Sistema NieX

Daniel Brito dos Santos

Ausberto S. Castro Vera

UENF - CCT - LCMAT - CC

2 de dezembro de 2022

UENF - UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO

CCT - CENTRO DE CIÊNCIA E TECNOLOGIA
LCMAT - LABORATÓRIO DE MATEMÁTICAS
CC - CURSO DE CIÊNCIA DA COMPUTAÇÃO

Sumário

1. Introdução

Paradigma de Desenvolvimento de Sistemas Orientado a Objetos é uma disciplina orientada a desenvolver um sistema utilizando a metodologia orientada a Objetos em todas as etapas do Ciclo de Vida de Desenvolvimento de um Sistema (CVDS). As referências bibliográficas básicas a serem consultadas são: [DWR14], [Hel13], [Gue11], [Som18] e [Waz11]. Como bibliografia complementar serão considerados: [SJB12], [SR12] e [Fur13].

Neste documento serão apresentadas as principais atividades realizadas para o desenvolvimento COMPLETO de uma aplicação OO.

O sistema a ser desenvolvido é o NieX, um sistema de ponto de venda (PDV) para supermercados, especialmente pensado para oferecer praticidade, confiabilidade e rapidez.

1.1 Escopo ou Contextualização do Sistema OO

O sistema proposto busca atender necessidades de grande pertinência de supermercados de modo que o seu desenvolvimento permita ao aluno praticar e aprofundar os conceitos do paradigma da programação orientada a objetos.

1.1.1 Desafios dos supermercados

A velocidade de atendimento é uma das características mais importantes para clientes e gestores de supermercados. Os maiores fatores que contribuem para essa velocidade é a eficiência dos sistemas de atendimento ao cliente. Assim, um sistema reduzido às funções mais fundamentais permite maior clareza de operação, maior velocidade de processamento e portanto maior satisfação das partes interessadas.

1.1.2 Escopo da disciplina

Além dos requisitos não funcionais mencionados, a possibilidade de ser flexível e resiliente é de grande valia em um sistema PDV. Portanto, propomos desenvolver um sistema web no qual cada operador de caixa irá interagir com uma interface cliente, e todas elas estarão conectadas com um servidor. Desse modo, a manutenção preventiva e criação de planos de contingência em casos de falha poderão ser concentrados de forma mais eficiente no servidor suas conexões; as máquinas clientes terão um sistema mais leve, e a escalabilidade também será facilitada.

Finalmente, a linguagem Ruby foi selecionada por ser uma linguagem puramente orientada a objetos muitas vezes comparada a Smalltalk, com recursos modernos, sintaxe simples e semântica transparente. Além de contar com o framewok "Rails"que facilita a estruturação dos componentes web, permitindo que o foco do desenvolvimento seja na modelagem do sistema, suas classes, objetos e comportamentos. [TFH04]

1.2 Objetivo do Sistema

O objetivo geral do sistema é oferecer uma interface simples e intuitiva para supermercados venderem de forma eficiente.

Nesse sentido, podemos enumerar os seguintes objetivos específicos:

1. Permitir que funcionários de diferentes cargos se cadastrem e entrem no sistema.
2. Oferecer uma tela para operadores de caixa registrarem vendas.
3. Oferecer uma tela para criação e modificação do cadastro de produtos apenas a funcionários autorizados para tal.

1.3 Justificativa

O sistema NieX foi proposto visando construir uma aplicação que propõem abordar as necessidades de supermercados bem atender os objetivos da disciplina. Assim, acreditamos que o projeto é pertinente ao contexto que se apresenta, tanto como uma prova de conceito, quanto um exercício teórico-conceitual, e ainda como uma proposta para mitigar um problema enfrentado por diversos trabalhadores e empresários de nosso país.

2. Requisitos do Sistema OO

Neste capítulo é apresentado listas, definições e especificações de Requisitos do sistema ser desenvolvido. Os requisitos são declarações abstratas de alto nível sobre os *serviços* que o sistema deve prestar à organização, e as *restrições* sobre as quais deve operar. Os requisitos sempre refletem as necessidades dos clientes do sistema.

Sobre os requisitos, Raul S. Wazlawick afirma:

> A *etapa de levantamento de requisitos* corresponde a buscar todas as informações possíveis sobre as funções que o sistema deve executar e as restrições sobre as quais o sistema deve operar. O produto dessa etapa será o documento de requisitos, principal componente do anteprojeto de software.
>
> A *etapa de análise de requisitos* serve para estruturar e detalhar os requisitos de forma que eles possam ser abordados na fase de elaboração para o desenvolvimento de outros elementos como casos de uso, classes e interfaces.
>
> O levantamento de requisitos é o processo de descobrir quais são as *funções* que o sistema deve realizar e quais são as *restrições* que existem sobre estas funções [Waz11].

2.1 Requisitos Funcionais

- Subsistema de login
  1. Verificar credenciais de usuário
  2. Gerar sessão de usuário de acordo com o seu cadastro
  3. Realizar login
  4. Realizar logout
  5. Alterar senha de usuário
- Subsistema de vendas
  1. Registrar produtos para compor uma venda
  2. Permitir a finalização de uma venda
  3. Contabilizar o subtotal da venda
  4. Permitir o pagamento da venda
  5. Armazenar cada venda e suas informações

- Subsistema cadastro de usuários
  1. Criar novo usuário
  2. Alterar cadastro de usuário
  3. Criar cargos
  4. Atribuir permissões à cargos
  5. Alterar cargo de usuário
- Subsistema de gerenciamento de produtos
  1. Permitir o cadastro e alteração de produtos em tela própria
  2. Permitir criação e alteração de produtos apenas aos usuários credenciados para tal
  3. Armazenar preço de venda, código de barras, descrição de produtos
  4. Atribuir código único para cada produto
  5. Permitir consulta de produtos

2.2 Requisitos Não-Funcionais

2.2.1 Requisitos de Usabilidade

1. A interface do sistema deve ser facilmente legível para o usuário
2. Todas as funcionalidades do sistema devem ter uso claro e consistente
3. Todo o sistema deve oferecer desempenho satisfatório aos seus operadores
4. Todos os aspectos estéticos do sistema devem seguir a identidade visual da empresa
5. O sistema não deve oferecer tempo de espera maior do que 2 segundos

2.2.2 Requisitos de Confiabilidade

1. Todas as falhas do sistema devem ser documentadas e direcionadas a equipe de manutenção do software
2. Todas as entradas do sistema devem ser validadas
3. O sistema deve oferecer feedback continuo aos usuários
4. Deve ser implementado um subsistema de backups para assegurar a permanência dos dados
5. O sistema deve ser tolerante a falhas

2.2.3 Requisitos de Disponibilidade

1. O sistema deve funcionar interrupto durante o horário comercial
2. Deve haver planos de contingência para um sistema emergencial funcionar caso o principal apresente alguma falha crítica
3. O servidor do sistema deve ser capaz de servir até 20 unidades clientes simultâneas.
4. O sistema deve oferecer a possibilidade de servidor local ou remoto
5. O sistema deve ser tolerante a falhas de internet

2.2.4 Requisitos de Privacidade

1. Todos os dados de usuários devem armazenados de forma criptografada
2. O fornecimento de qualquer dado pessoal adicional será opcional
3. Apenas o sistema deve ter acesso aos dados armazenados
4. Todas as operações do sistema devem estar de acordo com a LGPD
5. Apenas gerente deve ter acesso aos dados dos funcionários no sistema

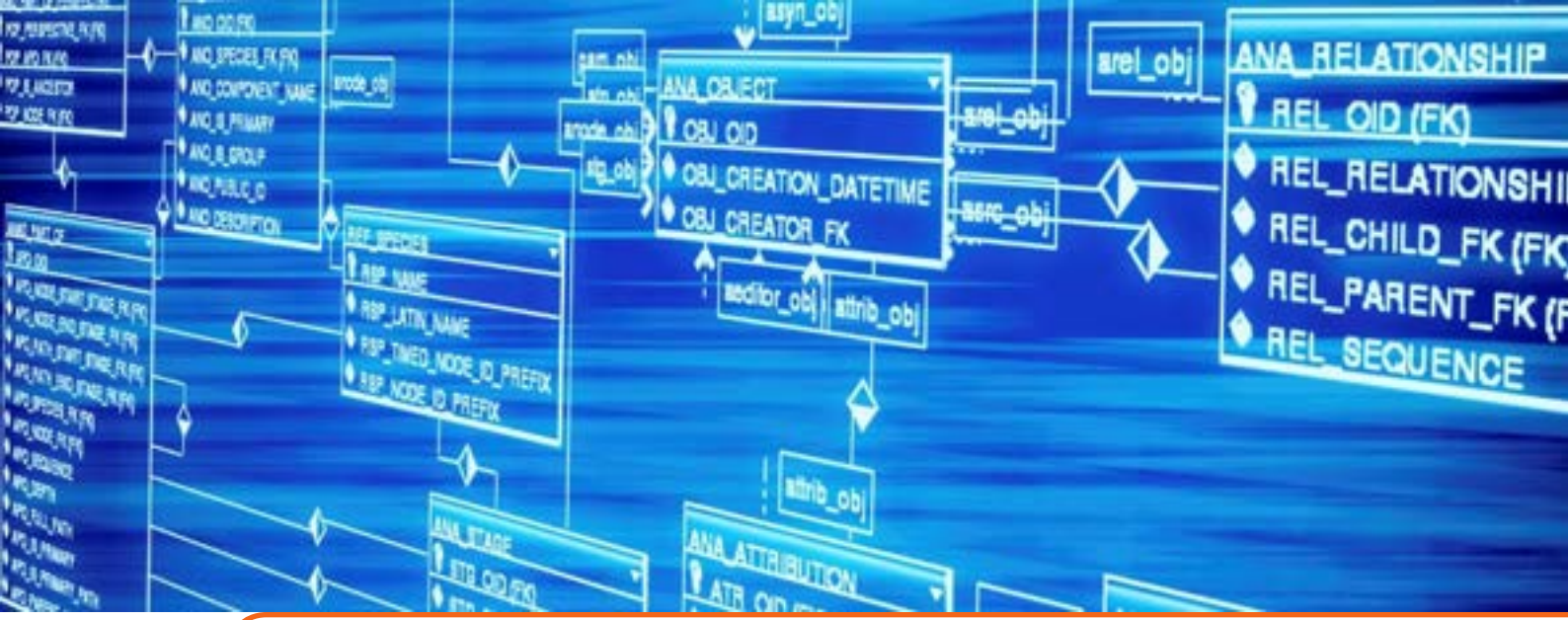
2.2.5 Requisitos de Acesso

1. Apenas usuários cadastrados devem ser direcionados ao sistema
2. Usuários não cadastrados podem visualizar a tela de login e a tela "Sobre"
3. Todas as funcionalidades do sistema devem ser restritas aos cargos correspondentes
4. Uma vez autenticado cada usuário será direcionado a sua respectiva tela

2.3 Requisitos de Negócios

Requisitos do negócio são requisitos de alto nível que explicam e justificar qualquer projeto. Os requisitos de negócios são as atividades críticas de uma empresa que devem ser executadas para atender ao(s) objetivo(s) organizacional(is) enquanto permanecem independentes do sistema solução.

1. Reduzir as vendas processadas erroneamente em 30% até o final do ano.
2. Incrementar o número de atendimentos a clientes em 5% cada mês.
3. Diminuir em 30% o tempo médio de espera dos clientes em fila a partir do terceiro mês de implementação.
4. Aumentar a satisfação dos clientes em pelo menos 10% no sexto mês após a implementação.
5. A implementação e treinamento dos operadores deve durar menos de dois meses.



3. Modelagem do Sistema

Neste capítulo apresentamos a representação gráfica da arquitetura do sistema. Nesse sentido, utilizaremos diagramas de fluxo de dados, entidades e seus relacionamentos, classes do sistema, casos de uso, sequências de processos, atividades e estados do sistema.

3.1 Diagramas DFD

Diagramas de Fluxo de Dados apresentam os processos, entidades e a forma como os dados fluem em determinado sistema. Aqui apresentamos o diagrama geral de contexto (figura 3.1), o diagrama nivel 1 com todos os subsistemas (3.2), e em detalhes o subsistema de vendas chamado Sale (3.3).

- Diagrama de Contexto (Mostra o relacionamento e fluxo de dados entre o sistema e as entidades externas)

Figura 3.1: Diagrama de contexto

- Nível 1 do Sistema (O sistema como um todo junto com seus subsistemas)

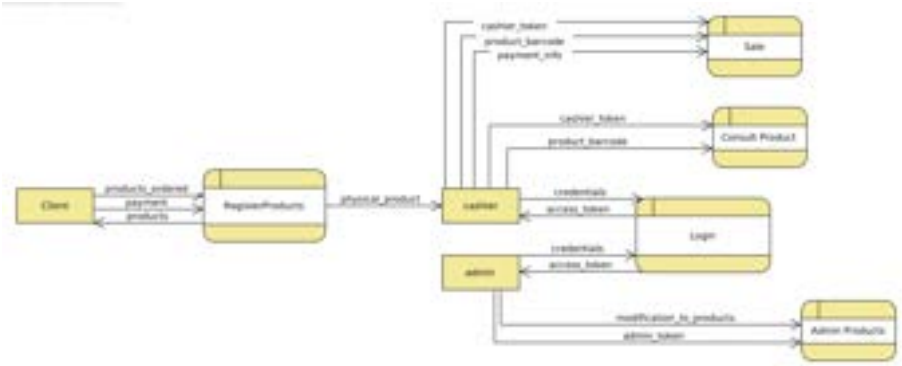


Figura 3.2: DFD dos subsistemas

- Nível 2

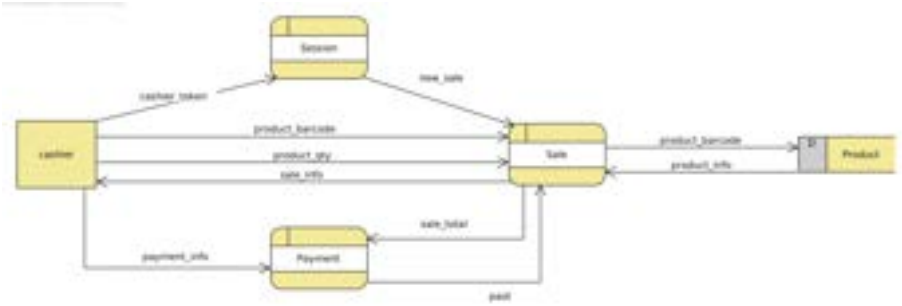


Figura 3.3: DFD do subsistema Sale

3.2 Diagramas E-R

Diagramas Entidade Relacionamento modelam a arquitetura de um banco de dados. Desse modo, na figura 3.4 temos as entidades que populam nossas tabelas internas bem como as relações entre cada uma delas, seus atributos e tipos de dados.

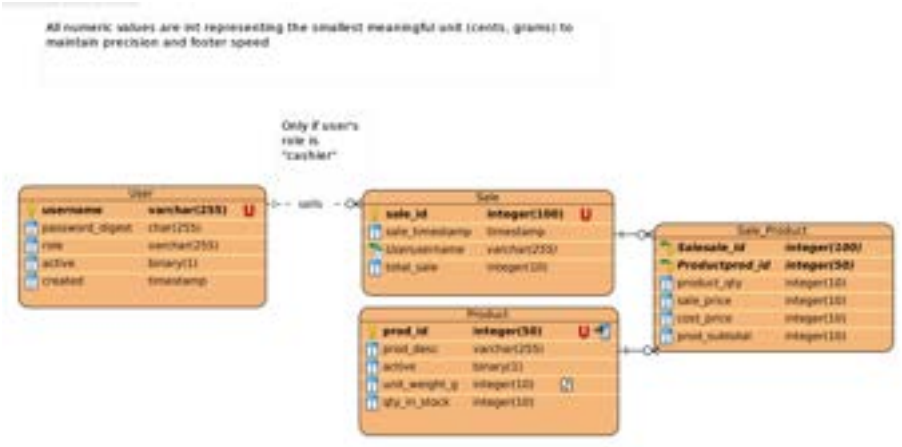


Figura 3.4: Diagrama Entidade-Relacionamento do sistema

3.3 Diagramas de Classes

Temos, na figura 3.5, as classes do sistema, seus atributos e métodos.

Figura 3.5: Diagrama de classes

3.4 Diagramas Casos de uso

A figura 3.6 demonstra os diferentes casos do uso do programa.

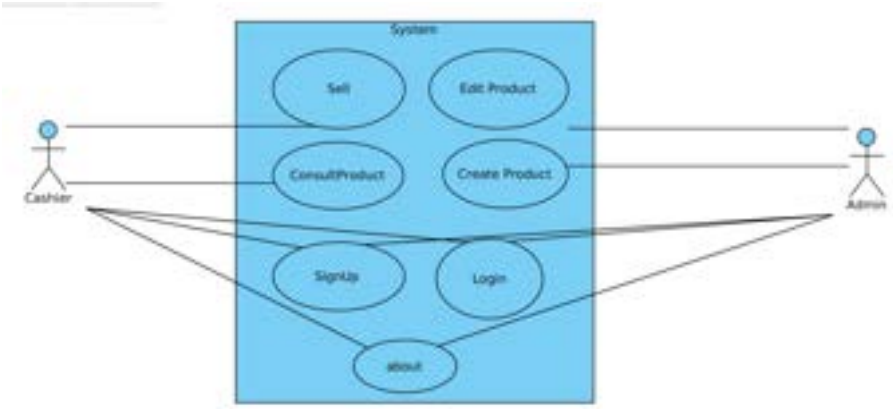


Figura 3.6: Diagrama de casos de uso

3.5 Diagramas Sequência

Como o nome sugere, diagramas de sequência apresentam a ordem que diferentes componentes do sistema são acionados e suas mensagens para atingir determinado objetivo.

3.5.1 Diagrama de sequência da página inicial

Na figura 3.7 demonstramos como há complexidade até mesmo para abrir a página inicial do programa. Podemos observar que ao entrar com o endereço do programa no navegador ele faz uma requisição do tipo GET para a página raiz. Essa requisição é direcionada pelo componente das rotas para o método "index"da classe "MainController". Esta classe por sua vez busca e executa a

View correspondente (Main/index.html.erb) que enfim retorna o html a ser retornado como resposta a requsição inicial do navegador.

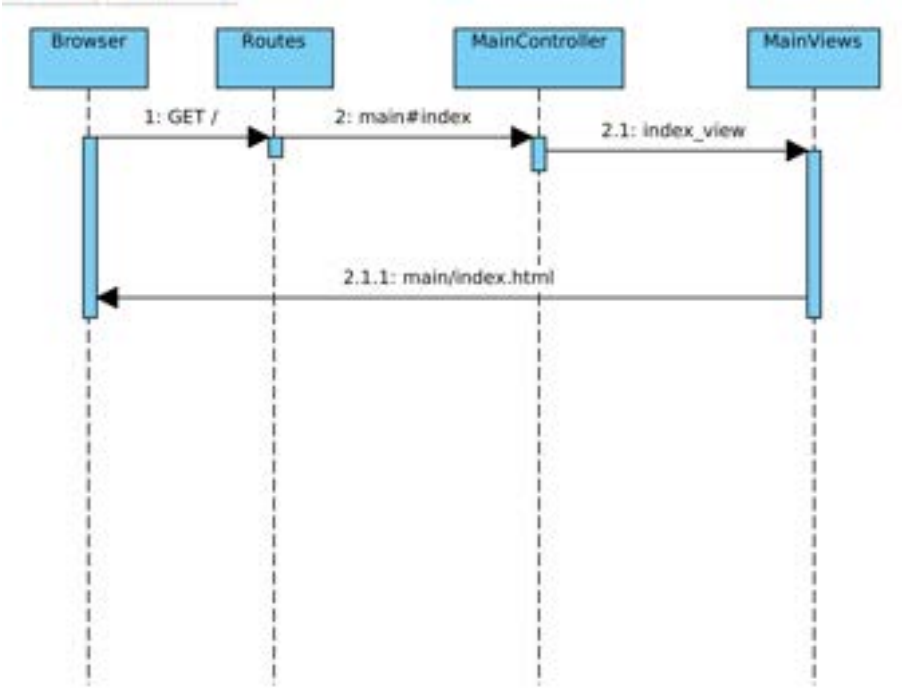


Figura 3.7: Diagrama de sequência da página principal

3.5.2 Diagrama de sequência da página about

Na figura 3.8 podemos observar a sequência necessária para apresentar a página "about"para o usuário. Mais uma vez temos toda a mensageria da página inicial, e ainda a solicitação da página about, seguida da resposta do arquivo html. Podemos ver como a complexidade cresce com já nessa que é a página mais simples do sistema.

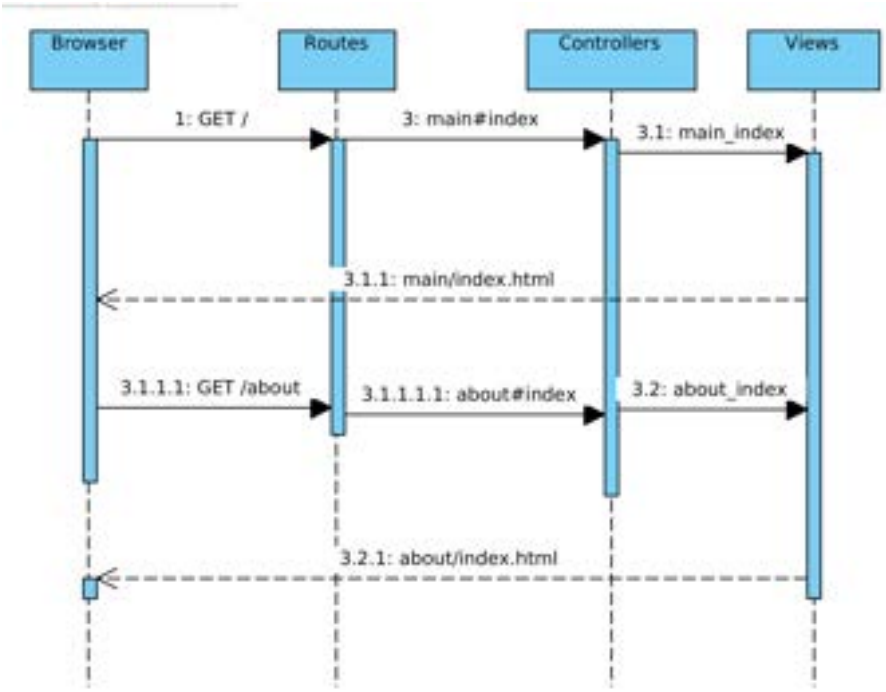


Figura 3.8: Diagrama de sequência da página about

3.5.3 Diagrama de sequência de login

Para efetuar o login, temos que o navegador primeiramente requisita a página raiz. Na página de resposta com os hiperlinks para login, registro e sobre ele seleciona o primeiro.

Assim temos uma segunda requisição, que é novamente direcionada pelo "Routes"até o método da classe de controle adequada, no caso o método "new"da classe "Sessions".

Esse método apenas direciona o sistema para processar a sua respectiva View (sessions/-new.html.erb), que gera nada mais que um formulário html para receber as informações de login.

Esse formulário é devolvido a requisição original de login do navegador, e uma vez preenchido suas respostas são enviadas ao programa com o verbo POST.

Mais uma vez a rota estabelescida define o método que cuidará da requisição. Nesse caso será o método create da classe SessionController.

Este sim, de posse dos dados, utiliza o Model "Product"para buscar no banco de dados o usuário que está tentando logar.

Uma vez encontrado o usuário, o modelo devolve a sessão criada ao controller que por sua vez gera um novo html para o usuário, dessa vez redirecionando-o com um novo GET para sua respectiva tela. Ou seja, caso o usuário seja um caixa, ele será direcionado para a tela de vendas, e caso seja administrador ele será direcionado para a tela de gestão de produtos.

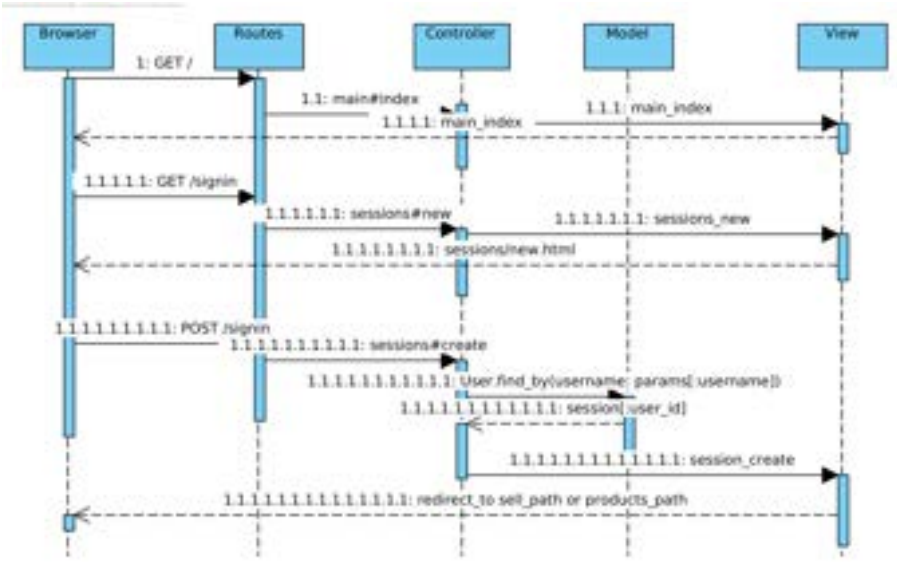


Figura 3.9: Diagrama de sequência de login

3.6 Diagramas de Atividades

3.6.1 Editar produto

O diagrama da figura 3.10 apresenta as ações necessárias para editar um produto do cadastro. Nesse caso se inicia na tela inicial, efetua-se o login, e caso o usuário seja administrador ele será apresentado com o índice dos produtos cadastrados para selecionar o que deseja editar. Em seguida lhe será apresentada uma tela de edição onde poderá alterar a informação desejada e enfim confirmar ou não as alterações feitas.

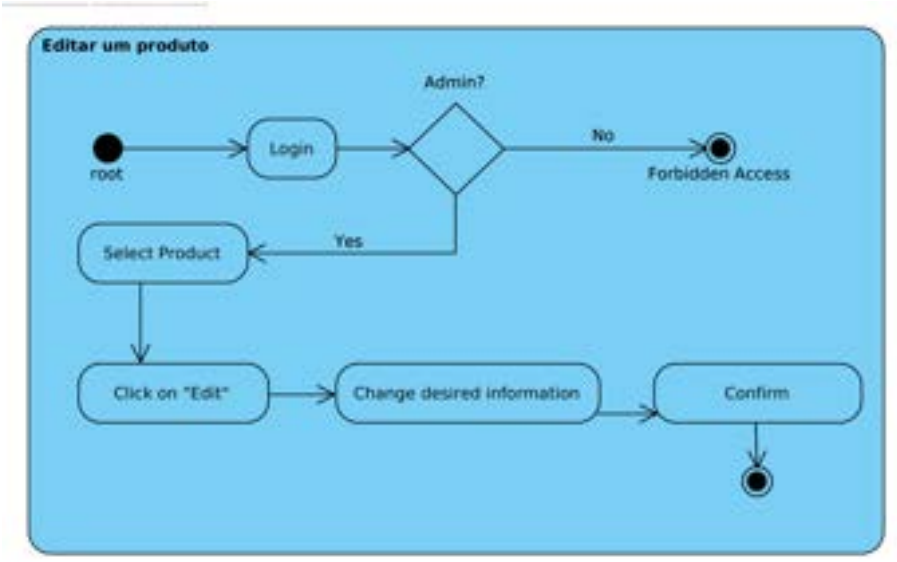


Figura 3.10: Diagrama de Atividade para editar um produto

3.6.2 Efetuar uma venda

Na figura 3.11 vemos as ações necessárias para se efetuar uma venda. Semelhante a atividade anterior temos o login, que aqui já assumimos ser corretamente de um caixa e portanto ele será imediatamente direcionado para a tela de vendas. Na tela de vendas ele poderá registrar quantos produtos forem necessários e a qualquer momento pode cancelar ou finalizar a venda.

Imediatamente após finalizar ou cancelar a venda, o interface já está pronto para iniciar a próxima, nesse caso o usuário tem a opção de encerrar a atividade ou continuar vendendo.

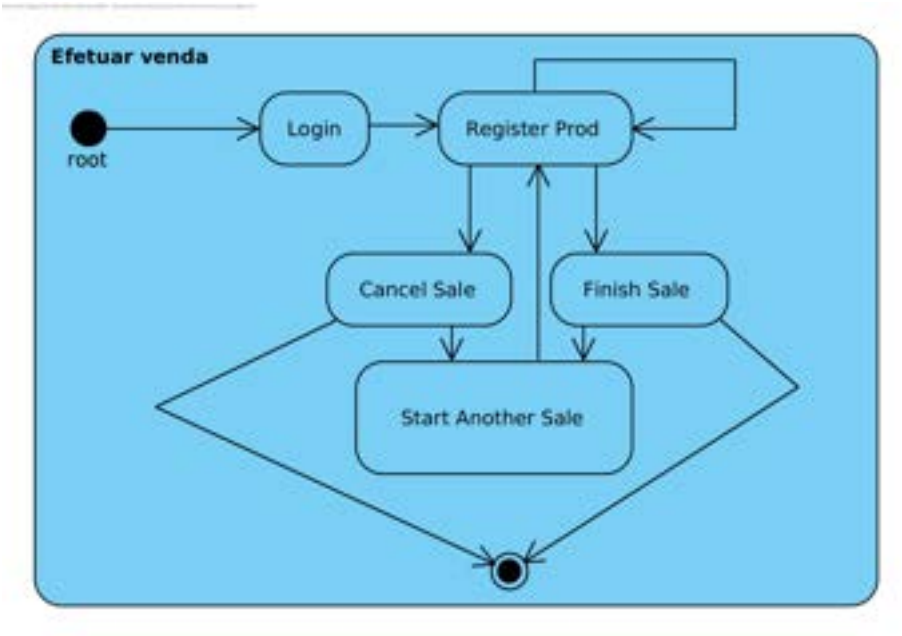


Figura 3.11: Diagrama de Atividade para efetuar uma venda

3.6.3 Cadastrar usuário

O cadastro de usuário segue uma sequencia parecida de ações. Da tela inicial o usuário seleciona a tela de registro, na tela de registro ele preenche seus dados, seleciona o seu cargo e confirma para concluir seu cadastro.

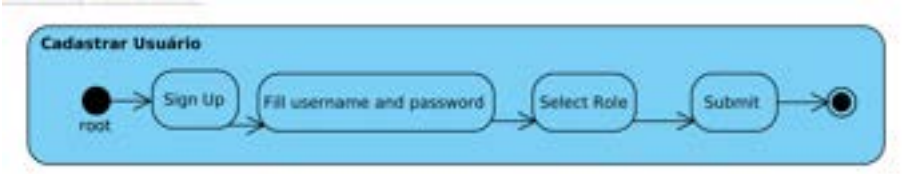


Figura 3.12: Diagrama de Atividade para cadastrar um usuário

3.7 Diagramas Estado

3.7.1 Editar produto

Na figura 3.13 podemos observar os estados envolvidos na edição de um produto. A partir da página inicial deve-se estabelescer uma sessão por meio do login ou do registro de um novo usuário. Desde que o usuário seja administrator ele será redirecionado para a tela de produtos. O estado seguinte é o produto selecionado, que leva ao estado de buscar e adquirir seus dados, que por sua vez podem ser modificados pelo usuário e então salvos.

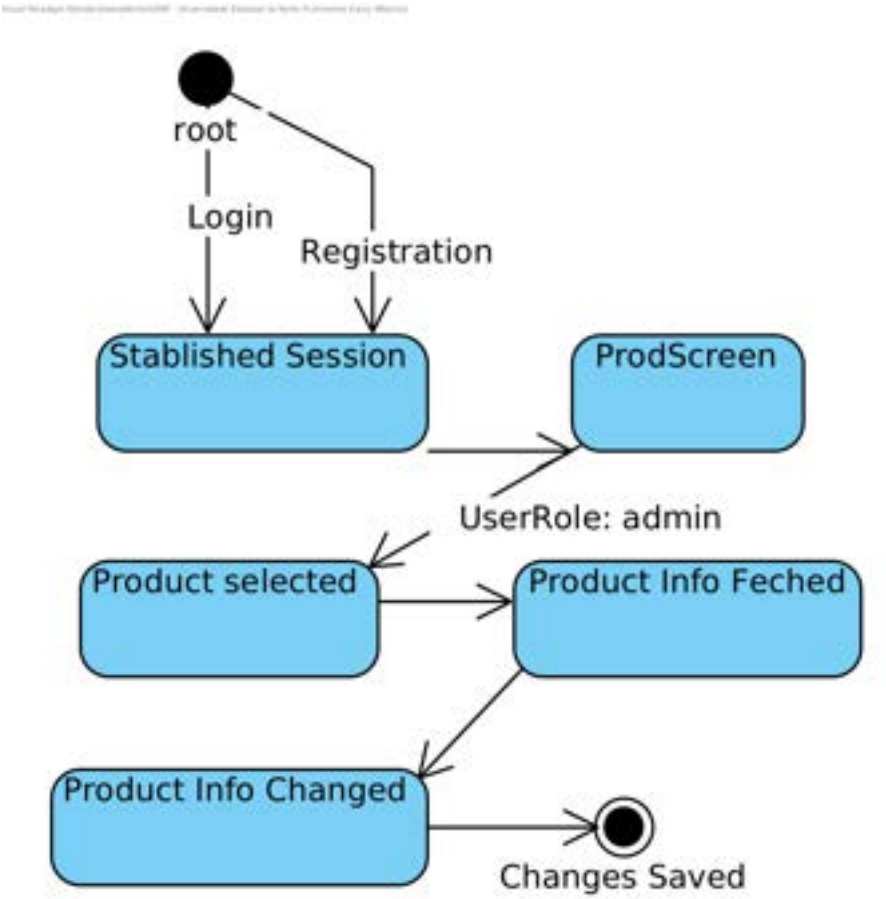


Figura 3.13: Diagrama de Atividade para editar um produto

3.7.2 Efetuar uma venda

A máquina de estados de vendas (Figura 3.14) é muito semelhante ao a sequência de ações no diagrama de atividades. A partir da tela de venda o carrinho se inicia vazio com o subtotal de zero. Esse estado pode ser final agora ao não efetuar venda nenhuma, ou pode transitar com a adição do produto para o próximo estado de um produto a mais no carrinho e o seu acréscimo no subtotal, indefinidamente até que a venda seja encerrada. Ao ser encerrada ela pode ser ou não salva no banco de dados, e de qualquer forma o próximo estado é novamente da tela de venda inicial.

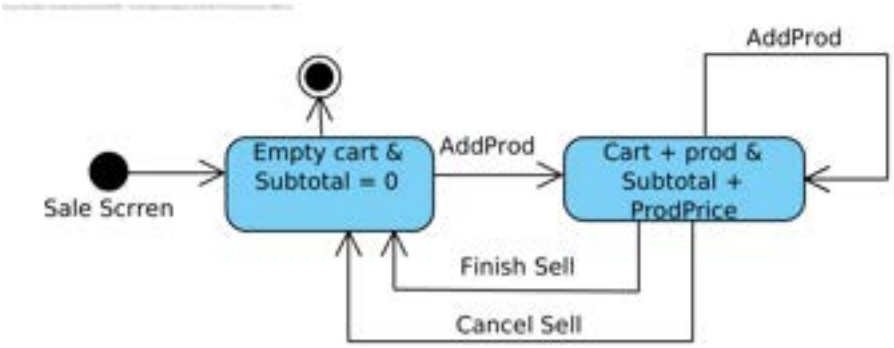


Figura 3.14: Diagrama de Atividade para editar um produto

3.7.3 Cadastrar usuário

Os estados que compreendem o cadastro de usuário são apresentados na Figura 3.15. O usuário transita da página inicial para a página de registro com um clique, onde ele se depara com um formulário em branco. Que ao ser preenchido transita para o estado preenchido, bem como o cargo transita de vazio para selecionado, e o próximo estado ocorre quando o cadastro é confirmado.

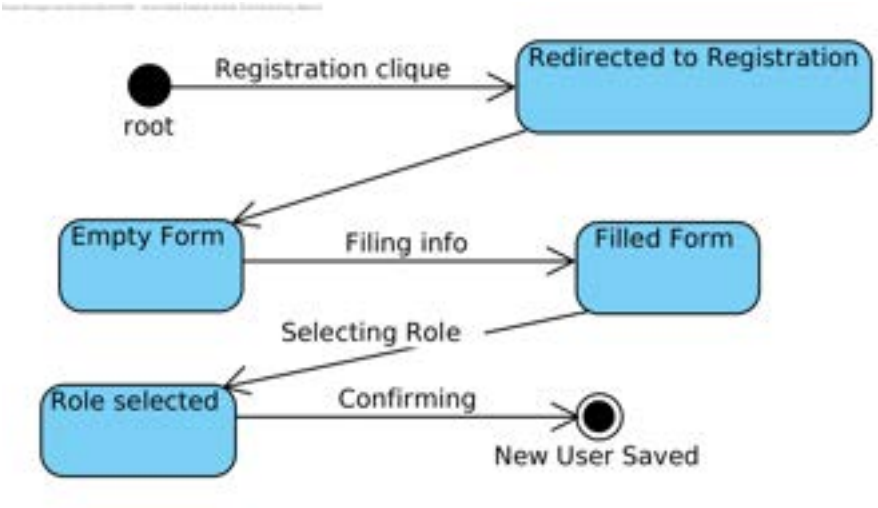


Figura 3.15: Diagrama de Atividade para editar um produto

4. Projeto do Sistema OO

Neste capítulo apresentamos os componentes em diagramas que representem sua arquitetura, bem como imagens com as diferentes interfaces do sistema.

Um Projeto Orientado a Objetos é um processo por meio do qual um conjunto de modelos de projeto orientados a objetos são construídos pra posteriormente, ser utilizados por programadores para escrever e testar o novo sistema sendo desenvolvido [SJB12].

4.1 Arquitetura

4.1.1 Arquitetura do sistema

Na figura 4.1 apresentamos os componentes do sistema. Nela podemos observar tem uma interface com três possíveis entradas: login de usuários cadastrados, registro de novos usuários ou a consulta Sobre a aplicação.

As duas primeiras entradas são ficam a cargo do controlador de sessões, de modo que só a partir de uma sessão válida o usuário tem acesso ao sistema.

A sessão pode ser de dois tipos de acordo com o cargo do usuário: administração ou caixa. Assim, imediatamente após a sessão o usuário é direcionado a sua respectiva interface adequada.

Caso o usuário seja administrador ele será direcionado ao cadastro de produtos onde pode visualizar, cadastrar e alterar qualquer item. Já caso o usuário seja caixa, ele será direcionado a tela de vendas, onde poderá apenas digitar ou escanear o código de produtos e a sua quantidade, confirmar ou cancelar a venda em questão e nada mais. Ambos os usuários além de suas respectivas funções podem apenas encerrar suas sessões.

Assim, asseguramos a segurança do sistema e a clareza de seu uso.

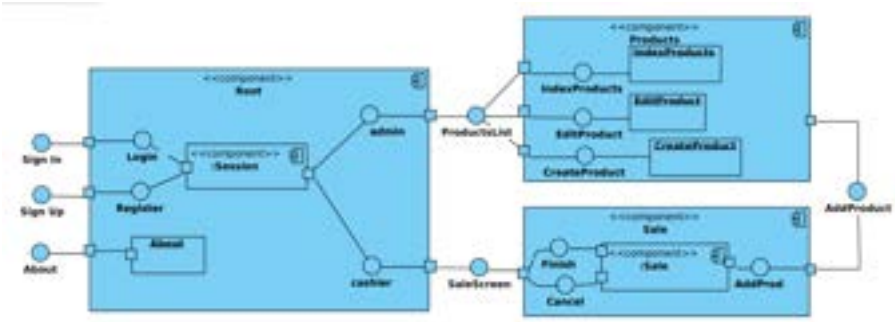


Figura 4.1: Diagrama De Componentes da Arquitetura do Sistema

Mais especificamente temos os seguintes componentes e suas respectivas responsabilidades:

- **Root**: Abarca as primeiras interfaces do sistema, apresentando a tela inicial e os três possíveis fluxos de uso.
- :**Session**: responsável por receber cadastrar as informações de usuário fornecidas no banco de dados (registro) ou autentica-las com o banco (login). Apos a autenticação, esse componente gera e encripta o token de sessão utilizado nas funções seguintes.
- **Products**: composto por controladores para as funções de apresentar a lista de produtos cadastrados e as funções de criar e edita-los. Também é a abstração que se encarrega modelar e comunicarse com os elementos da tabela Products no banco de dados.
- **Sale**: componente no qual ocorrem as vendas em si, com componentes visuais de input para receber um código de barras, que é direcionado pela interface AddProd para buscar no banco de dados dos produtos e retornar com suas informações. Além de armazenar as respectivas quantidades e valor final de cada linha. Bem como registrar a venda ou cancelá-la.

4.1.2 Arquitetura do subsistema de Produtos

Na figura 4.2 podemos ver os componentes do subsistema de Produtos. Nele, a partir do token adequado o usuário pode consultar a lista de produtos disponíveis, criar um novo produto, editar um existente ou recuperar as informações a partir de seu código de barras.

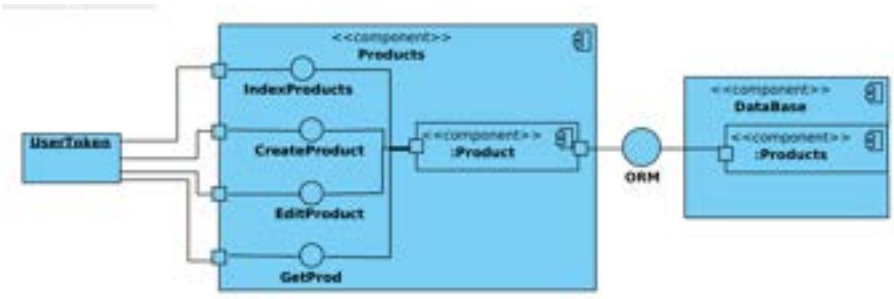


Figura 4.2: Diagrama De Componentes da Arquitetura do Sistema

O **UserToken** é gerado pelo subsistema de sessão de usuário. A partir dele é possível acessar as interfaces referentes a cada função sobre os produtos:

- **IndexProducts**: Se encarrega de buscar todos os registros de produtos cadastrados no sistema.

- **CreateProduct**: Cria uma nova entrada de produto e a salva no banco de dados.
- **EditProduct**: Busca a tupla relacionada ao ID de determinado produto para permitir que esses dados sejam alterados e salvos sobre os anteriores.
- **GetProd**: Busca os dados de um produto a partir de seu código de barras. É utilizado no contexto de vendas para adicionar o tal produto a venda em questão.
- **:Product**: Representa o Modelo Rails, isto é, o objeto que encapsula a comunicação com o ORM que gerencia o banco de dados do sistema.
- **Products (table)**: Tabela no banco de dados que de fato armazena os dados de cada produto cadastrado no sistema.

4.2 Interfaces do Usuário

Nesta seção apresentamos as telas do sistema seguido de seus códigos fontes bem como uma breve explicação de seus principais mecanismos e componentes.

4.2.1 Tela Inicial

A tela inicial é apresentada na figura 4.3. Logo em seguida apresentamos o seu código fonte. Nele podemos ver o uso das classes Bootstrap para gerar os componentes gráficos e dinâmicos sobre um HTML simples. Cada um dos botões redireciona para sua respectiva rota na qual a funcionalidade será servida.

Figura 4.3: Tela inicial

```
<body class="d-flex h-100 text-center text-bg">

  <main class="px-3">
    <div class="mx-auto pb-5 mt-5">
    <img src="/Niex_logo.png" alt="Niex logo" width="150"/>
    </div>

    <div class="d-grid gap-2 col-6 mx-auto">
        <a class="btn btn-outline-primary mb-1" aria-current="page" href="/
    sign_in">Sign In</a>

```

```
        <a class="btn btn-outline-primary mb-1" aria-current="page" href="/sign_up">Sign Up</a>

        <a class="btn btn-outline-secondary mb-1" aria-current="page" href="/about">About</a>
    </div>
  </main>
</body>
```

4.2.2 Tela Sobre

A figura 4.4 apresenta a tela "Sobre"onde apresentamos o aplicativo e as suas principais informações.

Figura 4.4: Tela Sobre

```
<body class="d-flex h-100 text-center text-bg">

  <main class="px-3">
    <div class="mx-auto pb-5 mt-5">
    <a href="/">
    <img src="/Niex_logo.png" alt="Niex logo" width="150"/>
    </a>
    </div>

    <p class="text-start d-grid col-8 mb-3 mx-auto">
Niex    um sistema web de ponto de vendas (PDV). Desenvolvido no contexto
    da disciplina <u>Programa  o Orientada a Objetos para Desenvolvimento
     de software (POODS)</u> ministrada pelo professor Ausberto S. Castro
    Vera.  Buscamos aplicar os conceitos da POO em um sistema an logo a
    sistemas utilizados em supermercados de fato. Buscamos oferecer uma
    experi ncia de usu rio simples e intuitiva que ao mesmo tempo
    proponha uma prova de conceito funcional, e demonstre na pr tica os
    conceitos vistos em sala.

    </p>
    <div>
        <img src="/logo-uenf.png" alt="Logo UENF" width="150"/>
    </div>
    <p>
```

```
18      UENF - Universidade Estadual do Norte Fluminense Darcy Ribeiro <br>
19  CCT - Centro de Ci ncia e Tecnologia <br>
20  LCMAT - Laborat rio de Ci ncias Matem ticas <br>
21  CC - Bacharelado em Ci ncia da Computa  o
22      </p>
23          <img src="/logo-cc.png" alt="Logo CC" width="150"/>
24
25      <p class="text-secondary">
26      Desenvolvido por <a href="github.com/dbs-97"> Daniel Brito dos Santos</
        a> em 2022.
27      </p>
28
29
30    </main>
31  </body>
```

4.2.3 Tela de Login

Nesta seção apresentamos a tela de login na figura 4.5, e seu código fonte logo em seguida. Nele podemos ver o html que é enviado ao navegador quando o usuário acessa esse endpoint. Bem como o código em Ruby responsável por lidar com sua requisição. Assim, temos a criação de um formulário com os campos "username"e "password".

Ao receber esses dados o método "create"do controlador de sessões busca um usuário correspondente no banco de dados por meio do objeto "User". Caso o usuário exista e a senha esteja correta, será criada uma sessão e o cargo do usuário será utilizado para redirecioná-lo adequadamente. Isto é, para a tela de vendas se for caixa ou para tela de produtos caso seja administrador.

Figura 4.5: Tela inicial

```
1  <body class="d-grid gap-2 col-3 mx-auto align-items-center">
2
3  <h1>Sign In</h1>
4
5  <%=form_with url: sign_in_path do |form| %>
6      <div class="mb-3">
7      <%= form.label :username %>
8      <%= form.text_field :username, class:"form-control" %>
```

```
    </div>

    <div class="mb-3">
    <%= form.label :password %>
    <%= form.password_field :password, class:"form-control" %>
    </div>

    <div class="col-sm-12 text-center">
    <%= form.submit :Submit, class: 'btn btn-primary'%>
     <a class="btn btn-outline-primary" aria-current="page" href="/">Back</a>
    </div>
<%end%>

</body>
```

```
class SessionsController < ApplicationController
    def new
    end

    def create
        user = User.find_by(username: params[:username])

        if user.present? && user.authenticate(params[:password])
            session[:user_id] = user.id
            if user.role == "admin"
                redirect_to products_path
            else
                redirect_to sell_path
            end
        else
            flash[:alert] = "Invalid username or password"
            render :new, status: :unprocessable_entity
        end
    end

    def destroy
        session[:user_id] = nil
        redirect_to root_path, notice: "Logged out"
    end
end
```

4.2.4 Tela de Registro de usuário

Assim como na tela de login, podemos ver os campos para o usuário inserir o seu "Username"e a sua senha. Nesse caso temos ainda os campos para selecionar o cargo do usuário na empresa e confirmar a senha para se certificar que não houve nenhum erro de digitação.

Também de modo análogo ao "login"o usuário cadastrado será prontamente redirecionado a sua página correspondente.

Figura 4.6: Tela login

```
<body class="d-grid gap-2 col-3 mx-auto align-items-center">

<h1 class="px-10" >User registration</h1>
<%= form_with model: @user, url: sign_up_path do |form| %>
    <% if @user.errors.any?%>
    <div class="alert alert-danger">
        <% @user.errors.full_messages.each do |message| %>
            <div> <%=message%> </div>
        <%end%>
    </div>
    <%end%>

    <div class="mb-3">
    <%= form.label :username %>
    <%= form.text_field :username, class:"form-control" %>
    </div>



    <%= form.select :role, ['cashier', 'admin']%>

    <div class="mb-3">
    <%= form.label :password %>
    <%= form.password_field :password, class:"form-control" %>
    </div>

    <div class="mb-3">
    <%= form.label :password_confirmation%>
    <%= form.password_field :password_confirmation, class:"form-control" %>
    </div>

    <div class="col-sm-12 text-center">
    <%= form.submit :Submit, class: 'btn btn-primary'%>
     <a class="btn btn-outline-primary" aria-current="page" href="/">Back</
    a>
    </div>
<% end %>

</body>
```

```
class UsersController < ApplicationController
    def new
        @user = User.new
    end
    def create
        @user = User.new(user_params)
        if @user.save
            session[:user_id] = @user.id
            if @user.role == "admin"
                redirect_to products_path, notice: "Sucessfully created admin user"
            else
                redirect_to sell_path, notice: "Sucessfully created cashier user"
            end
        else
            flash[:alert] = "Something went wrong"
            render :new
        end

        #  render plain: "Thanks"
    end
    def user_params
        params.require(:user).permit(:username, :role, :password, :password_confirmation)
    end
end
```

4.2.5 Tela de Vendas

A figura 4.7 apresenta a tela de venda recém iniciada. Podemos observar que o total da compra é 0 e não há produtos registrados. Também podemos ver as opões disponíveis ao caixa: finalizar a compra, cancelar a compra ou finaliza-la. Principalmente o campo superior para entrada de dados que é automaticamente selecionado para registro rápido dos códigos de barras dos produtos conforme demonstrado na figura 4.8.

Nesta tela temos as classes SalesController para lidar com o fluxo de dados da funcionalidade e a Cart para representar o carrinho de compras do cliente. Assim, após a validação do usuário uma nova venda é iniciada, e a cada novo código de barras inserido o método search processa a entrada de dados e direciona para a classe Cart que se encarrega de adicionar o produto ao carrinho de compras e efetuar as mudanças necessárias no subtotal e variáveis correspondentes. Logo em seguida a tela de vendas é recarregada com os produtos adicionados sendo apresentados.

O caixa pode ainda finalizar a compra, o que faz com que a mesma seja salva. Ou cancelar a compra, o que apenas reseta as variáveis o subtotal e os produtos registrados.

Figura 4.7: Tela de vendas vazia

Figura 4.8: Tela de vendas com produtos

```
<nav class="navbar navbar-light bg-light">
    <a class="navbar-brand">
    <img src="/Niex_logo.png"  height="30" alt="">
  </a>
  <a class="navbar-brand">Sales Screen</a>
  <div class="my-2 my-sm-0">
    <%= form_for sell_path, method: :post do |f|%>
      <p>
      <%= f.text_field :query, autofocus: true %>
      <%f.submit%>
      </p>
    <% end %>
  </div>
<div class="row">
  <div class="col">
```

```
    <%= button_to "Finish sale", sell_path, method: :get, class: "btn btn-outline-primary my-2 my-sm-0"%>
  </div>
  <div class="col">
    <%= button_to "Cancel sale", sale_path, method: :delete, class: "btn btn-outline-danger my-2 my-sm-0"%>
  </div>
</div>
</nav>



<h2>Total: <%=@total%></h2>

<div>
  <div class="table-wrapper-scroll-y">

  <table class="table table-bordered table-striped mb-0">
    <thead>
      <tr>
        <th scope="col">#</th>
        <th scope="col">Quantity</th>
        <th scope="col">Barcode</th>
        <th scope="col">Description</th>
        <th scope="col">Unit Price</th>
        <th scope="col">Total Price</th>

      </tr>
    </thead>

  <tbody>
    <% @sale_products.last(5).each do |p| %>
    <tr>
      <th scope="row"> <%= p[:prod_count] %> </th>
      <td>  <%= p[:prod_quantity] %>  </td>
      <td> <%= p[:prod_code] %> </td>
      <td> <%= p[:prod_name] %> </td>
      <td> <%= p[:prod_price] %> </td>
      <td> <%= p[:prod_total] %> </td>
    </tr>
    <% end%>
  </tbody>
  </table>
</div>

    <%# <%= button_to "Finish sale", sell_path, method: :get, class: "btn btn-outline-primary mt-2"%>
    <%# <%= button_to "Cancel sale", sale_path, method: :delete, class: "btn btn-outline-danger mt-2"%>

<footer class="footer">
<%= "----" * 36%>
    <% if @user %>
       <%= button_to "Logout", logout_path, method: :delete, class: "btn btn-outline-danger mt-2"%>
       Logged in as <%= @user.username %>. Role: <%= @user.role%>
    <% end %>
</footer>
```

```
class SalesController < ApplicationController
    def index
```

```
3         if session[:user_id]
4             @user = User.find(session[:user_id])
5             @sales = Cart.fetch_sales()
6         end
7     end
8
9     def create
10        if session[:user_id]
11            @user = User.find(session[:user_id])
12            Cart.save(@user.username)
13            redirect_to sale_url
14        end
15    end
16
17    def new
18        if session[:user_id]
19            @user = User.find(session[:user_id])
20            if not @total
21                Cart.build_inventory
22            end
23            @sale_products = Cart.show_products
24            @total = Cart.show_total
25
26        end
27    end
28
29    def search
30        query = params["/sell"]["query"]
31        quantity, prod_code = treat_query(query)
32        Cart.new_product(quantity, prod_code)
33        redirect_to sale_url
34    end
35    def destroy
36        Cart.destroy
37        redirect_to sale_url
38    end
39
40    private
41     def treat_query(query)
42         split_query = query.split("*")
43             if split_query.count == 2
44                     quantity, prod_code = split_query
45                     return [quantity, prod_code]
46                 else
47                     return ["1", split_query[0]]
48             end
49    end
50 end
51
52 class Cart
53
54    @@products = []
55    @@products_count = 0
56    @@inventory = {}
57    @@sale_total = 0
58    @@sales_archive = []
59
60   def self.new_product(quantity, prod_code)
61        @@products_count += 1
62        prod_info = get_product(prod_code)
63        prod_total = prod_info[:prod_price].to_f * quantity.to_f
```

```
64          @@sale_total += prod_total
65
66          prod_row = {prod_count: @@products_count, prod_quantity: quantity,
    prod_total: prod_total }.merge(prod_info)
67          @@products.append(prod_row)
68      end
69
70      def self.show_products
71       @@products
72      end
73
74      def self.show_total
75       @@sale_total
76      end
77
78      def self.destroy
79       @@products = []
80       @@products_count = 0
81       @@sale_total = 0
82      end
83
84       def self.get_product_test(prod_code)
85        @@inventory[prod_code] || {prod_code: prod_code,
86                                   prod_name: "abc_" + prod_code,
87                                   prod_price: 9.99}
88       end
89       def self.get_product(prod_code)
90          prod = Product.find_by(barcode: prod_code)
91          if prod
92              {prod_code: prod.barcode, prod_name: prod.description,
    prod_price: prod.price}
93          else
94              {prod_code: prod_code,
95              prod_name: "abc_" + prod_code,
96              prod_price: 9.99}
97          end
98       end
99  end
100 end
```

4.2.6 Tela da Listagem de Produtos

A imagem 4.9 apresenta a tela com a listagem dos produtos cadastrados no sistema. Podemos observar seu código interno no banco de dados, sua descrição, seu código de barras, e seu preço de venda. Também podemos observar que cada descrição é também um hiperlink para a tela de vizualização daquele produto como veremos na Subseção 4.2.7.

Nesse caso temos que o método "index"da classe "ProductsController"preenche a variável "@products"com todos os produtos registrados no banco ordenados pelo atributo ":id". Para tanto utilizamos o modelo "Product", um objeto que abstrai a comunicação com o banco de dados, nesse caso com a tabela Products.

Em seguida o programa gera um html dinâmico utilizando da variável "@products"para criar a tabela que vemos na imagem 4.9.

Figura 4.9: Tela da listagem de produtos

```
<nav class="navbar navbar-light bg-light">
    <a class="navbar-brand">
    <img src="/Niex_logo.png"  height="30" alt="">
  </a>
  <a class="navbar-brand">Products List</a>
<%= button_to "New Product", new_product_path, method: :get, class: "btn
    btn-outline-primary mt-2"%>

</nav>

  <table class="table table-bordered table-striped mb-0">
    <thead>
      <tr>
        <th scope="col">ID</th>
        <th scope="col">Barcode</th>
        <th scope="col">Description</th>
        <th scope="col">Unit Price</th>
      </tr>
    </thead>

  <tbody>
    <% @products.each do |p| %>
    <tr>
      <th scope="row"> <%= p.id %> </th>
      <td> <%= p.barcode %> </td>
      <td> <%= link_to p.description, p %> </td>
      <td> <%= p.price %> </td>
    </tr>
    <% end%>
  </tbody>
  </table>


<footer class="footer">
<%= "----" * 36%>
    <% if @user %>
       <%= button_to "Logout", logout_path, method: :delete, class: "btn
    btn-outline-danger mt-2"%>
       Logged in as <%= @user.username %>
```

```
    <% end %>
</footer>
```

```
class ProductsController < ApplicationController
    def index
        @products = Product.all.order(:id)
         if session[:user_id]
            @user = User.find(session[:user_id])
         end
    end
end
```

4.2.7 Tela de Visualização de Produto

Na imagem 4.10 vemos a visão detalhada do produto "Café Pilão 250g". Nela podemos ver além das informações do produto as opções de editá-lo, voltar para a tela de "produtos"ou deletá-lo.

De modo análogo a tela anterior, temos o método "show"que preenche a variável "@product"com o produto referente ao "id"selecionado para visualização, utilizando o método ".find"do modelo "Product". Essa variável "@product"por sua vez será utlizada pela camada de visualização para apresentar suas informações ao usuário.

Figura 4.10: Tela de visualização de produto

```
<nav class="navbar navbar-light bg-light">
    <a class="navbar-brand">
    <img src="/Niex_logo.png"  height="30" alt="">
  </a>
  <a class="navbar-brand">Product View</a>

<div class="row">
  <div class="col">
 <%= button_to "Edit", edit_product_path(@product), method: :get, class: "
    btn btn-outline-primary mt-2"%>
  </div>
  <div class="col">
    <%= button_to "<- Products", products_path, method: :get, class: "btn
    btn-outline-secondary mt-2"%>
```

```
  </div>
    <div class="col">
    <%= button_to "Delete", product_path(@product), method: :delete, class:
     "btn btn-outline-danger mt-2"%>
  </div>
</div>
</nav>



<h1><%= @product.description %></h1>

<h2>Barcode: <%= @product.barcode %></h2>
<h2>Price: <%= @product.price %></h2>
```

```
class ProductsController < ApplicationController
    def index
        @products = Product.all.order(:id)
         if session[:user_id]
            @user = User.find(session[:user_id])
         end
    end
    def show
        @product =  Product.find(params[:id])
    end
end
```

4.2.8 Telas de Criação e Edição de Produto

Na figura 4.11 podemos ver a criação do produto "Café Pilão". Essa tela nada mais é que um formulário semelhante aos encontrados no login e no registro de usuário. Com o detalhe de também ser utilizado para editar determinado produto, bastando primeiro preencher a variável "@product"com o produto que se pretende editar, de modo que os campos sejam preenchidos com as informações salvas, permitindo a modificação, permitindo assim salvar tais alterações ou descartá-las. Uma após a criação ou edição de produto o usuário é redirecionado de volta a tela dos produtos.

Figura 4.11: Tela de visualização de produto

```
<%= form_with model: product do |form| %>
  <div>
    <%= form.label :description %><br>
    <%= form.text_field :description %>
    <% product.errors.full_messages_for(:description).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.label :barcode %><br>
    <%= form.text_field :barcode %><br>
    <% product.errors.full_messages_for(:barcode).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.label :price %><br>
    <%= form.text_field :price %><br>
    <% product.errors.full_messages_for(:price).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>


  <div>
    <%= form.submit %>
  </div>
<% end %>
```

```
class ProductsController < ApplicationController
    def index
        @products = Product.all.order(:id)
         if session[:user_id]
            @user = User.find(session[:user_id])
         end
    end
    def show
        @product =  Product.find(params[:id])
    end
    def new
         @product = Product.new
    end

    def create
        @product = Product.new(product_params)

            if @product.save
            redirect_to @product
            else
            render :new, status: :unprocessable_entity
            end
    end

    def edit
        @product = Product.find(params[:id])
    end

    def update
        @product = Product.find(params[:id])
```

```
31
32          if @product.update(product_params)
33          redirect_to @product
34          else
35          render :edit, status: :unprocessable_entity
36          end
37      end
38  end
```

4.3 Tabelas de Dados

Nesta seção veremos as tabelas de dados utilizados pelo programa.

4.3.1 Schema

A figura 4.12 nos apresenta a estrutura das tabelas do programa declarada de acordo com o *framework* "Rails"que se encarrega de aplicar e se certificar do mesmo na comunicação com o banco de dados. Nesse caso utilizamos um contêiner docker executando uma instância do SGBD (Sistema de Gerenciamento de Banco de Dados) PostgreSQL.

Nesse sentido, criamos as seguintes tabelas:

- **users**: armazena os dados de login e cargo de cada usuário do sistema.
- **products**: armazena os dados de cada produto a ser vendido, como código de barras, descrição e preço de venda.
- **sales**: registro de cada venda com o nome do caixa responsável por ela, preço total e chave estrangeira para a tabela com os produtos registrados naquela venda.
- **sale_products**: os produtos de fato vendidos em cada venda, ou seja, o id da venda e o id de cada produto, junto de seu preço naquela ocasião, bem como a quantidade vendida e o subtotal dessa entrada.

```
ActiveRecord::Schema.define(version: 2022_11_28_225046) do

  enable_extension "plpgsql"

  create_table "users", force: :cascade do |t|
    t.string "username"
    t.string "password_digest"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.string "role"
  end

  create_table "products", force: :cascade do |t|
    t.string "barcode"
    t.string "description"
    t.float "price"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

  create_table "sales", force: :cascade do |t|
    t.string "cashier"
    t.float "total_price"
    t.products "sale_products_id"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

  create_table "sale_products", force: :cascade do |t|
    t.string "product_id"
    t.string "sale_id"
    t.float "price"
    t.float "quantity"
    t.float "prod_total"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end
end
```

Figura 4.12: Schema das tabelas de dados

5. Implementação do Sistema OO

Neste capítulo abordaremos mais alguns aspectos da implementação do sistema.

5.1 Programação

5.1.1 Classes

As classes que o aluno implementou no sistema foram foram:

- **Cart**: classe inteiramente criada pelo aluno com o objetivo de implementar a funcionalidade de um carrinho de compras referente aos produtos sendo registrados pelo operador de caixa. Essa é a principal classe da tela de vendas. Diz respeito ao carrinho de compras, ou seja, faz o papel de receber o código lido e registrar o novo produto junto de sua quantidade. Armazena temporariamente os produtos sendo comprados, se comunica com a classe Sale, que faz a comunicação com a tabela de vendas do banco de dados para adquirir os dados de produto.
- **SalesController**: classe implementada de acordo com o padrão Rails para operar o controle da entidade abstrata das vendas. Isto é, essa classe é responsável por receber, direcionar e operacionalizar tudo referente ao objeto Sale (venda), inclusive funcionando em sincronia com a classe Cart. Esta classe é a responsável por receber as requisições referentes a classe abstrata de vendas. Isto é, ela que abarca os métodos necessários para renderizar e servir páginas de resposta, popular variáveis utilizadas na interface das vendas como as variáveis do total da compra e dos produtos no carrinho. Além de ser ela que processa a adição de produtos.
- **Sale**: Sale é o modelo da tabela sales. Ele apresenta os métodos para criar um novo registro, consultar um registro existente,
- **ProductsController**: Classe que apresenta todos os métodos necessários para criar, atualizar, apresentar uma lista, gerenciar as interfaces referentes ao produto e principalemente utilizar a classe Product para tudo isso.
- **Product**: Classe modelo que assim como Sale. Nesse caso emcapsula a tabela products e fornece os métodos necessários para se comunicar com o banco de dados.

- **UsersController**: Classe que é responsável por permitir as interfaces e métodos referentes aos usuários do sistema, isto é, criar usuários, averiguar cadastro, permitir o login e o cadastro de novos usuários.
- **User**: Outro modelo, que encapsula a comunicação com a tabela users.
- **SessionsController**: Classe responsável por gerenciar as sessões no aplicativo. Através dela estabelecemos uma sessão criptografada utilizando cookies e averiguamos os cargos dos usuários para direciona-los corretamente.
  Todas elas podem ser vista com maior detalhe na apresentação das telas Seção 4.2.

5.1.2 Módulos - Arquitetura MVC

O sistema por ter sido desenvolvido em *Ruby on Rails* segue a arquitetura *Model View Controller* (MVC). Ela constitui na ideia de separar um programa nessas três camadas, de modo que o Controller se encarregue de lidar com as requisições, chamar métodos, e transitar com dados, enquanto o Model cria uma abstração de uma tabela do banco de dados, como por exemplo o model "Product"para se comunicar com a tabela "products"e finalmente a camada de View é a interface com o usuário. Geralmente cada view está atrelada a determinado método de um controller, como por exemplo o método "index"do ProductsController tem o seu respectivo View "Products/index.html.erb"que tem acesso as variáveis no escopo do método e retorna pra ele eventuais POSTs. Apresentamos diversos exemplos dessa arquitetura na Seção 4.2

Assim, para cada uma das quatro principais entidades do sistema (User, Session, Product, Sale) nos temos os seus respectivos e homônimos modelos para encapsular a comunicação com suas respectivas tabelas por meio de métodos como .find para buscar determinado registro, .create(x,y,z) para criar um novo registro com os atributos x,y,z. Cada um deles têm o seu respectivo controller com diversos métodos, e boa parte desses métodos tem sua própria view.

A arquitetura interna do Ruby on Rails é uma demonstração interessante desse princípio, como podemos observar no diagrama da Figura 5.1. Nele vemos que o navegador envia requests para o servidor web do rails, normalmente utilizamos o software Puma hoje em dia. Esse servidor encaminha com o Dispacher para o Controller. O controler é o cérebro por trás do programa e é quem comanda, delega e redireciona, incluindo se comunicando com o Active Record que por sua vez é uma interface para um SGBD. Uma vez processado a requisição a Active View se encarreda de renderizar e dispor na tela do navegador.

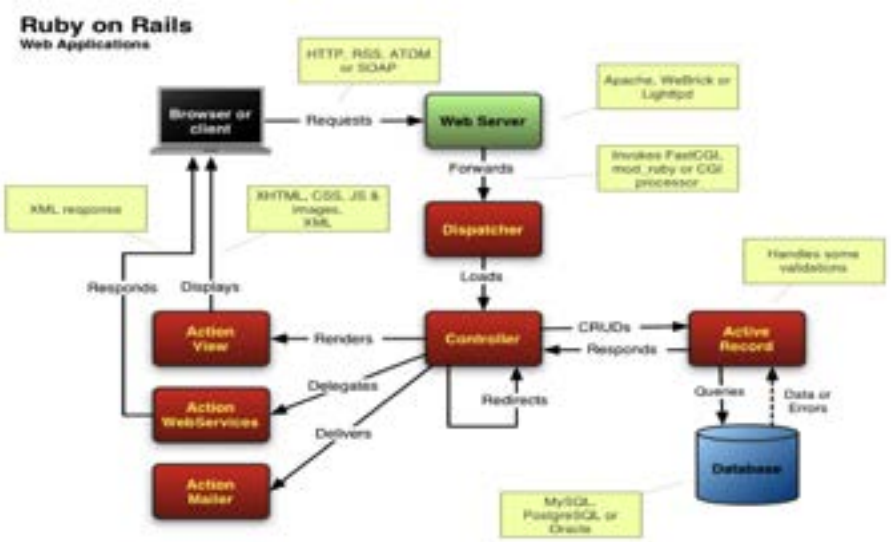


Figura 5.1: Arquitetura Ruby On Rails. Fonte: https://adrianmejia.com/ruby-on-rails-architectural-design/

Podemos, portanto, representar o sistema conforme a Figura 5.2. Nela temos um diagrama de componentes demonstrando a arquitetura de fato implementada no sistema, de acordo com a arquitetura MVC.

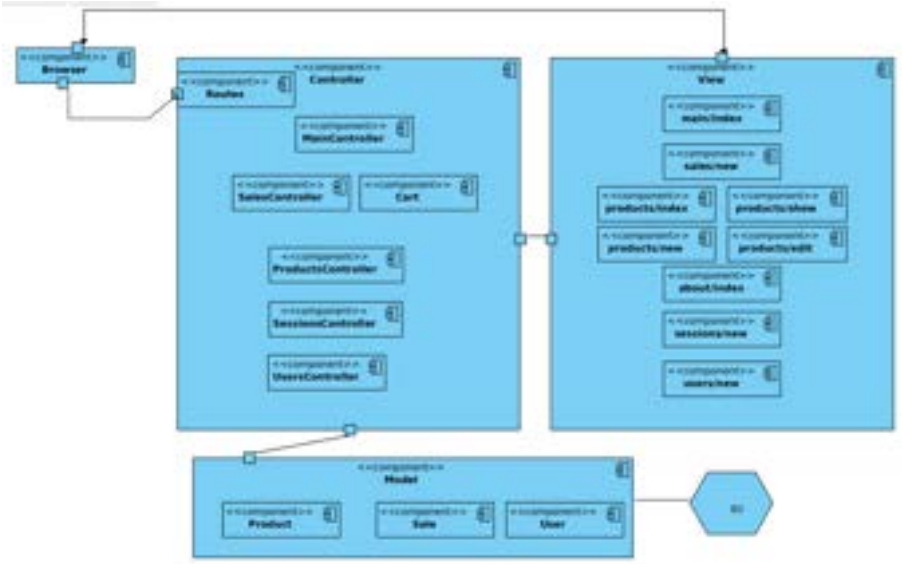


Figura 5.2: Diagrama de Atividade para editar um produto

Também é importantíssimo citar a biblioteca *Bootstrap* que apresenta classes prontas com css, e por meio dela temos um versão micro do javascript rodando em tempo de execução. As gemas de ruby: pg, bcrypt e puma utilizado respectivamente para comunicação com o postgress, criptografar as senhas e sessões e o próprio servidor web que processa as requisições antes de encaminhá-las ao Rails.

5.1.3 Partes fundamentais do código fonte

Além do código fonte das classes demonstrada anteriormente na Subseção 5.1.1, outro aspecto absolutamente fundamental é o papael das rotas. Todo o funcionamento do programa depende dessas rotas direcionando corretamente endereços e funções requeridas aos métodos adequados para manipulá-las. A seguir trazemos o arquivo das rotas utilizado no projeto. Nela podemos observar como cada palavra no endereço é uma palavra chave para determinado método de determinada classe. Também observmaos como Ruby tem a linguagem poderosa, pois ao invés de escrever as rotas para todos os verbos comuns para métodos (new, create, destroy, etc) podemos apenas utilzar o "resources :products"para criá-las e precisamos apenas implementar seus métodos.

```
1  Rails.application.routes.draw do
2    get root to: "main#index"
3    get "about", to: "about#index"
4
5    get "sign_up", to: "users#new"
6    post "sign_up", to: "users#create"
7
8    delete "logout", to: "sessions#destroy"
9    get "sign_in", to: "sessions#new"
10   post "sign_in", to: "sessions#create"
11
12  get "sale", to: "sales#new"
13  post "sale", to: "sales#search"
14  delete "sale", to: "sales#destroy"
15  get "sell", to: "sales#create"
16
17  get "sales", to: "sales#index"
18
19 #  get "products", to: "products#index"
20 #  get "products/:id", to: "products#show"
21  resources :products
22 end
```

5.1.4 Base de dados implementadas

Conforme vimos na Seção 4.3, utilizamos o SGBD "Postgress"devido a sua prevalência no mercado e gratuidade. Ainda, tendo em vista a modularização do sistema utilizamos um contêiner *Docker* para abarcar a execução do sistema e seu contexto *Ruby on Rails* que por sua vez se conecta com outro contêiner executando uma instância do PostgreSQL.

Essa implementação trouxe a vantagem de ser plenamente executável em qualquer circunstância... Desde que tenha o docker instalado. Mas logo descobrimos que a instalação do docker por si só é um desafio considerável. Além disso, por estar encapsulado no contêiner as melhoras formas de uma conexão com ele são pouco factíveis nesse contexto. Por essa razão não foi possível tirar os prints das tabelas, uma vez que o utilitário necessário deveria ser construído em um quarto conteiner mais complexo para se conectar com o conteinter de postgress. Assim, seguiremos desenvolvendo nesse nestido, mas não foi possível concluir no escopo desse projeto.

5.2 Documentação do Software

5.2.1 Manual de Instalação

O sistema Niex foi desenvolvido para ser abarcado em um servidor de modo que para se utilizar bastasse inserir o seu endereço adequado no seu navegador de preferência e utilizá-lo. Assim, disponibilizamos a seguir o passo a passo necessário para executar localmente o programa.

1. Instalar o Docker conforme as instruções oficiais disponíveis em: Install Docker Compose
2. Clonar o repositório github [1] com a versão mais recente disponível:

```
git clone https://github.com/dbs-97/niex.git
```

3. Entrar no diretório "niex"criado ao clonar o projeto e executar o seguinte comando:

```
docker compose up
```

4. Pronto. Basta abrir o endereço "0.0.0.0:3000"em seu navegador e começar a vender!

5.2.2 Manual de usuário

Nesta seção abordaremos o aspecto mais prático do que foi abordado na Seção 4.2.

Tela inicial

1. Na tela inicial o você pode logar no sistema, cadastrar-se, ou ler ler mais informações na tela Sobre.
2. Logando ou cadastrando-se a tela automaticamente será alterada para a interface adequada ao seu cargo.

Para Caixas

1. Assim que finalizar seu login ou cadastro no sistema, ele automaticamente lhe apresentará a tela de vendas.
2. - O estado padrão dessa tela é receber códigos de produto, ou a quantidade multiplicada pelo código do produto.
   - Por exemplo, para registrar cinco pães francêses basta digitar "5*3", pois "3"é o código do pão francês.
   - O sistema sempre interpretará "multiplicações"como sendo essa descrição de quantidade * código de produto.
   - Quando for apenas o código do produto o programa interpretará a quantidade como sendo uma unidade.
   - Escanear um código de barras com um leitor é em termos de entrada idêntico a digitar, portanto todos os produtos podem ser apenas escaneados. Ganhando assim bastante velocidade.
3. Além da entrada de produtos, o caixa ainda pode registrar a compra para finalizá-la, ou cancelar, ou ainda finalizar a sua sessão no sistema.
4. O fato de só haverem essas opções auxilia na simplicidade do sistema. Direcionado o seu uso da forma mais simples possível.

Para administradores

1. Assim que adquire a sua sessão no sistema ele é apresentado a tela de produtos. Nela ele pode conferir todas as informações de cada produto disponível no cadastro da empresa.

[1] https://github.com/dbs-97/niex.git

2. Cada linha da tabela de produtos tem um hiperlink que permite visualizar em detalhes o produto em questão, inclusive apresentando as opções de editá-lo ou ainda excluí-lo. Além de poder voltar a tela de produtos.
3. Também na tela de produto o administrador pode criar um novo produto clicando no botão "New Product"e preenchendo corretamente as informações pedidas a seguir.
4. Finalmente, em cada uma dessas telas o usuário também pode voltar ou ainda encerrar a sua sessão no sistema.

6. Considerações Finais

Este trabalho teve como objetivo a construção de um protótipo completo para demonstrar na prática o desenvolvimento de um programa orientado a objeto. A ideia é aplicar os conceitos aprendidos ao longo do curso junto da prática de uma habilidade fundamental a qualquer especialista nas áreas de tecnologia da informação e correlatas: auto didática. É necessário aprender muito, e muito rápido para conseguir qualquer resultado nesse meio. O trabalho foi desenvolvido em forma resumida devido a complexidade do objetivo.

Nesse sentido, foi desenvolvido o sistema web Niex, um programa para caixas de comércio registrarem os produtos sendo vendidos e as principais funcionalidades necessárias nesse entorno.

Os problemas enfrentados neste trabalho foram principalmente devido ao salto de complexidade entre este e os projetos anteriores desenvolvidos no curso. Pela primeira vez o aluno desenvolveu um projeto de ponta a ponta, desde a ideação até a entrega de uma demonstração funcional, tudo estruturado em um contêiner Docker para ser instalado em um servidor remoto. Assim, cada detalhe significou um novo universo de aprendizado sendo explorado, tornando todo esse projeto particularmente desafiador. Aprendi como apresentar informações em html dinâmico, utilizar a biblioteca online Bootstrap para criar componentes visuais, toda infraestrutura do framework Rails, incluindo a manipulação do banco de dados PostgreSQL, fundamentos de desenvolvimento web, rotas e verbos HTTP, arquitetura MVC, a própria sintaxe da linguagem Ruby e o trabalho de design da marca do systema e sua apresentação visual.

Os principais exemplos de aspectos importantes não considerados são: a capacidade de deletar determinado produto de uma venda, diferentes formas de pagamento e suas implicações, geração de notas fiscais, gerenciamento de estoque, mais ferramentas administrativas como a possibilidade de editar usuários, visualização dos dados gerados e outros critérios de Bussiness Inteligence. Aumento de segurança implementando uma senha mestre para permitir a criação de usuários admin. Dentre diversas outras.

Justamente em função dessa perspectiva podemos concluir que o projeto foi uma grande oportunidade de aprendizado e desenvolvimento em diversas áreas e de diversas habilidades muito importantes a um profissional de qualidade.

Figura 6.1: Logo do Sistema

Referências Bibliográficas

[DWR14] Alan Dennis, Barbara Haley Wixom, and Roberta M. Roth. *Análise e Projeto de Sistemas*. LTC, Rio de Janeiro, 5 edition, 2014. Citado na página 1.

[Fur13] Sérgio Furgeri. *Modelagem de Sistemas Orientados a Objetos*. Érica Editora, Sã Paulo, SP, 1 edition, 2013. Citado na página 1.

[Gue11] Gilleanes T.A. Guedes. *UML 2 : uma abordagem prática*. Novatec Editora, 2011. Citado na página 1.

[Hel13] Helio Engholm Jr. *Análise e Design Orientados a Objetos*. Novatec, 2013. Citado na página 1.

[SJB12] John W. Satzinger, Robert B. Jackson, and Stephen D. Burd. *Introduction to Systems Analysis and Design: An Agile, Iterative Approach*. Course Technology, CENGAGE Learning, Mason, Ohio, 6 edition, 2012. Citado 2 vezes nas páginas 1 e 17.

[Som18] Ian Sommerville. *Engenharia de Software*. Pearson Education do Brasil, São Paulo, 10 edition, 2018. Citado na página 1.

[SR12] Gary B. Shelly and Harry J. Rosenblat. *Analysis and Design for Systems*. Course Technology, CENGAGE Learning, 9 edition, 2012. Citado na página 1.

[TFH04] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby*. Pragmatic, 2004. Citado na página 2.

[Waz11] Raul Sidnei Wazlawick. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. Editora Campus SBC. Elsevier, Rio de Janeiro, RJ, 2 edition, 2011. Citado 2 vezes nas páginas 1 e 3.