

**SOCIEDADE EDUCACIONAL PINHALZINHO - HORUS FACULDADES  
SISTEMAS DE INFORMAÇÃO**

**CARGACERTA**

Sistema para conectar empresas e transportadores de carga

**Gabriel Luiz Kunz**

GABRIEL LUIZ KUNZ

## **CARGACERTA**

Sistema para conectar empresas e transportadores de carga

Trabalho de conclusão de curso para o curso de Sistemas de Informação apresentado à Horus Faculdades como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Ricardo J. Hendges.

Pinhalzinho – SC  
2023

## RESUMO

O fluxo logístico de distribuição de mercadorias no Brasil é um processo complexo envolvendo diversos participantes, exigindo planejamento eficiente para otimizar custos e maximizar resultados (Pereira, 2015). Administradores, gerentes logísticos e motoristas desempenham papéis cruciais ao compreender o fluxo logístico como um todo e analisar resultados através de indicadores chave. Este trabalho aborda os principais pontos da cadeia logística de distribuição, propondo uma solução que atenda às necessidades, como a otimização de rotas, aumento do aproveitamento de cargas e garantia de retornos consistentes.

O transporte rodoviário representa a maior parcela, movimentando bilhões de toneladas e reais anualmente no Brasil (Pereira, 2015). Empresas de Transporte Rodoviário de Carga (ETC), Cooperativas de Transporte Rodoviário de Cargas (CTC) e Transportadores Autônomos de Carga (TAC) constituem as categorias de transportadores. Enquanto ETCs e CTCs têm fluxo constante de fretes, os TACs enfrentam incertezas devido à dependência de contatos diretos para obter cargas.

Este trabalho propõe o desenvolvimento de uma plataforma para conectar empresas com cargas a motoristas autônomos disponíveis, visando superar os desafios enfrentados pelos TACs e melhorar a eficiência do transporte rodoviário de cargas no Brasil.

**Palavras chave:** Logística, transporte rodoviário, cadeia de distribuição, otimização de rotas, motoristas autônomos, eficiência operacional, plataforma de conexão, transporte de cargas, Brasil.

## **ABSTRACT**

The logistics flow of goods distribution in Brazil is a complex process involving various stakeholders, requiring efficient planning to optimize costs and maximize results (Pereira, 2015). Administrators, logistics managers, and drivers play crucial roles in understanding the logistics flow as a whole and analyzing results through key indicators. This work addresses the main points of the distribution logistics chain, proposing a solution to meet needs such as route optimization, increased cargo utilization, and consistent return assurance.

Road transport represents the majority, moving billions of tons and reais annually in Brazil (Pereira, 2015). Freight transporters are categorized into Companies of Road Cargo Transport (ETC), Cooperatives of Road Cargo Transport (CTC), and Autonomous Cargo Transporters (TAC). While ETCs and CTCs have a constant flow of freight, TACs face uncertainties due to dependence on direct contacts to obtain cargos.

This work proposes the development of a platform to connect companies with cargos to available autonomous drivers, aiming to overcome challenges faced by TACs and improve the efficiency of road cargo transport in Brazil.

**Keywords:** Logistics, road transport, distribution chain, route optimization, autonomous drivers, operational efficiency, connection platform, cargo transport, Brazil.

## LISTA DE FIGURAS

Figura 1 - Tabela de usuários.....	37
Figura 2 - Tabela de cidades.....	38
Figura 3 - Tabela de pessoas.....	39
Figura 4 – Tabela de veículos .....	40
Figura 5 – Tabela de entregas .....	41
Figura 6 – Tabela das candidaturas dos motoristas para realização das entregas...	43
Figura 7 – Criação do projeto da API .....	44
Figura 8 – Arquivo package.json com as configurações do projeto .....	46
Figura 9 – Estrutura de pastas da API .....	47
Figura 10 – Definição de modelo de usuário .....	49
Figura 11 – Método de registro de novos usuários .....	51
Figura 12 – Método de login de usuários .....	52
Figura 13 – Método de solicitação de recuperação de senha .....	54
Figura 14 – E-mail de recuperação de senha.....	55
Figura 15 – Método de alteração de senha do usuário .....	56
Figura 16 – Implementação de método de registro de usuários no controller .....	58
Figura 17 – Método de tratamento de erro .....	59
Figura 18 – Implementação das rotas de usuário .....	61
Figura 19 – Middleware de verificação de autenticação.....	62
Figura 20 – Configuração inicial do projeto do frontend .....	64
Figura 21 - Opções de configuração do projeto .....	65
Figura 22 – Estrutura das pastas do projeto Vue.JS .....	65
Figura 23 – Implementação da configuração do axios .....	67
Figura 24 – Implementação de Layout .....	68
Figura 25 – Implementação de componente de cabeçalho de páginas .....	69
Figura 26 – Código Javascript do componente de cabeçalho de páginas .....	70
Figura 27 – Configuração das rotas do projeto de frontend .....	71
Figura 28 – Implementação de service de usuários no frontend .....	72
Figura 29 – Implementação de store de usuário .....	73
Figura 30 – Cadastro de carga.....	75
Figura 31 – Listagem de entregas cadastradas pela empresa logada .....	75
Figura 32 – Listagem de veículos cadastrados .....	76

Figura 33 – Tela de cadastro de veículos .....	76
Figura 34 – Listagem de cargas disponíveis .....	77
Figura 35 – Confirmação de manifestação de interesse .....	77
Figura 36 – Manifestações de interesse para a entrega .....	78
Figura 37 – Listagem das cargas do motorista.....	79

# SUMÁRIO

<b>1. INTRODUÇÃO – TEMA E PROBLEMATIZAÇÃO .....</b>	<b>9</b>
<b>2. JUSTIFICATIVA .....</b>	<b>10</b>
<b>3. OBJETIVOS.....</b>	<b>11</b>
3.1 GERAL.....	11
3.2 ESPECÍFICOS.....	11
<b>4. REFERENCIAIS TEÓRICOS.....</b>	<b>12</b>
4.1 PANORAMA DA LOGÍSTICA NO BRASIL.....	12
4.2 O FLUXO LOGÍSTICO .....	14
4.3 AS INFORMAÇÕES PARA A LOGÍSTICA E DISTRIBUIÇÃO .....	14
<b>5. METODOLOGIA DA PESQUISA .....</b>	<b>25</b>
<b>6. CRONOGRAMA .....</b>	<b>27</b>
<b>7. LEVANTAMENTO DE REQUISITOS .....</b>	<b>28</b>
<b>8. ESTUDO DAS TECNOLOGIAS .....</b>	<b>30</b>
8.1. JAVASCRIPT .....	30
8.2. NODE.JS.....	31
8.3. EXPRESS.JS .....	32
8.4. VUE.JS.....	32
8.5. QUASAR .....	33
8.6. POSTGRESQL.....	34
8.7. SEQUELIZE .....	35
<b>9. DESENVOLVIMENTO DO PROJETO.....</b>	<b>36</b>
9.1. DEFININDO A ESTRUTURA DO BANCO DE DADOS.....	36
9.2. DESENVOLVIMENTO DA API.....	44
<b>9.2.1 Criação do projeto da API.....</b>	<b>44</b>
<b>9.2.2 Models .....</b>	<b>48</b>
<b>9.2.3 Services.....</b>	<b>50</b>
<b>9.2.4 Controllers .....</b>	<b>57</b>
<b>9.2.5 Routers.....</b>	<b>60</b>
<b>9.2.6 Middlewares.....</b>	<b>61</b>
9.3. DESENVOLVIMENTO DO FRONTEND .....	63
<b>9.3.1 Configuração inicial do projeto com Quasar .....</b>	<b>64</b>
<b>9.3.2 O diretório boot .....</b>	<b>67</b>
<b>9.3.3 O diretório layouts.....</b>	<b>68</b>
<b>9.3.4 O diretório components.....</b>	<b>69</b>
<b>9.3.5 O diretório pages.....</b>	<b>70</b>
<b>9.3.6 O diretório router.....</b>	<b>71</b>
<b>9.3.7 O diretório services.....</b>	<b>72</b>
<b>9.3.8 O diretório stores .....</b>	<b>73</b>
<b>9.3.9 Considerações sobre o frontend .....</b>	<b>74</b>

<b>10. PROTÓTIPO .....</b>	<b>75</b>
<b>11. CONSIDERAÇÕES FINAIS .....</b>	<b>80</b>
<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>82</b>

## **1. INTRODUÇÃO – TEMA E PROBLEMATIZAÇÃO**

O fluxo logístico de distribuição de mercadorias no país compreende um grande e complexo conjunto de partes envolvidas para o bom andamento do processo. É preciso um bom planejamento logístico para que o processo de distribuição e transporte seja otimizado, com o objetivo de otimizar custos e maximizar resultados.

Atingir esses resultados não é uma tarefa fácil e exige que administradores, gerentes logísticos e motoristas entendam o fluxo logístico como um todo e saibam analisar resultados com base em indicadores chave.

Encontrar a melhor rota, cargas com aproveitamento maior, retornos garantidos com cargas de volta são apenas alguns dos pontos almejados por quem controla o fluxo de transporte de alguma empresa transportadora ou motorista autônomo.

Esse trabalho tratará de entender os principais pontos que compreendem a cadeia logística de distribuição, apontando para uma o que uma solução possível pode entregar a fim de atender às necessidades expostas anteriormente.

## 2. JUSTIFICATIVA

O transporte de cargas por meio rodoviário representa 65,6% da circulação de mercadorias em solo brasileiro, transportando cerca de 936 bilhões de toneladas por quilometro útil (Pereira, 2015), movimentando anualmente cerca de 365 bilhões de reais. De acordo com a ANTT, há três categorias de transportadores, sendo eles as Empresas de Transporte Rodoviário de Carga (ETC), as Cooperativas de Transporte Rodoviário de Cargas (CTC) e os Transportadores Autônomos de Carga (TAC). No caso das duas primeiras categorias, há muita movimentação no que diz respeito a aquisição de contratos para a realização do transporte, o que garante um fluxo mais constante de fretes e conseqüentemente uma receita mais estável. Enquanto isso, os transportadores autônomos dependem de contatos diretos com responsáveis pelas empresas para a realização de fretes pontuais e nem sempre frequentes, o que gera certa incerteza no fato de conseguir ou não cargas para realizar o transporte, já que nem sempre a comunicação com os responsáveis por direcionar as cargas é fácil.

Buscando resolver esse problema, esse trabalho visa desenvolver uma plataforma com o objetivo de conectar empresas que tenham cargas para serem transportadas e motoristas autônomos com disponibilidade para a realização desses fretes.

### **3. OBJETIVOS**

#### **3.1 GERAL**

O objetivo geral desse trabalho é realizar um estudo no que se refere ao mercado do transporte rodoviário no Brasil, englobando questões como a disponibilidade e distribuição de cargas, acesso à contratos de transporte, logística para a realização de fretes, pagamento com relação ao transporte realizado, e com base nesse estudo, desenvolver uma plataforma digital para conectar motoristas autônomos de caminhões e empresas com disponibilidade de cargas para transporte, visando o aumento da disponibilidade geral de mercadorias para consumo, além de um aumento de receita para esses motoristas e para as empresas que estão inclusas no processo.

#### **3.2 ESPECÍFICOS**

- Realizar um estudo de disponibilidade e distribuição de cargas para transporte rodoviário em nível nacional.
- Buscar informações de como é realizado o contato de motoristas autônomos com empresas que disponibilizam cargas para transporte.
- Desenvolver um protótipo de plataforma digital online, com disponibilidade de acesso em computadores e celulares, para que seja realizada a conexão dos motoristas autônomos com as empresas com cargas a serem transportadas.

## 4. REFERENCIAIS TEÓRICOS

A cadeia logística aplicada no Brasil compreende uma extensa rede que trabalha de forma coordenada e de acordo com as demandas que o consumo impõe no mercado. O ponto de estudo desse trabalho compreende a última etapa da cadeia logística implementada e seguida no Brasil e se refere especificamente à distribuição dos produtos e serviços produzidos no decorrer da cadeia logística.

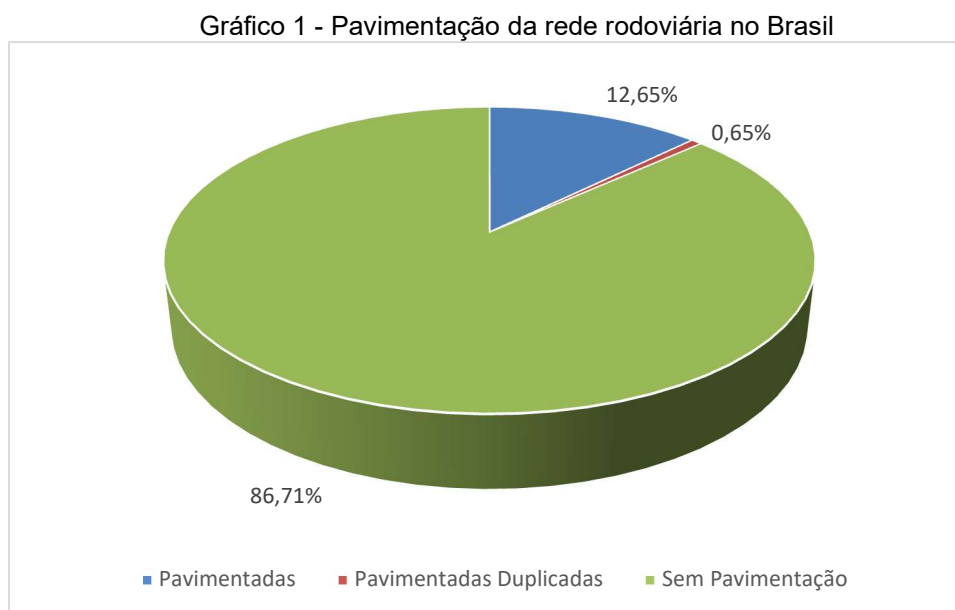
### 4.1 PANORAMA DA LOGÍSTICA NO BRASIL

Conforme Schlüter (2013), a logística de distribuição de produtos de uma empresa é o elemento formador do sistema logístico de transportes. Isso significa que uma empresa, ao necessitar de serviços de transporte, desencadeia todos os processos relacionados às operações, envolvendo outros atores que participam da cadeia logística desde seu início.

Além de desencadear ações de outros atores envolvidos no processo logístico, o processo de distribuição sofre fortes influências externas. Essas influências vão desde o grau de conhecimento de logística pelos tomadores de decisão, que são os usuários dos sistemas de transportes, passando pelo nível de competitividade de cada operador das empresas de cada modal de transporte, até a forte interferência do governo no que tange à infraestrutura e à regulamentação dos modos de transporte. No que diz respeito à infraestrutura de transporte, o governo interfere nas esferas municipal, estadual e federal, sendo responsável por melhorias nas condições de estradas. Já a regulamentação, é feita estritamente pelo governo federal, através da ANTT - Agência Nacional de Transporte Terrestre (Schlüter, 2013).

Em termos de infraestrutura de transportes terrestres, constata-se que nossas redes rodoviárias são modestas e com uma distribuição concentrada das regiões Sudeste, Sul e Centro-Oeste. No total, a rede rodoviária nacional compreende mais de 1,7 milhões de quilômetros, porém apenas 215 mil quilômetros estão pavimentados, representando algo em torno de 12,65% do total, conforme

apresentado no Gráfico 1.



Fonte: Pereira, 2015.

A rede pavimentada divide-se em federal, 64 mil quilômetros, estaduais, 124 mil, e municipais, 27 mil, sendo que as regiões Sul e Sudeste correspondem a mais de 50% desse total pavimentado (Pereira, 2015).

Apesar da responsabilidade dos governos na manutenção das vias terrestres, são realizadas concessões ao setor privado para administração de rodovias e isso ocorre principalmente na região Sudeste. Por esse motivo, apenas cerca de 11 mil quilômetros das vias terrestres estão pavimentadas e duplicadas, com a grande maioria entregue à concessões privadas que administram, dão manutenção e realizam a cobrança de pedágio para os motoristas que trafegam nessas vias (Pereira, 2015).

Quanto à regulamentação da atividade, pode-se afirmar que ela somente teve início em 2007, quando a ANTT regulamentou pela Lei 11.442 de 5 de janeiro de 2007, o Registro Nacional de Transporte Rodoviário de Cargas (RNTRC). A essa iniciativa básica, seguiram-se outras pela ANTT, relativas à contratação de terceiros, à cobrança dos fretes e mais recentemente a regulamentação da atividade e jornada dos motoristas. Porém, nenhuma dessas alterações tem impacto ou aplica qualquer restrição de mercado na atividade de transporte propriamente dita (Pereira, 2015).

## 4.2 O FLUXO LOGÍSTICO

Ao visualizarmos a cadeia logística aplicada atualmente, ela está dividida em procedimentos estáticos e dinâmicos. Os procedimentos estáticos compreendem o momento em que o produto englobado pela cadeia logística está parado para a realização de algum processo da cadeia produtiva ou quando está em centros de armazenagem. Os procedimentos dinâmicos compreendem as diversas fases do processo de distribuição do produto, desde a aquisição da matéria-prima para a sua produção até a entrega ao consumidor final (Schlüter, 2013).

De acordo com Izidoro (2016), além dos procedimentos citados anteriormente, são dois os principais fluxos da cadeia logística: o de materiais e o de informações. O fluxo de materiais compreende a movimentação física da matéria-prima, de insumos, de produtos e serviços, em toda a rede, desde os fornecedores até a entrega ao cliente. Já o fluxo de informações, é o movimento de dados detalhados entre os membros de uma cadeia logística. Esses dados são informações sobre o que está sendo transportado, o cliente, o status da entrega, o comprovante de entrega, entre outros. Essas informações são de grande importância, pois representam a oportunidade de exercer um poder acima da média em relação aos concorrentes e aumentam a confiabilidade na realização do serviço de transporte.

## 4.3 AS INFORMAÇÕES PARA A LOGÍSTICA E DISTRIBUIÇÃO

O fluxo de informações, porém, vai muito além de dar informações sobre o serviço de transporte e deixar os participantes do processo cientes do que está ocorrendo da realização do frete. É muito importante que os dados que estão nesse fluxo sejam úteis para a tomada de decisão pelos responsáveis logísticos de empresas, transportadoras e motoristas, indicando pontos dentro da cadeia logística que precisam de melhorias. (Izidoro, 2016).

Um dos principais pontos em que os dados provenientes do fluxo de informações da cadeia logística são muito úteis e necessários é em relação à busca pela redução de custos na realização do serviço de transporte. O custo do transporte rodoviário aumenta em função de alguns fatores que serão abordados na sequência.

O principal fator de custo elevado na realização de um serviço de transporte rodoviário é a distância entre os pontos de origem e destino da entrega. Por motivos óbvios, quanto maior a distância, maior o consumo de combustível, maior o desgaste de pneus e maior a depreciação do veículo que está sendo utilizado (Pereira, 2015).

Em contexto nacional, conforme abordado pela revista CNN Business, após o anúncio da Petrobras de uma alta de 8,9% no preço do diesel para distribuidoras no dia 10 de maio de 2022, a Associação Nacional do Transporte de Cargas e Logística aponta que o reajuste representaria um aumento de pelo menos 3,1% no custo dos fretes no país. De acordo com a mesma entidade, nos 12 meses antecedentes a esse reajuste, o preço do diesel acumula cerca de 49% de aumento, o que encarece o custo dos transportadores e conseqüentemente o custo para a realização do serviço de transporte.

Conforme Pereira (2015), outro fator que impacta em um aumento do custo da realização do transporte rodoviário, é a idade média dos veículos utilizados na prestação do serviço. Uma boa parte dos veículos que estão em circulação realizando os fretes são relativamente antigos, isso se dá em função do seu custo de aquisição, que é geralmente mais atrativo do que a compra de veículos novos ou seminovos. O problema é que a economia realizada na hora da compra do veículo não reflete o comportamento do bem na realização do transporte, visto que tendem a consumir mais combustível, apresentar mais problemas de manutenção e quanto mais antigo o veículo, mais difícil é encontrar peças para reposição e conserto.

Questões de infraestrutura de rodovias também tem impacto direto no custo, em função da qualidade do pavimento, duplicações, pontos de parada e rotas coerentes com a viagem.

Outro ponto muito forte, que se visa amenizar com o desenvolvimento dessa pesquisa, é o desbalanceamento nos fluxos de ida e volta do frete. Esse ponto, segundo Pereira (2015), é amplamente justificado em função de que um frete se torna mais custoso quando não há a possibilidade de um frete de retorno para o motorista. Nessas situações, o motorista ou a empresa de logística tem a necessidade de andar com o veículo muitas vezes descarregado no retorno, o que

gera os custos já mencionados anteriormente, e ainda sem retorno financeiro por não estar realizando a prestação de nenhum serviço. Dessa forma, a logística busca organizar um fluxo de cargas onde o destino de um transporte pode ser a origem, ou estar próximo da origem de um novo serviço. Com essa organização sendo realizada corretamente, os resultados tendem a ser o aumento do faturamento, a diminuição de custos e uma frequência maior de realização de fretes.

De encontro com o que foi abordado anteriormente, Schlüter (2013) destaca que os custos do modelo da função logística sofrem uma série de influências de fatores de demanda e oferta. O primeiro grupo está relacionado às características dos produtos e ao perfil da demanda do mercado em que a empresa prestadora do serviço de transporte atua. O segundo grupo está relacionado às características dos modais que farão parte de análise do cenário logístico.

Os fatores de demanda dimensionam as necessidades logísticas dos usuários e a infraestrutura para atendê-los. Nesse grupo de fatores, destaca-se a análise da característica do produto a ser transportado em alguns pontos: peso, volume, valor, prazo de validade, embalagem e risco.

O peso do produto está diretamente ligado ao valor do frete, uma vez que tem impacto sobre a performance do veículo e conseqüentemente sobre o consumo de combustível, além de que um veículo com um produto pesado, gera mais danos às rodovias e exige a longo prazo, mais investimentos para manutenção das vias.

O volume é muito considerado no mercado de cargas fracionadas, pois representa quanto espaço aquele produto ocupa no veículo em relação aos demais. Fatores como valor do produto e risco tem impacto no custo de um frete, em função de que podem ser alvos de ações criminosas, o que gera maior preocupação com a integridade do veículo, da carga e do motorista.

A embalagem e o prazo de validade do produto que está sendo transportado também devem ser analisados para mensurar o custo da realização do serviço de transporte. Produtos com o prazo de validade curto, naturalmente demandam uma entrega mais rápida, e o custo pela agilidade na realização do frete é amplamente considerado.

Ainda de acordo com Schlüter (2013), os fatores da demanda de mercado são originados pelo consumidor final e repassados às demais empresas integrantes da cadeia logística. A elaboração de um projeto logístico se inicia pela compreensão da relação entre os consumidores e o seu produto. Com base nesse entendimento,

analisam-se as variáveis quantitativas de demanda espaço-temporal. Conhecer o produto inserido no mercado garante uma razoável previsibilidade da demanda, o que facilita a formação da infraestrutura e organização da logística de distribuição desse produto no mercado.

Outros fatores de demanda tratam das questões relativas às quantidades que são comercializadas no âmbito geográfico ao longo do tempo. Entre esses fatores, pode-se destacar a quantidade demandada pelo mercado, a localização dos pontos de origem e destino do produto em questão e as sazonalidades, que definem o aumento ou diminuição da demanda em determinados períodos (Schlüter, 2013).

Agora em relação ao segundo grupo, de acordo com Schlüter (2013), os fatores de oferta de serviços de logística estão ligados às características dos modais de transporte e ocorrem em todos eles, porém com intensidades diferentes de acordo com o meio em que estão inseridos. Como já visto anteriormente, o Brasil possui a grande parte do transporte sendo realizado pelo modal rodoviário.

Em geral, alguns fatores são levados em consideração no caso da oferta para que seja criado o fluxo logístico adequado para a cada situação. Entre eles, destacam-se a acessibilidade, disponibilidade, velocidade, tarifa, valor relativo do veículo e vias (Schlüter, 2013).

A acessibilidade é a condição proporcionada pela via do modal que executará o longo curso em relação ao cumprimento do trajeto, dada pelo percentual de cobertura do modal no percurso total.

A disponibilidade é a capacidade de dispor do veículo do modal de longo curso no momento em que ocorre a necessidade de transporte.

A velocidade, nesse caso, deve ser tratada como prazo de entrega e não apenas como velocidade do veículo do modal. Embora cada veículo possua uma velocidade nominal, muitas vezes, as condições de operação levam a prazos de entregas mais longos que aqueles que efetivamente poderiam ser realizados.

A tarifa é o valor pago ao operador do modal pelo transporte de uma carga de uma origem para um destino.

O valor relativo do veículo refere-se ao valor do investimento na aquisição de um veículo, relacionado com a sua capacidade dinâmica de carga, analisando o tempo de retorno do investimento realizado.

Por último, as vias, é o valor de implantação de vias em relação ao seu compartilhamento por outras formas de uso e outros operadores. Uma via pode ter o

seu valor maior do que outra, desde que tenha uma utilização que propicie retorno sobre o investimento realizado (Schlüter, 2013).

Para Arbache (2006), a movimentação de produtos cria para a sociedade o valor de lugar, pois permite que os produtores coloquem o produto exatamente onde os consumidores desejam. Se um produto não estiver disponível na data exata em que se precisar dele, isso poderá gerar vendas perdidas, insatisfação do cliente ou parada da produção.

A velocidade com que a distribuição acontece é vital para a economia moderna, na qual as empresas procuram trabalhar com o menor estoque possível, seja de matéria-prima, seja de produtos acabados ou em processo de produção.

O transporte tem um peso enorme no custo de distribuição da maioria dos produtos e é muito importante para os resultados obtidos no serviço ao cliente. Seu desempenho pode ter impacto no resultado final de uma operação, tendo influência na percepção que o comprador tem da qualidade do serviço. Muitas vezes, o agente responsável pelo transporte é o único elo real entre o vendedor e o comprador, como é o caso do comércio eletrônico.

Há ainda que ser destacado o papel do transporte na integração das diversas etapas da função logística e na geração de redes logísticas suficientemente flexíveis e velozes para atender às demandas do mercado consumidor (Arbache, 2006).

Dadas as informações a respeito do meio logístico, dos fluxos aplicados e dos fatores que atuam sobre o processo logístico, o próximo passo é entender o papel que a tecnologia vem representando para a evolução do controle da atividade como um todo.

De acordo com Arbache (2006), para entender a importância da tecnologia da informação para a logística, é necessário fazer uma reflexão a respeito das mudanças que estão ocorrendo no mercado. O mercado está se modificando continuamente, se adequando às tecnologias que estão surgindo e se inserindo em todos os segmentos. As novas aplicações e equipamentos são, na verdade, responsáveis pelos novos produtos que vêm revolucionando o mercado de consumo e produtivo. A partir dessas inovações são criados novos processos nas empresas, aumento a produtividade, reduzindo os preços e tornando acessível às diversas classes sociais os mais inimagináveis produtos, como o telefone celular.

Esta nova realidade vem formando o que podemos chamar de novos consumidores, cada vez mais seletivos, exigindo produtos mais personalizados e

exclusivos. As mudanças nos hábitos de consumo são cada vez mais imprevisíveis, demandando das empresas o aprimoramento de sua capacidade de análise de tendências, por meio da utilização de e-mails, conversas online e chamadas de áudio pela internet, que aumentam a aproximação com os clientes, facilitando um retorno contínuo (Arbache, 2006).

Portanto, a tecnologia proporciona maior integridade e velocidade na troca de informações, o que otimiza diversas atividades na logística como, por exemplo, identificar onde e quando os produtos deverão ser distribuídos, o que e quando estocar, quais locais estão demandando mais produtos, etc.

Os benefícios diretos alcançados com o acesso à informação são melhorias na previsão das demandas, na coordenação estratégica entre os membros da cadeia e na gestão dos estoques, assim como uma rápida reação às demandas do mercado e a redução do tempo de espera para a entrega dos produtos (Arbache, 2006).

Ainda de acordo com Arbache (2006), a era dos serviços está sendo caracterizada por focalizar a entrega de benefícios, além de produtos, aos consumidores. A orientação do gerenciamento está sendo voltada para o cliente. A garantia está sendo a medida de eficácia. Quando um consumidor adquire um computador com garantia de desempenho por todo o período de seu ciclo esperado de vida, está adquirindo capacidade de processamento, e não apenas um computador. A tendência é que os produtos venham a cada vez mais agregados de serviços em sua composição, para agradar ao novo gosto dos consumidores. Um desses serviços que com total certeza é considerado pelos clientes, é o processo de transporte para recebimento do seu produto. Uma promoção de frete grátis, entrega expressa ou desconto em compras de valor mais expressivo, tendem a chamar mais a atenção do cliente, conquistá-lo e o fidelizar.

Para Arbache (2006), o uso da tecnologia da informação na cadeia logística tende a focar em três áreas principais: a minimização de tempos de espera, a administração da capacidade de serviços e a entrega por meio de canais de distribuição mais acessíveis aos clientes.

Como resultado da evolução da tecnologia e de sua implementação no processo logístico, nos últimos anos surgiram os Sistemas de Gerenciamento de Transporte, do inglês *Transportation Management System*, conhecido também pela sigla TMS.

Conforme destaca Izidoro (2016), o objetivo do TMS é gerenciar os fluxos de transporte de uma empresa até seus clientes e dos fornecedores até a empresa. Ele compartilha informações como o conteúdo dos pedidos transportados, a quantidade, o peso e o volume dos bens e as datas previstas de entregas. De forma bem ampla, o TMS deve entregar as seguintes funcionalidades.

O TMS deve possibilitar atender os mais diversos modais de transporte, de acordo com a necessidade do serviço e com a infraestrutura da empresa que está utilizando o sistema. Deve entregar ao cliente destino do serviço de transporte informações atualizadas sobre o frete, com dados atualizados em tempo real. Deve ser capaz de organizar os fretes de forma que possam ser realizados da forma mais otimizada o possível. Deve definir a melhor rota para a realização da entrega. Deve entregar um rastreamento em tempo real para os administradores da empresa para que possam acompanhar o andamento das entregas, o tempo de viagem, as pausas dos motoristas e assim garantir que o prazo de entrega seja cumprido. Por último, deve ser capaz de reunir todas as informações dos fretes realizados e entregar análises de desempenho, com parâmetros estabelecidos que devem ser considerados para melhorar a qualidade da prestação do serviço de transporte (Izidoro, 2016).

Ainda de acordo com Izidoro (2016), o planejamento ajuda muito na tomada de decisões. A informação é fundamental para o sucesso do planejamento logístico, sendo assim, depende-se de um bom acesso, da confiabilidade e da qualidade das informações.

Izidoro (2016) destaca que um sistema integrado de informações logísticas deve coletar as informações, armazená-las e movimentá-las em todo o processo. O principal objetivo do planejamento logístico é prever o comportamento do mercado e se adaptar a possíveis mudanças com máxima antecedência possível. Além disso, o planejamento logístico tende a reduzir os custos operacionais da realização do serviço de transporte. Deve-se sempre levar em consideração os riscos, os benefícios, os custos e os retornos sobre investimentos realizados na operação. O planejamento apoia a operação da empresa e, no caso da logística, especificamente em relação à distribuição, entrega eficaz em otimização de processos e custos.

Conforme Grabara (2014), o gerenciamento do transporte é a coisa mais importante no que tange à logística. Custos associados à distribuição representam mais de 25% de todos os custos do processo logístico. Dentro das empresas, os

gerentes de transporte e logística são responsáveis pelas decisões de quando optar por determinado tipo de transporte, qual a melhor rota para a carga que será transportada e como deve ocorrer o fluxo logístico, buscando sempre otimização de tempos de entrega e redução de custos.

Ainda de acordo com Grabara (2014), é muito importante que os transportadores entendam todos os custos envolvidos no processo de transporte, para que possam tomar decisões mais cada vez mais assertivas em relação ao gerenciamento de recursos, sejam eles humanos ou financeiros. Uma das coisas que pode ajudar nesse ponto, é que o responsável pela logística da empresa, ou no caso de um transportador autônomo, o próprio, tenha acesso constante a controles de monitoramento de seus resultados.

Destaca-se que no passado, no início da formação da economia moderna como a conhecemos hoje, o gerenciamento de transporte era normalmente deixado de lado. Essa falta de interesse em ter um bom gerenciamento de transportes se dava pela baixa competitividade do mercado transportador. Nos dias atuais, o mercado de transportadores de carga é extremamente competitivo, com diversas empresas e transportadores engajadas diariamente no transporte de bens e serviços, buscando incorporar cada vez mais novas tecnologias em seus processos, com o objetivo de otimizar a entrega e aumentar a satisfação dos usuários, que normalmente estão sendo acompanhadas de estratégias de marketing para atrair mais clientes (Grabara, 2014).

Para Grabara (2014), a informação forma o sistema nervoso que será base para a tomada de decisões. Esse sistema nervoso criado pela união de informações provenientes de diferentes fontes, vem se tornando cada vez mais um elemento essencial em qualquer operação de logística e transporte. No caso dos transportes, cada etapa do processo pode ser facilmente distinguida e gerar uma informação específica para o processo como um todo. Grabara (2014) destaca que em cada etapa, a informação tem um objetivo específico e deve atender às necessidades impostas buscando atender ao objetivo:

Na etapa de planejamento de transporte, as informações geradas tem relação ao modo de transporte que deverá ser utilizado para o caso, buscando compreender as necessidades da operação e a infraestrutura disponível para a realização do transporte. Na etapa seguinte, serão geradas informações em relação à carga a ser transportada, como peso, tipo, quantidade e dimensões.

Na sequência do processo, estarão as informações referentes aos participantes da operação de transporte. No caso do Brasil, há seis possíveis participantes diferentes em uma operação de transporte: o transportador, o remetente, o destinatário, o expedidor, o recebedor e o pagador. Cada um dos participantes da operação gera informações de nome, contato, endereços e dados para pagamento.

Por último, refletindo a materialização da realização do serviço de transporte de cargas, são geradas informações referentes à entrega. Dados como início do transporte, entrega, tempo de espera, paradas realizadas, fluxo de entregas, roteiro.

Destaca-se também a importância de que um processo de transporte bem realizado, é um requisito tão importante quanto a qualidade do produto a ser transportado para a avaliação do grau de satisfação dos envolvidos no processo (Grabara, 2014).

Com o objetivo de melhorar o processo de transporte, foram desenvolvidas muitas tecnologias e ferramentas no que diz respeito à preparação e transmissão de informações do processo. Dessa forma, empresas transportadoras estão aumentando o uso de ferramentas e sistemas de informação cada vez mais modernos (Grabara, 2014).

Conforme Grabara (2014), a necessidade das empresas e o atual comportamento da busca por uso de novas tecnologias, levaram à criação do que é chamado de *telematics*. O termo é a combinação das expressões “telecomunicação” e “tecnologia da informação”, e parte da ideia que de é necessária a digitalização de processos, buscando soluções de controle cada vez mais automatizado com base em dados obtidos no decorrer do processo físico realizado.

Destaca-se que soluções de *telematics* podem ser definidos com base em alguns critérios a serem atendidos, conforme Grabara (2014) os define na sequência.

É preciso que a solução possua a capacidade de escalabilidade, ou seja, não se limitem às funcionalidades atualmente implementadas e possa ter seu escopo expandido para o atendimento de novas necessidades e tendências.

Possui também grande importância o fato de que a quantidade de fonte de informações seja ampla e de qualidade. Nesse ponto, utilizam-se sensores de posicionamento geográfico, sensores de estado do veículo e da carga, comunicação direta com o condutor do veículo, etc.

Destaca-se a importância de interatividade da solução, onde a informação transmitida do veículo para um painel de controle central, possa ser facilmente analisada de diferentes perspectivas, facilitando a identificação de pontos de melhoria, esses podendo serem transmitidos de volta ao condutor do veículo de forma rápida e segura.

Para Grabara (2014), o maior diferencial no que tange às soluções *telematics* está em relação à coleta contínua de dados para análise e também a capacidade de integração desse sistema com outras ferramentas que facilitem o processo de tomada de decisão.

Como já mencionado anteriormente, o custo é algo que sempre busca-se reduzir na realização da operação de transporte, e uma solução de *telematics*, que propõe entregar todas as informações mencionadas em tempo real, com qualidade, confiabilidade, integridade e segurança certamente não terá um preço tão baixo.

Por esse motivo, Grabara (2014) destaca que o primeiro critério na escolha de uma solução de *telematics* para um transportador, não é o seu custo propriamente dito, mas sim o tempo em que o investimento realizado passará a oferecer retornos para a operação. Observe que a contratação de uma solução como essa, em um primeiro momento, gera custos elevados de implantação, treinamento e capacitação de uso, porém, com o uso correto da ferramenta e uma administração coerente com as informações que a solução entrega há a tendência de diminuição de custos, aumento do faturamento e consequentemente o lucro.

Com todo o processo do transporte organizado, com uma cadeia logística organizada e bem estruturada e com a realização do serviço de transporte feito, deve-se imaginar como é realizado o pagamento do frete aos prestadores desse serviço.

Conforme a Agência Nacional de Transportes Terrestres - ANTT destaca, de acordo com o art. 5º-A da Lei nº 11.442, de 2007, o pagamento pelo serviço de transporte realizado por Transportadores Autônomos de Cargas (TAC), por Empresas de Transporte de Cargas (ETC) com até três veículos ou por membros de uma Cooperativa de Transportadores de Carga (CTC), deve ser realizado por meio de crédito em conta depósito mantida por instituição bancária ou por outro meio de pagamento regulado pela ANTT. Dessa forma, é proibido o uso da Carta-Frete para pagamento de frete.

Para viabilizar o controle e a fiscalização dos meios de pagamento de frete,

garantindo o cumprimento da legislação e com objetivo de centralizar e organizar o mercado regulado pela ANTT, foram criadas as regras do Pagamento Eletrônico de Frete (PEF), em que se inserem também as Instituições de Pagamento Eletrônico de Frete – IPEFs.

Atualmente, o PEF é regulamentado por meio da Resolução ANTT nº 5.862, de 17 de dezembro de 2019, norma que substituiu a Resolução ANTT nº 3.658, de 2011, estabelecendo regras referentes ao processo de habilitação de instituições de Pagamento Eletrônico de Frete e às hipóteses de obrigatoriedade de cadastramento da Operação de Transporte e respectiva geração do Código Identificador da Operação de Transporte. A norma define, ainda, as infrações e respectivas penalidades a que estão sujeitos aqueles que venham a descumprir a regulamentação do PEF.

## 5. METODOLOGIA DA PESQUISA

Para esse trabalho utiliza-se a metodologia aplicada em seu desenvolvimento, para que os objetivos gerais e específicos citados sejam atendidos.

Em um primeiro momento, define-se e estuda-se a área de abrangência que a plataforma será capaz de atender. Como trata-se de uma plataforma digital, é válido definir que terá possibilidade de atender e operar em todo o território nacional, desde que os usuários cumpram com os requisitos da plataforma. Nesse processo, há a necessidade de entender como funciona o fluxo de contratação, realização e pagamento de um frete.

Com o entendimento da realização do processo, deve-se partir para a parte de estruturação da aplicação, envolvendo as entidades que fazem parte do processo para que possa se desenhar a regra de negócio. Define-se então que há sempre um motorista e uma empresa disponibilizando uma carga para a realização de um frete. Além disso, o motorista precisa, indispensavelmente, de um veículo para a realização do transporte. Com esses pontos levantados, tem-se 5 entidades principais para definição da regra de negócio da aplicação: motorista, veículo, empresa, carga e o frete em si, como a materialização da realização do serviço de transporte.

Com base nas entidades, se iniciará o desenho do banco de dados a ser utilizado, e como opção de código aberto, gratuita e de muita qualidade, opta-se pelo uso do PostgreSQL na versão 14. O processo a ser realizado no banco de dados, inicialmente, é a definição de informações que cada tabela, representando cada entidade, deverá conter. Deve-se analisar os dados relevantes para o uso na aplicação, buscando normalização de dados, para aumentar a coerência dos dados e facilitar o entendimento da estrutura como um todo.

Para o desenvolvimento da aplicação em si, em um primeiro momento, será construída uma API para manipulação, consulta, inserção e exclusão das informações, fazendo a ponte com o banco de dados criado. Como opção nesse caso, será utilizado a tecnologia do Node.js, que permite a execução de código Javascript e Typescript do lado do servidor, e não apenas no cliente.

Para a construção da interface da aplicação, opta-se pelo uso do framework Vue.js, que permite uma construção sólida visualmente, agregando praticidade de

uso, rapidez no carregamento das informações e um design atrativo para o usuário.

## 6. CRONOGRAMA

Atividades	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov
Pesquisa do tema	X									
Pesquisa bibliográfica		X	X	X						
Coleta de Dados (se for o caso)										
Apresentação e discussão dos dados										X
Elaboração do trabalho				X	X	X	X	X	X	

## 7. LEVANTAMENTO DE REQUISITOS

O objetivo desse levantamento de requisitos é estabelecer uma base sólida para o desenvolvimento do projeto. Com base nos requisitos levantados é possível compreender as necessidades dos possíveis usuários, incluindo empresas e motoristas autônomos.

Os requisitos desse projeto foram levantados com base em uma entrevista realizada com 12 motoristas autônomos residentes no estado de Santa Catarina. Dos motoristas entrevistados, 11 deles relataram que a maior dificuldade encontrada está relacionada a obtenção de contatos para a realização de fretes.

Dada a necessidade, o processo de levantamento de requisitos é iniciado visualizando inicialmente uma necessidade de que a plataforma esteja em ambiente online e que não dependa de instalações adicionais nos dispositivos dos usuários.

Analisando o contexto de uma forma geral, teremos alguns atores envolvidos no processo, visto que são partes integrantes do fluxo a ser desenvolvido, sendo eles: usuário (motorista e empresa), veículo e carga.

Para que o processo seja iniciado, é necessário que o usuário tenha um cadastro na plataforma e esteja com os dados de cadastro atualizados. Os usuários podem ser empresas ou motoristas de caminhão autônomos.

Os usuários do tipo “empresa” poderão cadastrar novas cargas na plataforma, detalhando informações como origem e destino, valor a ser pago pelo serviço, detalhes sobre o tipo de material a ser transportado e datas estimadas para carregamento e entrega.

Os usuários do tipo “motorista” poderão realizar o cadastro de seus veículos na plataforma, e a partir desse cadastro, buscar pela plataforma cargas que sejam compatíveis com sua localização atual, disponibilidade e condição de transporte de acordo com o tipo do caminhão. Com o cadastro e busca realizados, o usuário do tipo “motorista” pode realizar uma aplicação para uma carga disponível, isto é, mostrar disponibilidade para a realização do frete.

Com a aplicação realizada pelo usuário do tipo “motorista”, o usuário do tipo “empresa” poderá acompanhar todos os possíveis motoristas para a realização do frete, entrar em contato diretamente e possivelmente o definir como transportador para a realização do frete.

A partir desse momento, a conexão entre os dois está realizada, podendo se estender a mais parcerias dentro ou fora da plataforma, de acordo com a necessidade dos usuários.

Dados os requisitos funcionais levantados, define-se que uma boa alternativa para esse caso é utilizar tecnologias voltadas para o desenvolvimento web, como Javascript, Node, Vue.js e PostgreSQL. Todas as tecnologias são de uso livre e garantem um desenvolvimento ágil, entregando qualidade no software, além de segurança e integridade, que são pilares muito importantes. Com as tecnologias definidas, deve-se realizar um estudo aprofundado de cada uma delas, visando entender as peculiaridades, funcionamento, restrições e vantagens de uso.

## 8. ESTUDO DAS TECNOLOGIAS

Na seção de “Estudo das Tecnologias” deste trabalho, estaremos imersos em uma análise aprofundada das tecnologias que impulsionam o desenvolvimento de sistemas *web* no contexto de sistemas de informação. Este segmento representa uma exploração minuciosa das ferramentas, linguagens de programação, ambientes de desenvolvimento e *frameworks* que desempenham um papel central na concepção, construção e operação de sistemas *web* modernos. Através desse estudo, nosso objetivo é compreender como essas tecnologias se entrelaçam e se complementam, desempenhando um papel fundamental na capacidade de proporcionar experiências interativas, eficiência operacional e segurança em sistemas *web*. A análise aprofundada dessas tecnologias servirá como alicerce sólido para as discussões subsequentes, nas quais exploraremos suas aplicações específicas e os impactos no contexto de sistemas de informação com foco no desenvolvimento de sistemas *web* de alto desempenho.

### 8.1. JAVASCRIPT

Javascript é uma linguagem de programação amplamente utilizada no desenvolvimento web. Sendo interpretada e orientada a objetos, Javascript é executada no lado do cliente, nos navegadores, e desempenha um papel fundamental na criação de experiências interativas na *web*. Uma característica distintiva do Javascript é sua tipagem dinâmica, o que significa que as variáveis não são rigidamente tipadas e podem mudar de tipo durante a execução do programa. A linguagem segue as especificações do ECMAScript, que padronizam a sintaxe e as funcionalidades da linguagem, tornando-a uma linguagem versátil e de fácil aprendizado (MOZILLA, 2023).

Além de suas capacidades no navegador, o Javascript também é usado no lado do servidor, graças ao Node.js. Isso amplia o escopo do Javascript para além do cliente, permitindo o desenvolvimento de aplicativos *web* completos e eficientes

em termos de recursos. O Javascript desempenha um papel crucial no desenvolvimento moderno, capacitando a criação de aplicativos *web* complexos e aprimorando a experiência do usuário por meio da manipulação do *Document Object Model* (DOM) para interações dinâmicas em tempo real. Juntamente com bibliotecas e *frameworks* populares, como React, Angular e Vue.js, o Javascript é a base de muitas aplicações da *web* moderna (MOZILLA, 2023).

## 8.2. NODE.JS

Node.js é um ambiente de tempo de execução Javascript no lado do servidor que tem revolucionado o desenvolvimento *web*. Baseado no mecanismo V8 do Google Chrome, ele oferece um desempenho excepcional na execução de código Javascript. O Node.js adota uma arquitetura não bloqueante, tornando-o altamente eficiente para lidar com operações de entrada/saída assíncronas e, como resultado, é ideal para aplicações de alta concorrência. Uma de suas características mais notáveis é o *Node Package Manager* (npm), que simplifica o gerenciamento de pacotes e dependências, facilitando a inclusão de bibliotecas de terceiros em projetos (NODEJS, 2023).

Node.js capacita o desenvolvimento de servidores *web* escaláveis, aplicativos em tempo real, APIs e muito mais. Sua combinação de eficiência, escalabilidade e a capacidade de lidar com milhares de conexões simultâneas faz com que seja uma escolha atraente para projetos de larga escala. Além disso, a vasta biblioteca de módulos disponíveis no ecossistema *npm* acelera o desenvolvimento e amplia a funcionalidade das aplicações Node.js. Em resumo, Node.js desempenhou um papel significativo na evolução da computação na *web*, permitindo que os desenvolvedores utilizem Javascript tanto no *frontend* quanto no *backend* (NODEJS, 2023).

### 8.3. EXPRESS.JS

Express.js é um *framework web* minimalista construído em cima do Node.js, projetado para simplificar o desenvolvimento de aplicativos *web* e APIs. Com uma ênfase na simplicidade e flexibilidade, fornece uma estrutura organizada para o tratamento de solicitações HTTP e o roteamento de URLs. Uma de suas principais características é a facilidade de definição de rotas e manipuladores de rota, tornando a criação de *endpoints* de API e páginas da *web* mais acessível (EXPRESS, 2023).

Outro ponto forte do Express.js é seu sistema de *middleware*. Isso permite a execução de funções intermediárias que podem manipular solicitações e respostas antes de chegar ao manipulador de rota final. Essa funcionalidade é valiosa para tarefas como autenticação, validação de dados, registro de solicitações e muito mais. A extensibilidade do Express.js permite que os desenvolvedores personalizem suas aplicações de acordo com suas necessidades específicas, adicionando facilmente pacotes de terceiros a partir do vasto ecossistema *npm*

Devido à sua simplicidade e eficiência, o Express.js se tornou uma escolha popular para o desenvolvimento de aplicativos *web*, especialmente quando é necessário criar APIs REST e *endpoints* de serviços *web*. Além disso, o Express.js possui uma comunidade ativa de desenvolvedores e oferece uma abundância de recursos, tutoriais e bibliotecas de terceiros para atender a uma variedade de necessidades de desenvolvimento (EXPRESS, 2023).

### 8.4. VUE.JS

Vue.js é um *framework* Javascript progressivo e orientado a componentes, utilizado para construir interfaces de usuário interativas e ricas. Ele se destaca por sua simplicidade e facilidade de integração em projetos existentes. Vue.js é executado no lado do cliente, no navegador, e é particularmente eficaz para o desenvolvimento de *Single-Page Applications* (VUEJS, 2023).

De acordo com a documentação oficial do Vue.js, uma característica central

do Vue.js é o seu sistema de componentes, que permite a construção de interfaces modulares e reutilizáveis. Cada componente encapsula seu próprio HTML, CSS e Javascript, facilitando o desenvolvimento e manutenção de código limpo e organizado. O Vue.js também oferece uma reatividade robusta, o que significa que as alterações nos dados do aplicativo automaticamente refletem nas visualizações, proporcionando uma experiência de desenvolvimento eficiente e fluida.

Além disso, Vue.js tem uma comunidade ativa e uma variedade de recursos, incluindo bibliotecas de terceiros e plugins que expandem sua funcionalidade. Seu sistema de roteamento e a integração com ferramentas de compilação, como o Vue CLI, tornam-no uma escolha versátil para o desenvolvimento de aplicações *web* complexas. Vue.js se destaca por sua curva de aprendizado amigável e é amplamente adotado no desenvolvimento *web* moderno, tornando-se uma opção valiosa para a construção de interfaces de usuário dinâmicas e responsivas (VUEJS, 2023).

## 8.5. QUASAR

O Quasar Framework é uma estrutura de código aberto altamente versátil e abrangente para o desenvolvimento de aplicativos *web* e móveis com base em Vue.js. Ele oferece um conjunto robusto de componentes reutilizáveis, juntamente com uma ampla gama de recursos que simplificam e aceleram o processo de desenvolvimento de aplicativos. Uma característica notável do Quasar é sua capacidade de construir aplicativos para várias plataformas, incluindo navegadores, dispositivos móveis e *desktop*, a partir de um único código-fonte (QUASAR, 2023).

Com o Quasar, os desenvolvedores têm acesso a uma biblioteca rica de componentes prontos, como botões, menus, barras de progresso e muito mais, facilitando a criação de interfaces de usuário consistentes e atraentes. Além disso, o *framework* oferece suporte para temas personalizáveis, permitindo a adaptação visual dos aplicativos ao estilo desejado (QUASAR, 2023).

Uma das características mais notáveis do Quasar é a integração perfeita com as funcionalidades do Vue.js, juntamente com o suporte nativo para o Vue Router e

o Pinia para gerenciamento de estados. O Quasar CLI oferece um ambiente de desenvolvimento poderoso, simplificando a configuração do projeto e o processo de compilação (QUASAR, 2023).

Outra vantagem é a sua comunidade ativa e extensa documentação, que oferece tutoriais, exemplos e recursos para ajudar os desenvolvedores a aproveitar ao máximo o Quasar Framework. O Quasar tem se tornado uma escolha popular para o desenvolvimento de aplicativos modernos, devido à sua capacidade de simplificar tarefas complexas e garantir uma experiência de desenvolvimento mais eficiente e produtiva.

## 8.6. POSTGRESQL

O PostgreSQL é um sistema de gerenciamento de banco de dados relacional de código aberto amplamente reconhecido por sua robustez, desempenho e capacidade de extensibilidade. Este sistema de gerenciamento de banco de dados relacional é notável por sua conformidade com padrões SQL e seu suporte para recursos avançados de banco de dados (POSTGRESQL, 2023).

Uma das características distintivas do PostgreSQL é sua arquitetura orientada a objetos, que permite o armazenamento de tipos de dados complexos, como *arrays* e JSON, bem como a definição de funções e procedimentos armazenados. Isso o torna uma escolha popular para uma variedade de casos de uso, desde aplicativos web até aplicações empresariais de missão crítica.

Outra característica notável é a ênfase na extensibilidade. O PostgreSQL suporta extensões personalizadas, permitindo que desenvolvedores criem funcionalidades adicionais para atender a requisitos específicos do projeto. Isso é aprimorado pela vasta comunidade de desenvolvedores que contribuem com extensões e plugins para o ecossistema do PostgreSQL (POSTGRESQL, 2023).

Além disso, o PostgreSQL é elogiado por sua confiabilidade e recursos de recuperação de falhas, tornando-o adequado para aplicações que exigem alta disponibilidade e segurança de dados. Sua documentação abrangente e comunidade ativa tornam o PostgreSQL uma escolha confiável para organizações

que buscam um sistema de gerenciamento de banco de dados confiável, poderoso e altamente personalizável (POSTGRESQL, 2023).

## 8.7. SEQUELIZE

O Sequelize é uma poderosa biblioteca de mapeamento objeto-relacional (ORM) desenvolvida para o ecossistema Javascript. Essa ferramenta desempenha um papel essencial na integração de aplicativos Javascript com bancos de dados relacionais, facilitando a comunicação e a manipulação de dados de maneira eficaz. O Sequelize atua como uma camada intermediária entre a aplicação e o banco de dados, permitindo que os desenvolvedores interajam com os dados usando objetos e métodos em vez de consultas SQL diretas (SEQUELIZE, 2023).

O Sequelize oferece suporte a uma ampla variedade de sistemas de gerenciamento de banco de dados relacionais, como PostgreSQL, MySQL, SQLite e SQL Server, tornando-o altamente flexível e adaptável a diferentes ambientes de desenvolvimento. Sua funcionalidade de criação automática de esquemas e migrações simplifica a configuração inicial e a evolução do banco de dados ao longo do ciclo de vida do aplicativo (SEQUELIZE, 2023).

Além disso, de acordo com a documentação oficial, o Sequelize permite que os desenvolvedores definam modelos de dados, relacionamentos e consultas complexas de maneira simples e eficiente, tornando o acesso aos dados consistente e intuitivo. Sua extensa lista de recursos inclui-se suporte a transações, validações, gatilhos e a capacidade de trabalhar com consultas CRUD (*Create, Read, Update, Delete*) de maneira eficaz.

Em resumo, o Sequelize ORM é uma ferramenta inestimável no desenvolvimento de aplicativos Javascript que dependem de bancos de dados relacionais, proporcionando uma camada de abstração para gerenciar dados de maneira eficiente, aumentando a produtividade do desenvolvedor e promovendo a manutenção de código organizado e escalável.

## 9. DESENVOLVIMENTO DO PROJETO

Nesta seção do trabalho iremos adentrar no cerne da criação do software, levando em consideração as tecnologias previamente exploradas. Após um estudo aprofundado das tecnologias, nossa análise se concentra em como essas ferramentas desempenharam um papel fundamental na concepção, arquitetura e implementação do software.

Este segmento é o coração do nosso estudo, onde traduzimos o conhecimento adquirido das tecnologias em prática, demonstrando como essas ferramentas moldaram o desenvolvimento do software e sua influência na criação de um produto eficaz, responsivo e com alta qualidade. Ao longo deste processo, destacamos os desafios enfrentados, as soluções encontradas e a aplicação das melhores práticas de desenvolvimento de software, oferecendo uma visão abrangente da jornada de criação de um sistema de informação moderno.

Nesta parte do trabalho será detalhado o processo de codificação, a arquitetura do software, a interação com as tecnologias estudadas, os desafios de implementação e as decisões tomadas durante o processo de desenvolvimento.

### 9.1. DEFININDO A ESTRUTURA DO BANCO DE DADOS

Como primeiro passo no desenvolvimento do projeto, é definida a estrutura de tabelas do banco de dados da aplicação. Com o uso do SQL e mais precisamente do DDL, concentramo-nos na criação da estrutura do banco de dados, determinando as tabelas, campos, relacionamentos e índices que serão essenciais para atender aos requisitos da aplicação. A importância desse processo reside na garantia da integridade e eficiência dos dados, bem como na manutenção de um esquema de banco de dados que atende às necessidades do software. Ao definir cuidadosamente a estrutura do banco de dados, garantimos que os dados sejam organizados de forma coerente e otimizada, facilitando consultas e análises. Além

disso, contribui para a escalabilidade do sistema, permitindo a evolução e expansão à medida que novos requisitos surgem ao longo do ciclo de vida do software.

Para começar, vamos definir a estrutura da tabela de usuários, que servirá como base para a criação das demais tabelas, além de que o usuário irá interagir com todos os demais processos sistêmicos. Para a necessidade do software, optou-se por armazenar os dados de código, e-mail, senha, tipo do usuário (motorista ou empresa), código de confirmação da conta e as datas de criação e última atualização do registro, conforme exibido na Figura 1.

Figura 1 - Tabela de usuários.

```
CREATE TABLE IF NOT EXISTS users (  
  id SERIAL PRIMARY KEY,  
  email VARCHAR(255) NOT NULL UNIQUE,  
  password VARCHAR(255) NOT NULL,  
  type VARCHAR(255) NOT NULL CHECK (type IN ('1', '2')),  
  confirmation_token INTEGER,  
  created_at TIMESTAMPTZ DEFAULT NOW(),  
  updated_at TIMESTAMPTZ DEFAULT NOW()  
);
```

Fonte: elaborado pelo autor (2023).

Todas as tabelas seguem o padrão de chave primária na coluna “*id*”, que será um sequencial automático administrado pelo banco de dados. A coluna “*email*” é definida com o uso das palavras chave “NOT NULL” e “UNIQUE”, o que garante que um registro de usuário não possa ter a coluna sem valor informado e também que um mesmo e-mail não seja utilizado por mais de um registro da tabela. A coluna “*password*” irá guardar a informação da senha do usuário encriptada para que não ocorram problemas relacionados à segurança do sistema. A coluna “*type*” identifica se um usuário é motorista (tipo 1) ou empresa (tipo 2), para que futuramente seja possível limitar as ações dentro do software, de acordo com as permissões concedidas a cada tipo de usuário. A coluna “*confirmation\_token*” serve para armazenar um código de 6 dígitos que é enviado para o e-mail do usuário no caso

de solicitação de alteração de senha. Os campos “*created\_at*” e “*updated\_at*” registram a data de criação e última atualização de cada registro, sendo atualizados automaticamente por uma trigger do banco de dados, e estando presentes em todas as próximas tabelas que serão definidas.

A próxima tabela para definição da estrutura é a tabela de cidades. Foi julgado importante a criação de uma tabela de cidades para que não ocorra divergências na descrição dos nomes das cidades, que serão utilizadas para definição dos cadastros das pessoas e das entregas posteriormente, garantindo assim que uma mesma cidade não esteja escrita com nomes diferentes em locais diferentes do software. A estrutura se dá conforme exibido na Figura 2.

Figura 2 – Tabela de cidades.

```
CREATE TABLE IF NOT EXISTS cities (  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(255) NOT NULL,  
  state VARCHAR(255) NOT NULL,  
  created_at TIMESTAMPTZ DEFAULT NOW(),  
  updated_at TIMESTAMPTZ DEFAULT NOW()  
);
```

Fonte: elaborado pelo autor (2023).

A tabela “*cities*” segue o padrão comum de chave primária na coluna “*id*”, que é um número de série único gerado automaticamente pelo banco de dados para identificar exclusivamente cada cidade. A coluna “*name*” armazena o nome da cidade, enquanto a coluna “*state*” guarda a informação sobre o estado ao qual a cidade pertence, garantindo que ambas sejam campos de preenchimento obrigatório, marcados com a restrição “*NOT NULL*”. Além disso, a tabela inclui campos de data e hora para o registro da criação (*created\_at*) e da última atualização (*updated\_at*) de cada cidade, que são automaticamente atualizados pelo banco de dados com a data e hora atuais sempre que um registro é criado ou modificado.

Com usuários e cidades tendo sua estrutura definida, iniciaremos a definição das tabelas com relacionamento de chaves estrangeiras. A primeira delas será a tabela responsável por armazenar os dados das pessoas, sejam pessoas físicas ou jurídicas, que irão realizar interações dentro do software. Aqui teremos o uso da chave estrangeira da tabela “users”, identificando que cada pessoa cadastrada está diretamente relacionada ao cadastro de um usuário, gerando uma relação de 1 para 1. A chave estrangeira para a tabela “cities” será para identificar qual a cidade correspondente para o endereço da pessoa. A estrutura se dá conforme exibido na Figura 3.

Figura 3 – Tabela de pessoas.

```
CREATE TABLE IF NOT EXISTS persons (  
  id SERIAL PRIMARY KEY,  
  cpfnpj VARCHAR(255) NOT NULL UNIQUE,  
  rgie VARCHAR(255),  
  name VARCHAR(255) NOT NULL,  
  birth_date DATE NOT NULL,  
  driver_licence_number VARCHAR(255),  
  driver_licence_category VARCHAR(255),  
  genre VARCHAR(255) CHECK (genre IN ('M', 'F')),  
  phone VARCHAR(255),  
  address_zip_code VARCHAR(255) NOT NULL,  
  address VARCHAR(255) NOT NULL,  
  address_number INTEGER,  
  address_district VARCHAR(255) NOT NULL,  
  address_complement VARCHAR(255) NOT NULL,  
  extra JSONB,  
  created_at TIMESTAMPTZ DEFAULT NOW(),  
  updated_at TIMESTAMPTZ DEFAULT NOW(),  
  user_id INTEGER NOT NULL,  
  address_city_id INTEGER NOT NULL,  
  FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE NO ACTION ON UPDATE NO ACTION,  
  FOREIGN KEY (address_city_id) REFERENCES cities (id) ON DELETE NO ACTION ON UPDATE NO ACTION  
);
```

Fonte: elaborado pelo autor (2023).

A tabela “persons” é projetada para armazenar informações detalhadas sobre pessoas e/ou entidades. Ela começa com uma chave primária “id” que é um número de série único gerado automaticamente para identificar exclusivamente cada registro. A coluna “cpfnpj” armazena números de CPF (Cadastro de Pessoa Física)

ou CNPJ (Cadastro Nacional da Pessoa Jurídica) e é marcada como única para garantir que não haja duplicação de números. A coluna “*rgie*” armazena os números de Registro Geral (RG) ou Inscrição Estadual, quando aplicável.

Os campos “*name*” e “*birth\_date*” armazenam o nome e a data de nascimento da pessoa, ambos são obrigatórios. A tabela também inclui informações sobre a carteira de motorista, como número da carteira (“*driver\_licence\_number*”) e categoria da carteira (“*driver\_licence\_category*”). O campo “*genre*” define o gênero da pessoa, com restrição para “M” (masculino) ou “F” (feminino).

Os campos de contato incluem o número de telefone (“*phone*”) e endereço, abrangendo CEP (“*address\_zip\_code*”), endereço (“*address*”), número do endereço (“*address\_number*”), bairro (“*address\_district*”), e complemento do endereço (“*address\_complement*”). A coluna “*extra*” é do tipo JSONB e permite armazenar informações adicionais.

Além disso, a tabela “*persons*” registra a data e hora de criação (“*created\_at*”) e a última atualização (“*updated\_at*”) de cada registro, ambos atualizados automaticamente pelo banco de dados. Os campos “*user\_id*” e “*address\_city\_id*” estabelecem chaves estrangeiras que se relacionam com as tabelas “*users*” e “*cities*”, respectivamente, vinculando os registros de pessoas a usuários e cidades específicas.

Com a estrutura da tabela “*persons*” definida, vamos para a definição da estrutura da tabela que irá armazenar os dados de veículos dos usuários do tipo 1 (motoristas). A estrutura se dá conforme exibido na Figura 4.

Figura 4 – Tabela de veículos.

```
CREATE TABLE IF NOT EXISTS vehicles (
  id SERIAL PRIMARY KEY,
  licence_plate VARCHAR(255) NOT NULL UNIQUE,
  type VARCHAR(255) NOT NULL CHECK (type IN ('TRACTION', 'TRAILER')),
  renavam VARCHAR(255) NOT NULL,
  brand VARCHAR(255) NOT NULL,
  model VARCHAR(255) NOT NULL,
  year VARCHAR(255) NOT NULL,
  color VARCHAR(255) NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW(),
  person_id INTEGER NOT NULL,
  traction_vehicle_id INTEGER,
  FOREIGN KEY (person_id) REFERENCES persons (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY (traction_vehicle_id) REFERENCES vehicles (id) ON DELETE NO ACTION ON UPDATE NO ACTION
);
```

Fonte: elaborado pelo autor (2023).

A tabela “*vehicles*” tem como objetivo armazenar informações detalhadas sobre veículos, seja para fins de rastreamento, gerenciamento ou qualquer outra finalidade. A coluna “*id*” é uma chave primária que gera automaticamente números de série exclusivos para identificar cada veículo.

A coluna “*licence\_plate*” armazena o número da placa do veículo, com a restrição “*NOT NULL*” para garantir que esse campo seja sempre preenchido e “*UNIQUE*” para evitar duplicatas. O campo “*type*” define o tipo de veículo, com opções “*TRACTION*” (veículo de tração) ou “*TRAILER*” (reboque).

Outros campos importantes incluem “*renavam*” (Registro Nacional de Veículos Automotores), “*brand*” (marca do veículo), “*model*” (modelo do veículo), “*year*” (ano do veículo) e “*color*” (cor do veículo). Todos esses campos são obrigatórios para garantir que as informações do veículo sejam completas.

A tabela “*vehicles*” também mantém registros das datas de criação (“*created\_at*”) e última atualização (“*updated\_at*”) de cada veículo, com ambas as colunas atualizadas automaticamente pelo banco de dados. Os campos “*person\_id*” e “*traction\_vehicle\_id*” são chaves estrangeiras que estabelecem relacionamentos com a tabela “*persons*” e, quando aplicável, com outros veículos de tração.

Com a definição das tabelas de usuário, cidades, pessoas e veículos, vamos agora definir a estrutura da tabela responsável por armazenar os dados das entregas. A estrutura definida está exibida na Figura 5.

Figura 5 – Tabela de entregas.

```
CREATE TABLE IF NOT EXISTS deliveries (
  id SERIAL PRIMARY KEY,
  expected_loading_date DATE NOT NULL,
  expected_delivery_date DATE NOT NULL,
  total_km NUMERIC(15,3) NOT NULL,
  value_per_km NUMERIC(15,2) NOT NULL,
  additional_value NUMERIC(15,2) NOT NULL,
  delivery_value NUMERIC(15,2) NOT NULL,
  status VARCHAR(255) NOT NULL CHECK (status IN ('AVAILABLE', 'PICKED', 'IN PROGRESS', 'DELIVERED', 'CANCELED')) DEFAULT 'AVAILABLE',
  details TEXT,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW(),
  "from" INTEGER NOT NULL,
  "to" INTEGER NOT NULL,
  company_id INTEGER NOT NULL,
  driver_id INTEGER,
  driver_vehicle_id INTEGER,
  FOREIGN KEY ("from") REFERENCES cities (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY ("to") REFERENCES cities (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY (company_id) REFERENCES persons (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY (driver_id) REFERENCES persons (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY (driver_vehicle_id) REFERENCES vehicles (id) ON DELETE NO ACTION ON UPDATE NO ACTION
);
```

Fonte: elaborado pelo autor (2023).

A tabela “*deliveries*” tem como objetivo rastrear informações relacionadas a entregas, fornecendo detalhes essenciais para o gerenciamento eficiente do processo logístico. A coluna “*id*” serve como uma chave primária e gera automaticamente um número inteiro único para identificar cada entrega.

Os campos “*expected\_loading\_date*” e “*expected\_delivery\_date*” armazenam a data prevista para o carregamento e a data prevista para a entrega da carga, ambas marcadas como obrigatórias para garantir que sejam sempre informadas. A coluna “*total\_km*” guarda a distância total da entrega em quilômetros, enquanto “*value\_per\_km*” e “*additional\_value*” representam os valores associados à entrega por quilômetro e custos adicionais, respectivamente. A coluna “*delivery\_value*” registra o valor total da entrega.

O campo “*status*” define o estado da entrega, com opções como “*AVAILABLE*” (disponível), “*PICKED*” (com motorista já definido), “*IN PROGRESS*” (em andamento), “*DELIVERED*” (entregue) e “*CANCELED*” (cancelado), com um valor padrão de “*AVAILABLE*”. A coluna “*details*” permite a inclusão de informações adicionais, como observações ou notas relacionadas à entrega.

Além disso, a tabela “*deliveries*” mantém registros das datas de criação (“*created\_at*”) e última atualização (“*updated\_at*”) de cada entrega, ambas atualizadas automaticamente pelo banco de dados. Os campos “*from*” e “*to*” estabelecem chaves estrangeiras que se relacionam com a tabela “*cities*”, indicando as cidades de origem e destino da entrega.

Outras chaves estrangeiras incluem “*company\_id*”, que se relaciona com a tabela “*persons*” para identificar a empresa envolvida na entrega, “*driver\_id*” que se relaciona com a tabela “*persons*” para identificar o motorista responsável, e “*driver\_vehicle\_id*” que se relaciona com a tabela “*vehicles*” para identificar o veículo utilizado na entrega.

A última tabela a ser definida será a responsável por armazenar as “candidaturas” dos motoristas para determinada carga. Por meio dessa tabela, usuários do tipo 2 (empresas) poderão verificar todos os motoristas que se colocaram à disposição para a realização de uma entrega específica e então realizar o contato para a concretização da realização do serviço de transporte. A definição da estrutura da tabela é conforme exibido na Figura 6.

Figura 6 – Tabela das candidaturas dos motoristas para realização das entregas.

```
CREATE TABLE IF NOT EXISTS delivery_driver_offers (
  id SERIAL PRIMARY KEY,
  status VARCHAR(255) NOT NULL CHECK (status IN ('APPLIED', 'APPROVED', 'DENIED', 'CANCELED')),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW(),
  driver_id INTEGER NOT NULL,
  driver_vehicle_id INTEGER,
  delivery_id INTEGER NOT NULL,
  FOREIGN KEY (driver_id) REFERENCES persons (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY (driver_vehicle_id) REFERENCES vehicles (id) ON DELETE NO ACTION ON UPDATE NO ACTION,
  FOREIGN KEY (delivery_id) REFERENCES deliveries (id) ON DELETE NO ACTION ON UPDATE NO ACTION
);
```

Fonte: elaborado pelo autor (2023).

A tabela “*delivery\_driver\_offers*” tem como objetivo rastrear as candidaturas feitas por motoristas para assumir entregas específicas, registrando detalhes cruciais do processo de atribuição de entregas a motoristas. A coluna “*id*” serve como uma chave primária e gera automaticamente números de série exclusivos para identificar cada oferta.

O campo “*status*” define o estado da candidatura, com opções como “*APPLIED*” (aplicada), “*APPROVED*” (aprovada), “*DENIED*” (negada) e “*CANCELED*” (cancelada). Essa coluna garante que o status da oferta seja sempre uma das opções válidas.

A tabela “*delivery\_driver\_offers*” mantém registros das datas de criação (“*created\_at*”) e última atualização (“*updated\_at*”) de cada oferta, ambas atualizadas automaticamente pelo banco de dados. Os campos “*driver\_id*” e “*driver\_vehicle\_id*” estabelecem chaves estrangeiras que se relacionam com a tabela “*persons*” e “*vehicles*” para identificar o motorista e o veículo associados à oferta. Além disso, a chave estrangeira “*delivery\_id*” se relaciona com a tabela “*deliveries*” para vincular a oferta a uma entrega específica.

Essa estrutura de tabela é crucial para rastrear o processo de atribuição de entregas a motoristas, permitindo o registro claro do status de cada oferta e sua relação com motoristas, veículos e entregas específicas. Ela oferece suporte à comunicação eficaz entre motoristas e operadores logísticos, permitindo o acompanhamento de quem está disponível e disposto a assumir uma entrega específica. Além disso, garante a integridade dos dados e a organização das

informações relacionadas a ofertas de entrega.

## 9.2. DESENVOLVIMENTO DA API

Com a estrutura do banco de dados definida, o próximo passo é dar início ao desenvolvimento da API. A API é a parte do software responsável por receber requisições da interface de usuário, realizar interações com o banco de dados, tratar os dados e os retornar para o usuário. Com o estudo realizado das tecnologias e seguindo padrões de desenvolvimento consolidados na comunidade de programadores Node.JS, o projeto da API é separado em diferentes níveis, cada um com a sua responsabilidade, sendo basicamente: *models*, *services*, *controllers*, *routers* e *middlewares*. Antes de detalhar a responsabilidade de cada um deles, é necessária criação do projeto, utilizando o Gerenciador de Pacotes do Node.JS – NPM. O NPM é uma plataforma aberta para que desenvolvedores publiquem seus projetos e os disponibilizem para uso de outros desenvolvedores, sem custo.

### 9.2.1 Criação do projeto da API

O primeiro passo para o desenvolvimento é a criação do arquivo principal do nosso projeto, o qual irá armazenar as informações como versão, autor, dependências, *scripts* e demais informações. Para isso, usa-se o comando representado na Figura 7.

Figura 7 – Criação do projeto da API.

```
$ npm init -y  
$ npm install bcryptjs cors dotenv express jsonwebtoken moment nodemailer pg sequelize
```

Fonte: elaborado pelo autor (2023).

O comando “npm init -y” inicia um projeto base com as configurações padrões do NPM, podendo ser personalizado conforme a necessidade. Abaixo, o comando “npm install”, realiza a instalação das dependências desejadas no projeto. Conforme já explicado anteriormente no estudo das tecnologias, o Express é responsável pelo *core* da aplicação. A dependência do Sequelize e do PostgreSQL também são adicionadas ao projeto para que possam realizar as interações com o banco de dados. As demais bibliotecas são responsáveis por realizar tratamentos específicos no desenvolvimento da API.

O “bcryptjs” é uma biblioteca que oferece uma maneira direta de realizar o *hash* de senhas, garantindo que as senhas dos usuários estejam criptografadas, protegendo assim as contas de usuários contra acessos não autorizados e eventuais violações de dados.

O “cors” desempenha um papel crucial no que diz respeito a segurança da API. Essa dependência possibilita ou restringe solicitações entre origens diferentes, garantindo que aplicativos web possam interagir de forma segura com recursos provenientes de domínios diversos. Isso ajuda a mitigar o risco de ataques de falsificação de solicitações entre sites e outras ameaças de segurança.

O “dotenv” simplifica a gestão de variáveis de configuração em aplicações Node.JS. Essa biblioteca permite que os desenvolvedores armazenem configurações, como chaves de API ou *strings* de conexão de banco de dados, em um arquivo de ambiente separado. Essa abordagem aprimora a segurança ao manter informações sensíveis fora do código-fonte e facilita alterações de configuração sem a necessidade de modificar o código-fonte diretamente.

O “jsonwebtoken”, popularmente conhecido como JWT, permite a implementação de mecanismos de autenticação seguros. Os tokens JWTs são comumente utilizados para gerenciamento de sessões e autenticação de usuários, proporcionando uma solução escalável e sem estado para a transmissão segura de informações entre cliente e servidor.

O “moment” simplifica o manuseio e manipulação de datas e horários em JavaScript. Essa biblioteca fornece uma API fácil de usar para análise, formatação e manipulação de datas e horas, superando algumas das limitações associadas ao objeto nativo Date em JavaScript. Os desenvolvedores frequentemente utilizam o “moment” para realizar várias operações relacionadas ao tempo, como calcular durações, formatar datas para exibição ou trabalhar com fusos horários.

Por último, o “nodemailer” facilita o envio de e-mails a partir de aplicações Node.JS. Ele oferece suporte à criação e envio de e-mails usando vários métodos de transporte, incluindo SMTP e sendmail.

Com a criação do projeto e a instalação das dependências, é criado o arquivo raiz, chamado de “package.json”. Conforme descrito anteriormente, ele armazena todas as informações do projeto e é representado na Figura 8.

Figura 8 - Arquivo package.json com as configurações do projeto.

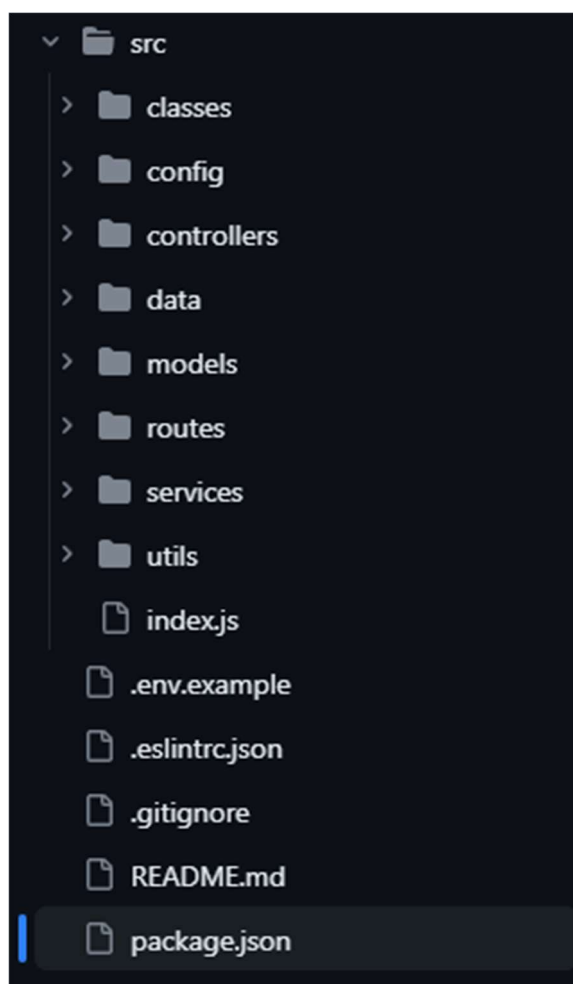
```
{
  "name": "carga-certa-api",
  "version": "1.0.0",
  "description": "",
  "type": "module",
  "main": "index.js",
  "scripts": {
    "dev": "nodemon --experimental-specifier-resolution=node src/index.js",
    "commit": "cz"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "commitizen": "^4.2.6",
    "cors": "^2.8.5",
    "dotenv": "^16.0.3",
    "express": "^4.18.2",
    "jsonwebtoken": "^8.5.1",
    "moment": "^2.29.4",
    "morgan": "^1.10.0",
    "nodemailer": "^6.9.4",
    "pg": "^8.8.0",
    "sequelize": "^6.25.5"
  },
  "devDependencies": {
    "cz-conventional-changelog": "^3.3.0",
    "eslint": "^8.31.0",
    "nodemon": "^2.0.20"
  },
  "config": {
    "commitizen": {
      "path": "./node_modules/cz-conventional-changelog"
    }
  }
}
```

Fonte: elaborado pelo autor (2023).

É possível observar outras dependências, além das que instalamos, mas opta-se por não detalhar seu papel no projeto por serem dependências utilizadas para facilitar a atividade de desenvolvimento em si, como a padronização de mensagens de *commit*, o registro de *logs* e definição de padrões do código, como indentação, por exemplo.

A configuração inicial do projeto se finaliza com a criação da estrutura de pastas, para melhor organização e separação de responsabilidades de cada parte da API. A estrutura de pastas pode ser visualizada na Figura 9.

Figura 9 – Estrutura de pastas da API.



Fonte: elaborado pelo autor (2023).

A organização acima contempla os níveis que foram citados no trabalho (*models*, *services*, *controllers*, *routers* e *middlewares*) e também outras pastas como *classes*, *config*, *data* e *utils*, que serão mencionadas durante o trabalho.

## 9.2.2 Models

Os *models* constituem a camada fundamental na API, implementados em conformidade com as diretrizes do *Sequelize* para representar de maneira precisa e eficiente as tabelas do banco de dados no contexto da aplicação. Cada *model* encapsula a estrutura de uma tabela específica, definindo um objeto Javascript que reflete as propriedades correspondentes a cada coluna da tabela associada. Esta abordagem visa fornecer uma interface coesa e orientada a objetos para interagir com os dados armazenados, simplificando operações de criação, leitura, atualização e exclusão.

Além de serem a representação fiel das tabelas, os *models* no *Sequelize* desempenham um papel crucial na abstração do acesso ao banco de dados, permitindo que os desenvolvedores interajam com os dados usando métodos e propriedades familiares, sem a necessidade de consultas SQL diretas. Esta camada de abstração facilita a manutenção do código e aumenta a legibilidade, ao mesmo tempo em que oferece um conjunto de funcionalidades avançadas.

Os *models* também incorporam poderosas capacidades, como definição de relações entre tabelas, validações de dados e a inclusão de métodos personalizados. Essas características adicionam uma camada de sofisticação ao acesso aos dados, possibilitando uma modelagem de dados mais complexa e garantindo a integridade e consistência dos dados ao longo da aplicação.

Para exemplificar, o modelo de usuário no *Sequelize*, referente à tabela *users* definida anteriormente, é construído com precisão, atribuindo propriedades Javascript a cada coluna correspondente. Utilizando tipos de dados como *INTEGER* e *STRING*, as opções adicionais, como *primaryKey*, *autoIncrement*, *unique* e *allowNull*, configuram restrições específicas. A validação, incorporada pelo *Sequelize*, inclui verificações como “*isEmail: true*” para garantir a validade dos dados. As opções finais, como *freezeTableName*, *timestamps*, *createdAt* e *updatedAt*, contribuem para o comportamento do modelo, preservando nomes e automatizando registros temporais. Assim, este modelo encapsula a estrutura da tabela e suas validações, proporcionando uma representação eficaz da entidade

“usuário” no contexto da API. O código de criação do *model* de usuário se dá conforme exibido na Figura 10.

Figura 10 – Definição de modelo de usuário.

```
import { DataTypes } from 'sequelize';
import { sequelize } from '../config';

const User = sequelize.define(
  'users',
  {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    email: {
      type: DataTypes.STRING,
      unique: true,
      allowNull: false,
      validate: {
        isEmail: true
      }
    },
    password: {
      type: DataTypes.STRING,
      allowNull: false
    },
    type: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        isIn: [[1, 2]] // 1 - Motorista, 2 - Empresa
      }
    },
    confirmationToken: {
      type: DataTypes.INTEGER,
      field: 'confirmation_token'
    }
  },
  {
    freezeTableName: true,
    timestamps: true,
    createdAt: 'created_at',
    updatedAt: 'updated_at'
  }
);

export default User;
```

Fonte: elaborado pelo autor (2023).

Diferente da estrutura de banco de dados, no caso dos *models*, *services*, *controllers*, *routers* e *middlewares* não serão exemplificados todos os arquivos de código, já que muitas vezes a única coisa que muda são as propriedades, que variam de acordo com a estrutura de banco de dados previamente explicada. Na próxima sessão do trabalho será exemplificado como os *models* são os utilizados dentro dos *services* para facilitar a interação com o banco de dados.

### 9.2.3 Services

Os *services* são a camada responsável pela lógica de negócios e pela execução de operações específicas dentro da API. Eles encapsulam a funcionalidade da aplicação, implementando as operações principais necessárias para atender às solicitações dos clientes. Eles interagem com os *models* para realizar operações no banco de dados, fornecendo uma camada de abstração que isola as regras de negócios complexas do restante da aplicação.

Cada *service* é projetado para realizar tarefas específicas, como no caso dessa aplicação, cadastro de usuários, realização de autenticação, cadastro de cargas, veículos, entre outros.

Além de comunicação com banco de dados, qualquer lógica adicional, como cálculos, verificações de dependências entre entidades, chamadas de APIs externas, gerenciamento de *cache*, manipulação de arquivos e tratamento de exceções são implementados dentro do *service*.

A nível de exemplo, documenta-se aqui nesse trabalho a criação do *service* responsável por lidar com toda a lógica que trabalha com a estrutura de usuários. Entende-se como necessário para a aplicação, que um usuário possa ser registrado, possa realizar login no sistema, possa redefinir sua senha (seja por esquecimento, via recuperação após instruções enviadas por e-mail, ou dentro do sistema) e que possa visualizar seus dados de cadastro.

É importante que cada função dentro do *service* observe a necessidade de ser assíncrona ou não, isto é, que aguarde determinadas instruções serem

executadas antes de dar continuidade na execução. Para isso são utilizadas as palavras chave *async/await*.

A criação do método de registro de novos usuários pode ser visualizada na Figura 11.

Figura 11 – Método de registro de novos usuários.

```
async function register (params) {
  const { email, password, type } = params;

  let userExists = await User.findOne({ where: { email } });

  if (userExists) {
    throw new ApiException(400, 'Já existe um usuário cadastrado com esse e-mail!');
  }

  let passwordHash = await bcrypt.hash(password, 10);

  return await User.create({
    email,
    password: passwordHash,
    type
  });
}
```

Fonte: elaborado pelo autor (2023).

O método acima faz o cadastro de novos usuários na aplicação. O método é definido como assíncrono, o que indica que haverá interações com recursos onde é necessário aguardar uma resposta antes de dar continuidade na execução do restante da função. Esse método recebe como parâmetro um objeto que contém os dados do usuário que deseja se registrar, como e-mail, senha e o tipo (empresa ou motorista).

Antes de inserir um novo usuário no sistema, é realizada uma verificação para garantir que o e-mail informado ainda não esteja em uso por outro usuário da plataforma. Caso o e-mail já esteja em uso, é lançada uma exceção que irá retornar uma mensagem de erro ao cliente, indicando o problema que ocorreu.

Caso o e-mail não esteja em uso ainda, significa que um novo usuário pode

ser cadastrado no sistema. Para isso, antes de armazenar o novo usuário no banco de dados, a biblioteca *bcrypt* faz a criptografia da senha para garantir que não seja armazenada a senha “crua” do usuário, o que pode gerar problemas de segurança em uma eventual situação de vazamento de dados.

Com a senha criptografada e agora realizando o uso da camada explicada anteriormente, o método “*User.create()*” cria uma nova instancia do *model* de usuário e o armazena no banco de dados, conforme os dados passados por parâmetro para o método.

Os dados do usuário cadastrado são retornados ao final da chamada, para dar um retorno visual de que os dados informados foram de fato persistidos no banco de dados da aplicação.

Observa-se que nesse caso, o método “*User.create()*” faz o equivalente à instrução SQL de “*INSERT INTO*”, e o método “*User.findOne()*” faz o equivalente à instrução SQL de “*SELECT \* FROM USERS*”, adicionando a condição de “*WHERE*” para retornar apenas o registro que contenha o e-mail informado na chamada do método.

Dando continuidade ao *service* de usuários, vamos ao desenvolvimento do método para que usuários possam realizar login na plataforma. O método se dá conforme exibido na Figura 12.

Figura 12 – Método de login de usuários.

```
async function login (params) {
  const { email, password } = params;

  const user = await User.findOne({ where: { email } });

  if (user || !(await bcrypt.compare(password, user.password))) {
    throw new ApiException(401, 'E-mail ou senha incorretos!');
  }

  const token = jwt.sign(
    { userId: user.id, email: user.email, type: user.type },
    process.env.TOKEN_PRIVATE_KEY,
    { expiresIn: '1h', algorithm: 'HS256' }
  );

  return {
    user,
    token
  };
}
```

Fonte: elaborado pelo autor (2023).

Assim como o método de registro, o login também precisa ser assíncrono, por estar realizando interações com o banco de dados da aplicação. Nesse método, é esperado que seja passado por parâmetro um objeto com as informações de e-mail e senha do usuário para login no sistema.

Nesse método, assim como no método de registro do usuário, é buscado no banco de dados um registro de usuário que contenha o e-mail informado na tentativa de login. Na sequência é feita uma validação com a biblioteca *bcrypt* para verificar se a senha “crua” informada pelo usuário na requisição corresponde a senha criptografada que está guardada no banco de dados. Caso o usuário não seja encontrado pelo e-mail informado ou caso a validação de senha do *bcrypt* falhe, é retornada uma exceção, informando ao cliente que a requisição não pode ser completada por e-mail ou senha incorretos. Essa abordagem foi escolhida para que não sejam retornadas informações de que apenas o e-mail ou a senha estão incorretas, para dificultar tentativas de invasão por quebra de senha de usuários.

Caso o e-mail e senha estejam corretos, com a biblioteca *jsonwebtoken* é gerado um token de acesso para o usuário. Esse token irá armazenar um *payload*, que é um objeto Javascript que guarda informações relevantes para o fluxo da aplicação, sendo nesse caso, o código, e-mail e tipo do usuário que está fazendo login. O token é então “assinado” com uma chave privada configurada no projeto, garantindo que o token não seja facilmente decifrado. Por último, são passadas informações de tempo de expiração do token gerado e qual o algoritmo utilizado para a criptografia do token.

Ao final da requisição, são retornados os dados do usuário logado e o token de autenticação gerado pelo *jsonwebtoken*. Posteriormente, nesse trabalho, será demonstrado como o token gerado é utilizado e tratado pelos *middlewares* para que seja possível o controle de acesso por autenticação e perfis de acesso à aplicação.

Outra funcionalidade implementada no *service* de usuários é a que permite que o usuário solicite uma redefinição de senha por e-mail. Nesse caso, foi utilizado a biblioteca *nodemailer*, que faz a abstração do envio de e-mail no Javascript. O usuário informa seu e-mail e caso corresponda a um cadastro já existente no sistema, é disparado uma notificação para o e-mail informado, contendo um token de 6 dígitos para ser posteriormente informado na tela de redefinição de senha. O código desse método pode ser visualizado na Figura 13.

Figura 13 – Método de solicitação de recuperação de senha.

```

async function forgotPassword (params) {
  const { email } = params;

  const user = await User.findOne({ where: { email } });

  if (!user) {
    throw new ApiException(404, 'Usuário não encontrado');
  }

  const confirmationToken = Math.floor(Math.random() * 1000000) % 1000000;
  user.confirmationToken = confirmationToken;
  await user.save();

  const emailParams = {
    to: [email],
    subject: 'Solicitação de Recuperação de Senha - Carga Certa',
    html: `
      <div style="background-color: #f4f4f4; padding: 20px;">
        <h2>Recuperação de Conta</h2>
        <p>Olá,</p>
        <p>Você está recebendo este e-mail porque solicitou a recuperação da sua conta.</p>
        <p style="font-size: 20px; background-color: #e2e2e2; padding: 10px; text-align: center;">
          <p>Insira o código acima no sistema para confirmar a recuperação da sua conta.</p>
          <p>Se você não solicitou essa recuperação de conta, ignore este e-mail.</p>
          <p>Atenciosamente,<br>Equipe Carga Certa</p>
        </div>
      `;
  };

  await emailUtil.sendMail(emailParams);
}

```

Fonte: elaborado pelo autor (2023).

Novamente, o e-mail é parâmetro para a função, que verifica inicialmente se existe o cadastro de algum usuário com o e-mail informado. Caso não seja encontrado nenhum registro com o e-mail informado, é retornada uma mensagem de erro com essa informação para o usuário. Caso o registro exista para o e-mail informado, com o uso da biblioteca *Math* do Javascript é gerado um número aleatório de 6 dígitos. Esse número aleatório será o token de confirmação que o

usuário irá receber por e-mail, para garantir que o usuário que fez a solicitação seja mesmo o proprietário da conta. O mesmo token é guardado no banco de dados, junto da informação do usuário cadastrado. A variável *emailParams* recebe as informações necessárias para que o *nodemailer* consiga fazer o envio do e-mail. Para isso, é informado o e-mail de destino, o assunto da mensagem e o texto que irá compor o corpo do e-mail. A biblioteca permite que seja realizado o uso de *HTML*, o que permite personalizações de estilo conjuntamente com a aplicação de *CSS*. Com os parâmetros definidos, o método “*emailUtil.sendMail()*” faz o envio da mensagem de acordo com os dados definidos nos parâmetros. O e-mail recebido pelo usuário pode ser visualizado na Figura 14.

Figura 14 – E-mail de recuperação de senha.



Fonte: elaborado pelo autor (2023).

O próximo método do *service* de usuários é o responsável por fazer de fato a alteração de senha do usuário. Esse método é utilizado em duas situações: no caso de solicitação de recuperação da conta por e-mail ou simplesmente quando o usuário desejar realizar a alteração, já estando logado no sistema. Optou-se nesse caso por manter o tratamento para as duas situações no mesmo método, para evitar

a duplicidade de código dentro do *service*. O código do método pode ser visualizado na Figura 15.

Figura 15 – Método de alteração de senha do usuário.

```
async function changePassword (params) {
  const { email, newPassword, oldPassword, type, confirmationToken } = params;

  const user = await User.findOne({ where: { email } });

  if (type === 'reset' && confirmationToken !== user.confirmationToken) {
    throw new ApiException(400, 'Código de recuperação de conta inválido');
  }

  if (type === 'change' && !(await bcrypt.compare(oldPassword, user.password))) {
    throw new ApiException(400, 'Senha atual incorreta');
  }

  let passwordHash = await bcrypt.hash(newPassword, 10);
  user.password = passwordHash;
  user.confirmationToken = null;
  return await user.save();
}
```

Fonte: elaborado pelo autor (2023).

Nesse método, o objeto que é passado por parâmetro contém as informações de e-mail, nova senha, senha antiga, tipo de alteração (dependendo da situação, conforme explicado anteriormente) e o token de confirmação para alteração da senha.

Para começar, o usuário é buscado no banco de dados conforme o e-mail enviado na requisição. Conforme a situação da alteração de senha, são feitas validações específicas visando garantir a segurança de que é o proprietário da conta que está realizando a alteração.

No caso de ser uma recuperação de conta o *type* será “reset” e é feita a verificação se o token de confirmação informado é o mesmo token que foi gerado no método anterior de recuperação de conta. Caso o código não corresponda, é retornado uma mensagem de erro informando a situação.

No caso de ser uma alteração de senha com o usuário já logado no sistema, o *type* será “change” e é verificado se a senha antiga informada na requisição corresponde com a senha atualmente gravada no cadastro do usuário. Caso não corresponda, é retornada uma mensagem de erro informando a situação.

Se todas as verificações forem atendidas, é realizada a encriptação da nova senha informada pelo usuário e então essa senha é atribuída ao cadastro do usuário no banco de dados.

Com esses métodos, o *service* de usuários está completo e atendendo as necessidades para as situações e requisitos da plataforma. Assim como foi mencionado anteriormente, nesse trabalho não será explicado em integralidade todo o código de *services* para todas as entidades do sistema. Os métodos utilizados na explicação do *service* de usuários são os mesmos utilizados em todos os outros *services*, com a diferença de que os parâmetros vão ser diferentes e algumas regras específicas vão ser aplicadas de acordo com a necessidade da funcionalidade.

## 9.2.4 Controllers

Os *controllers* representam a camada na qual as requisições da API são recebidas, processadas e respondidas. Sua função é atuar como intermediário entre as solicitações do cliente e a lógica de negócios implementada nos *services*. Cada *controller* está associado a um conjunto específico de rotas e é responsável por interpretar os dados da requisição, acionar as operações apropriadas nos *services* correspondentes e enviar uma resposta de volta ao cliente.

Dentro do contexto desta aplicação, os *controllers* desempenham um papel crucial na manipulação das diversas funcionalidades oferecidas, como o cadastro de usuários, autenticação, gestão de cargas e veículos, entre outras. Eles recebem os parâmetros das requisições HTTP, validam esses dados conforme necessário e encaminham as instruções aos *services* para execução.

A criação de *controllers* específicos para cada entidade ou funcionalidade contribui para uma arquitetura organizada e escalável. Cada função dentro de um *controller* é projetada para responder a um tipo específico de requisição, seja ela

relacionada a inserções, consultas, atualizações ou exclusões.

É importante observar que os *controllers* não realizam lógica de negócios diretamente; em vez disso, eles coordenam as operações delegando a execução real para os *services* correspondentes. Isso promove uma clara separação de responsabilidades e facilita a manutenção e expansão do código.

Assim como nos *services*, nesse trabalho será exemplificado o código com o contexto da entidade de usuários. Sendo assim, o primeiro método do *controller* de usuários será o responsável por receber e tratar a requisição de cadastro de novos usuários. A implementação pode ser visualizada na Figura 16.

Figura 16 – Implementação de método de registro de usuários no controller.

```
async function register (req, res) {
  try {
    if (!req.body.email || !req.body.password || !req.body.type) {
      throw new ApiEception(400, 'Campos obrigatórios não informados');
    }

    const response = await service.register(req.body);
    return res.status(200).send({
      type: 'success',
      message: 'Usuário cadastrado com sucesso!',
      data: response
    });
  } catch (err) {
    handleError(res, err);
  }
}
```

Fonte: elaborado pelo autor (2023).

Observa-se nesse caso que o método recebe dois parâmetros. O primeiro deles, o parâmetro “req”, contém todos os dados da requisição realizada pelo cliente, como cabeçalhos que indicam origem, cookies de autenticação e o corpo da requisição onde estão os dados que desejamos nesse caso salvar. O segundo parâmetro (“res”) é utilizado para que possamos dar uma resposta para a requisição.

Inicialmente há uma validação de campos obrigatórios para que a interação com o banco de dados posteriormente no *service* não tenha problemas. Nessa situação, caso o usuário não informe e-mail, senha ou o tipo de conta que está criando será retornada uma mensagem de erro informando que os campos obrigatórios estão ausentes na requisição.

Se os campos obrigatórios tiverem sido informados, o *controller* aguarda a execução da função de cadastro de usuários no *service*, a qual já vimos anteriormente. Com a função do *service* retornando o usuário cadastrado, utilizamos o parâmetro “res” para definir um status 200, que simboliza sucesso na requisição, e retornamos para o cliente informações como o tipo do retorno, uma mensagem indicando sucesso e o usuário cadastrado.

Caso algum erro ocorra na requisição, seja algum erro não esperado ou os erros que são indicados manualmente conforme as validações realizadas no *service* e no *controller*, o bloco “catch” da função do controller vai tratar o erro chamando a função “handleError()”, que faz o tratamento conforme implementação que pode ser visualizada na Figura 17.

Figura 17 – Método de tratamento de erro

```
async function handleError (res, err) {
  if (err instanceof ApiException) {
    return res.status(err.status).json({
      type: 'error',
      message: err.message,
      data: err.data
    });
  }
  return res.status(500).send({
    type: 'error',
    message: err.message,
    err
  });
}
```

Fonte: elaborado pelo autor (2023).

O método acima verifica se o erro que chegou para tratamento é um erro gerado pela API propositalmente ou se é um erro de Javascript. Conforme a situação, serão exibidas as mensagens definidas na geração do erro e será dado o feedback coerente para que o cliente saiba exatamente qual foi o problema e saiba o que está fazendo de errado.

Todos os métodos de *controller* seguem o mesmo exemplo do que foi exemplificado nesse caso, por esse motivo não estará sendo detalhado cada um dos métodos do *controller* de usuários. Importante é ressaltar que o *controller* pode conter validações referentes a requisição, no que diz respeito a obrigatoriedade de campos, por exemplo, mas não é o ideal que contenha validações da regra de negócio.

### 9.2.5 Routers

Os *routers* desempenham um papel crucial na definição e roteamento das rotas da API. Eles são responsáveis por mapear as URLs para os controladores correspondentes, direcionando as requisições HTTP para as operações específicas que manipulam dados e processam as solicitações. Essa camada é essencial para estruturar e facilitar o acesso às funcionalidades oferecidas pela aplicação, proporcionando uma interface clara para interações externas. Em resumo, os *routers* são o ponto de entrada que organiza e direciona o tráfego, conectando as requisições externas às operações internas da API.

As funções dos *routers* apontam para determinados métodos dos *controllers*, e além disso podem passar por *middlewares* no fluxo da execução da requisição. O exemplo de implementação para as rotas relacionadas à entidade de usuário no sistema pode ser visualizado na Figura 18.

Figura 18 – Implementação das rotas de usuário.

```
import controller from '../controllers/usersController';
import { verifyJWT } from '../utils/auth';

export default (app) => {
  app.post('/users/register', controller.register);
  app.post('/users/login', controller.login);
  app.post('/users/forgot-password', controller.forgotPassword);
  app.post('/users/change-password', controller.changePassword);
  app.get('/users/info', verifyJWT, controller.getUserInfo );
};
```

Fonte: elaborado pelo autor (2023).

Para cada rota é definido o método HTTP que permite acesso ao conteúdo e uma URL dentro da aplicação para ter acesso ao método. Na sequência definem-se os métodos que serão executados quando alguma requisição “bater” na rota em questão. Observa-se que os métodos são executados na ordem em que estão definidos no cadastro da rota, ou seja, no exemplo da rota “/users/info”, é executada primeiro a função “verifyJWT()” e na sequência o método do *controller* desejado.

## 9.2.6 Middlewares

Os *middlewares* desempenham um papel crucial na manipulação das requisições e respostas ao longo do ciclo de vida da API. Colocados entre os *routers* e os *controllers*, eles oferecem uma camada de flexibilidade para introduzir lógicas específicas em diferentes pontos do fluxo de execução.

Essas funcionalidades adicionais podem incluir verificações de autenticação, validações de dados de entrada, tratamento de erros e controle de acesso. Os *middlewares* são aplicados a nível global, afetando todas as requisições, ou de maneira específica em rotas selecionadas, proporcionando uma abordagem modular

para estender as capacidades da API.

No contexto desta aplicação, os *middlewares* são utilizados para garantir a segurança, autenticando usuários por meio de tokens, validando os dados recebidos e registrando informações relevantes para monitoramento e análise. Além disso, permitem a customização do fluxo de execução com base em requisitos específicos da aplicação.

Em suma, os *middlewares* enriquecem o pipeline de execução da API, introduzindo camadas adicionais de funcionalidades que melhoram a segurança, confiabilidade e flexibilidade do sistema. Essa abordagem modular facilita a manutenção do código, uma vez que as lógicas específicas podem ser adicionadas ou removidas sem impactar diretamente a lógica dos *controllers*.

Para exemplificar, nesse trabalho será demonstrado o método por realizar a verificação se o usuário que fez a requisição na API está de fato autenticado para tal ação. O método de validação de autenticação pode ser visualizado na Figura 19.

Figura 19 – Middleware de verificação de autenticação.

```
async function verifyJWT (req, res, next) {
  try {
    const authorization = req.headers.authorization;

    if (!authorization) {
      throw new ApiException(401, 'Token não informado');
    }

    const encodedToken = authorization.split(' ');

    if (encodedToken[0] !== 'Bearer') {
      throw new ApiException(401, 'Token invalido');
    }

    const decodedToken = jwt.verify(encodedToken[1], process.env.TOKEN_PRIVATE_KEY, { algorithms: ['HS256'] });

    const user = await User.findOne({ where: { id: decodedToken.userId }});

    if (!user) {
      throw new ApiException(401, 'Token invalido');
    }

    req.body.userId = decodedToken.userId;
    req.body.userType = decodedToken.type;
    next();
  } catch (err) {
    handleError(res, err);
  }
}
```

Fonte: elaborado pelo autor (2023).

O método captura as informações enviadas nos cabeçalhos de autorização da requisição. Caso um token não tenha sido informado nos cabeçalhos, é gerado um erro informando a situação. O token então tem sua estrutura validada e é decriptado com base nas chaves de assinatura que foram utilizadas no momento da criação do token no login do usuário.

Caso o token seja decriptado com sucesso, são obtidos os dados do usuário que estavam armazenados no *payload* do token e então as informações são armazenadas no corpo da requisição. O método “next()” utilizado indica que a próxima função da “fila” do *router* pode ser executada.

Observe que nesse exemplo, são utilizados os dois mesmos parâmetros iniciais dos métodos dos *controllers*, sendo possível utilizar esse terceiro parâmetro também naqueles métodos, de acordo com a necessidade.

Todas as camadas apresentadas para o desenvolvimento da API se complementam e são implementadas da forma que foi apresentado visando uma boa modularidade e escalabilidade do software. Essas camadas são “unificadas” pela configuração do arquivo principal do projeto, que realiza a importação dos métodos desenvolvidos e os disponibiliza para acesso externo à API. Na próxima etapa do projeto será apresentado como é realizada a configuração do *frontend* da aplicação.

### 9.3. DESENVOLVIMENTO DO FRONTEND

O *frontend*, ou interface do usuário, constitui a camada visual e interativa de uma aplicação, proporcionando a experiência direta dos usuários finais. Essa parte da aplicação engloba a interface gráfica (UI), que inclui elementos visuais como botões, campos de entrada e gráficos, proporcionando a interação intuitiva com os recursos do software. Além disso, a experiência do usuário (UX) é um componente essencial do *frontend*, visando garantir que a interação seja não apenas funcional, mas também agradável e eficiente.

A interatividade dinâmica é uma característica distintiva do *frontend*, permitindo a atualização de elementos em tempo real sem a necessidade de

recarregar a página. Isso é viabilizado por tecnologias como Javascript, que possibilita a manipulação dinâmica da interface. Apesar do foco na interação direta com os usuários, o *frontend* também depende da comunicação eficiente com o *backend*, responsável por processar dados e fornecer informações à interface.

No caso desse projeto, optou-se por utilizar o Vue.JS aliado ao framework de componentes visuais Quasar. O Quasar oferece uma lista extensa de componentes que deixam o desenvolvimento mais ágil e prático. Esse framework pode ser adicionado à um projeto Vue já existente, mas também pode ser utilizado para a criação de um novo projeto, utilizando o Quasar CLI, que é o caso aplicado nesse projeto.

### 9.3.1 Configuração inicial do projeto com Quasar

Nessa parte do trabalho será exemplificado como é gerado um projeto Vue do zero, utilizando a ferramenta de criação de projetos do Quasar. Inicialmente, é necessário criar uma pasta para receber o projeto e rodar o comando demonstrado na Figura 20 para dar início às configurações.

Figura 20 – Configuração inicial do projeto do frontend.

```
C:\Users\bi_bi\Projetos>npm init quasar
Need to install the following packages:
  create-quasar
Ok to proceed? (y)

.d88888b.
d88P" "Y88b
888      888
888      888 888 888 888 8888b. .d8888b 8888b. 888d888
888      888 888 888      "88b 88K      "88b 888P"
888 Y8b 888 888 888 .d888888 "Y8888b. .d888888 888
Y88b.Y8b88P Y88b 888 888 888      X88 888 888 888
"Y888888" "Y88888 "Y888888 88888P' "Y888888 888
      Y8b

? What would you like to build? » - Use arrow-keys. Return to submit.
> App with Quasar CLI, let's go! - spa/pwa/ssr/bex/electron/capacitor/cordova
  AppExtension (AE) for Quasar CLI
  Quasar UI kit
```

Fonte: elaborado pelo autor (2023).

O comando acima inicia a configuração do projeto, e na sequência são feitas algumas perguntas para definirmos detalhes da aplicação. Nesse caso, optamos por realizar a criação do projeto com o Quasar CLI e respondemos às perguntas de configuração do projeto conforme a Figura 21.

Figura 21 - Opções de configuração do projeto.

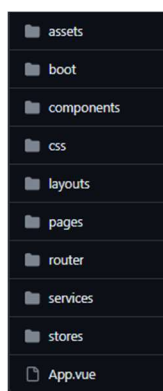
```
✓ What would you like to build? » App with Quasar CLI, let's go!  
✓ Project folder: ... carga-certa  
✓ Pick Quasar version: » Quasar v2 (Vue 3 | latest and greatest)  
✓ Pick script type: » Javascript  
✓ Pick Quasar App CLI variant: » Quasar App CLI with Vite  
✓ Package name: ... carga-certa  
✓ Project product name: (must start with letter if building mobile apps) ... Carga Certa  
✓ Project description: ... Projeto Vue JS  
✓ Author: ... kunzgabriel <gabrielkunz2012@gmail.com>  
✓ Pick your CSS preprocessor: » Sass with SCSS syntax  
✓ Check the features needed for your project: » ESLint, State Management (Pinia), Axios  
✓ Pick an ESLint preset: » Prettier
```

Fonte: elaborado pelo autor (2023).

Definimos o diretório onde o projeto será iniciado, qual a versão do Vue.JS e do Quasar que serão utilizadas, a linguagem Javascript (há também suporte para Typescript) e alguns detalhes técnicos como tecnologia utilizada para estilização, indentação do código e plugins como Axios e Pinia, para lidar com as requisições HTTP e o gerenciamento de estado da aplicação, respectivamente.

Essa instalação vai gerar uma estrutura de pastas dentro de um padrão adotado na comunidade de desenvolvimento de aplicações que utilizam Vue.JS e algumas particularidades do Quasar. A estrutura pode ser visualizada na Figura 22.

Figura 22 – Estrutura das pastas do projeto Vue.JS



Fonte: elaborado pelo autor (2023).

A pasta “assets” contém arquivos estáticos, como imagens, fontes ou qualquer recurso que seja referenciado diretamente nos componentes Vue. Esses recursos são geralmente importados nos arquivos de componentes para uso na interface do usuário.

A pasta “boot” é responsável por inicializações do aplicativo. Os arquivos nesta pasta são executados automaticamente durante o processo de inicialização do aplicativo. Isso é útil para configurar bibliotecas externas, configurar interceptadores de requisições ou qualquer lógica que precise ser executada globalmente.

A pasta “components” armazena os componentes Vue reutilizáveis da aplicação. Cada componente encapsula uma parte específica da lógica e da interface do usuário, facilitando a manutenção e a reutilização do código.

A pasta “css” contém arquivos de estilo global que afetam toda a aplicação. Esses estilos podem incluir estilos padrão para elementos HTML, ajustes no layout global ou qualquer outra personalização que afete múltiplos componentes.

A pasta “layouts” agrupa layouts da aplicação, que definem a estrutura geral da página. Cada layout pode ter diferentes estruturas de grade, barras de navegação e outros elementos que são consistentes em determinadas partes da aplicação.

A pasta “pages” contém os componentes que representam páginas específicas da aplicação. Cada arquivo neste diretório geralmente corresponde a uma rota específica e define a estrutura e o comportamento da página associada.

A pasta “router” gerencia a configuração de roteamento da aplicação. Aqui, são definidas as rotas da aplicação, associando cada rota a um componente específico. O roteador permite a navegação entre as diferentes páginas da aplicação.

A pasta “services” armazena serviços que encapsulam lógica de negócios, comunicação com o *backend* ou qualquer outra funcionalidade que não pertença diretamente a um componente. Esses serviços podem ser injetados em componentes para facilitar a reutilização e a modularidade.

A pasta “stores” contém os arquivos de gerenciamento de estado do Pinia. As stores Pinia facilitam o compartilhamento de dados entre componentes e a manutenção do estado da aplicação de forma centralizada.

Nesse trabalho abordaremos com mais detalhes as pastas “boot”, “layouts”, “pages”, “components”, “router”, “services” e “store”, exemplificando o processo de

codificação adotado em cada uma delas.

### 9.3.2 O diretório boot

No contexto dessa aplicação, o diretório “boot” contém o arquivo de configuração do Axios. O Axios é uma biblioteca que abstrai o processo de requisições HTTP. No caso desse projeto, a configuração do Axios irá ter apontamentos para a API desenvolvida anteriormente, com algumas configurações adicionais para tratamento da requisição e da resposta obtida da API. A configuração do arquivo de inicialização do Axios pode ser visualizada na Figura 23.

Figura 23 – Implementação da configuração do Axios.

```
import { boot } from 'quasar/wrappers';
import { Notify } from 'quasar';
import axios from 'axios';

const api = axios.create({
  baseURL: process.env.API_URL
});

api.interceptors.request.use((config) => {
  config.headers.Authorization = `Bearer ${sessionStorage.getItem('carga-certa-token')}`;
  return config;
});

function showMessage (response) {
  // ...
}

api.interceptors.response.use(
  response => {
    showMessage(response);
    return response;
  },
  error => {
    showMessage(error.response);
    return error.response;
  }
);

export default boot(({ app }) => {
  app.config.globalProperties.$axios = axios;
  app.config.globalProperties.$api = api;
});

export { api };
```

Fonte: elaborado pelo autor (2023).

Define-se uma URL padrão para as requisições HTTP por meio da variável de ambiente “API\_URL” e é adicionado um interceptador de requisições que garante que em todas as requisições realizadas a configuração do cabeçalho de autorização contenha o token de acesso do usuário logado na aplicação.

Há ainda a configuração de interceptadores da resposta da requisição, para que sejam exibidas em tela as mensagens correspondentes ao retorno da API.

### 9.3.3 O diretório layouts

Nesse diretório são criados os componentes Vue que serão exibidos em várias páginas de forma compartilhada, como os menus, rodapés e barras de navegação. É importante que os layouts contêmam a *tag* “<router-view/>” no seu *template* de criação, para que as páginas “filhas” desse layout sejam exibidas corretamente em tela. Na Figura 24 pode ser visualizado um exemplo de implementação de menu lateral com opções de acesso e o *container* do layout que exibirá os componentes de página.

Figura 24 – Implementação de Layout.

```
<q-drawer
  v-model="leftDrawerOpen"
  bordered
  overlay
  behavior="mobile"
>
  <q-list>
    <q-item-label
      header
    >
      Menu
    </q-item-label>
    <Essentiallink
      v-for="link in menuOptions"
      :key="link.title"
      v-bind="link"
    />
  </q-list>
</q-drawer>

<q-page-container>
  <router-view />
</q-page-container>
```

Fonte: elaborado pelo autor (2023).

### 9.3.4 O diretório components

Nessa pasta serão armazenados todos os componentes da aplicação. É importante observar que a abordagem com uso de componentes reutilizáveis não é obrigatória no desenvolvimento de uma aplicação com Vue. Mas nesse caso, optou-se pela criação de alguns componentes que precisavam ser utilizados em mais de uma página. Um exemplo disso, é o componente de cada página que exibe um título, um botão para retornar à página anterior e a possibilidade da exibição de um botão para novos cadastros (quando for o caso de a página conter cadastros).

Nas Figuras 25 e 26, há o exemplo de implementação desse componente, chamado “CrudPageHeader”, com o *template* visual e o código Javascript, respectivamente.

Figura 25 – Implementação de componente de cabeçalho de páginas.

```
<template>
  <q-toolbar class="bg-grey-4">
    <q-btn
      flat
      dense
      round
      fab
      icon="mdi-arrow-left-circle"
      aria-label="Voltar"
      @click="$router.go(-1)"
    />

    <q-toolbar-title>
      {{ title }}
    </q-toolbar-title>

    <q-space></q-space>

    <q-btn
      v-if="!hideAddBtn"
      color="primary"
      no-caps
      icon="mdi-plus"
      label="Adicionar"
      @click="$emit('onClickAdd')"
    ></q-btn>
  </q-toolbar>
</template>
```

Fonte: elaborado pelo autor (2023).

Figura 26 – Código Javascript do componente de cabeçalho de páginas.

```
<script>
export default {
  name: 'CrudPageHeader',

  emits: ['onClickAdd'],

  props: {
    title: {
      type: String,
      default: ''
    },
    hideAddBtn: {
      type: Boolean,
      default: false
    }
  }
};
</script>
```

Fonte: elaborado pelo autor (2023).

No exemplo desse componente, ele possui duas propriedades para ser utilizado: o título que será exibido e uma condição que indica se o botão de “adicionar” deve ficar oculto em tela. Quando o botão é clicado, o Vue gera um evento que pode ser “ouvido” pela página que utilizou o componente e dessa forma realizar os tratamentos que forem necessários. Nesse caso, as páginas que utilizam esse componente exibem uma caixa de diálogo para cadastro das informações da página correspondente.

### 9.3.5 O diretório pages

As “pages” são basicamente componentes Vue que utilizam componentes previamente criados e os juntam em algum contexto específico. Por exemplo, uma

página de listagem de usuários pode conter o componente de cabeçalho exemplificado anteriormente, uma tabela para listagem dos usuários, botões para exclusão e edição, campos de pesquisa e filtro, entre outros.

### 9.3.6 O diretório router

Essa é a pasta que armazena os arquivos de configuração do roteamento da aplicação, isto é, que define quais URLs vão exibir as páginas do projeto. Para as rotas podem ser definidos parâmetros, o componente que será exibido e pode também ser feito o gerenciamento de acesso de acordo com a autorização que o usuário em questão está contemplado.

Um exemplo de implementação das rotas do sistema pode ser visualizado na Figura 27.

Figura 27 – Configuração das rotas do projeto de frontend.

```
const routes = [
  {
    path: '/login',
    component: () => import('pages/auth/LoginPage.vue')
  },
  {
    path: '/register',
    component: () => import('pages/auth/RegisterPage.vue')
  },
  {
    path: '/',
    component: () => import('src/layouts/MainLayout.vue'),
    children: [
      { path: '', component: () => import('pages/IndexPage.vue') },
      { path: 'account', component: () => import('pages/user/UserPage.vue') },
      { path: 'deliveries', component: () => import('pages/company/DeliveriesPage.vue') }
    ],
    beforeEnter: verifyToken
  },
]
```

Fonte: elaborado pelo autor (2023).

No exemplo acima, observa-se que as rotas de login e cadastro de usuários não estão vinculadas a nenhuma rota “pai”, enquanto as rotas de conta do usuário e entregas estão dentro da estrutura de componentes “filhos” do layout de menus do sistema. Ainda nesse último caso, antes de acessar a página de fato é realizada uma verificação da autenticação, para que seja validado se o usuário em questão tem permissão de acesso, funcionando como se fosse um *middleware* dos que foram aplicados na API.

### 9.3.7 O diretório services

Nesse diretório são criados os arquivos responsáveis por armazenar os arquivos que contém os métodos que fazem as requisições para a API da aplicação. É importante que cada arquivo contenha os métodos do contexto específico e no caso desse projeto, optou-se por manter um arquivo para cada estrutura de rotas da API. Por exemplo, o *service* de usuário contém todas as requisições para as rotas que lidam com os usuários na API, conforme exemplo da Figura 28.

Figura 28 – Implementação de service de usuários no frontend.

```
import { api } from 'src/boot/axios';

async function login (params) {
  return await api.post('/users/login', params)
    .then(res => res.data)
    .catch(err => err);
}

async function register (params) {
  return await api.post('/users/register', params)
    .then(res => res.data)
    .catch(err => err);
}

async function forgotPassword (params) {
  return await api.post('/users/forgot-password', params)
    .then(res => res.data)
    .catch(err => err);
}

async function resetPassword (params) {
  return await api.post('/users/reset-password', params)
    .then(res => res.data)
    .catch(err => err);
}
```

Fonte: elaborado pelo autor (2023).

### 9.3.8 O diretório stores

As *stores* armazenam os dados de estado da aplicação Vue. Geralmente armazenam dados de requisições em *cache* evitando a necessidade de repetidas requisições, principalmente no caso de dados que pouco são alterados durante o uso da aplicação, como é o caso dos dados do usuário logado. Além disso, as *stores* possuem os métodos que chamam as funções implementadas nos *services*. Essa abordagem é utilizada para que seja possível a aplicação de alguma regra de negócio de *frontend* necessária antes ou depois de uma requisição realizada, como pode ser visualizado na Figura 29.

Figura 29 – Implementação de store de usuário.

```
import { defineStore } from 'pinia';
import userService from 'src/services/userService';

export const useUserStore = defineStore('user', {
  state: () => ({
    user: {}
  }),
  actions: {
    async login (params) {
      const response = await userService.login(params).then(res => res.data).catch(err => err);
      if (response) {
        this.user = response.user;
        return response;
      }
    },

    async register (params) {
      await userService.register(params);
    },

    async forgotPassword (params) {
      await userService.forgotPassword(params);
    },

    async resetPassword (params) {
      await userService.resetPassword(params);
    },

    async getUserInfo () {
      return await userService.getUserInfo().then(res => res.data);
    }
  }
});
```

Fonte: elaborado pelo autor (2023).

### 9.3.9 Considerações sobre o frontend

A implementação correta e organizada na estrutura descrita nos pontos anteriores tende a garantir uma aplicação de fácil manutenção, escalável, ágil e de uso intuitivo para os usuários.

Páginas, componentes, *services*, rotas e *stores* devem ser criadas de acordo com a necessidade da aplicação e é importante que não exerçam mais do que sua responsabilidade, o que poderia gerar um sistema de difícil compreensão e limitado a implementações futuras.

## 10. PROTÓTIPO

Nessa etapa do trabalho é exibido o resultado obtido com base em todos os estudos de tecnologias e na descrição do processo de desenvolvimento descrito neste trabalho, demonstrando o fluxo de usabilidade do protótipo.

O fluxo inicia-se quando algum usuário do tipo “empresa” faz o cadastro de alguma entrega pela tela de “Cadastro de carga”, conforme exibido na Figura 30.

Figura 30 - Cadastro de carga.

A interface de 'Cadastro de carga' apresenta os seguintes campos e informações:

- De: SC
- Para: SC
- Destino: Chapecó
- Destino: Pinhalzinho
- Data de carregamento: 05/12/2023
- Data de entrega: 08/12/2023
- Km total: 55.00
- R\$/Km: 32.00
- R\$ adicional: 230.00
- Observações: Carga frágil
- Botões: Cancelar, Salvar

Fonte: elaborado pelo autor (2023).

O usuário logado poderá visualizar todas as cargas que foram cadastradas por ele na tela de “Entregas”, conforme exibido na Figura 31.

Figura 31 – Listagem de entregas cadastradas pela empresa logada.

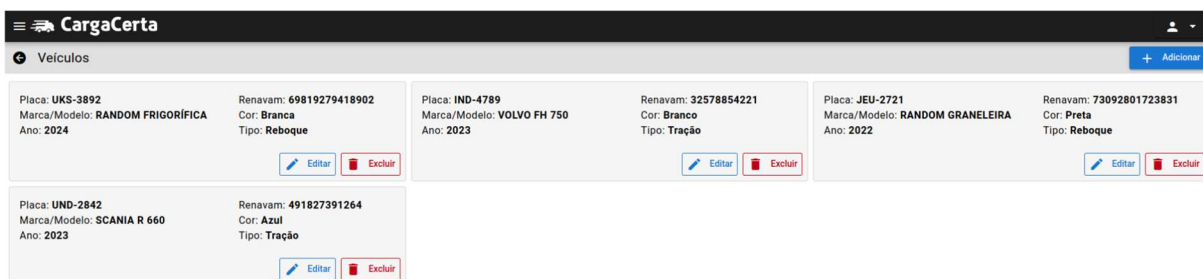
A interface de 'Entregas' mostra uma lista de entregas com o seguinte conteúdo:

Status	Origem	Destino	Carregamento	Entrega	Valor	Ações
Em Andamento	Pinhalzinho - SC	Chapecó - SC	01/12/2023	02/12/2023	R\$ 779.40	Status, Aplicações
Disponível	Pinhalzinho - SC	Carambel - PR	02/12/2023	05/12/2023	R\$ 6537.50	Aplicações
Disponível	Cordelópolis - SP	São José - SC	04/12/2023	07/12/2023	R\$ 11295.00	Aplicações
Em Andamento	Agrolândia - SC	Ferreira Gomes - AP	01/12/2023	02/12/2023	R\$ 94400.00	Status

Fonte: elaborado pelo autor (2023).

Nessa listagem, a empresa pode acompanhar o andamento de cada uma das entregas, visualizar os motoristas interessados para a realização do frete e realizar a alteração de *status* da entrega. Com o frete cadastrado, usuários do tipo “motorista” podem manifestar interesse em realizar esse frete, mas antes é necessário que realizem o cadastro de seus veículos na plataforma. Os veículos cadastrados por cada motorista são listados na tela “Veículo”, conforme exibido na Figura 32.

Figura 32 – Listagem de veículos cadastrados.



Fonte: elaborado pelo autor (2023).

Novos veículos podem ser cadastrados pela tela “Cadastro de veículo”, conforme exibido na Figura 33.

Figura 33 – Tela de cadastro de veículos.

A tela de cadastro de veículo, intitulada 'Cadastro de veículo', possui uma aba 'DADOS GERAIS' selecionada. Os campos de entrada são:

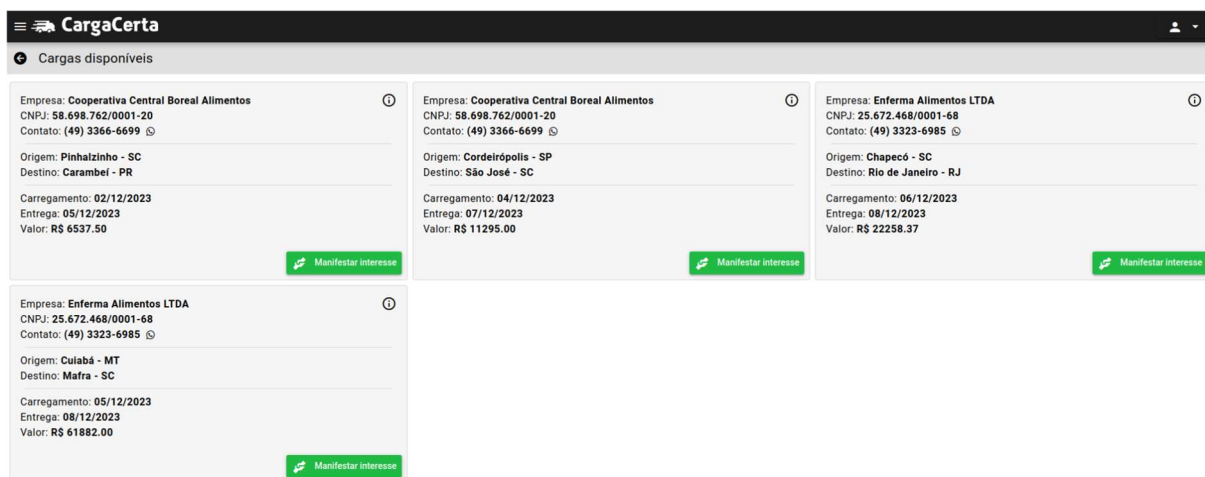
- Tipo: Reboque (menu suspenso)
- Placa: UKS-3892
- Renavam: 69819279418902
- Marca: RANDOM
- Modelo: FRIGORÍFICA
- Ano: 2024
- Cor: Branca

Na base da tela, há dois botões de ação: 'Cancelar' (em vermelho) e 'Salvar' (em azul).

Fonte: elaborado pelo autor (2023).

Com os veículos devidamente cadastrados, os usuários do tipo “motorista” podem acessar a tela de “Cargas disponíveis”, escolher a carga de seu interesse e clicar na opção “Manifestar interesse”. A listagem das cargas é exibida na Figura 34.

Figura 34 – Listagem de cargas disponíveis.



Fonte: elaborado pelo autor (2023).

Ao clicar na opção “Manifestar interesse”, será exibida uma caixa de diálogo na qual o usuário deve informar o veículo que usará na realização do frete, em caso de aprovação da empresa. A caixa de diálogo é exibida na Figura 35.

Figura 35 – Confirmação de manifestação de interesse.



Fonte: elaborado pelo autor (2023).

Vários motoristas podem manifestar interesse para a mesma entrega, ficando a cargo da empresa realizar contato com esses motoristas e definir quem será o responsável pela prestação do serviço de transporte. Para visualizar todos os interessados os usuários do tipo “empresa” acessam a tela de “Aplicações para entrega” a partir da listagem de entregas exibida anteriormente. Todas as manifestações de interesse para a entrega selecionada poderão ser visualizadas pela empresa, que terá a opção de aprovar, negar ou cancelar essa manifestação, conforme exibido na Figura 36.

Figura 36 – Manifestações de interesse para a entrega.



Fonte: elaborado pelo autor (2023).

Caso a empresa aprove a manifestação de interesse, o motorista poderá visualizar a carga na tela “Entregas”, com acesso às informações pertinentes a prestação do serviço de transporte, conforme exibido na Figura 37.

Figura 37 – Listagem das cargas do motorista.

Status	Origem	Destino	Carregamento	Entrega	Valor
Em Andamento	Pinhalzinho - SC	Chapecó - SC	01/12/2023	02/12/2023	R\$ 779.40
Retirada	Pinhalzinho - SC	Carambeí - PR	02/12/2023	05/12/2023	R\$ 6537.50

Fonte: elaborado pelo autor (2023).

Após esse momento, a empresa apenas realizará o controle do *status* das entregas para fins de acompanhamento da realização do serviço de transporte, podendo realizar contatos com o motorista pelo telefone cadastrado na plataforma.

## 11. CONSIDERAÇÕES FINAIS

No decorrer do desenvolvimento deste projeto, diversas etapas e tecnologias foram exploradas, proporcionando uma visão abrangente do processo de construção de uma aplicação *web*. Inicialmente, a escolha das tecnologias fundamentais, como Vue.js para o *frontend*, Node.js para o *backend*, e PostgreSQL como sistema de gerenciamento de banco de dados, trouxe desafios significativos, mas também oportunidades de aprendizado substanciais.

O emprego do Node.js no *backend* possibilitou a criação de servidores eficientes e escaláveis, enquanto o uso do Express.js agilizou o desenvolvimento da API, oferecendo um conjunto robusto de ferramentas. No entanto, a adaptação ao modelo de programação assíncrona do Node.js demandou uma compreensão aprofundada para evitar problemas de concorrência e assegurar a execução suave das operações.

A integração do *Sequelize* como ORM para a interação com o PostgreSQL simplificou a manipulação do banco de dados, proporcionando uma abstração poderosa para a criação de modelos e consultas. A definição e configuração de tabelas no PostgreSQL revelaram-se críticas, destacando a importância de uma estrutura de banco de dados bem planejada para suportar as necessidades do sistema.

No *frontend*, a combinação do Vue.js com o framework Quasar ofereceu uma experiência de desenvolvimento ágil, possibilitando a criação de interfaces modernas e responsivas. A definição de *models* utilizando o *Sequelize* e a implementação de serviços no *backend* contribuíram para uma arquitetura consistente e modular, permitindo a manutenção eficiente do código.

Ao longo do desenvolvimento, enfrentamos desafios relacionados à integração entre as camadas *frontend* e *backend*, bem como à configuração inicial do projeto Quasar CLI. Esses obstáculos proporcionaram aprendizados valiosos sobre o processo de depuração, configuração de ambientes de desenvolvimento e práticas recomendadas para a comunicação eficiente entre as diferentes partes da aplicação.

Concluindo, este projeto representou uma jornada enriquecedora,

proporcionando uma compreensão mais profunda das tecnologias escolhidas, dos desafios enfrentados durante o desenvolvimento e das soluções implementadas. As dificuldades encontradas serviram como oportunidades para aprimorar habilidades técnicas e estratégicas, contribuindo para o crescimento profissional e a consolidação do conhecimento adquirido ao longo deste processo de desenvolvimento.

## REFERÊNCIAS BIBLIOGRÁFICAS

ANTT. **Pagamento Eletrônico de Frete**. 2022. Disponível em: <https://portal.antt.gov.br/pagamento-eletronico-de-frete>. Acesso em: 15 jun. 2022.

BARRETO, Elis. **Setor de fretes e cargas prevê aumento de 3% nos contratos com reajuste do diesel**. CNN Business Brasil, 2022. Disponível em: <https://www.cnnbrasil.com.br/business/setor-de-fretes-e-cargas-preve-aumento-de-3-nos-contratos-com-reajuste-do-diesel/>. Acesso em: 04 jun. 2022.

EXPRESS. **Guia do Express.js**. Disponível em: <https://expressjs.com/en/guide>. Acesso em: 10 nov. 2023.

GRABARA, Janusz. **THE ROLE OF INFORMATION SYSTEMS IN TRANSPORT LOGISTICS**. Vol. 2. Czestochowa: International Journal of Education and Research . 2 fev. 2014.

IZIDORO, Cleyton. **Gestão de Tecnologia e Informação em Logística**. 1. ed. São Paulo: Pearson Education do Brasil, 2016.

MOZILLA. **Documentação do JavaScript**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 09 nov. 2023.

NODEJS. **Documentação do Node.js**. Disponível em: <https://nodejs.org/en/docs/>. Acesso em: 10 nov. 2023.

PEREIRA, Vicente de Britto. **Transportes: História, Crises e Caminhos**. 1. ed. Rio de Janeiro: Civilização Brasileira, 2015.

POSTGRESQL. **PostgreSQL**. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 10 nov. 2023.

QUASAR. **Introdução ao Quasar Framework**. Disponível em: <https://quasar.dev/introduction-to-quasar>. Acesso em: 10 nov. 2023.

SCHLÜTER, Mauro Roberto. **Sistemas logísticos de transporte**. 1. ed. Curitiba: InterSaberes, 2013.

SEQUELIZE. **Documentação do Sequelize ORM (v6)**. Disponível em: <https://sequelize.org/docs/v6/>. Acesso em: 10 nov. 2023.

VUEJS. **Guia do Vue.js**. Disponível em: <https://vuejs.org/guide>. Acesso em 10 nov. 2023.