GUILHERME DA SILVA TEIXEIRA

DESENVOLVIMENTO E MANUTENÇÃO DE RESTFUL WEB SERVICES COM O FRAMEWORK DROPWIZARD

LAVRAS – MG

2020

GUILHERME DA SILVA TEIXEIRA

DESENVOLVIMENTO E MANUTENÇÃO DE RESTFUL WEB SERVICES COM O FRAMEWORK DROPWIZARD

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação, para obtenção do título de Bacharel.

Prof. DSc. Marluce Rodrigues Pereira
Orientadora

LAVRAS – MG

2020

GUILHERME DA SILVA TEIXEIRA

DESENVOLVIMENTO E MANUTENÇÃO DE RESTFUL WEB SERVICES COM O FRAMEWORK DROPWIZARD

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 02 de Setembro de 2020.

Dra. Marluce Rodrigues Pereira	UFLA
Dr. Neumar Costa Malheiros	UFLA
Daniel Henrique Pinheiro	RODOSAFRA

Prof. Dra. Marluce Rodrigues Pereira

Orientadora

LAVRAS – MG
2020

Dedico aos meus pais que sempre me apoiaram, impulsionaram e nunca mediram esforços para garantir o meu sucesso nessa jornada, à minha irmã mais velha que foi a primeira pessoa a me falar sobre Ciência da Computação e aos meus pequenos irmãos, Bernardo e Giovanna, que recém chagaram neste mundo cheios de amor e alegria.

AGRADECIMENTOS

Agradeço à Deus pela minha vida.

Agradeço à Universidade Federal de Lavras pela elevada qualidade de ensino. Agradeço aos professores do Departamento de Ciência da Computação pela excelência da qualidade técnica e por fazerem parte dessa jornada, em especial, à minha orientadora Prof. DSc. Marluce Rodrigues Pereira.

Agradeço à Equipe RodoSafra, cuja participação foi essencial na construção do meu conhecimento profissional, pelo acolhimento e por todo conhecimento transmitido.

Agradeço aos meus colegas de classe por toda e qualquer colaboração durante as atividades do curso.

Agradeço ao Rotaract Club de Lavras pelas amizades, pelo companheirismo e por todos os ensinamentos sobre liderança.

Por fim, agradeço a todas a pessoas que fizeram parte desta etapa da minha vida.

RESUMO

Este trabalho apresenta o relatório de estágio na empresa RodoSafra Transportes Rodoviários. O objetivo do estágio foi a evolução e manutenção do sistema *web* utilizado para o gerenciamento das atividades da empresa. Para a realização das atividades foram necessários conhecimentos sobre programação orientada a objetos com Java, conceitos de *web services* e *RESTful web services*, modelagem e gerenciamento de banco de dados, mapeamento objeto-relacional e outros temas relacionados. O relatório descreve a metodologia de desenvolvimento, as atividades e as ferramentas utilizadas. As atividades desenvolvidas permitiram que o estagiário conhecesse novas tecnologias, aplicasse os conhecimentos adquiridos no curso de graduação e trabalhasse em equipe para gerar produto dentro de uma organização.

Palavras-chave: Web services. RESTful. Dropwizard. Transporte Rodoviário.

ABSTRACT

The present document presents an internship report carried out at the company *RodoSafra Transportes Rodoviários*. The internship goal is the company's web system evolution and maintenance. The system is used for operational activities management. To carry out the activities, knowledge about object-oriented programming with Java, concepts of web services and RESTful web services, database modeling and management, object-relational mapping and other related topics were required. The report describes the development methodology, activities and used tools. The activities developed allowed the trainee to know new technologies, apply the knowledge acquired in the undergraduate course and work as a team to generate products within an organization.

Keywords: *Web services*. *RESTful*. *Dropwizard*. Road transport.

.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

No Brasil, quando se fala sobre transporte, o principal setor é o rodoviário que é responsável por 61,1% do transporte de cargas no país (CNT, 2019b). O setor conta com uma malha rodoviária de 1.720.700 quilômetros, sendo 280.355 em Minas Gerais, estado detentor da maior malha rodoviária do país (CNT, 2019). O transporte rodoviário é, também, o setor de transporte que mais recebe investimentos da União, 71,5% (CNT, 2019a). A RodoSafra é uma empresa que nasceu a partir dessa alta demanda do setor e, para facilitar o gerenciamento, utiliza sistemas computacionais. Por isso, possui um setor de TI, em crescimento, onde foi realizado o estágio descrito nesse documento.

A empresa possui um sistema *web*, o Sistema RodoSafra, que é um sistema de gestão para todas as atividades da empresa, como o gerenciamento de operações de transporte realizadas, o controle de demanda e disponibilidade das cargas de seus clientes, gestão financeira e administrativa.

No presente documento são detalhadas as principais atividades realizadas no estágio, que ocorreu de 16 de agosto de 2019 à 28 de fevereiro de 2020, expondo o processo de desenvolvimento, as dificuldades encontradas, decisões tomadas e possíveis melhorias. Durante o estágio foram criadas novas funcionalidades para o sistema e também funcionalidades pré-existentes foram melhoradas e mantidas.

1.1 Objetivos

O objetivo principal deste trabalho é apresentar as principais atividades realizadas durante o período de estágio na RodoSafra[1], as quais abarcam a evolução e manutenção do sistema web da empresa.

[1] Capítulo 2

1.2 Estrutura do Trabalho

O presente trabalho é composto por cinco capítulos incluindo esta introdução. O segundo capítulo contém a apresentação da empresa na qual o estágio foi realizado. No terceiro capítulo, são apresentadas as ferramentas, conceitos e metodologias utilizadas na realização das atividades do estágio. As principais atividades desenvolvidas são apresentadas no Capítulo 4. E por último, no Capítulo 5, são levantadas algumas considerações sobre o estágio e as contribuições do mesmo para a empresa.

2 A EMPRESA E O ESTÁGIO

A RodoSafra Transportes LTDA foi fundada no ano de 2014 com o objetivo de atender uma crescente demanda no setor do transporte rodoviário do sul do estado de Minas Gerais. Atualmente, com sede em 5 cidades brasileiras nos estados de Minas Gerais e São Paulo, a RodoSafra conta com 270 clientes fornecedores e 10000 veículos cadastrados. No ano de 2018 foram transportadas pela empresa 500000 toneladas de carga. Todos esses dados podem ser encontrados em (RODOSAFRA, 2019). O setor de TI da empresa encontra-se em crescimento, contando, atualmente com apenas um estagiário.

2.1 O negócio

A empresa possui dois tipos de clientes, o proprietário de uma carga ou fornecedor e o proprietário de um veículo. O objetivo é fazer a conexão entre os dois tipos de clientes de forma segura e dinâmica. O proprietário da carga precisa transportá-la. Então a RodoSafra, gera ordens de carregamento desta carga para proprietários de veículos cadastrados, contrata seguradoras, libera os embarques e confirma as descargas no local de destino. Todos os veículos e motoristas passam por uma fase de consulta onde busca-se referências, garantindo a segurança de todos os embarques realizados.

2.2 O sistema RodoSafra

Para melhor atender seus clientes a empresa possui um sistema *web* desenvolvido sob demanda que é utilizado internamente na empresa. O sistema possui funcionalidades como, cadastro de clientes, veículos e cargas, controle de cada embarque por todo o seu tempo de vida desde sua criação até seu fim, envio de *e-mails* para as partes interessadas informando cada estado do embarque, gerência das cargas cadastradas, geração de relatórios para a equipe, dentre muitas outras

funcionalidades que auxiliam na concretização dos objetivos da empresa. Esse sistema precisa ser mantido e adaptado às necessidades da equipe da RodoSafra ao longo do tempo.

Além do sistema *web* já existente, outra aposta tecnológica é o desenvolvimento do aplicativo mobile RodoSafra. No aplicativo, o próprio motorista cadastrado poderá consultar as cargas disponíveis próximas a ele e solicitar a abertura de uma ordem de carregamento, facilitando assim, a comunicação entre o motorista e a equipe de logística. Além disso, o aplicativo mostrará notificações ao motorista recomendando cargas próximas a ele, utilizando serviços de geolocalização.

3 FERRAMENTAS E METODOLOGIAS UTILIZADAS

Neste capítulo são apresentados conceitos e tecnologias utilizados durante as atividades do estágio.

3.1 Metodologia de desenvolvimento

Como já apresentado anteriormente, o setor de TI da empresa encontra-se em crescimento, o que influenciou em como as atividades foram realizadas durante o estágio. A equipe é formada por apenas dois programadores, sendo um contratado e um estagiário, autor desse trabalho. Para o planejamento do desenvolvimento foi utilizada a ferramenta Trello[1], que é apresentada na próxima seção. A princípio todas as demandas de desenvolvimento eram recebidas por mensagens e adicionadas no Trello para distribuição entre os programadores.

Com o decorrer do estágio foram encontradas dificuldades em relação ao cumprimento de metas e minimização de retrabalho, quando se viu a necessidade de uma reformulação em toda a metodologia de desenvolvimento. A equipe passou a realizar reuniões semanais onde eram pautados os seguintes assuntos: entrega de demandas cumpridas, apresentação de novas demandas e por fim planejamento das atividades da semana seguinte. Essa nova metodologia apresentada se aproxima das metodologias ágeis com entregas rápidas e frequentes (semanais).

Estão entre os objetivos da empresa em relação ao setor de TI o aumento da equipe e a evolução para uma metodologia ágil bem estruturada.

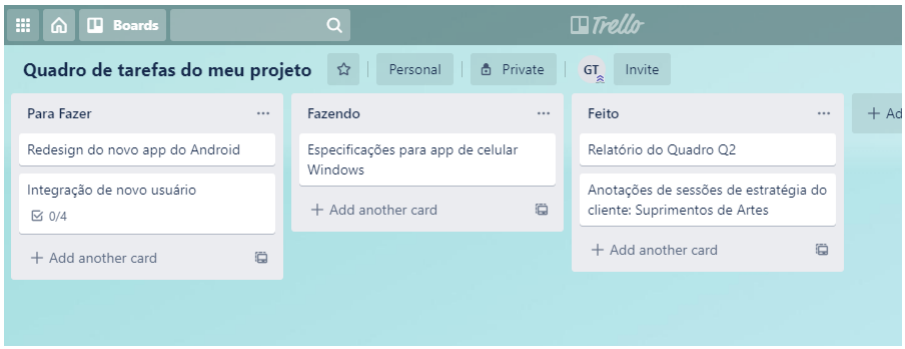
3.1.1 Trello

O Trello é uma ferramenta de produtividade simples e intuitiva utilizada para organizar tarefas e atividades. A ferramenta utiliza o formato de quadros e cartões, onde um quadro está relacionado a um projeto ou programa e cada cartão é

[1] <https://trello.com/pt-BR>

uma atividade sendo possível organizar as atividades em categorias. Um exemplo mais simples é apresentado na figura 3.1.

Figura 3.1 – Quadro de tarefas no Trello



Fonte: <*https://trello.com/pt-BR*>

Cada cartão pode possuir: título, descrição, *checklist*, observações, comentários, imagens e várias outras informações sobre determinada atividade, podendo conter um grande grau de detalhamento sobre a atividade auxiliando o indivíduo que irá realizá-la.

3.2 *Web Services*

Para definir o que são *Web Services* é preciso entender primeiro o conceito de serviço. De acordo com (SPROTT; WILKES, 2004), um serviço é um componente capaz de realizar uma tarefa específica. No entanto, o termo "serviço" não é recente e pode possuir inúmeras definições de acordo com o contexto aplicado. O trabalho de (MARZULLO, 2009), por exemplo, aponta que outra possível definição para um serviço é "um relacionamento entre um provedor e um consumidor, que possuem um objetivo de solucionar uma determinada atividade comum".

Com base nessas definições, neste trabalho, um serviço é um componente disponibilizado por um provedor com um padrão de consumo bem definido com a finalidade de ser consumido e realizar uma determinada atividade. A definição apresentada acima pode ser exemplificada com um serviço que está no nosso dia-a-dia, o fornecimento de energia elétrica que:

- É realizado por um provedor
- Para ser consumido, as instalações do consumidor precisam estar de acordo com os padrões de consumo
- Sua finalidade é alimentar os equipamentos eletro-eletrônicos do consumidor.

Com a definição de serviço apresentada, *Web Service* pode ser entendido como a disponibilização de um serviço e seus padrões de consumo na *Web*. Assim pode ser requisitado de múltiplas origens (clientes).

> Um *Web Service* representa a materialização da ideia de um serviço que é disponibilizado na Internet e que pode ser acessado em qualquer lugar do planeta. Representa uma lógica de negócio que permite que um ou mais clientes enviem requisições de um tipo bem definido de informação e recebam respostas síncronas ou assíncronas. (MARZULLO, 2009)

A disponibilização de serviços na Internet facilita a integração entre diferentes aplicações de uma empresa, mesmo aplicações desenvolvidas em plataformas diferentes. Para isso, basta que estas aplicações utilizem os padrões definidos pelo serviço. Suponha que uma empresa de vendas possui uma infinidade de aplicações que atendem de diversas formas suas necessidades. Todos esses softwares são capazes de realizar um cadastro de um novo produto na base de dados, e sempre, antes de inserir um produto, deve ser realizada uma validação dos dados. Cada um destes *softwares* é mantido por um prestador de serviços diferente e foi escrito em uma linguagem diferente. Essa situação gera alguns desafios. São eles:

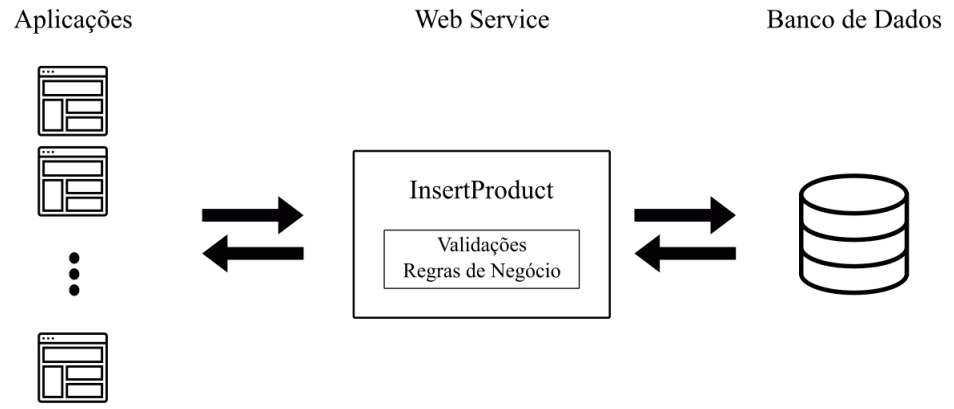
- Garantia de que as validações sejam feitas corretamente para todas as aplicações
- Alteração ou adição de novas etapas na validação mantendo a consistência para todas as aplicações
- Integração com possíveis novas aplicações.

Uma solução para estes desafios é a implementação de um *Web Service* denominado *'InsertProduct'*, o qual pode ser consumido por todas as aplicações. Neste caso, a validação, ou regra de negócio, está centralizada e é executada sempre que o serviço for consumido, assim garante-se a consistência da transação para todas as aplicações dessa empresa. E em caso de alteração nos passos da validação, apenas altera-se o serviço, facilitando a manutenção. A Figura 3.2 ilustra a solução.

Percebe-se que apenas o *Web Service* acessa o banco de dados, o que garante que toda inserção no banco de dados será validada de acordo com as regras de negócio.

Como apontado anteriormente, para que uma aplicação consuma um serviço é preciso que ela siga os padrões de consumo deste. Esses padrões precisam ser bem definidos e alguns deles são: uma representação universal para troca de

Figura 3.2 – *Web Service 'InsertProduct'*.



Fonte: Próprio autor

informações, um protocolo de envio e recebimento de mensagens e uma identificação para o serviço.

Uma característica forte dos modelos de *Web Services* é utilizarem tecnologias *web* padronizadas e abertas como *Extensible Markup Language* (XML), *Hypertext Transfer Protocol* (HTTP), *JavaScript Object Notation* (JSON) e *Uniform Resource Identifier* (URI). A vantagem de usar tecnologias como essas é que qualquer aplicação, independente da linguagem ou plataforma em que foi desenvolvida, será capaz de consumir os serviços fornecidos.

3.2.1 Protocolo HTTP

O HTTP é um protocolo de comunicação cliente/servidor criado para troca de dados na Internet. O cliente é quem inicia a comunicação enviando uma requisição HTTP para o servidor que o responde fornecendo recursos ao cliente como, arquivos de texto ou HTML (*Hypertext Markup Language*), imagens, dentre outros, ou executa funções requisitadas pelo cliente. O HTTP é a base da *Web*.

"O *Hypertext Transfer Protocol* (HTTP), o protocolo da camada de aplicação da *Web*, está no coração da *Web*" (KUROSE; ROSS, 2017).

A Figura 3.3 ilustra a estrutura de uma mensagem de requisição HTTP. Um exemplo de mensagem de requisição HTTP é como a seguir:

GET /hello/helloWorld.html HTTP/1.1
Host: www.world.example
Connection: close
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Accept-language: en-gb

O exemplo acima segue os padrões de mensagem de requisição HTTP, onde a primeira linha é chamada linha de requisição e é formada por três campos: método, URL (*Uniform Resource Locator*) e versão.

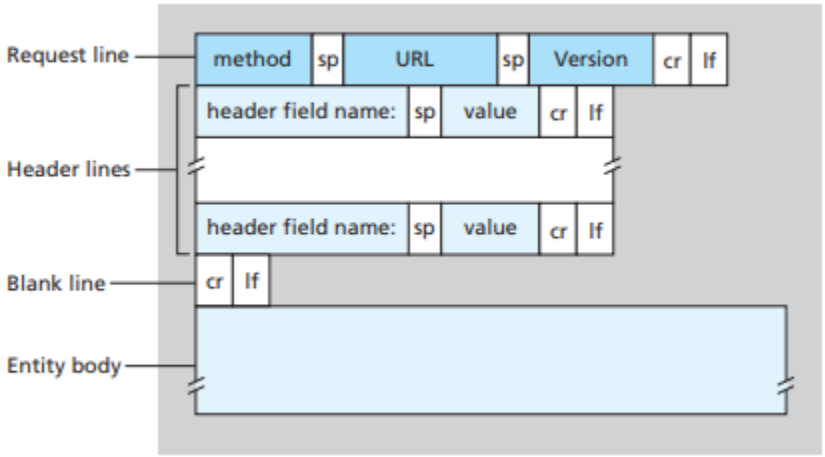
O campo método pode possuir diferentes valores, e eles são:

- *GET:* Solicitar um recurso.
- *HEAD:* Solicitar uma resposta igual ao método *GET*, no entanto sem o corpo da resposta.
- *POST:* Adicionar ou atualizar um recurso.
- *PUT:* Adicionar ou substituir um recurso.
- *DELETE:* Remover um recurso.
- *OPTIONS:* Descobrir as opções de requisição para um recurso.
- *TRACE:* Realizar teste de *loopback*. Envia requisição até o recurso, passando por todo o caminho. Muito usado para *debug*.
- *PATCH:* Modificar parcialmente um recurso.

O campo URL é utilizado para localizar o recurso, e versão é a versão do HTTP sendo utilizada. As demais linhas de mensagem de requisição são chamadas

de linhas de cabeçalho que servem para definir algumas variáveis como no exemplo são definidos *host, connection, user-agent, accept-language*. Esse exemplo possui 5 linhas mas uma requisição pode possuir mais, ou menos, linhas, depende do objetivo da requisição.

Figura 3.3 – Estrutura de uma mensagem de requisição HTTP



Fonte: (KUROSE; ROSS, 2017)

O formato da mensagem de resposta é um pouco diferente, como ilustrado na Figura 3.4. Um exemplo deste tipo de mensagem é como segue:

HTTP/1.1 200 OK

Connection: close

Date: Tue, 09 Aug 2011 15:44:04 GMT

Server: Apache/2.2.3 (CentOS)

Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT

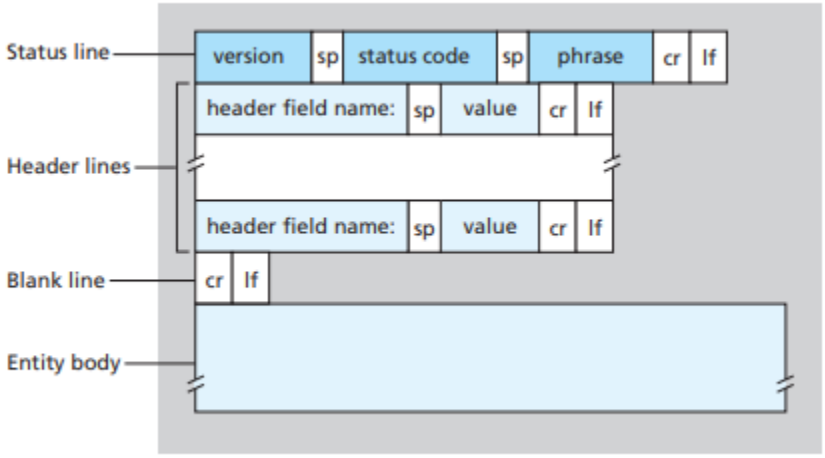
Content-Length: 6821

Content-Type: text/html

{"data"= "Hello World!"}

Neste formato, a primeira linha é chamada de linha de *status* e também possui três campos: versão do HTTP utilizada, código do *status* e uma frase correspondente ao *status*. Após a linha de *status*, temos as linhas de cabeçalho, uma linha em branco e o corpo da resposta. No exemplo, o corpo da resposta é um objeto JSON.

Figura 3.4 – Estrutura de uma mensagem de resposta HTTP



Fonte: (KUROSE; ROSS, 2017)

Existem diversos *status* HTTP que informam o tipo da resposta recebida, se a operação foi bem sucedida ou não, e em caso negativo, que tipo de erro ocorreu. Esses *status* são agrupados em cinco classes:

- Respostas de Informação - 100 a 199
- Respostas de Sucesso - 200 a 299
- Respostas de Redirecionamento - 300 a 399
- Respostas de Erros do Cliente - 400 a 499

- Respostas de Erros do Servidor - 500 a 599

3.3 *RESTful Web Services*

Representational State Transfer (*REST*), em português, Transferência de Estado Representacional, é um conjunto de princípios e critérios de arquitetura de software. No entanto, não é uma arquitetura, mas sim "um conjunto de critérios de desenvolvimento. Pode-se dizer que uma arquitetura satisfaz esses critérios melhor que outra, mas não existe uma 'arquitetura *REST*'" (RICHARDSON; RUBY, 2008).

Um *Web Service* que utiliza esses princípios é denominado *RESTful*. Então quando fala-se dos princípios e critérios *REST* neste trabalho, fala-se sobre o que um *Web Service* necessita para ser denominado um *RESTful Web Service*.

3.3.1 Princípios *REST*

Os princípios básicos do padrão *REST* são:

- Ausência de estado entre mensagens, ou seja, cada requisição HTTP contém todas as informações necessárias para entender uma tarefa.
- Endereçamento dos recursos. Cada recurso é identificado por uma URI.
- Interface uniforme, ou um conjunto de métodos bem definidos que se aplicam a todos os recursos.
- Utilização de *links* (hipermídia) para as transições de estado da aplicação.

Em suma, a dinâmica desses princípios é a seguinte: um *RESTful web-service* trata as requisições como uma chamada para um recurso. O servidor não armazena estados entre requisições HTTP, mas pode guiar o cliente para outros recursos por meio de hipermídia, o que é possível devido ao endereçamento dos recursos. Nota-se que a utilização de hipermídia se dá tanto para a informação

da aplicação quanto para as transições de estado da aplicação como exposto em (WIKIPÉDIA, 2020). Além disso, para cada recurso, há operações bem definidas e padronizadas, o que denomina-se interface uniforme.

Exemplo:

GET http://localhost:3333/animal/cachorro/1

Essa requisição está buscando o animal cachorro que possua *id* 1. O método HTTP *GET* é o que informa que se quer buscar um recurso no domínio especificado na URL. Então o método HTTP é utilizado para identificar a ação que será realizada com aquele recurso, ou seja, para aplicar o princípio de interface uniforme.

3.3.2 *REST Vs. RPC*

O estilo *RPC* ou *Remote Procedure Call* também utiliza o protocolo HTTP, com isso gera confusão e muitas vezes esse estilo é confundido com *REST*. Ao contrário do padrão *REST*, o estilo *RPC* não trata as requisições como uma chamada para um recurso mas sim como uma chamada para um procedimento ou função.

Exemplo de requisição no estilo *RPC*:

GET http://localhost:3333/buscaNomeDoCachorroPorId?id=1

No exemplo acima nota-se que a URL identifica uma função que busca o nome de um cachorro e não um recurso no domínio cachorro. Basicamente, o foco do *RPC* é criar serviços que realizam bem um trabalho específico, já no

REST o foco é a separação dos dados em domínios. Devido a essa diferença, o padrão *REST* torna-se uma melhor opção quando tem-se mais de uma aplicação com diferentes objetivos fazendo requisições de dados.

3.4 Banco de Dados Objeto-Relacional

Nesta seção é apresentada uma breve definição de banco de dados objeto-relacional para auxiliar no entendimento do próximo assunto, que é o controle de versão de banco de dados.

3.4.1 Introdução

Um banco de dados relacional é baseado no modelo relacional de dados, que é descrito a partir de três princípios:

- Aspecto estrutural: os dados no banco são apresentados como tabelas.
- Aspecto de integralidade: existem regras de integridade e consistência para as tabelas.
- Aspecto manipulador: as manipulações dos dados são realizadas por meio de operadores que derivam tabelas a partir de outras tabelas.

O banco de dados objeto-relacional é similar ao relacional mas possui também características do modelo de banco de dados orientado a objetos, como objetos, classes e herança. A linguagem *SQL* (*Structured Query Language*) é uma linguagem de pesquisa estruturada para banco de dados que seguem o modelo relacional. No estágio descrito por este trabalho foi utilizado o *Postgres SQL*.

> O *Postgres SQL* é um poderoso sistema de banco de dados objeto-relacional de código aberto que usa e estende a linguagem *SQL* combinado com muitas ferramentas que armazenam com segurança e escala as mais complexas cargas de trabalho de dados (POSTGRESQL, 2020).

3.5 Controle de versão

Controlar a versão de um arquivo ou artefato é guardar todas as alterações que foram realizadas no mesmo. Esse controle é fundamental no desenvolvimento de software porque ele implica em uma rastreabilidade das alterações, permitindo ao desenvolvedor revertê-las, em um arquivo ou projeto inteiro, identificar o seu autor, e tomar inúmeras outras ações. O controle de versão facilita o trabalho em equipe. Um exemplo de ferramenta para controlar a versão do código fonte e da documentação de um *software* é o Git[2].

O Git é uma ferramenta gratuita e de código aberto para controle distribuído de versões, e foi utilizada durante o estágio. O fato de ser distribuído o torna essencial para projetos com equipes numerosas.

3.5.1 Controle de versão de banco de dados

Assim como o código da aplicação, o banco de dados também pode ser versionado. Imagine que uma nova funcionalidade para a aplicação dependa de uma alteração no banco de dados. A alteração no código foi versionada mas a alteração no banco de dados foi feita manualmente sem nenhum versionamento. No futuro essa funcionalidade precisa ser revertida. O código pode ser revertido facilmente utilizando a ferramenta de versionamento mas o banco de dados teria que ser revertido manualmente.

Para solucionar esse problema deve-se utilizar o versionamento não só no código mas também no banco de dados. As *Migrations* são uma maneira de realizar esse controle. Segundo (ANDRADE, 2012), a técnica de *Migrations* é implementada a partir de 3 princípios:

- Um conjunto de representações das alterações feitas no banco de dados.
- Um registro informando qual a última alteração aplicada.

[2] https://git-scm.com/

- Operações para avanço e retrocesso de versão.

O conjunto de representações das alterações nada mais é que um conjunto de arquivos com as alterações realizadas no banco de dados, podendo ser em SQL ou abstraídas do sistema de banco de dados utilizado com uma representação mais simples como o XML. A ferramenta de *Migrations* armazenará em uma tabela no próprio banco de dados quais alterações foram ou não aplicadas.

Cada arquivo de alteração geralmente possui informações para aplicá-la e revertê-la, para, como dito no item 3, serem realizadas operações de avanço e retrocesso de versão no banco de dados. Exemplos de ferramentas de *Migrations* são *Carbon 5*[3] e *Liquibase*[4]. Durante o estágio foi utilizada a ferramenta *Liquibase*.

3.5.2 Controle de versão de banco de dados com *Liquibase*

Liquibase é uma ferramenta de *Migrations* de código aberto para o versionamento de banco de dados. A ferramenta utiliza, como representação das alterações, arquivos XML, YAML, JSON ou SQL. Para este trabalho utilizou-se XML. Como no exemplo abaixo:

```
<changeSet author="gteixeira" id="create-table-programacao-carga">
    <createTable tableName="programacaoCarga">
        <column name="id" type="bigint" autoIncrement="true">
            <constraints primaryKey="true" nullable="false"/>
        </column>
        <column name="cargaId" type="bigint">
            <constraints nullable="false"/>
        </column>
        <column name="tipoProgramacao" type="varchar(10)">
            <constraints nullable="false"/>
        </column>
```

[3] <https://code.google.com/archive/p/c5-db-migration/>
[4] <https://www.liquibase.org/>

```
        <column name="data" type="date"/>
        <column name="viagens" type="integer"/>
        <column name="volume" type="decimal">
            <constraints nullable="false"/>
        </column>
    </createTable>
    <rollback>
        <dropTable tableName="programacaoCarga"/>
    </rollback>
</changeSet>
```

Cada arquivo XML possui um *Database Change Log* onde pode-se registrar um ou mais grupos de alterações chamados *Change Set*. Neste exemplo, tem-se uma criação de uma tabela. Dentro da *tag 'changeSet'* tem-se outra *tag* chamada *'createTable'*. Todas as *tags*, exceto *rollback*, dentro do *change set* informam qual alteração está sendo aplicada, no caso a criação da tabela *programacaoCarga*.As ações do *rollback* estão desfazendo tudo o que foi feito no *change set*. Assim a ferramenta saberá como reverter as alterações de uma versão.

Cada *change set* possui um autor e um id, usados para identificação da alteração e do desenvolvedor que a criou. A ferramenta utiliza esse id para armazenar no banco quais alterações foram aplicadas. A tabela de alterações se chama *databasechangelog*. Para executar as *Migrations* deve-se passar como parâmetro um arquivo XML contendo o *database change log*, então para utilizar múltiplos arquivos deve-se incluir todos os arquivos em um principal, como a seguir.

```
<?xml version="1.0" encoding="UTF-8"?>

<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
```

```
    http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <include file="migrations/programacao_carga.xml"/>
    <include file="migrations/exemplo2.xml"/>
    <include file="migrations/exemplo3.xml"/>
</databaseChangeLog>
```

Na próxima seção, é descrito o *framework* utilizado durante as atividades do estágio e sobre como as configurações são feitas neste caso. Caso não haja utilização do mesmo *framework* o projeto terá um arquivo chamado *liquibase.properties* onde são armazenados os dados de acesso ao seu banco de dados assim como o arquivo principal das *Migrations*.

Com todas as configurações em regularidade deve-se rodar os comandos *liquibase update* para aplicar as alterações ou *liquibase rollback* para reverter alterações. No caso do *rollback* deve-se especificar um limite, que pode ser por número de *change sets (-Dliquibase.rollbackCount=n)* ou por data (”-Dliquibase.rollbackDate=Sep 03, 2019").

3.6 *Framework Dropwizard*

O *Dropwizard*[5] é um *framework Java* de código aberto para desenvolvimento de *RESTful Web Services*. Ele agrupa bibliotecas *Java* como *Jetty* para HTTP, *Jackson* para JSON, *Jersey* para REST e *Hibernate* para banco de dados. Nesta seção é descrito como desenvolver um *Web Service* do zero com *Dropwizard*. Como pré-requisito para utilizar esse *framework*, é necessária a instalação do *Java (JDK)*, disponível no site oficial da *Oracle*[6], do gerenciador de dependên-

[5] <https://www.dropwizard.io/en/latest/>
[6] <https://www.oracle.com/br/index.html>

cias *Maven*, disponível no site oficial da *Apache*[7] e do PostgreSQL[8]. Com tudo instalado precisa-se criar um projeto *Dropwizard* utilizando o *Maven*.

3.6.1 Criando um projeto

O *Maven* possui *templates* chamados arquétipos e pode-se, a partir deles, criar projetos com uma configuração padrão já pronta. Para criar um projeto *Dropwizard*, utiliza-se o arquétipo *dropwizard-archetype*. Para isso, basta executar o comando abaixo na pasta escolhida para o projeto, substituindo o valor *'DROPWIZARD-VERSION'* por uma versão válida do *Dropwizard*[9].

$ mvn archetype:generate
-DarchetypeGroupId=io.dropwizard.archetypes
-DarchetypeArtifactId=java-simple
-DarchetypeVersion=DROPWIZARD-VERSION

Ao executar o comando, serão pedidas algumas informações:

- *groupId*: identificador único do seu projeto seguindo as regras de nomeação de pacotes do *Java*. Exemplo: com.example.myproject
- *artifactId*: um nome para seu arquivo .jar. Exemplo: myproject
- *version*: versão do seu projeto. Exemplo: 1.0
- *package*: nome do pacote principal da aplicação. Geralmente igual ao *groupId*.
- *name*: nome da aplicação. Será o nome da classe principal da aplicação. Exemplo: MyApp

[7] <https://maven.apache.org/download.cgi>
[8] <https://www.postgresql.org/download/>
[9] <https://www.dropwizard.io/en/latest/index.html>

Ao finalizar, será gerada uma estrutura padrão do *Dropwizard* dentro do pacote principal do projeto, com vários outros pacotes, sendo os principais, *core, db e resources*. O pacote *core* é onde armazena-se os *models* da aplicação, ou seja, as classes que representam os dados da aplicação. Por exemplo, considerando uma aplicação de um *pet shop* que deve manter dados sobre o *pet* de seus clientes, deve-se ter uma classe *Pet* similar ao código que segue, onde tem-se o nome da classe, seus atributos e métodos de acesso aos mesmos.

```
package com.example.myproject.core;

public class Pet{

    private String nome;
    private String dono;
    private String especie;
    private String avatarFileName;

    public Pet(String nome, String dono, String especie ){
        this.nome = nome;
        this.dono = dono;
        this.especie = especie;
    }

    public String getNome(){
        return nome;
    }

    ...

    public String getAvatarFileName(){
        return avatarFileName;
    }
```

```
}
```

Nas próximas seções deste trabalho é mostrado como alterar essa classe para utilizar o *Hibernate ORM (Object/Relational Mapping)* e adicionar dados no banco de dados postgreSQL. No pacote *db*, são armazenadas as classes de acesso ao banco de dados para cada *model* criado. E finalmente o pacote *resources*, onde armazena-se as classes dos recursos da aplicação, ou seja, onde são declaradas as URIs dos recursos (endereçamento) e criada a interface uniforme do modelo REST para nosso *Web Service*.

Além dos pacotes, são criadas duas classes *MyAppApplication.java* e *MyAppConfiguration.java*, sendo a primeira a classe que inicializa a aplicação, e a segunda a que armazena as configurações da aplicação.

Com o básico sobre a estrutura do *Dropwizard* apresentado, são detalhados os próximos passos da criação do projeto. Na raiz do projeto foi criado um arquivo *pom.xml* que o *Maven* utiliza para controlar as dependências do projeto. É preciso conferir se na *tag 'properties'* desse arquivo tem uma versão para o *Dropwizard* e adicionar como dependência a biblioteca *dropwizard-core*, como abaixo:

```
<properties>
        ...
        <dropwizard.version>X.X.X</dropwizard.version>
        ...
</properties>
...
<dependencies>
    <dependency>
        <groupId>io.dropwizard</groupId>
        <artifactId>dropwizard-core</artifactId>
        <version>${dropwizard.version}</version>
```

```
    </dependency>
</dependencies>
```

3.6.2 Criando o arquivo de configurações para a aplicação

O Dropwizard permite externalizar a configuração da aplicação utilizando um arquivo YAML. O YAML é um padrão de serialização de dados com um visual amigável e simples de se entender. Neste arquivo, serão armazenadas algumas variáveis necessárias para a execução da aplicação, como: dados para o acesso ao banco de dados, caminhos para diretórios, porta utilizada, dentre outras informações. Essas informações estarão disponíveis na classe *MyAppConfiguration*. Exemplo de arquivo YAML:

```
# O caminho para a pasta onde estão os avatares dos pets
petAvatarPath: /storage/pet/avatar
database:
  # O driver JDBC utilizado pela aplicação (PostgreSQL)
  driverClass: org.postgresql.Driver
  # O usuário que a aplicação utilizará ao acessar
  # o banco de dados
  user: postgres
  # A senha para acesso ao banco de dados
  password: 123456
  # A URL para o banco de dados
  url: jdbc:postgresql://localhost:5432/meu_pet_shop
```

No arquivo apresentado são definidos o banco de dados da aplicação e o diretório onde serão armazenados os avatares de todos os *pets*. Para ter acesso a esses dados é necessário que na classe *MyAppConfiguration* haja funções que retornam esses dados. Como no exemplo:

```
package com.example.myapp;
```

```
...

public class MyAppConfiguration extends Configuration {
    @NotEmpty
    private String petAvatarPath;
    @Valid
    @NotNull
    private DataSourceFactory database = new DataSourceFactory();

    @JsonProperty
    public String getPetAvatarPath() {
        return petAvatarPath;
    }

    @JsonProperty
    public void setPetAvatarPath(String petAvatarPath) {
        this.petAvatarPath = petAvatarPath;
    }

    @JsonProperty("database")
    public DataSourceFactory getDataSourceFactory() {
        return database;
    }

    @JsonProperty("database")
    public void setDataSourceFactory(
        DataSourceFactory dataSourceFactory
    ) {
        this.database = dataSourceFactory;
    }
}
```

A classe *DataSourceFactory* é uma classe padrão do framework e contém as informações para acesso ao banco de dados. Os atributos da classe *MyAppConfiguration* serão inicializados com os valores inseridos no arquivo YAML. As anotações *NotNull, Valid e Not Empty* são restrições aos atributos e a anotação *JsonProperty* informa uma propriedade devendo definir o nome da propriedade como está no arquivo.

3.6.3 Hibernate e banco de dados

Na classe principal da aplicação, *MyAppApplication*, há três métodos sobrescritos: *getName, initialize e run*. O primeiro apenas retorna o nome da aplicação, o segundo inicializa dados da aplicação, instanciando objetos, e o terceiro é a execução da aplicação em si. Para configurar o acesso ao banco de dados utilizando Hibernate, deve-se criar um atributo do tipo *HibernateBundle* passando como parâmetro todos os *models* da aplicação, como segue:

```
public final HibernateBundle<MyAppConfiguration> hibernateBundle =
    new HibernateBundle<MyAppConfiguration>(Pet.class){
        @Override
        public DataSourceFactory getDataSourceFactory(
            MyAppConfiguration configuration
        ) {
            return configuration.getDataSourceFactory();
        }
    };
```

No método *initialize* deve-se adicionar o *Hibernate bundle* ao *bootstrap* da aplicação.

```
@Override
public void initialize(
    final Bootstrap<MyAppConfiguration> bootstrap
```

```
) {
    bootstrap.addBundle(hibernateBundle);
}
```

Para que o Hibernate utilize os *models* da aplicação como as estruturas dos dados que serão armazenados no banco, cada classe deve conter anotações com informações sobre os dados. Inicia-se informando que a classe é uma entidade utilizando a anotação *'@Entity'*, a anotação *'@Table'* permite definir os nomes da tabela, catálogo e esquema para a entidade. Para os atributos da classe a anotação *'@Column'* informa o nome da coluna referente aquele atributo. Para definir a *Primary Key* da tabela utiliza-se *'@Id'*. Há diversas outras anotações utilizadas para mapear a entidade, recomenda-se a leitura da documentação[10] da biblioteca para a identificação de todas elas. Outro exemplo de anotação é a *'@GeneratedValue'* que serve para definir a estratégia de geração da *Primary Key* da tabela. A classe *Pet* apresentada em seções anteriores deve ser mapeada, como segue:

```
@Entity
@Table(name = "pet")
public class Pet{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column(name = "nome", nullable = false)
    private String nome;
    @Column(name = "dono", nullable = false)
    private String dono;
    @Column(name = "especie", nullable = false)
    private String especie;
    @Column(name = "avatar")
    private String avatarFileName;
```

[10] https://hibernate.org/orm/documentation/5.4/

```
    ...

}
```

As relações entre as entidades também devem ser mapeadas. No caso de uma relação M para N, será necessário a criação de outro *model* que também deve ser mapeado. Após realizado o mapeamento de todos os *models* deve-se criar uma classe de acesso aos dados para cada *model*, geralmente chamada pelo nome do *model* + DAO, exemplo: *PetDAO*. Essas classes devem estender *AbstractDAO*.

```
package com.example.myapp.db;

public class PetDAO extends AbstractDAO<Pet> {
    public PetDAO (SessionFactory factory){
        super(factory);
    }
}
```

É a classe *PetDAO* que deve conter todas as funções que acessam a tabela *pet* da aplicação. Para isso a classe deve ser instanciada em *MyAppApplication*, no método *run*. Perceba que no construtor da classe deve-se passar uma *SessionFactory*, é o *Hibernate bundle* criado anteriormente que a fornecerá:

```
final PetDAO petDao = new PetDAO(
    hibernateBundle.getSessionFactory()
);
```

3.6.3.1 Criando métodos de acesso ao banco de dados

Nesta seção é apresentado como criar métodos de acesso ao banco de dados. As classe *Pet* e *PetDAO* criadas anteriormente são utilizadas como exemplos. Os métodos para adicionar, remover, alterar ou buscar os dados são implementados na classe *PetDAO*. Para criar ou alterar uma entidade utiliza-se o mesmo

método chamado *persist*. Esse método funciona como um *update* caso o *id* do objeto exista, caso contrário, adiciona o objeto no banco de dados criando um novo registro. Para deletar um registro do banco de dados utiliza-se o método *delete*.

```
public class PetDAO extends AbstractDAO<Pet> {
    public PetDAO (SessionFactory factory){
        super(factory);
    }

    public Pet persist(Pet pet){
        return currentSession().persist(pet);
    }

    public void delete(Pet pet){
        currentSession().delete(pet);
    }
}
```

Para as consultas ao banco de dados é utilizada a *Criteria API*[11] que é uma API usada para definir as consultas de uma entidade. Ela permite que sejam realizados diversos tipos de consultas simples ou complexas.

```
public List<Pet> getAll(){
    CriteriaBuilder builder = currentSession().getCriteriaBuilder();
    CriteriaQuery<Pet> query = builder.createQuery(Pet.class);
    Root<Pet> root = query.from(Pet.class);

    query.select(root);

    return list(query);
}
```

[11] https://www.objectdb.com/java/jpa/query/criteria

No exemplo acima, a consulta retorna todos os *pets* cadastrados no banco. Pega-se a instância da classe *CriteriaBuilder* da seção atual que é utilizada para criar uma instância da classe *CriteriaQuery* que armazenará os dados da consulta. Cria-se uma instância *Root* para informar a raiz da consulta. A linha *query.select(root);* informa que será retornada toda a entidade, essa linha não é necessária nesse caso, pois como há apenas uma raiz a seleção de toda a entidade seria implícita. Por último é retornado o resultado da consulta. Caso o resultado pretendido seja apenas o nome do *pet*, o *select* da consulta seria alterado. Mais além, considere que a busca seja de um nome por um *id*, ou seja, tem-se o *id* do *pet* e quer-se o seu respectivo nome. Para tal consulta deve-se definir um filtro para a consulta utilizando a cláusula *where*.

```
public String getNameById(long id){
    CriteriaBuilder builder = currentSession().getCriteriaBuilder();
    CriteriaQuery<String> query = builder.createQuery(String.class);
    Root<Pet> root = query.from(Pet.class);

    query.select(root.get("nome"));
    query.where(
        builder.equal(root.get("id"),id)
    );

    return currentSession().createQuery(query).getSingleResult();
}
```

Utiliza-se a instância da classe *CriteriaBuilder* para criar as condições da cláusula *where*. Nota-se que para identificar uma coluna da tabela utiliza-se a raiz da consulta, tanto para o *select* quanto para o *where*. Como a entidade da consulta (*String*) e a entidade da raiz da consulta (*Pet*) não são a mesma, deve-se usar *currentSession().createQuery(query)* para conseguir criar a consulta. Caso

o resultado esperado seja uma lista de objetos, para obtê-lo utiliza-se o seguinte método:

```
return currentSession().createQuery(query).getResultList();
```

Caso contrário, se for apenas um objeto, como o exemplo acima, utiliza-se o método:

```
return currentSession().createQuery(query).getSingleResult();
```

3.6.4 Criando os *Resources* da aplicação

Nas seções anteriores, foi apresentado como criar um projeto dropwizard utilizando o *maven*, como configurar a aplicação utilizando um arquivo YAML, como criar os *models* da aplicação e mapeá-los utilizando anotações Hibernate e como acessar os dados no banco de dados. Nesta seção, é apresentado como criar os *resources* da aplicação, que são os recursos disponibilizados pelo *webservice*. Cada recurso é uma classe com métodos acessíveis por requisições HTTP. É nessas classes que a interface uniforme[12] é implementada. A interface uniforme é um conjunto de métodos bem definidos para todos os recursos. Seguindo o exemplo do *PetShop*, suponha que também são armazenados no banco de dados as informações do dono de cada *pet*, tem-se então duas entidades, *Pet* e Dono. É criada uma classe de recurso para cada uma dessas entidades onde os métodos implementados são padronizados, por exemplo, *inserir, alterar, deletar, listarTodos, etc.*

Os métodos da classe de recurso não acessam diretamente o banco de dados. A classe *PetDAO* é utilizada em *PetResource* para realizar esse acesso. Todo método que acessa o banco de dados deve conter a anotação *@UnitOfWork* que é responsável por criar uma seção de acesso ao banco. Como os métodos da classe recurso são serviços disponibilizados na web, deve-se informar o método HTTP[13]

[12] Seção 3.3.1
[13] Seção 3.2.1

e o caminho para aquele serviço, gerando um endereçamento[14] para os recursos. Deve haver endereçamento para a classe também.

```
package com.example.myapp.resources;
...
@Path("/pet")
@Produces(MediaType.APPLICATION_JSON)
public class PetResource{
    private final PetDAO petDao;

    public PetResource(PetDAO petDao){
        this.petDao = petDao;
    }

    @GET
    @UnitOfWork
    public Response listAll(){
        List<Pet> pets = petDao.listAll();
        return Response.ok().entity(pets).build();
    }

    ...
}
```

A anotação *@Path* é utilizada para informar o caminho para o recurso e pode ser utilizada tanto na classe recurso quanto nos métodos implementados nessa classe. *@Produces* informa que tipo de dado é gerado pelo recurso, nesse caso, um objeto JSON. Nota-se que o construtor da classe recebe uma instância da classe *PetDAO* que realiza o acesso ao banco de dados. Como apresentado anteriormente, *@UnitOfWork* cria uma seção para consulta e manipulação dos dados e por último

[14] Seção 3.3

a anotação *@GET* informa que o método HTTP[15] utilizado para o acesso àquele recurso é o método *GET*.

Todo recurso criado para a aplicação deve ser registrado no método *run* da classe *MyAppApplication* para que ele seja disponibilizado.

```
@Override
    public void run(final HelloWorldConfiguration configuration,
                    final Environment environment) {
        final PetDAO petDao = new PetDAO(
            hibernateBundle.getSessionFactory()
        );

        environment.jersey().register(new PetResource(petDao));
    }
```

Dessa forma, toda requisição HTTP no formato ***GET /pets*** listará todos os *pets* cadastrados no banco de dados.

3.6.4.1 Recebendo dados da requisição HTTP

Há diversas formas de receber dados de uma requisição, nesta seção são apresentadas 3 delas: os *path parameters*, os *query parameters* e o corpo da requisição. Os dois primeiros são informados na URI da requisição mas se diferem no propósito. Os *path parameters* são utilizados para identificar um recurso, por exemplo, quer-se acessar o recurso *Pet* de id igual a 12, então tem-se a seguinte requisição:

GET /pets/12

[15] Seção 3.2.1

Mas para receber esse valor e realizar a busca corretamente devemos informar ao método do recurso que será enviado um parâmetro do tipo *path* utilizando a anotação *@PathParam*. O parâmetro do tipo *path* deve ser informado dentro da *tag* *@Path* também, como a seguir:

```
@GET
@Path("/{id}")
@UnitOfWork
public Response getPetById(@PathParam("id") long id){
    Pet pet = petDao.getPetById(id);
    return Response.ok().entity(pet).build();
}
```

Já os parâmetros do tipo *query* são utilizados para filtrar ou ordenar os recursos, por exemplo, para buscar todos os *pets* existentes no banco de dados ordenados por nome.

GET /pets?order=name

Os parâmetros *query* também devem ser informados no método do recurso utilizando a anotação *@QueryParam*, mas não são informados na *tag* *@Path*:

```
@GET
@UnitOfWork
public Response listAll(@QueryParam("order") String order){
    List<Pet> pets = petDao.listAll(order);
    return Response.ok().entity(pets).build();
}
```

Por último os parâmetros do corpo da requisição, estes não precisam de anotações, pode-se declará-los diretamente. Por exemplo, ao cadastrar um *pet*:

```
@POST
@UnitOfWork
public Response create(Pet pet){
    pet = petDao.create(pet);
    return Response.ok().entity(pet).build();
}
```

4 ATIVIDADES DESENVOLVIDAS

Neste capítulo, são apresentadas as atividades de maior importância realizadas durante o período do estágio. Essas atividades foram realizadas com o intuito de manter e evoluir o sistema *web* da RodoSafra que é utilizado para gerenciamento das atividades da empresa. Apesar deste trabalho não ter como foco o *front-end* da aplicação, também foram realizadas atividades relacionadas ao mesmo.

4.1 Treinamento

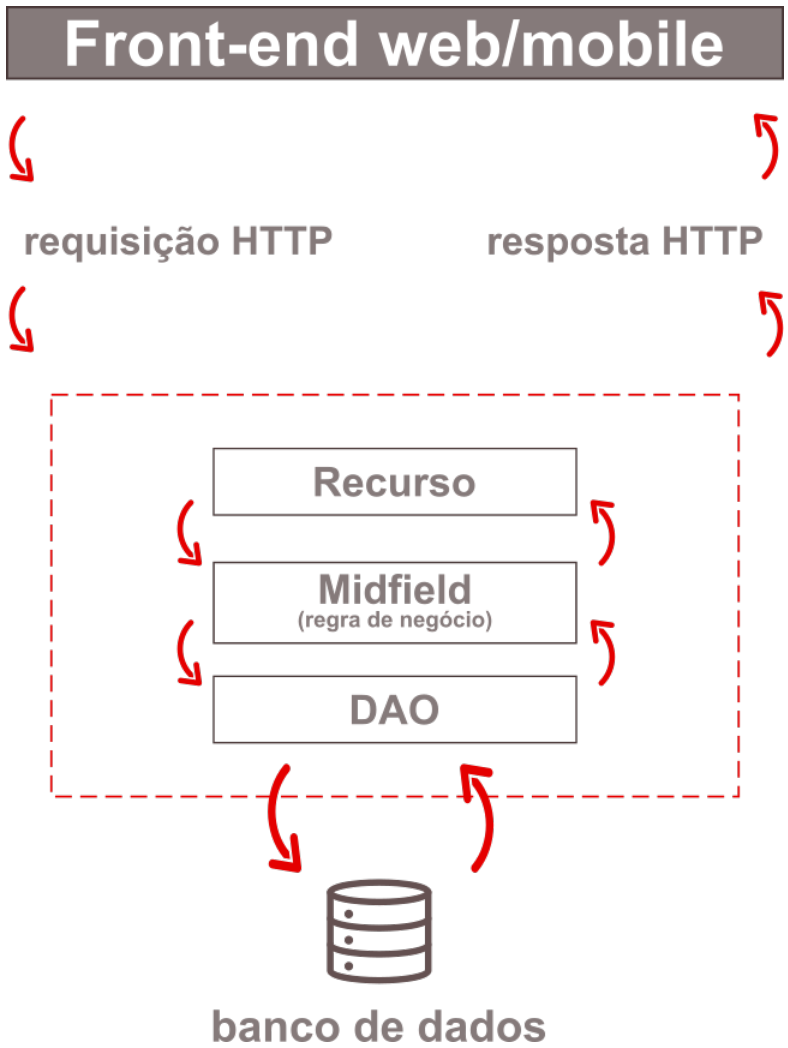
Para o desenvolvimento do sistema são utilizadas ferramentas específicas que exigem um conhecimento mais detalhado, então foi necessária a realização de um treinamento com duração de três meses. Foi disponibilizado para o estagiário um material com todo o conteúdo necessário para que o treinamento fosse concluído, com vídeos, PDFs e exercícios. Foram realizadas também reuniões com o outro membro da equipe para esclarecimento de dúvidas e avaliações do andamento do treinamento.

Além da realização de estudos voltados às tecnologias utilizadas, na fase de treinamento foi discutida a estrutura atual do *webservice* e do *framework Dropwizard* adotado pela empresa. Como apresentado na Seção 3.6, ao criar-se um projeto Dropwizard tem-se uma estrutura padrão gerada automaticamente pelo framework. Nessa estrutura padrão a classe que representa o recurso acessa diretamente a classe DAO alterando os dados no banco de dados. Mas, na prática isso não ocorre porque as aplicações devem seguir as regras de negócio da empresa que existem em diversos formatos para todas as empresas, então foi necessário a realização de adaptações na estrutura para atender essas necessidades.

Durante o treinamento foi possível entender como foram realizadas as adaptações no sistema RodoSafra para que as regras de negócio fossem melhor estruturadas. Foi criado um pacote chamado *midfield* que representa uma camada

intermediária entre os recursos e as DAOs. É nessa camada onde implementam-se as regras de negócio, por exemplo, ao criar um embarque deve-se definir seu *status* inicial levando em consideração o índice de confiabilidade do motorista, requisitos do cliente fornecedor para a carga em questão, quem enviou a solicitação, dentre outras informações. Esse cálculo é realizado em um *midfield* criado para os recursos da entidade embarque, com um nome parecido com *EmbarqueMidfield*. A Figura 4.1 demonstra o fluxo da aplicação incluindo a camada *midfield*.

Figura 4.1 – Ilustração da camada *midfield* da aplicação



O sistema RodoSafra possui integrações com outros *webservices* para realizar operações como emissão de notas fiscais para o embarque. Para que essas integrações fossem implementadas foi criado um pacote chamado *remote* onde são armazenadas todas as classes que representam um serviço externo, com exceção ao envio de e-mails que está em um pacote específico chamado *email*.

Durante o período de operação de um embarque são necessárias adições de diversos arquivos e documentos. Para tal tarefa foi criado um simples sistema de arquivos em um novo pacote chamado *fileSystem*. Outros pacotes foram criados conforme a necessidade de reestruturação do projeto.

Ao finalizar o treinamento, houve entendimento do funcionamento do sistema RodoSafra e das tecnologias utilizadas para em seguida realizar as outras atividades.

4.2 Melhoria na performance de listagens

As listagens dos dados no sistema não possuíam paginação nas consultas ao banco de dados, ou melhor, a paginação era realizada no *frontend* e com isso algumas delas mostravam problemas de performance devido à quantidade de dados listados. As principais entidades do sistema RodoSafra são Embarque, Carga, Veículo, Motorista e Parceiro.

Um embarque é sempre vinculado a uma carga e um motorista, e pode conter até 4 veículos, sendo um veículo cavalo ou caminhão (obrigatório) e os outros três veículos equipamentos (opcionais). Uma carga possui um parceiro fornecedor, e pode possuir parceiros recebedor e expedidor. Os registros de Embarque e Carga são numerosos mas as listagens mais utilizadas para essas entidades buscam apenas as que estão em operação, ou seja, ignorando embarques e cargas finalizados ou suspensos. Esse filtro faz com que a quantidade de registros buscados caia drasticamente. A quantidade de parceiros também não é suficiente para causar problemas de performance na listagem.

Já a quantidade de Motorista e Veículo é grande e gerava certa lentidão ao realizar consultas de listagem. Para a correção do problema foi alterada a maneira de realização da consulta e implementada uma paginação no *backend* e banco de dados. É apresentada como exemplo essa alteração realizada para a entidade Motorista embora ela também tenha ocorrido para outras entidades.

O código a seguir demonstra como a consulta era realizada antes das alterações:

```
public List<Motorista> findAll() {
    CriteriaBuilder cb = currentSession().getCriteriaBuilder();
    CriteriaQuery<Motorista> query = cb.createQuery(Motorista.class);
    Root<Motorista> root = query.from(Motorista.class);

    return list(query);
}
```

Nota-se que todos os registros da tabela motorista eram retornados. Para realizar a paginação no *backend* necessita-se saber o número da página e o tamanho de cada página. Uma classe foi criada para armazenar essas informações após recebidas a partir da requisição HTTP[1].

```
public class SearchWithPaginationRequest {
    private final Integer draw;
    private final Integer start;
    private final Integer length;
    private final String search;

    ...
}
```

O atributo *start* é o número da página (inicia-se em zero) vezes o tamanho da página, ou seja, o índice do primeiro registro da página, *length* é o tamanho

[1] Seção 3.6.4.1

da página, *search* é o valor do filtro que era realizado também no *frontend* e por fim *draw* é utilizado pelo *frontend* para validar a resposta recebida, o *backend* não altera esse valor, apenas o recebe e o devolve.

Para construir uma interface de paginação amigável ao usuário, o *frontend* necessita de algumas informações adicionais. Na nova implementação, além de enviar os registros da página atual, o *backend* deve também enviar a quantidade total de registros, a quantidade total de registros filtrados e o *draw*. Para isso foi criada uma classe utilizada para enviar esses dados na resposta HTTP, denominada *SearchWithPaginationResponse*.

```
public class SearchWithPaginationResponse {
    private int draw;
    private long recordsTotal;
    private long recordsFiltered;
    private List data;

    ...
}
```

Classes como *SearchWithPaginationRequest* e *SearchWithPaginationResponse* são utilizadas para receber e enviar dados e parâmetros por meio de mensagens HTTP. Essas classes representam a comunicação entre o *frontend* e o *backend* e foram chamadas de *modelviews* e são armazenadas no pacote de mesmo nome.

São realizadas três consultas ao banco de dados para obter os dados necessários. A primeira e mais simples conta o total de registros no banco de dados utilizando a operação *count* e foi implementada no método *getTotalRecords*.

```
private Long getTotalRecords(CriteriaBuilder builder){
    CriteriaQuery<Long> queryCount = builder.createQuery(Long.class);
    Root<Motorista> root = queryCount.from(Motorista.class);

    queryCount.select(builder.count(root));
```

```
    return currentSession().createQuery(queryCount).getSingleResult();
}
```

As outras duas consultas buscam a quantidade de registros filtrados e os registros filtrados da página atual. Nas duas situações precisa-se informar os filtros utilizando a cláusula *where*. Foi criada então uma função que retorna uma lista de predicados dos filtros, denominada *makeSearchInMotorista*.

```
private Predicate[] makeSearchInMotorista(
    String search,
    CriteriaBuilder builder,
    Root<Motorista> root
) {
    return new Predicate[]{
            builder.like(
                builder.lower(root.get("nome")),
                search.toLowerCase()
            ),
            builder.like(root.get("cpf"), search),
            builder.like(root.get("id").as(String.class), search),
            builder.like(root.get("fone1"),search),
            builder.like(root.get("fone2"),search)

    };
}
```

O filtro é utilizado para fazer buscas por nome, CPF, id e telefones do motorista. Para comparar *Strings* é utilizado o operador *like*. Para ignorar se o nome do motorista está em letras maiúsculas ou minúsculas são transformadas todas as letras em minúsculas utilizando o operador *lower*.

A segunda consulta, denominada *getRecordsFiltered*, é parecida com a primeira, a diferença é que ela recebe os predicados criados na função *makeSear-*

chInMotorista. Os predicados são utilizados com o operador *or*, ou seja, o motorista vai ser selecionado se satisfizer pelo menos um dos predicados.

```
private Long getRecordsFiltered(
    CriteriaBuilder builder,
    Predicate[] search
){
    CriteriaQuery<Long> queryRecordsFiltered
        = builder.createQuery(Long.class);
    Root<Motorista> rootCounter
        = queryRecordsFiltered.from(Motorista.class);

    queryRecordsFiltered.select(builder.count(rootCounter));
    queryRecordsFiltered.where(builder.or(search));

    return
        currentSession()
        .createQuery(queryRecordsFiltered)
        .getSingleResult();
}
```

A terceira consulta, denominada *getRecords*, é a mais complexa, que além de filtrar os registros, os ordena por *id* em ordem decrescente e possui uma subconsulta para realizar a paginação em si. A ordenação é realizada pelo operador *desc*.

```
private List getRecords(
    CriteriaBuilder builder,
    SearchWithPaginationRequest request,
    Predicate[] search
){
    CriteriaQuery<Motorista> query
        = builder.createQuery(Motorista.class);
```

```
    Root<Motorista> root = query.from(Motorista.class);

    queryMotoristas.select(root);
    queryMotoristas.where(builder.or(search));
    queryMotoristas.orderBy(builder.desc(root.get("id")));

    Query<Motorista> subQueryForPagination
        = currentSession().createQuery(query);
    subQueryForPagination.setFirstResult(request.getStart());
    subQueryForPagination.setMaxResults(request.getLength());

    return subQueryForPagination.list();
}
```

Com as três consultas criadas basta finalizar a função que as chama e retorna um objeto da classe *SearchWithPaginationResponse* que será enviado na resposta HTTP.

```
public SearchWithPaginationResponse searchWithPagination(
    SearchWithPaginationRequest request
) {
    CriteriaBuilder builder = currentSession().getCriteriaBuilder();
    Predicate[] predicatesSearch = makeSearchInMotorista(
        request.getSearch(),
        builder,
        root
    );

    Long recordsFiltered = getRecordsFiltered(
        builder,
        predicatesSearch
    );
    Long totalRecords = getTotalRecords(builder);
```

```
    List<MotoristaModelView> motoristas = getRecords(
        builder,
        request,
        predicatesSearch
    );

    return new SearchWithPaginationResponse(
        request.getDraw(),
        totalRecords,
        recordsFiltered,
        motoristas
    );
}
```

4.3 Melhorias na dinâmica das atividades do setor financeiro

Um custo é um gasto da empresa em algum produto ou serviço prestado. Os custos são cadastrados manualmente por um funcionário do setor financeiro e, na implementação anterior do sistema, caso houvesse algum erro durante o cadastramento não havia a funcionalidade de edição de um custo, ou seja, o usuário deveria deletar e recadastrar o mesmo. Após informado o pagamento de um determinado custo, este não podia ser cancelado ou deletado. Caso houvesse divergência nos dados do pagamento a equipe de TI era informada para que os dados fossem corrigidos. Essa dinâmica gerava atrasos de operação durante as atividades da empresa e para corrigir esses problemas foram criadas duas funcionalidades: a edição de um custo e o cancelamento de um pagamento de custo.

O setor financeiro da empresa também possuía o problema de ter atrasos no acesso dos recebíveis. Os recebíveis são valores a receber pela operação de transporte então cada um é relacionado a um embarque. Um embarque possui um ou mais CT-es (Conhecimento de transporte eletrônico) que são documentos

fiscais da operação de transporte, semelhantes a uma nota fiscal para produtos e serviços. Para gerar as faturas para os clientes são agrupados vários recebíveis e muitas das vezes os recebíveis são identificados pelos CT-es do embarque. O sistema não possuía uma listagem de recebíveis com filtro pelo número de CT-e, isso significa que o usuário primeiro deveria identificar o código do embarque daquele CT-e e depois filtrar o recebível pelo embarque. Para simplificar esse processo foi implementado um filtro de recebíveis por número de CT-e. A listagem de recebíveis também passou pelas modificações da atividade da seção 4.2 deste documento, então o filtro é implementado no *back-end* por meio de consultas no banco de dados utilizando a *Criteria API*[2].

4.4 Geração de Relatórios

A cada dia de operação do sistema há o armazenamento de muitos dados referentes aos serviços prestados e contratados. Foi exposta ao estagiário a necessidade de gerenciamento e análise desses dados para tomadas de decisão por parte da empresa. O sistema RodoSafra já possuía alguns relatórios com gráficos e métricas sobre alguns setores, como: Cadastro, Despachante, Financeiro, e outros. O objetivo desta atividade foi desenvolver novos relatórios de análise dos dados existentes. São apresentados os relatórios de volume embarcado e ação por usuário para ilustrar a atividade.

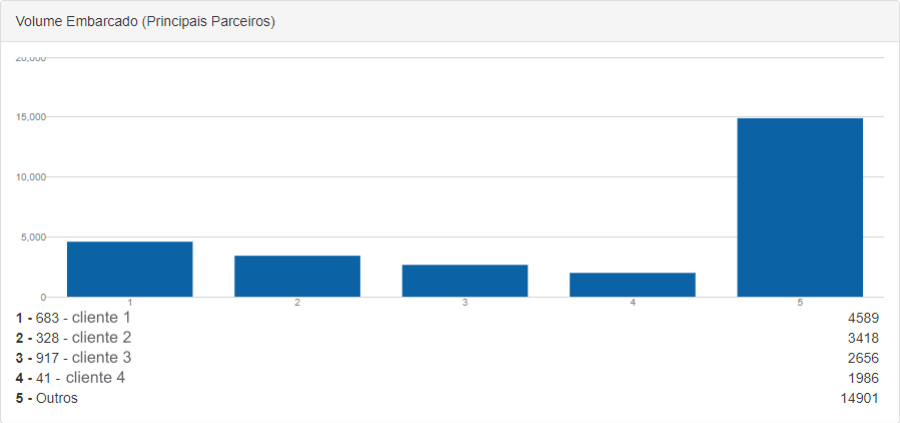
4.4.1 Volume Embarcado

O relatório de volume embarcado é responsável por apresentar a quantidade em toneladas de carga carregada para cada cliente em um determinado período. O relatório é formado por um valor total de toneladas carregadas, um gráfico com os principais clientes (apresentado pela figura 4.2) e uma tabela de clientes or-

[2] Seção 3.6.3.1

denada por volume embarcado. A principal utilidade deste relatório é saber quais são os clientes mais ativos durante o período selecionado.

Figura 4.2 – Gráfico dos principais clientes para o período de 6 de julho de 2020 à 4 de agosto de 2020



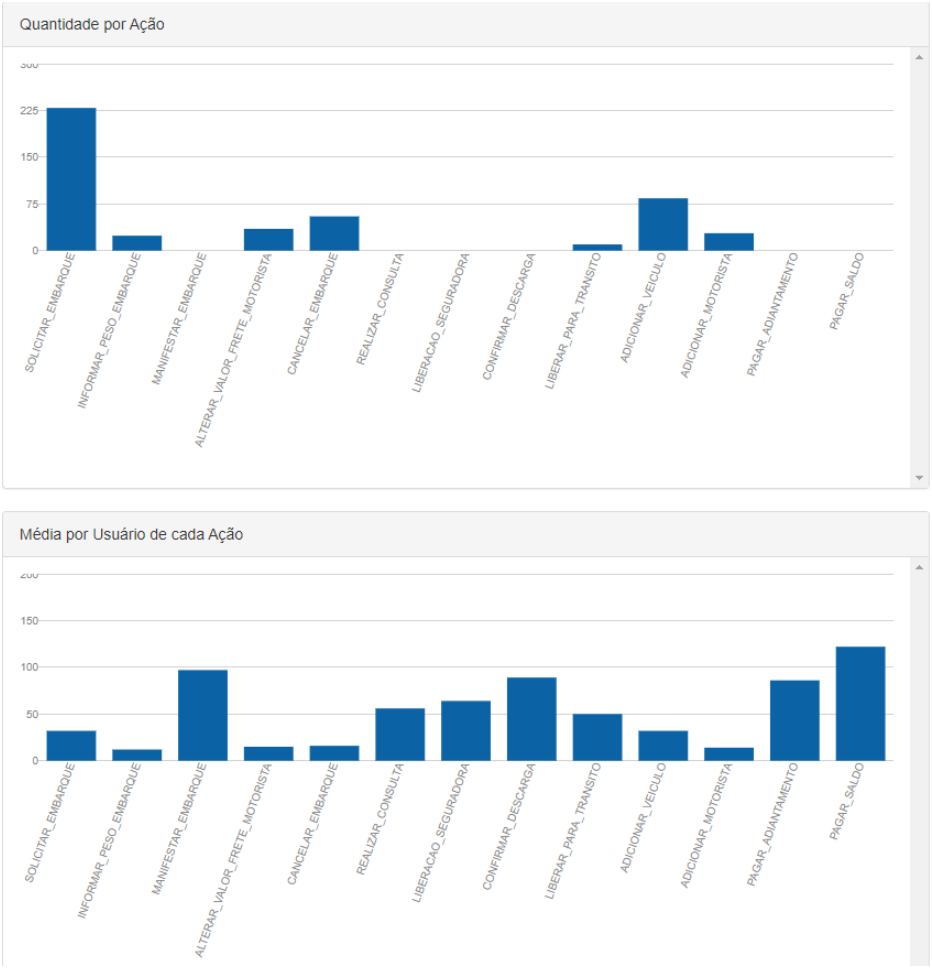
4.4.2 Ação por Usuário

O relatório de ações por usuário foi criado a partir da necessidade de monitorar as ações tomadas pelos usuários do sistema em um determinado período com o objetivo de avaliar a dinâmica dos setores da empresa e a atividade dos funcionários de cada setor. O relatório é formado por uma quantidade total de ações que o usuário selecionado realizou, uma média do total de ações dos usuários, e uma quantidade de embarques efetivos, ou seja, embarques que foram solicitados pelo usuário selecionado e que foram finalizados com sucesso.

Além dessas métricas há também dois gráficos (apresentados pela figura 4.3):

- Gráfico de quantidade por ação do usuário
- Gráfico da média de cada ação por usuário

Figura 4.3 – Gráficos de ação por usuário para um usuário específico no período de 6 de julho de 2020 à 4 de agosto de 2020



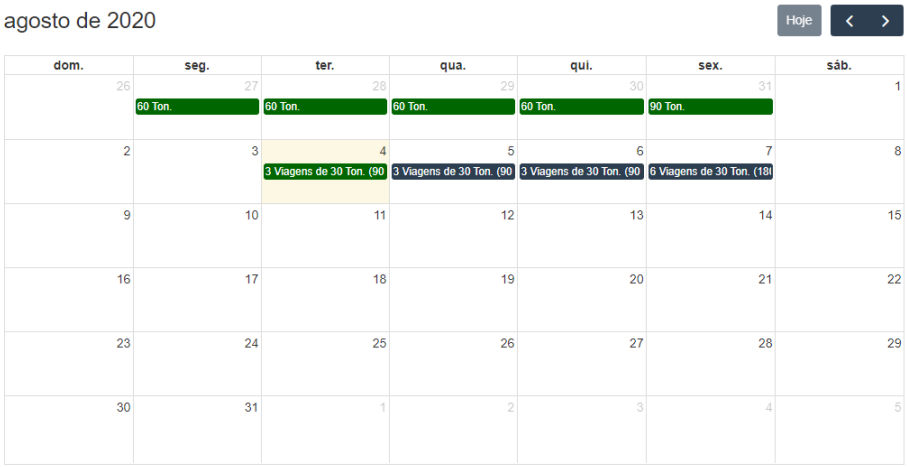
4.5 Gestão de Cargas

O sistema RodoSafra sempre teve maior foco no gerenciamento dos embarques, por isso, seu mapa operacional (tela inicial) sempre foi uma listagem, informando a situação e apresentando as ações que podem ser tomadas para cada embarque. Com o crescimento da empresa a quantidade de parceiros fornecedores e suas cargas aumentaram e notou-se uma dificuldade no gerenciamento dessas cargas. Por exemplo, controlar saldo da carga, cadência de carregamentos, valor

por frete, dentre outras informações. Em uma tentativa bem sucedida de amenizar essa dificuldade uma página específica para gestão de cargas foi elaborada.

Para o controle da cadência dos carregamentos foi criada uma funcionalidade de cadastrar manualmente uma programação para uma determinada carga. Suponha uma carga de açúcar com um saldo de 1000 toneladas, pode-se programar como esse saldo será carregado, por exemplo, em 10 dias corridos carregando 100 ton. cada dia ou nos próximos 7 dias carregando 200 ton. segunda, terça e quarta-feira e 100 ton. nos outros dias. A programação pode ser também por quantidade de viagens, por exemplo, nos próximos 10 dias sendo 4 viagens de 25 ton. por dia.

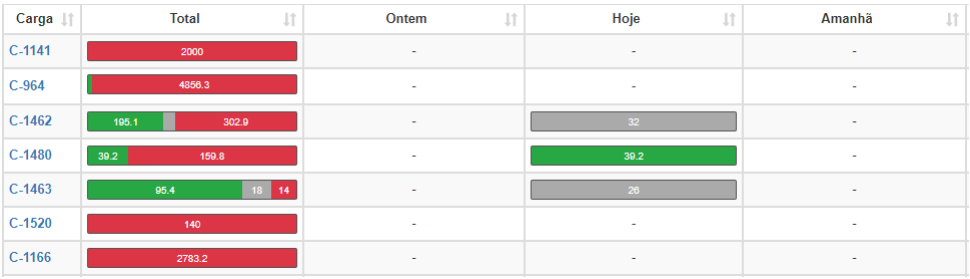
Figura 4.4 – Calendário para cadastro de programação para uma carga



A gestão de cargas tem por objetivo auxiliar o usuário a tomar decisões no dia a dia referente às cargas em operação, por exemplo, agendar ou não mais um embarque para uma determinada carga. Para que esse tipo de decisão seja tomada a gestão de cargas deve apresentar a situação atual da carga. Foi decidido então que a funcionalidade apresentaria o progresso da carga no período do dia anterior ao acesso, ao dia posterior (ontem, hoje e amanhã). Para cada dia há uma barra de progresso como as da Figura 4.5, sendo verde a quantidade de toneladas carregadas, cinza a quantidade de toneladas agendadas para carregamento e em

vermelho a quantidade que falta para atingir a programação da carga. Além dos três dias também foi criado um total onde vermelho é o saldo total da carga.

Figura 4.5 – Barras de progresso da gestão de cargas



Todas ações que um usuário pode realizar referente a um embarque está no mapa operacional, a tela de gestão de cargas tem a mesma função porém referente às cargas. Por exemplo, solicitar um embarque para aquela carga, alterar valor do frete, cadastrar ou alterar programação, dentre outras, são ações encontradas nessa página.

Ao clicar em uma barra de progresso o usuário é redirecionado ao mapa operacional com os devidos filtros aplicados. Por exemplo, caso o usuário clique na parte verde da barra de uma carga de código C-2020 na coluna 'Hoje', no mapa operacional serão aplicados os seguintes filtros: embarques da carga C-2020 carregados hoje. Assim tem-se um gerenciamento facilitado das cargas do sistema.

As atividades apresentadas foram as de maior impacto e que agregaram mais ao sistema e à empresa, no entanto, houveram várias atividades secundárias incluindo pequenas novas funcionalidades, correções de *bugs*, reestruturação de código e atividades relacionadas ao *frontend* da aplicação que não é o foco deste trabalho.

5 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo apresentar a evolução e manutenção do sistema *web* RodoSafra, assim como as ferramentas, conceitos e metodologias utilizadas durante a realização das atividades.

Durante o período do estágio notou-se a importância do currículo acadêmico do curso de Ciência da Computação que traz um embasamento da maioria dos conceitos apresentados neste trabalho. O estágio em si, por se tratar da prática desses conceitos, foi essencial no aprimoramento do conhecimento adquirido no decorrer do curso. Não só a presença dessas atividades práticas colaboraram com o crescimento profissional mas a falta delas também. Por exemplo os conceitos trabalhados na disciplina eletiva 'Processos de Software', por se tratar de um setor de TI imaturo com metodologias de desenvolvimento pouco estruturadas foram encontradas dificuldades relacionadas ao cumprimento de metas, priorização de demandas, retrabalho, dentre outras. Essas dificuldades foram minimizadas com a utilização de técnicas de metodologias ágeis como as reuniões e entregas semanais e a utilização da ferramenta Trello.

Por se tratar de *RESTful web services*[1] com acesso a um servidor de banco de dados, foi de extrema importância o *background* adquirido na disciplina de sistemas distribuídos. Outras disciplinas cujas contribuições foram essenciais são: Estruturas de Dados, Práticas de Programação Orientada a Objetos, Banco de Dados e Sistemas Gerenciadores de Banco de Dados.

As atividades descritas neste trabalho[2] foram de imenso impacto na realização e gerenciamento das atividades da empresa, reduzindo o tempo necessário para algumas operações e trazendo maior eficiência nas mesmas.

As possíveis melhorias para este projeto estão relacionadas a dar mais inteligência ao sistema utilizando análises dos dados armazenados durante as ope-

[1] Seções 3.2 e 3.3
[2] Capítulo 4

rações, prevendo perdas e atrasos em algumas situações, por exemplo, o cancelamento do embarque por parte do motorista ou apresentando relatórios que tenham efetividade no auxílio nas tomadas de decisões da empresa.

REFERÊNCIAS

ANDRADE, M. **Controle de versão em banco de dados com migrations**. 2012. Disponível em: <https://tasafo.wordpress.com/2012/10/13/controle-de-versao-em-bd/>. Online. Acessado em 02 jun. 2020.

CNT. **Anuário CNT do Transporte - 2019 - Malha Rodoviária Total, Confederação Nacional do Transporte**. 2019. Disponível em: <https://anuariodotransporte.cnt.org.br/2019/Rodoviario/1-3-1-1-1-/Malha-rodoviária-total>. Online. Acessado em 05 jan. 2020.

CNT. **Boletim Econômico - CNT - Novembro 2019**. 2019. Disponível em: <https://www.cnt.org.br/boletins>. Online. Acessado em 05 jan. 2020.

CNT. **Boletim Estatístico - CNT - Fevereiro 2019**. 2019. Disponível em: <https://www.cnt.org.br/boletins>. Online. Acessado em 05 jan. 2020.

KUROSE, J. F.; ROSS, K. W. **Computer Networking - A Top-Down Approach**. Seventh. [S.l.]: Pearson, 2017.

MARZULLO, F. P. **SOA na Prática**. [S.l.]: Novatec Editora, 2009.

POSTGRESQL. **Site oficial do Postgres SQL**. 2020. Disponível em: <https://www.postgresql.org/about/>. Online. Acessado em 23 mai. 2020.

RICHARDSON, L.; RUBY, S. **RESTful web services**. [S.l.]: "O'Reilly Media, Inc.", 2008.

RODOSAFRA. **Site oficial da RodoSafra**. 2019. Disponível em: <http://www.rodosafratransportes.com.br>. Online. Acessado em 09 dez. 2019.

SPROTT, D.; WILKES, L. Understanding service-oriented architecture. **The Architecture Journal**, MSDN Library, Microsoft Corporation, v. 1, n. 1, p. 10–17, 2004.

WIKIPÉDIA. **REST - Wikipédia, a enciclopédia livre**. 2020. Online; Acessado em 22 mai. 2020. Disponível em: <https://pt.wikipedia.org/w/index.php?title=REST&oldid=58326450>.