza seu Smartphone conectado a internet ?

29 respostas



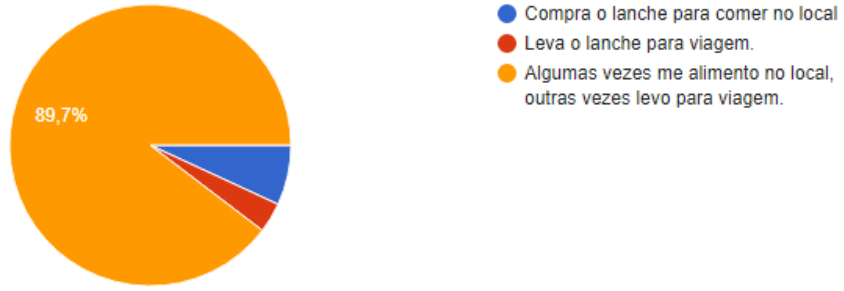
06 - Você costuma comprar lanches em lanchonetes?

29 respostas



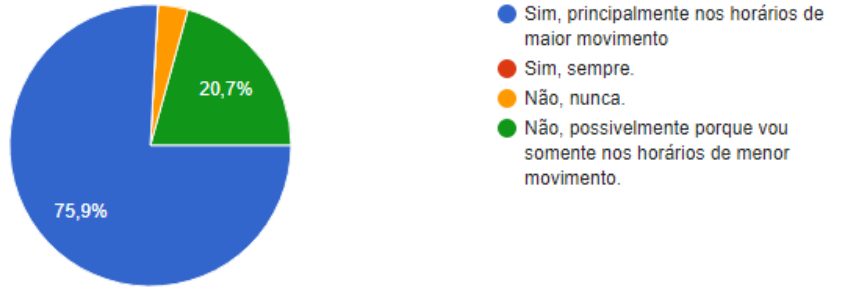
07 - Diga como utiliza a lanchonete?

29 respostas



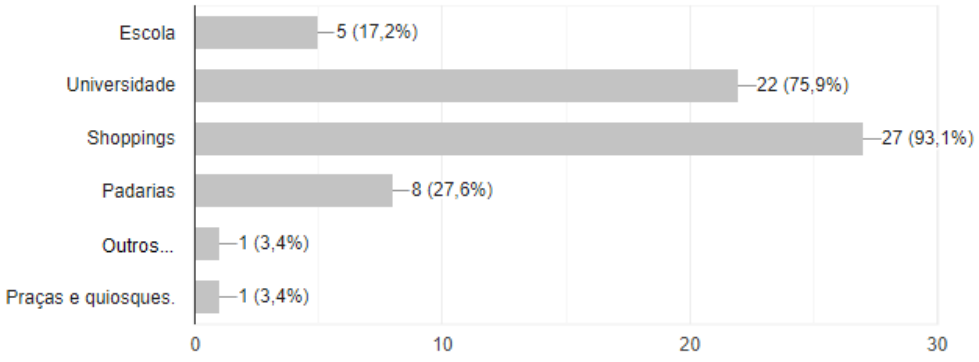
08 - Você quando vai a lanchonetes tem tido problema com filas nas compras de lanches?

29 respostas



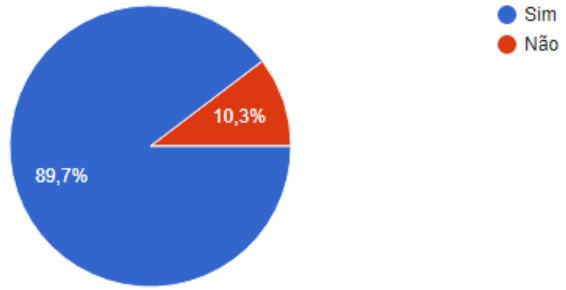
09 - Marque os locais das lanchonetes onde você encontra mais filas.

29 respostas



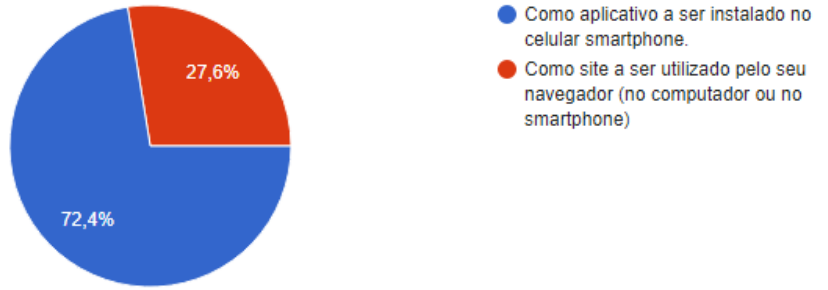
10 - Você aceitaria um aplicativo no seu Smartphone para amenizar seus problemas com filas?

29 respostas



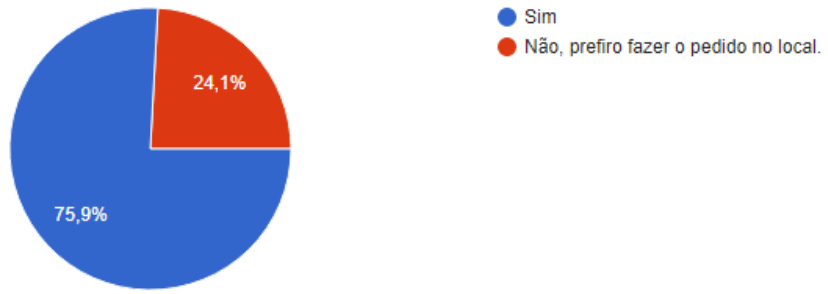
11 - Como você prefere utilizar a aplicação no seu Smartphone?

29 respostas



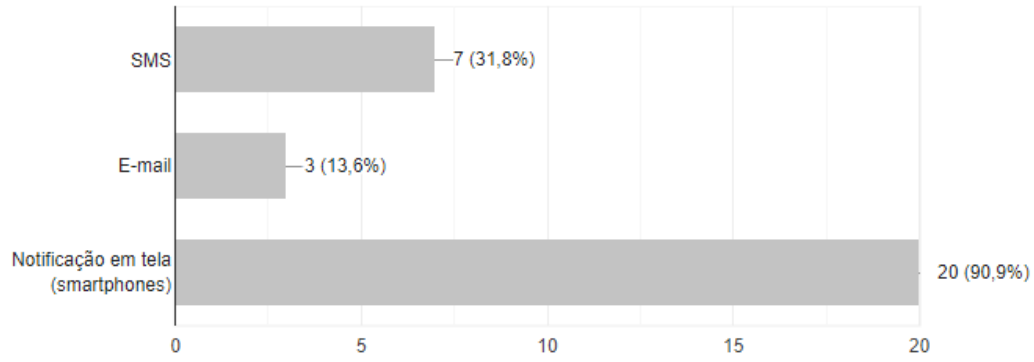
12 - Você gostaria de agendar seus pedidos de lanches via aplicativo em seu Smartphone?

29 respostas



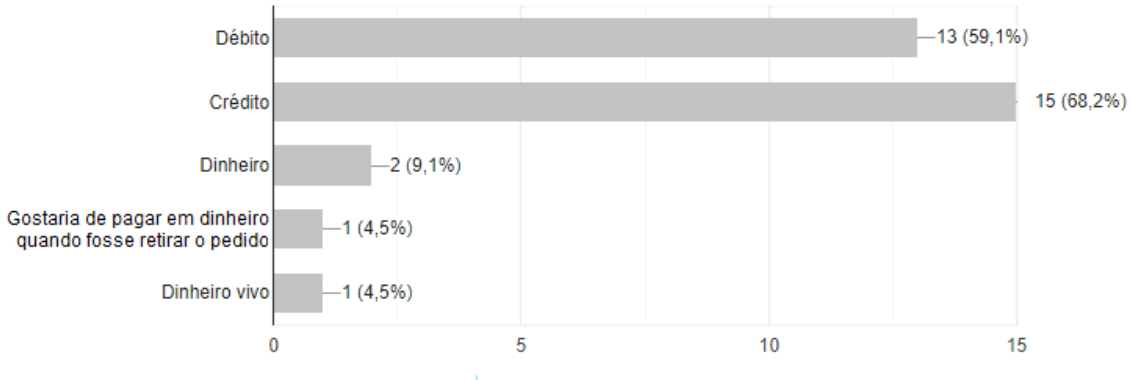
13 - Uma vez confirmado que seu pedido está pronto, por qual meio você gostaria de ser notificado?

22 respostas



14 - Quais os meios de pagamento você mais utilizaria para solicitar/agendar o pedido de um lanche por meio de uma aplicação digital como a que está sendo proposta?

22 respostas



Fonte: Pesquisa realizada pela equipe Time5 (2019).