UNIVERSIDADE FEDERAL DO PAMPA

EDUARDO KLEIN PANDOLFO

DESENVOLVIMENTO DE UM WEB SERVICE REST PARA UM PROTÓTIPO DE APLICATIVO NO CONTEXTO PECUÁRIO

Bagé
2019

EDUARDO KLEIN PANDOLFO

DESENVOLVIMENTO DE UM WEB SERVICE REST PARA UM PROTÓTIPO DE APLICATIVO NO CONTEXTO PECUÁRIO

Projeto de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Sandro da Silva Camargo
Coorientador: Vinícius do Nascimento Lampert

Bagé
2019

Ficha catalográfica elaborada automaticamente com os dados fornecidos pelo(a) autor(a) através do Módulo de Biblioteca do Sistema GURI (Gestão Unificada de Recursos Institucionais).

K64 Pandolfo, Eduardo Klein

Desenvolvimento de um Web Service REST para um protótipo de aplicativo no contexto pecuário / Eduardo Klein Pandolfo.

75 p.

Projeto de Trabalho de Conclusão de Curso (Graduação) – Universidade Federal do Pampa, ENGENHARIA DE COMPUTAÇÃO, 2019.

"Orientação: Sandro da Silva Camargo; Coorientação: Vinícius do Nascimento Lampert".

1. Interoperabilidade. 2. Tecnologia da informação. 3. Gestão rural. 4. Empresa Brasileira de Pesquisa Agropecuária. 5. Dispositivos móveis. I. Título.

EDUARDO KLEIN PANDOLFO

DESENVOLVIMENTO DE UM WEB SERVICE REST PARA UM PROTÓTIPO DE APLICATIVO NO CONTEXTO PECUÁRIO

Projeto de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Projeto de Trabalho de Conclusão de Curso defendido e aprovado em: 26 de novembro de 2019.

Banca examinadora:

Prof. Dr. Sandro da Silva Camargo
Orientador

Profª. Drª. Sandra Dutra Piovesan
Universidade Federal do Pampa

Prof. Me. Carlos Michel Betemps
Universidade Federal do Pampa

RESUMO

A computação está em constante evolução e está disseminada em diversas áreas auxiliando pessoas em suas atividades diárias. Usar TI (Tecnologia da Informação) pode ser uma boa forma de tornar o cotidiano mais fácil, o trabalho mais produtivo e as decisões mais efetivas. No contexto das atividades de um produtor rural, a sua posição no mercado pode ser mais competitiva com a inserção de tecnologias para gestão em suas áreas de negócio como produção e comercialização. Uma pesquisa realizada pela Fundação Getúlio Vargas aponta que em 2018 existiam 220 milhões de smartphones ativos no Brasil, um número tão expressivo de aparelhos gera uma quantidade considerável de informações que precisam ser armazenadas. Um banco de dados é a solução mais comum para uma empresa armazenar informações relevantes a sua organização. Smartphones têm um grande potencial para auxiliar o pequeno, médio e grande produtor rural. Além da alta tecnologia, um smartphone tem a característica de mobilidade, permitindo que um produtor rural possa utilizá-lo em qualquer local de sua propriedade.

Desta forma, este trabalho tem como objetivo a criação e documentação de um WS (*Web Service*, que terá a função de criar uma comunicação padronizada entre um banco de dados e um aplicativo de gestão pecuária produzido pela Embrapa. O *Web Service*, em conjunto com o banco de dados, possibilitará que informações possam ser armazenadas para posteriormente serem lidas e compartilhadas entre usuários do aplicativo. Para o desenvolvimento do trabalho utilizou-se REST (*Representational State Transfer*) como modelo de arquitetura do WS.

Palavras-chave: Interoperabilidade. Tecnologia da informação. Gestão rural. Empresa Brasileira de Pesquisa Agropecuária. Dispositivos móveis.

ABSTRACT

Computing is constantly evolving and is widespread in many areas helping people in their daily activities. Using IT (Information Technology) can be a good way to make life easier, more productive work, and more effective decisions. In the context of a rural producer's activities, its market position may be more competitive with the insertion of technologies for management in its business areas such as production and marketing. A survey conducted by the Getúlio Vargas Foundation points out that in 2018 there were 220 million active smartphones in Brazil, such an expressive number of devices generates a considerable amount of information that needs to be stored. A database is the most common solution for a company to store information relevant to its organization. Smartphones has great potential to assist the small, medium and large rural producer. In addition to high technology, a smartphone has the characteristic of mobility, a farmer can use it anywhere in his property.

Therefore, this work aims to create and document a WS (Web Service), which will have the function of creating a standardized communication between a database and a livestock management application produced by Embrapa. The proposed Web Service, combnined with the database, will enable that information can be stored so it can be read and shared later on between users of the app. For the development of the work REST (Representational State Transfer) was used as architecture model of the WS.

Keywords: Interoperability. Information technology. Rural management. Brazilian Agricultural Research Corporation. Mobile devices.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface* (Interface de Programação de Aplicativos)
CIENAGRO	Simpósio da Ciência do Agronegócio
DAO	*Data Access Object* (Objeto de acesso a dados)
DTO	*Data Transfer Object* (Objeto de Transferência de Dados)
EMBRAPA	Empresa Brasileira de Pesquisa Agropecuária
ER	Entidade Relacionamento
HTML	*Hypertext Markup Language* (Linguagem de Marcação de Hipertexto)
HTTP	*Hyper Text Transfer Protocol* (Protocolo de Transferência de Hipertexto)
IBGE	Instituto Brasileiro de Geografia e Estatística
IES	Instituição de Ensino Superior
IFSul	Instituto Federal Sul-rio-Grandense
JMS	*Java Message Service* (Serviço de Mensagens Java)
JPA	*Java Persistence API* (API de Persistência Java)
JSON	*JavaScript Object Notation* (Notação de Objeto JavaScript)
ORM	*Object-Relational Mapping* (Mapeamento Objeto Relacional)
OD	Origens-Destinos
POM	*Project Object Model* (Modelo de Objeto do Projeto)
REST	*Representational State Transfer* (Transferência de Estado Representacional)
ROA	*Resource Oriented Archicteture* (Arquitetura Orientada a Recursos)
RPC	*Remote Procedure Call* (Chamadas de Processamento Remoto)
SBIAgro	Associação Brasileira de Agroinformática
SciELO	*Scientific Electronic Library* Online (Biblioteca Eletrônica Científica Online)

SGBD	Sistema Gerenciador de Banco de Dados
SMTP	*Simple Mail Transfer Protocol* (Protocolo Simples de Transferência de Correio)
SOA	*Service Oriented Architecture* (Arquitetura Orientada a Serviços)
SOAP	*Simple Object Acess Protocol* (Protocolo Simples de Acesso a Objetos)
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
TI	Tecnologia da Informação
TIC	Tecnologia da Informação e Comunicação
UFRGS	Universidade Federal do Rio Grande do Sul
UNIPAMPA	Universidade Federal do Pampa
URI	*Uniform Resource Identifier* (Identificador de Recurso Uniforme)
UTFPR	Universidade Tecnológica Federal do Paraná
WS	*Web Service* (Serviço Web)
WSDL	*Web Services Description Language* (Linguagem de Descrição de Serviços Web)
W3C	*World Wide Web Consortium* (Consórcio World Wide Web)
XML	*eXtensible Markup Language* (Linguagem de Marcação Extensível)

SUMÁRIO

1 INTRODUÇÃO

Atualmente a evolução das TICs (Tecnologias da Informação e Comunicação) pode ser evidenciada pela popularização de smartphones. Esses aparelhos oferecem funcionalidades que vão além da possibilidade de fazer ligações, contando com outros recursos como: captura de áudio e imagens, downloads e instalação de aplicativos e acesso à internet. Um estudo realizado pelo IBGE (Instituto Brasileiro de Geografia e Estatística), em 2017, concluiu que o telefone móvel estava presente em 93,2% dos domicílios e 78,2% dos brasileiros (com mais de 10 anos) possuíam smartphones para uso próprio (IBGE, 2018).

Na área de desenvolvimento de aplicações em smartphones, uma barreira que pode ser identificada é a compatibilidade que a aplicação desenvolvida deve ter com plataformas (Android, iOS, Windows) e outras aplicações disponíveis no mercado. Como citado por Anacleto (2012), um problema encontrado no desenvolvimento de aplicações é a dificuldade de desenvolver aplicações para diversas plataformas quando há necessidade de aumentar o alcance do produto para seu público-alvo. Dificuldades como essa levam ao desenvolvimento de soluções como os *Web Services*, que permitem a troca de dados entre plataformas diferentes em um sistema heterogêneo através da internet. Entende-se sistema heterogêneo como um conjunto de sistemas de informação com finalidades próprias e desenvolvidos com tecnologias diferentes, que dificultam ou inibem uma comunicação direta entre esses sistemas.

A utilização de tecnologia de informação no setor rural permite equilibrar algumas desvantagens econômicas, reduzindo as barreiras de tempo e de distância dos principais mercados e possibilitando uma reestruturação das organizações por interligar pessoas, processos e empresas (MACHADO; NANTES, 2011). Os autores Hollifield e Donnermeyer (2003) indicam que a velocidade de acesso a informações é um fator significativo na competitividade dos negócios e que um atraso na adoção de novas TICs podem trazer desvantagens competitivas.

No trabalho de HOFER et al. (2011), os autores concluem que a maioria dos pequenos e médios produtores da região oeste do Paraná gerencia os processos produtivos e atividades informalmente através de anotações escritas em papel. BRUM (2019) complementa que estas atividades dificultam a execução de processos analíticos destas informações, devido ao trabalho adicional de inserção dos dados em computador e a potencial inconsistência dos dados pelo método de coleta. Em pesquisa feita pela EMBRAPA (2018),

fundamenta-se que os dois principais pontos de prioridade máxima do produtor pecuário na gestão e organização da propriedade rural são os custos de produção e a capacitação de recursos humanos e suporte técnico.

Dentro dos projetos que a Embrapa Pecuária Sul está atualmente desenvolvendo, há o *Mybeef* (EMBRAPA, 2015). Este projeto abrange a região Sul e Centro-Oeste do Brasil e visa a disseminação de conteúdo sobre a cadeia produtiva pecuária para pesquisadores e produtores rurais. Além da Embrapa, o projeto conta com contribuições de outras IES (Instituições de Ensino Superior) como a Unipampa (Universidade Federal do Pampa), UFRGS (Universidade Federal do Rio Grande do Sul) e IFSul (Instituto Federal Sul-rio-Grandense). As atividades abrangem os cursos de Engenharia de Computação, Engenharia de Produção, Zootecnia e Sistemas de Informação.

Este trabalho prevê a modelagem, o desenvolvimento e a documentação de um WS para um modelo lógico de banco de dados, sendo integrante do projeto da plataforma *MyBeef* de título: Desenvolvimento de Sistemas de Apoio à Decisão e de Métodos de Coleta, Análise de Dados e Monitoramento da Pecuária na Região Sul do Brasil. O trabalho de conclusão proposto espera auxiliar o desenvolvimento de aplicações dentro da plataforma *MyBeef* de forma a ajudar com que o projeto cumpra seus objetivos.

1.1 Justificativa

Para que o projeto *Mybeef* consiga obter dados relevantes sobre propriedades rurais da região sul do Brasil, o WS precisa realizar a intermediação de dados entre aplicativos móveis e o banco de dados da Embrapa. Com isso, o produtor rural poderá ter acesso a uma solução tecnológica para gestão de sua propriedade rural e a Embrapa poderá aumentar seu acervo de informações sobre a pecuária no sul do Brasil.

1.2 Problema de Pesquisa

Em um cenário onde uma organização possui diversos sistemas de informação com finalidades e linguagens de programação diferentes, que ficam isolados por não terem compatibilidade de comunicação com os outros sistemas. Nesse contexto, uma solução é a implementação de um padrão de comunicação através de um *Web Service*. O desenvolvimento de um WS pode estabelecer uma conexão de dados entre sistemas que

contêm finalidades e tecnologias diferentes, possibilitando assim que sistemas tenham comunicação direta entre si.

1.3 Objetivos

Nesta sessão serão abordados o objetivo geral e os objetivos específicos deste trabalho.

1.3.1 Objetivo Geral

Desenvolvimento de uma solução *Web Service* para permitir o intercâmbio de dados entre sistemas heterogêneos para aplicações desenvolvidas pela Embrapa Pecuária Sul.

1.3.2 Objetivos Específicos

- Identificar, estudar e comparar as tecnologias de desenvolvimentos de *Web Services* para intercâmbio de dados.
- Implementar uma solução adequada ao estudo das tecnologias.
- Realizar os testes dos recursos disponibilizados e a validação da solução através de aplicativos desenvolvidos pela Embrapa Pecuária Sul.
- Fazer a distribuição da solução proposta em um ambiente de produção da Embrapa Pecuária Sul.

1.4 Organização do Texto

O texto foi organizado em 6 capítulos, sendo o primeiro responsável pela introdução. No capítulo 2, serão apresentados os conceitos de API, *Web Services*, banco de dados e demais tecnologias e ferramentas envolvidas neste trabalho. No capítulo 3, é exposta a metodologia proposta para o desenvolvimento do trabalho. No capítulo 4, foi analisado o modelo do banco de dados utilizado e descrito como ocorreu o desenvolvimento da modelagem e do código do *Web Service*. No capítulo 5 são mostrados os testes do *Web*

Service. E, por último, no capítulo 6 são feitas as considerações finais.

2 REFERENCIAL TEÓRICO

Neste capítulo será apresentado o referencial teórico deste trabalho. As duas primeiras seções apresentam de forma gerais os conceitos de API e *Web Services*. Na seção 3, são expostos os conceitos sobre sobre REST e sua ligação com protocolo HTTP. Na seção 4, é explicado o que são os recursos em REST. Na seção 5, os conceitos de SOAP são apresentados e comparados com REST. Na seção 6, os conceitos de banco de dados são definidos. Na seção 7, são descritos os trabalhos correlatos. E na seção 8, são expostas as tecnologias.

2.1 API

Uma API (*Application Programming Interface*) é uma interface de comunicação de aplicações heterogêneas, que disponibiliza uma linguagem comum entre as aplicações. Vukovic et al. (2016), resumidamente, definem APIs como especificações que governam a interoperabilidade entre aplicações e serviços. Guillaud (2011) define que a função da API é permitir que dois programas consigam trocar dados. Uma das utilidades desse modelo de interface é prover acesso a serviços, como um banco de dados, sem que haja preocupação de implementação da lógica do banco de dados no usuário que necessita acessar esses dados.

Segundo HEFFNER (2014), existem três cenários distintos para uma organização ser provedora de APIs:

1. Uso interno com finalidade de aumentar agilidade nos negócios e engajamentos dos clientes.

2. Negócios com outras organizações para otimizar os relacionamentos e os processos visando garantir maior qualidade de serviços.

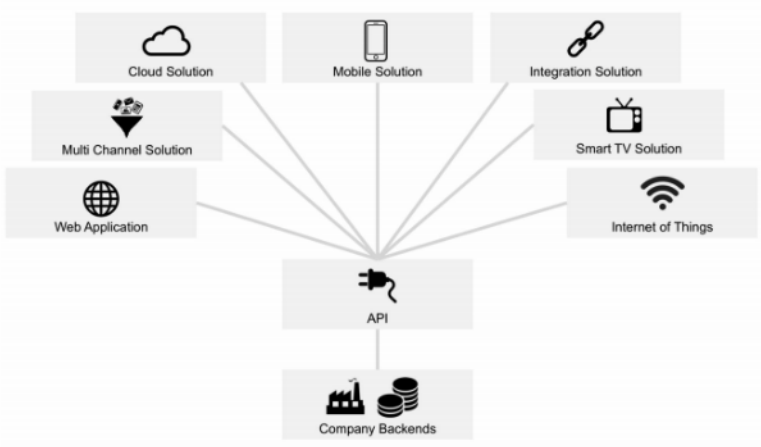
3. Disponibilização de APIs via web para que desenvolvedores possam acessá-las.

Grandes empresas como a Google e o Facebook disponibilizam uma API para realização de logins de usuários. Atualmente, é comum empresas oferecerem a oportunidade de um usuário se cadastrar em suas plataformas usando os dados de login do Facebook ou do Gmail. A Google também disponibiliza uma API para oferecer serviços de localização do *Google Maps* onde o detentor de um site consegue demonstrar a localização da

sua empresa em seu site, podendo utilizar esse recurso também na pagina de sua empresa do Facebook.

Biehl (2016) constata que normalmente uma solução típica de internet consiste em: aplicativos, APIs e sistemas de *back-end*. Os usuários finais utilizam os aplicativos que se aproveitam dos serviços de APIs para acessarem dados de sistemas *back-ends*. Vale notar que: o usuário final não é quem invoca a chamada na API, isso é feito pelo aplicativo. Logo, o cliente de uma API é o aplicativo que usa de seus serviços e não o usuário final que os desfruta. O autor exemplifica diferentes tipos de aplicações que podem fazer o uso de uma implementação de API para se conectarem a um banco de dados, neste caso, no banco estão centralizados os dados destas aplicações, os exemplos podem ser visualizados na Figura 1.

Figura 1 – Exemplos de soluções baseadas em API.



Fonte: Biehl (2016).

Pensando no *design* de uma API é necessário prestar atenção nas decisões que serão tomadas para o desenvolvimento. Um bom planejamento e processo de desenvolvimento induz em uma melhor qualidade do produto final. Biehl (2016) descreve 4 formas de analises que devem ser consideradas ao tomar uma decisão no desenvolvimento de uma API:

1. *Design* de Arquitetura: Analisar os padrões que serão usados para a comunicação entre os sistemas de informação, como formatos de mensagens e protocolos de comunicação.

2. *Design* de *Front-end*: Quais os parâmetros devem ser passados? E como os parâmetros devem ser passados? Essas decisões são importantes para o sucesso da API,

porque estão totalmente ligadas na sua usabilidade pelos usuários (desenvolvedores).

3. *Design* de *Back-end*: Como será feita a conexão com o *back-end*? Validar informações antes de enviar para o *back-end*? Tratar erros e exceções?

4. *Design* de Requisitos não Funcionais: Decisões sobre requisitos como: segurança, evolutividade, desempenho, etc.

Vaz et al. (2017) concluem em seu trabalho que o uso de APIs traz benefícios externos e internos para uma empresa, como por exemplo: viabilizar um maior alcance dos resultados adquiridos pela empresa detentora da API para seus parceiros e economia de recursos computacionais, com o compartilhamento de dados e serviços.

2.2 Web Services

Uma API disponibilizada na internet é popularmente conhecida como *Web Service* e atualmente sua existência é de extrema importância para a integração de aplicações baseadas na internet. Seu diferencial é utilizar a internet como meio de transporte de informações para conseguir conectar aplicações em qualquer lugar do mundo. Segundo W3C (2004), sistemas interagem com *Web Services* a partir de protocolos como HTTP (*Hyper Text Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*), JMS (*Java Message Service*) e formatos de mensagem como XML (*eXtensible Markup Language*) e JSON (*JavaScript Object Notation*). O objetivo de um *Web Service* é justamente disponibilizar uma plataforma de comunicação independente de *hardware* e linguagem utilizada no código programado (KREGER, 2001).

Eric Newcomer (2002) explica que *Web Services* trabalham com um nível de abstração similar a internet e são capazes de criar pontes entre qualquer sistema operacional. O autor complementa que a introdução dessa tecnologia mudou os mercados de venda de *hardwares* e *softwares*, porque com *Web Services* aplicações podem ser construídas reutilizando uma combinação de componentes disponibilizados na internet por múltiplos fornecedores. Sommervile (2007) esclarece que o reuso de *software* é indicado durante processos de desenvolvimento e manutenção, tal ação oferece benefícios como: redução do *time to market* e melhor qualidade e confiabilidade do produto final.

Um conceito apresentado muito recorrente com *Web Services* é SOA (*Service Oriented Architecture*), que tem como proposta a criação de uma aplicação a partir de vários

serviços. Segundo Joseph Bih (2006), SOA é uma coleção de serviços que comunicam através de uma troca simples de dados, ou uma coleção de serviços que juntas coordenam uma atividade, exigindo meios de conexão uns aos outros. Um serviço precisa ser uma função bem definida, independente e que não esteja atrelada a um contexto ou estados de outros serviços (BIH, 2006). Basicamente, a implementação de um SOA é um *Web Service*. Similar ao SOA, tem-se o ROA (*Resource Oriented Archicteture*): onde um recurso é qualquer componente ou entidade que será diretamente acessado ou representado na arquitetura. Todos os recursos em ROA são acessados pela interface comum do HTTP (LUCCHI; MILLOT, 2008).

Uma estrutura básica de SOA é similar a uma estrutura de comunicação cliente-servidor, sua comunicação é representada na Figura 2, ela é formada por um solicitador de serviço e um provedor do respectivo serviço. O solicitador envia um pedido para o provedor que o retorna com uma mensagem de resposta, os formatos das mensagens de pedido e resposta são definidos pelo provedor do serviço. Em algum momento nesta estrutura o provedor do serviço pode também se comportar como solicitador.

Figura 2 – Estrutura SOA.

Fonte: Adaptado de Joseph Bih (2006).

2.3 REST

Apresentado por Roy Fielding (2000) em sua tese de doutorado, REST é um modelo para projetar arquiteturas de *software* distribuído, como a *World Wide Web*. O modelo se baseia totalmente na forma atual de comunicação do protocolo de camada de aplicação HTTP. Fielding e Taylor (2002) explicam que REST é um conjunto de restrições que servem para minimizar a latência da comunicação de rede enquanto maximiza a independência e escalabilidade da implementação de componentes. Para contribuir com esses

benefícios, REST permite estratégias como reutilizar interações ao armazenar dados em cache e processamento de ações por intermediários.

Roy Fielding (2000), ao descrever a arquitetura REST a derivou em 6 principais conceitos:

1. Cliente-Servidor: É o estilo atual de arquitetura para aplicações baseadas na *web*. Sua base é composta por uma entidade que está disposta como um servidor que oferece um leque de serviços para serem solicitados e uma entidade cliente que dependendo do serviço desejado, envia requisições para o servidor. O servidor é capaz de aceitar ou rejeitar as chamadas, porém sempre é enviada uma resposta ao cliente.

2. Comunicação Sem Estado: O servidor não tem controle nenhum sobre o estado do cliente e vice-versa, o controle das ações fica por parte do cliente. As requisições ao servidor devem ser independentes de requisições anteriores. Tal restrição evita a necessidade de um sistema de monitoramento complexo, a leitura dos dados da requisição já é necessária para entender o contexto da solicitação. Também contribui para a confiabilidade porque a tarefa de recuperação de contexto é facilitada (WALDO et al., 1994). Como resultado, a escalabilidade é aumentada porque as requisições são independentes e com isso os recursos podem ser liberados diretamente após a utilização.

3. Cache: Tal conceito apresenta a possibilidade implícita ou explícita que dados sejam rotulados como armazenáveis em cache ou não. O cliente do serviço terá o direito de guardar o resultado de uma requisição para poder ser reusada posteriormente, podendo eliminar interações cliente-servidor futuras. Melhorando a eficiência, escalabilidade e desempenho percebido pelo usuário. Pautasso (2009), explica que REST usa os metadados das requisições para controlar propriedades da interação entre cliente e servidor como autenticação, compressão de dados e controle de cache.

4. Sistemas em camadas: Este conceito determina que o sistema deve ser dividido em camadas de forma hierárquica. A visão do sistema pelo ponto de vista de um componente é restringida e determinante na forma como o sistema deve funcionar: um componente só deve ter acesso as camadas nas quais ele interage. Com o sistema encapsulado pode-se criar políticas de segurança mais robustas, mas há desvantagens como a noção de uma latência maior por parte do usuário, devido

as constantes manipulações de dados, criando um *overhead* nesse tipo de sistema (CLARK; TENNENHOUSE, 1990).

5. Interface Uniforme: Os recursos da arquitetura devem ser acessados utilizando um conjunto fixo de operações do verbo HTTP: GET, PUT, POST e DELETE. Dentro do REST não ha restrição no formato dos dados, os dados podem ser representados, por exemplo, em HTML (*Hypertext Markup Language*), XML, JSON, etc. Generalizando os métodos de acessos aos recursos pode-se simplificar a interface e deixando sua utilização mais intuitiva, cada recurso também é disponibilizado através de um URI (Identificador de Recurso Uniforme).

6. Código sob demanda: No estilo do código sob demanda, um cliente pode enviar uma requisição para um servidor remoto para ter conhecimento de como processar um dado (FUGGETTA; PICCO; VIGNA, 1998). A interface da arquitetura pode auxiliar o cliente na utilização da própria arquitetura, estendendo a função de transferir parte da lógica ao cliente, por exemplo, disponibilizando endereços de recursos similares na resposta de uma requisição.

2.3.1 Mensagens HTTP

Como já dito anteriormente, REST é baseado no protocolo HTTP para realizar a comunicação entre o provedor e utilizador do serviço. HTTP é o protocolo atual utilizado pela *World Wide Web*, sua especificação está publicada na RFC 7540[1]. O protocolo é quem define a estrutura da mensagem entre cliente e servidor, basicamente as mensagens podem ser divididas em: requisição e resposta. Uma mensagem de requisição contém uma linha de requisição, linhas de cabeçalho e o corpo da mensagem. A mensagem de resposta contém o código de estado, linhas de cabeçalho e o corpo da mensagem.

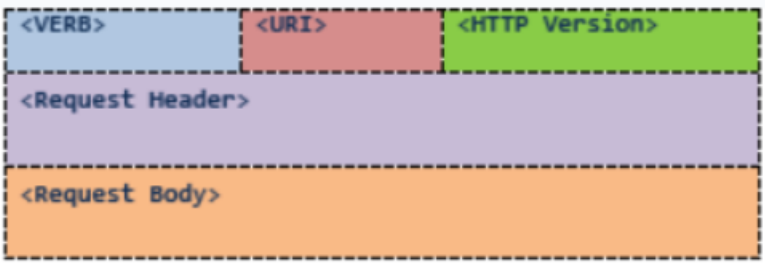
2.3.1.1 Requisição

Um exemplo de formato requisição pode ser visto na Figura 3. Já a Figura 4 contém um exemplo de requisição POST.

Verbo (*Verb*): Deve conter algum método válido HTTP como: GET, POST, PUT, DELETE, HEAD, OPTIONS.

[1] https://tools.ietf.org/html/rfc7540

Figura 3 – Formato de Mensagem de Requisição HTTP.



Fonte: Vaqqas (2014).

Identificador de Recurso Uniforme (URI): Link do recurso no qual deve ser processada a requisição.

Versão HTTP (*HTTP Version*): Versão do HTTP utilizada pelo navegador do usuário para que a resposta seja enviada com uma versão compatível ao usuário.

Cabeçalho de Solicitação (*Request Header*): Contém metadados referentes a mensagem e seu remetente, formatos de mensagens aceitos, versão do navegador utilizado, formato do corpo e etc.

Corpo de Solicitação (*Request Body*): É o local onde estão contidos os dados, no caso do REST é onde devem estar as representações dos recursos.

Figura 4 – Exemplo de Requisição POST com XML.

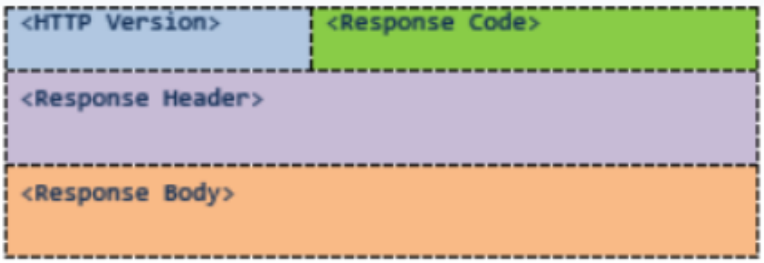
```
POST http://MyService/Person/
Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

Fonte: Vaqqas (2014).

2.3.1.2 Resposta

O formato de resposta de uma mensagem HTTP pode ser visto na Figura 5. Já a Figura 6 contém um exemplo resposta.

Figura 5 – Formato de resposta HTTP.



Fonte: Vaqqas (2014).

Código de Resposta (*Response Code*): O HTTP envia um código de 3 dígitos referentes ao estado da solicitação.

Cabeçalho de Resposta (*Response Header*): Contém os metadados e configurações da resposta.

Corpo de Resposta (*Response Body*): Contém a representação do recurso se a requisição for válida.

Figura 6 – Exemplo de resposta HTTP.

```
HTTP/1.1 200 OK
Date: Sat, 23 Aug 2014 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
Cache-Control: max-age=21600, must-revalidate
Expires: Sun, 24 Aug 2014 00:31:04 GMT
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns='http://www.w3.org/1999/xhtml'>
<head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head>
<body>
```

Fonte: Vaqqas (2014).

2.3.1.3 Verbos

Para uma implementação de uma REST API ter sucesso é necessário compreender muito bem os verbos HTTP e saber identificar quais operações serão implementadas para cada recurso que será disponibilizado. Um termo aliado aos verbos HTTP é o de idempotência: É uma propriedade de operações que podem ser aplicadas várias vezes sem que o valor do resultado se altere após a aplicação inicial. Por exemplo: Uma operação GET em um recurso não irá alterar o estado do recurso. Inúmeras requisições sucessivas po-

dem ser feitas e o recurso continuará no mesmo estado. A Tabela 1 mostra os principais verbos HTTP seguidos de uma descrição da sua função e se contém ou não a propriedade de idempotência, os 4 primeiros métodos são os mais comuns pois suas operações estão diretamente ligadas a manipulação de recursos.

Tabela 1 – Principais verbos HTTP.

Verbo	Descrição	Idempotente
GET	Solicita um recurso	Sim
POST	Cria um recurso	Não
PUT	Atualiza um recurso	Sim
DELETE	Deleta um recurso	Sim
HEAD	Solicita metadados do recurso	Sim
OPTIONS	Retorna métodos disponíveis para o recurso	Sim

Fonte: Autor (2019)

2.3.1.4 Códigos

Os códigos HTTP, também chamados de códigos de retorno ou estado, são usados em uma resposta HTTP com intuito de informar ao usuário se a requisição foi devidamente concluída. Uma REST API deve respeitar esse padrão para informar ao solicitador o estado mais correto da resposta. Matthias Biehl (2016) explica que os códigos contém 3 dígitos e complementa que o primeiro digito é responsável por dividi-los em 5 grupos distintos: Informativo (1xx), Sucesso (2xx), Redireção (3xx), Erro no Cliente (4xx), Erro no Servidor (5xx). Para melhor visualização, os grupos foram descritos na Tabela 2 e posteriormente serão apresentados na Tabela 3 os códigos mais comuns.

Tabela 2 – Grupos de Códigos HTTP.

Grupo	Dígitos	Descrição
Informativo	1xx	Informações não criticas.
Sucesso	2xx	Requisição foi concluída com sucesso.
Redireção	3xx	Para a solicitação ser completa deve ser algo a mais.
Erro no Cliente	4xx	Ocorreu um erro com por parte do cliente na solicitação.
Erro no Servidor	5xx	Ocorreu um erro na solicitação por parte do servidor.

Fonte: adaptado de Biehl (2016).

Tabela 3 – Códigos HTTP comuns.

Código	Descrição	Significado
100	*Continue*	Tudo certo com a requisição até o momento.
200	*Ok*	Requisição feita com sucesso.
201	*Created*	Sucesso, recurso criado.
202	*Accepted*	Sucesso, requisição aceita.
204	*No Content*	Sucesso, sem dados para responder.
301	*Moved Permanently*	Recurso está disponível somente através de endereço.
400	*Bad Request*	Solicitação não pode ser compreendida pelo servidor.
401	*Unauthorized*	Credenciais de autenticação inválidas.
404	*Not Found*	Recurso não encontrado.
405	*Method Not Allowed*	Método HTTP não é permitido para este recurso.
500	*Internal Server Error*	Erro interno no Servidor.
501	*Not Implemented*	Funcionalidade não implementada.
503	*Service Unavailable*	Servidor não está disponível no momento.

Fonte: Autor (2019)

2.4 Representação de Recursos REST

Para constituir uma conexão válida entre usuário solicitador e provedor do serviço é utilizado o HTTP. Entretanto, ainda precisamos definir o formato das mensagens contidas nos corpos de solicitação e resposta para que os dados consigam ser interpretados e consequentemente utilizados pelos dois lados da comunicação (como mostrados na Figura 3 e Figura 5, respectivamente). Atualmente os formatos utilizados mais comuns são XML e JSON. Sendo JSON considerado um formato leve, por ser mais reduzido em relação ao XML.

2.4.1 XML

A XML (*Extensible Markup Language*) é uma linguagem de marcação extensível para padronizar uma sequência de dados. Segundo a W3C (2008), detentora da linguagem XML, um dos propósitos da XML é auxiliar a comunicação de compartilhamento de dados entre diferentes sistemas de informações. Sua sintaxe é perfeitamente adequada para descrever dados semiestruturados (ABITEBOUL; BUNEMAN; SUCIU, 2000). Para fins de demonstração do formato XML, criou-se a Figura 7, onde é mostrado uma notação de um vetor de entidades do tipo Aluno com os seguintes campos: nome, sobrenome e

idade.

Figura 7 – Representação de um dado em XML.

```
<alunos>
  <aluno>
      <nome>Eduardo</nome><sobrenome>Pandolfo</sobrenome><idade>20</idade>
  </aluno>
  <aluno>
      <nome>José</nome><sobrenome>Moreira</sobrenome><idade>22</idade>
  </aluno>
  <aluno>
      <nome>Flávio</nome><sobrenome>Santos</sobrenome><idade>19</idade>
  </aluno>
</alunos>
```

Fonte: Autor (2019)

2.4.2 JSON

JSON (*JavaScript Object Notation*) é um formato de troca de dados, leve, baseado em texto e independe de idioma (BRAY; ED, 2017). É um formato simples e que pode ser compreendido facilmente por maquinas e humanos. Embora o nome indique que seja *JavaScript*, este formato não se limita para transações compatíveis somente com essa linguagem. Para fins de comparação foi criado um vetor similar ao da Figura 7, só que agora representando os mesmos dados no formato JSON, pode-se concluir ao analisar Figura 8, que o JSON tem uma uma carga útil maior em relação ao XML. Carga útil se refere aos dados de uma mensagem, onde são excluídas as informações que são utilizadas para a entrega desta mensagem.

Figura 8 – Representação de um vetor em JSON.

```
{"alunos":[
      {"nome": "Eduardo", "sobrenome": "Pandolfo", "idade": 20},
      {"nome": "José", "sobrenome": "Moreira", "idade": 22 },
      {"nome": "Flávio", "sobrenome": "Santos", "idade": 19 }
]}
```

Fonte: Autor (2019)

2.5 SOAP vs REST

Nesta seção será apresentada a tecnologia SOAP e em seguida será feito uma comparação entre as tecnologias SOAP e REST a fim de elucidar as diferenças entre as tecnologias atuais disponíveis para o desenvolvimento de um *Web Service*.

SOAP (*Simple Object Acess Protocol*) é um protocolo que tem a função de definir um padrão para troca de informações estruturadas dentro do contexto de *Web Services*. SOAP utiliza XML como formato de mensagens e pode funcionar com diferentes protocolos de camada de aplicação. Segundo a W3C (2000), o protocolo SOAP não define qualquer semântica de aplicação (como modelo de programação), mas define um mecanismo simples para expressar um formato de dados dentro de módulos.

SOAP é baseado em RPC (*Remote Procedure Call*), sendo que esta tecnologia possibilita que uma aplicação solicite um processamento remoto de um método ou objeto por outra aplicação. Dantas (2007) constata que RPC representa um problema de compatibilidade no contexto de segurança: *Firewalls* e servidores *proxy* tendem a bloquear este tipo de chamada. *Web Services* SOAP utilizam WSDL (*Web Services Description Language*) que são documentos XML que descrevem seus serviços, endereços, parâmetros das chamadas, tipos de dados e os formatos das respostas.

A comunicação deste protocolo é feita a partir de um envelope que é basicamente dividido em: cabeçalho e corpo. A estrutura básica de um envelope SOAP é mostrada na Figura 9. Quando utilizado com HTTP, o envelope SOAP é inserido no corpo das mensagens.

Figura 9 – Envelope SOAP básico.

Fonte: Autor (2019)

Envelope: Indica o inicio e o final do envelope, é o elemento raiz da mensagem SOAP.

Cabeçalho: Contém informações opcionais como os metadados da mensagem.

Corpo: Contém os dados efetivos da mensagem. Pode conter o elemento *fault* para carregar mensagens de erros para o remetente da solicitação.

O trabalho de Wagh e Thool (2012) apresenta um comparativo entre as tecnologia REST e SOAP dentro do contexto de *Web Services*. Os autores criaram em forma de tabela uma comparação entre as duas tecnologias, entre os pontos que foram comparados estão: Acoplamento de lógica entre servidor e cliente, viabilidade com *Wireless*, segurança, dentre outros. Estas comparações estão descritas na Tabela 4.

Tabela 4 – Comparação entre SOAP e REST.

SOAP	REST
Interação Cliente-Servidor é firmemente acoplada.	Interação Cliente-Servidor é fracamente acoplada.
SOAP contém conjuntos de ferramentas de desenvolvimento estabelecidos.	REST não tem padrão em ferramentas, mas sua interface é mais simples.
Mudança de serviços em um servidor requer considerável mudança no código por parte do cliente.	Mudanças de recursos em um servidor requer pouca, senão nenhuma, mudança no código por parte do cliente.
Não contém uma estrutura amigável para *Wireless*.	Contém uma estrutura amigável para *Wireless*.
Requisições SOAP utilizam o método POST e requerem um formato XML mais complexo (dificultando uso de cache).	Requisições utilizam os verbos HTTP e podem facilmente se aproveitar da utilização da cache.
Planejado para ambiente de sistemas de computação distribuídos.	Assume conexão ponto-a-ponto.
Embora SOAP tenha a extensão WS-Security não indica que é mais seguro que REST.	Pode utilizar o que está disponível em funções que podem ser construídas em HTTP (ou HTTPS).

Fonte: adaptado de Wagh e Thool (2012).

Por mais que as duas tecnologias tenham o mesmo objetivo de criar uma comunicação de dados entre sistemas de informação, depende do contexto para definir a mais adequada. SOAP foi planejado para um ambiente de sistemas de computação distribuídos, sua mensagem pode ser enviada para várias aplicações em uma solicitação. E REST, embora tenha somente conexão ponto-a-ponto com HTTP, é mais adequada para casos onde o tamanho dos dados e o processamento das mensagens são considerados críticos.

2.6 Banco de Dados

HEUSER (2009) define que banco de dados é um conjunto de dados integrados que tem como objetivo atender a uma comunidade de usuários. Tais dados necessitam ser gerenciados e para isso temos um SGBD (Sistema Gerenciador de Banco de Dados). Um SGBD é um *software* responsável por incorporar as funções de definição, recuperação e alteração de dados em um banco de dados e o modelo atual mais utilizado é o SGBD Relacional (HEUSER, 2009). Utilizar um sistema para gerenciar um banco de dados traz vantagens como: organização, segurança, integridade e garante um acesso facilitado aos dados.

O modelo relacional de banco de dados é baseado no agrupamento de tabelas e relações. Uma tabela é um conjunto de linhas de dados que representam entidades. Relações, como o nome já demonstra, são as conexões entre tabelas. Dentro dos exemplos de sistemas de gerenciamento de bancos de dados relacionais atuais temos: PostgreSQL, MySQL, Microsoft SQL, Oracle, etc.

Em um Banco de Dados Relacional é possível realizar funções como consulta e manipulação de dados, sendo a linguagem utilizada para tais ações a SQL (*Structured Query Language*). O comando para fazer uma consulta é o SELECT e para manipular os dados os comandos são INSERT, UPDATE e DELETE. Os comandos SQL citados quando englobados representam o notável conceito de manipulações CRUD (*Create*, *Read*, *Update* e *Delete*).

As funções de consulta e manipulação apresentadas podem ser facilmente mapeadas em verbos HTTP, a semântica dos comandos é bem definida e realizam as mesmas operações. A Tabela 5 reúne os verbos HTTP e os seus correspondentes em SQL e CRUD.

Tabela 5 – Relação de Comandos SQL, manipulações CRUD e verbos HTTP.

Comandos SQL	CRUD	Verbos HTTP
INSERT	*Create*	POST
SELECT	*Read/Retrieve*	GET
UPDATE	*Update*	PUT
DELETE	*Delete/Destroy*	DELETE

Fonte: Autor (2019)

2.7 Trabalhos Correlatos

Nesta seção serão apresentados trabalhos correlatos. Os trabalhos se relacionam no desenvolvimentos de APIs com pesquisas aplicadas, que não necessariamente utilizam a tecnologia REST no seu desenvolvimento.

Vaz et al. (2017) descrevem a plataforma da AgroAPI, que é uma iniciativa da Embrapa para promover a inovação tecnológica na Agricultura Digital por meio da divulgação de APIs. É citada uma API que provê acesso aos metadados de publicações da Embrapa e que auxilia no sistema de integração de informações internas. O trabalho apresenta o potencial do uso de APIs devidamente gerenciadas para a inovação tanto na empresa quanto na Agricultura Digital e trata de APIs tanto no contexto interno e externo: tratando como uma ferramenta para a integração de sistemas internos e como produto para prover acesso aos dados à empresas interessadas.

RODRIGUES (2015) em sua monografia desenvolveu uma API para exibir horários de transporte público para os acadêmicos da UTFPR (Universidade Tecnológica Federal do Paraná) em Londrina-PR. O autor utilizou REST como arquitetura de escolha para a API e descreve a elaboração do seu trabalho desde a análise de requisitos até a fase final do desenvolvimento. Além da API, também foi desenvolvido um aplicativo para consumir a API e disponibilizar seus serviços para os usuários finais.

O trabalho de KUHN (2012) apresenta o desenvolvimento de um protótipo de aplicativo para que usuários consigam se comunicar com taxistas para solicitarem seus serviços, o aplicativo provê serviço para compartilhamento de GPS e troca de mensagens. O objetivo do seu trabalho era automatizar a forma de como é feita a comunicação com as centrais de táxi, devido a alta demanda deste tipo de serviço. O trabalho apresenta uma abordagem para melhora deste serviço através do uso de smartphones, propondo um protótipo de aplicativo desenvolvido utilizando a plataforma Android em conjunto com Java (KUHN, 2012).

Em seu trabalho, Andry e Nicholson (2011) descrevem as motivações para criar uma aplicação que ofereça a médicos acesso aos registros de saúde de seus pacientes. A API descrita utiliza a metodologia REST em seu desenvolvimento e o aplicativo que irá utilizar os serviços da API tem compatibilidade somente com dispositivos iOS (iPhone e iPad). Os autores demonstram a importância da mobilidade de smartphones ou tablets no contexto de um ambiente hospitalar e exemplificam problemas como falta de espaço, problemas de segurança e custo para a implementação de computadores pessoais para

resolverem tal solução.

Ferreira (2015) apresenta em sua dissertação um estudo comparativo entre as tecnologias para o desenvolvimento de *WebServices* REST e SOAP. Na dissertação é apresentado testes comparativos de tempo de resposta e o tamanho das mensagens entre essas duas tecnologias. Também foram feitos testes de usabilidade do *Web Service* desenvolvido com usuários externos. Nos resultados a tecnologia REST se mostrou superior no tempo de resposta e tamanho das requisições.

2.8 Tecnologias

Nesta seção serão apresentadas as tecnologias utilizadas no desenvolvimento do trabalho.

2.8.1 Swagger

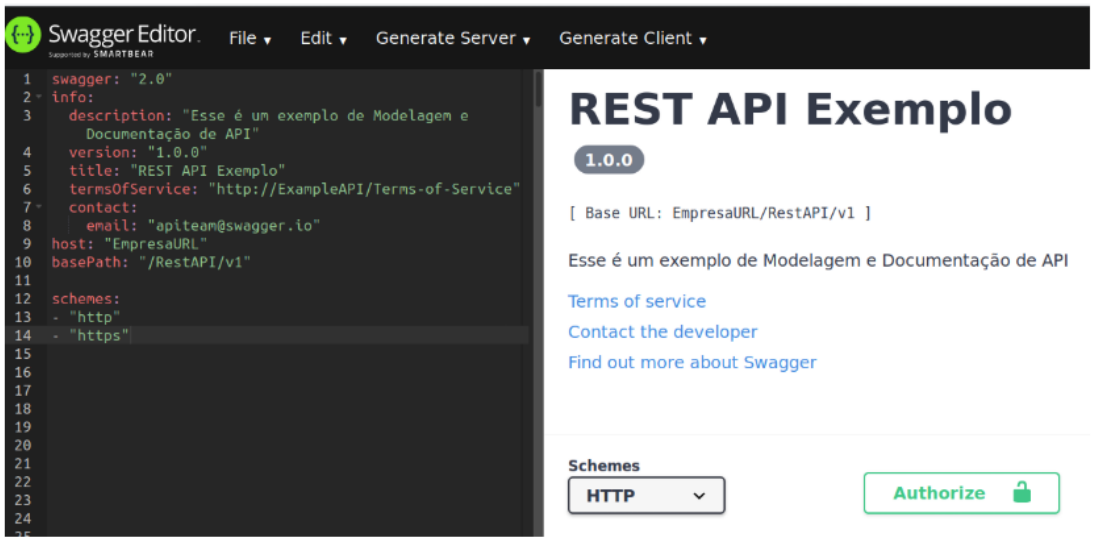
Open API Swagger [2] é um conjunto de ferramentas de código aberto que auxilia no *design*, desenvolvimento e documentação de interfaces de programação de aplicativo baseadas em REST. Surgiu com a necessidade de se utilizar ferramentas que sejam compatíveis com o paradigma proposto por REST. Dentre as ferramentas incluídas no Swagger estão: *Swagger Editor*, *Swagger UI*, *Swagger Codegen*. O primeiro é um editor online que o desenvolvedor pode utilizar para modelar e documentar a API. O segundo é um espaço para a publicação de documentação de APIs. E o ultimo é capaz de gerar os dados de cliente e servidor partindo da modelagem de uma API. O compilador do editor Swagger entende duas linguagens: yaml e JSON. A Figura 10 demonstra um exemplo de cabeçalho de um documento de uma API em yaml.

2.8.2 ORM

ORM (Mapeamento Objeto Relacional) explicita a forma como mapear objetos e os relacionar com os dados que os representam, é uma técnica focada em conectar aplicações a banco de dados para auxiliar nas consultas SQL. ORM surgiu para que o paradigma Orientado a Objetos consiga ser mapeado para o modelo objeto relacional

[2] https://swagger.io/docs/specification/about/

Figura 10 – Exemplo de um cabeçalho de documentação em yaml.



Fonte: Autor (2019)

de SGBDs. Com o mapeamento, é possível que desenvolvedores deixem a função para *frameworks* criarem os *scripts* de consultas de dados no banco, diminuindo assim o tempo de desenvolvimento de aplicações.

2.8.3 Hibernate

Produzido pela JBoss e desenvolvido em Java, o Hibernate [3] é um *framework* que implementa a técnica ORM, com ele podemos indicar que classes em java são tabelas no banco de dados e que os atributos dessas classes são colunas dessas tabelas. Este *framework* foi o responsável pela criação da JPA (*Java Persistence API*) que é a atual especificação de persistência de dados em Java.

Hibernate tem como responsabilidade a criação de códigos SQL em tempo de execução de acordo com as especificações da JPA. Para que o Hibernate consiga se comunicar com um banco de dados, inicialmente é preciso configurá-lo, porque: cada banco de dados possui um dialeto específico e também, dependendo do caso, pode ser necessário autenticação para obter acesso a algum banco.

A função da JPA é justamente padronizar a forma como é implementada a técnica ORM em Java para que diversos *frameworks* possam ser criados em cima desse formato de mapeamento. Na especificação, as classes que precisam ser persistidas devem ser

[3] https://hibernate.org/orm/

mapeadas a partir de notações, por exemplo: uma classe deve conter a notação @Entity e seus atributos devem conter a notação @Column. A JPA trabalha com o conceito de *Configuration By Exception*, isso quer dizer que caso não seja informado o nome da tabela na base de dados, é utilizado o nome da classe e assim também acontece com os atributos da classe (KUHN, 2012). Algumas notações que JPA padroniza para mapeamento de entidades e colunas:

1. @*Id*: Indica que o atributo é chave primária.
2. @*Temporal(TemporalType.DATE)*: Indica que o atributo é do tipo data.
3. @*Column(nullable = true)*: Indica que a linha da tabela não é obrigatória.
4. @*OneToMany*: Indica relação de um-para-muitos.

Além das notações para mapeamento, JPA contém um gerenciador de entidades, o *EntityManager*. É com ele que são implementadas as operações de persistência, como:

1. *find* : Busca um elemento no banco e o transforma na classe mapeada com ORM;
2. *merge*: Envia os dados da classe para atualizar suas respectiva tabela no banco;
3. *remove*: Executa a exclusão de uma determinada entidade;
4. *persist*: Cria uma unidade de persistência na base de dados.

3 METODOLOGIA

Neste capítulo é apresentada a metodologia definida para o desenvolvimento da API. A seção 3.1 discute a caracterização da pesquisa e a seção 3.2 apresenta os procedimentos metodológicos, onde são descritas as etapas de desenvolvimento para a execução do trabalho.

3.1 Caracterização da Pesquisa

A pesquisa, quanto à abordagem, é quantitativa. Quantitativa pelo estudo de práticas e tecnologias consolidadas para a implementação dessa solução, tendo como resultado dados numéricos de recursos implementados e disponibilizados pela API para serem utilizados pelo aplicativo.

Quanto à natureza, a pesquisa é aplicada porque sua função é resolver um problema cotidiano. O objetivo da pesquisa é desenvolver um *Web Service* que resolva os problemas reais de interoperabilidade entre aplicações dentro do contexto de atividades da Embrapa Pecuária Sul.

Quanto ao objetivo, a pesquisa é caracterizada como descritiva pela exposição das tecnologias atuais para o desenvolvimento de *Web Services*. Foram estudados e comparados os modelos REST e SOAP, também foram descritas ferramentas e modelos de dados que são utilizados em conjunto para implementação de um *Web Service* REST.

Quanto aos procedimentos, a pesquisa será bibliográfica e experimental. Bibliográfica devido a seleção de informações que sustentem as decisões tomadas dentro do trabalho. E experimental devido a realização de testes dos recursos implementados do *Web Service* através de ferramentas disponíveis na internet.

Por fim, a pesquisa é classificada como quantitativa, aplicada, descritiva, bibliográfica e experimental. Ainda pode se dizer que terá cunho tecnológico, devido as ferramentas utilizadas. O procedimento metodológico concluirá mais alguns pontos referentes à pesquisa, tais como os processos para o desenvolvimento deste trabalho e os métodos de pesquisa científica.

3.2 Procedimento Metodológico

A metodologia proposta para o desenvolvimento foi dividida em etapas para que o objetivo geral e os objetivos específicos do trabalho possam ser alcançados. As etapas estão demonstradas na Figura 11 e posteriormente serão apresentadas com maiores detalhes.

Figura 11 – Etapas para elaboração do TCC.

Fonte: Autor (2019)

1. Delimitação do Problema: Antes de começar o desenvolvimento da solução é preciso delimitar seu escopo. O trabalho precisa disponibilizar uma solução tecnológica para acesso a banco de dados. Esta solução é responsabilizada por receber solicitações de um aplicativo, identificar, executar e por último, enviar uma resposta ao aplicativo. Resumindo, a API intermediará dados entre um aplicativo e um banco de dados.

2. Levantamento do Referencial Teórico: A escrita do referencial teórico consistirá de uma análise, cujo foco inclui os assuntos requeridos no desenvolvimento e que servem para o embasamento e melhor compreensão do trabalho como um todo. Foram definidas as seguintes palavras-chaves para as pesquisas de referenciais: interfaces de programação de aplicativos, *REST API*, pecuária, banco de dados, *Web Services*. Como fontes primárias da pesquisa, temos o CIENAGRO (Simpósio da

Ciência do Agronegócio), SciELO (*Scientific Electronic Library Online*) e Simpósio de Pecuária de Corte, e de organizações como a SBIAgro (Associação Brasileira de Agroinformática) e publicações da própria Embrapa dentro do contexto do trabalho. Outras fontes foram utilizadas para fornecer conhecimento de ferramentas, linguagens, padrões de programação e protocolos que estão envolvidos na parte de desenvolvimento da API.

3. Análise do Banco de Dados: Partindo para a prática, nesta etapa é realizado um estudo sobre o atual banco de dados utilizado pelo projeto *MyBeef*. O objetivo é a obtenção de conhecimento sobre as tabelas do atual *database*, seus atributos e as suas relações. O estudo foi feito a partir de mais de 20 tabelas divididas em dois esquemas de banco de dados: Um público (denominado *public*) que contém principalmente as tabelas de usuários, fazendas, *logs* de usuários e algumas constantes como categorias de animais, cidades, estados e países. O segundo é sobre a Agenda Campeira, um aplicativo que está sendo desenvolvido pela Embrapa e que utilizará a API produzida nesse trabalho para realizar o acesso ao banco de dados já citado. O conhecimento obtido com a análise será útil para as próxima etapas, porque facilitará a definição dos requisitos e a modelagem dos recursos, que serão utilizados para manipular as tabelas no banco.

4. Definição de Requisitos: A definição dos requisitos ocorreu através de reuniões com os orientadores do trabalho e supervisores da Embrapa. Como resultado das conversas, criou-se um documento de requisitos da *Web* API, com o objetivo de identificar e registrar exigências ela precisa cumprir. O documento teve o objetivo de nortear o processo de documentação e desenvolvimento durante a execução prática deste trabalho.

5. Modelagem dos Recursos: Esta etapa realizar-se-á pela modelagem dos recursos do *Web Service*. A modelagem terá o auxílio da ferramenta Swagger, a qual possibilitará obter uma documentação visível dos recursos, seus métodos e as tabelas do banco manipuladas por cada recurso. Esta ferramenta foi escolhida por ser *open-source* e por estar padronizada pela Embrapa para o desenvolvimento de *Web Services* REST. Com o documento devidamente analisado, parte-se para o desenvolvimento da API. Como resultado dessa etapa, foram modelados recursos para manipulação de cerca de 12 tabelas e recursos auxiliares para autenticação e sincronização.

6. Desenvolvimento da API: A API foi programada com a linguagem orientada a objetos Java. O desenvolvimento contempla a utilização de ferramentas e *frameworks* complementares para gerenciamento do projeto Java (Maven), mapeamento e persistência dos dados (Hibernate) e também utiliza padrões para transferências de objetos Java (DTO). Assim como a escolha do Swagger, a linguagem de programação e as ferramentas também foram requeridas por já estarem padronizadas pela Embrapa, o que facilitará também o processo de manutenção da API pela própria empresa. Como resultado dessa etapa, os recursos modelados na etapa anterior foram implementados.

7. Testes e Validação: Nesta etapa foram feitos os testes dos recursos desenvolvidos para validação do *Web Service*. Para isso, será necessário realizar o *upload* do WS em um servidor para deixá-lo disponível e conseguir utilizar ferramentas que simulam chamadas HTTP para o endereço dos recursos. Assim, possibilitando os testes dos recursos de forma eficaz sem a necessidade da criação de um aplicativo ou de um ambiente para testes mais robustos. No modelo de APIs proposto por REST, as solicitações devem ser independentes e, por isso, os testes foram feitos por recurso. A validação se concluirá através da comparação entre os recursos modelados e implementados.

8. Análise dos Resultados e Conclusões: Etapa de finalização do trabalho, que está dividida duas etapas: uma de avaliações das tecnologias e outra de conclusões apresentando possíveis continuações do trabalho.

4 IMPLEMENTAÇÃO DO WEB SERVICE

Neste capítulo será apresentado a elaboração da modelagem e do desenvolvimento do *Web Service*. Será abordado inicialmente a modelagem dos recursos (seção 4.1) apresentando o diagrama ER (Entidade Relacionamento) do banco de dados que foi utilizado. Posteriormente, é apresentado a modelagem feita com a ferramenta Swagger. Na seção (4.2) mostrará como foram implementadas as classes em Java para o WS.

4.1 Modelagem de Dados e Documentação

A modelagem dos dados de um banco de dados pode ser mostrada a partir de um diagrama ER, que é uma representação visual de como as informações estão armazenadas e organizadas em um banco de dados. Em outras palavras, o diagrama ER mostra como foram implementadas as tabelas e suas relações. Como estudo de caso, agora serão apresentados os diagramas do banco que terão os dados manipulados pelo *Web Service*.

4.1.1 Modelo de Dados

O SGBD utilizado pelo projeto é o PostgreSQL, um diferencial deste sistema é a representação de esquemas que apresenta um nível a mais na abstração de organização das informações em um banco de dados. Como exemplo, pode ser citado a organização do banco de dados que será utilizado para os testes do *Web Service* (Figura 12). O esquema de nome *public* é considerado o esquema central por ter informações de usuários e fazendas e outras tabelas de padronização de dados. Já o esquema denominado *agenda campeira* está totalmente relacionado a uma proposta de aplicativo da Embrapa para auxiliar os pecuaristas na gestão de sua(s) propriedade(s).

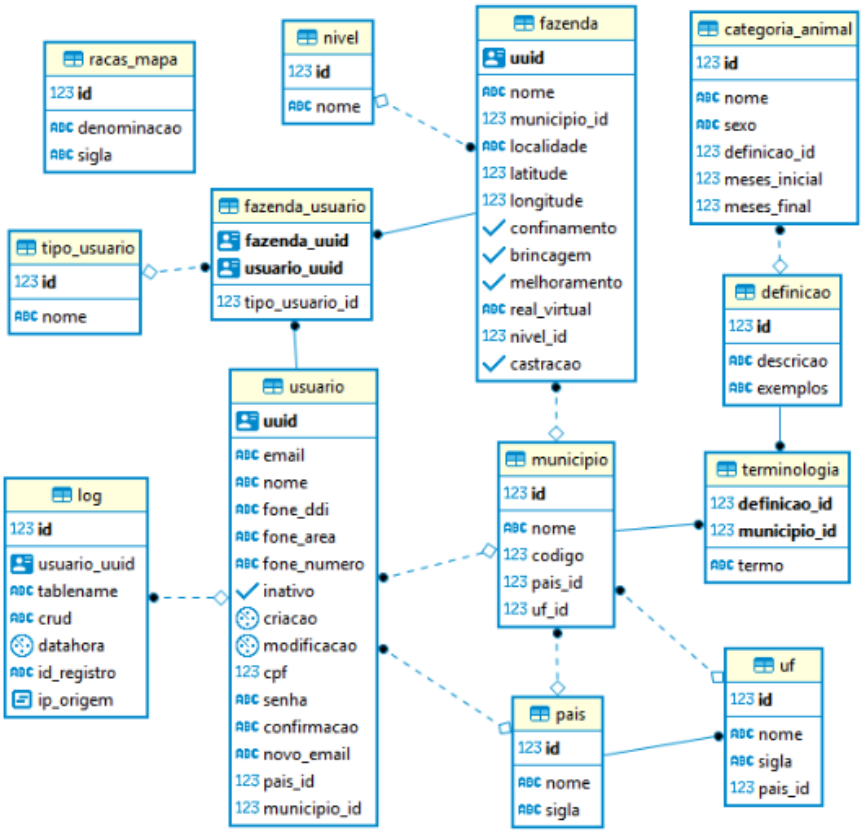
O esquema público contém as principais tabelas do banco de dados referentes a usuários e fazendas. A tabela que contém o registro de usuário, por exemplo, é requerida em qualquer aplicação desenvolvida no projeto *MyBeef* que necessite de login. O esquema também contém tabelas para padronização de registros como, por exemplo, as tabelas que representam cidades, estados, países e categorias de animais por raça. Assim, por ter os dados previamente cadastrados podem ser referenciadas dentro de qualquer aplicação, facilitando o processo de pesquisa e leitura dos dados dentro do banco. O

Figura 12 – Exemplificação dos esquemas do banco de dados.

Fonte: Autor (2019)

diagrama ER do esquema público pode ser visto na Figura 13.

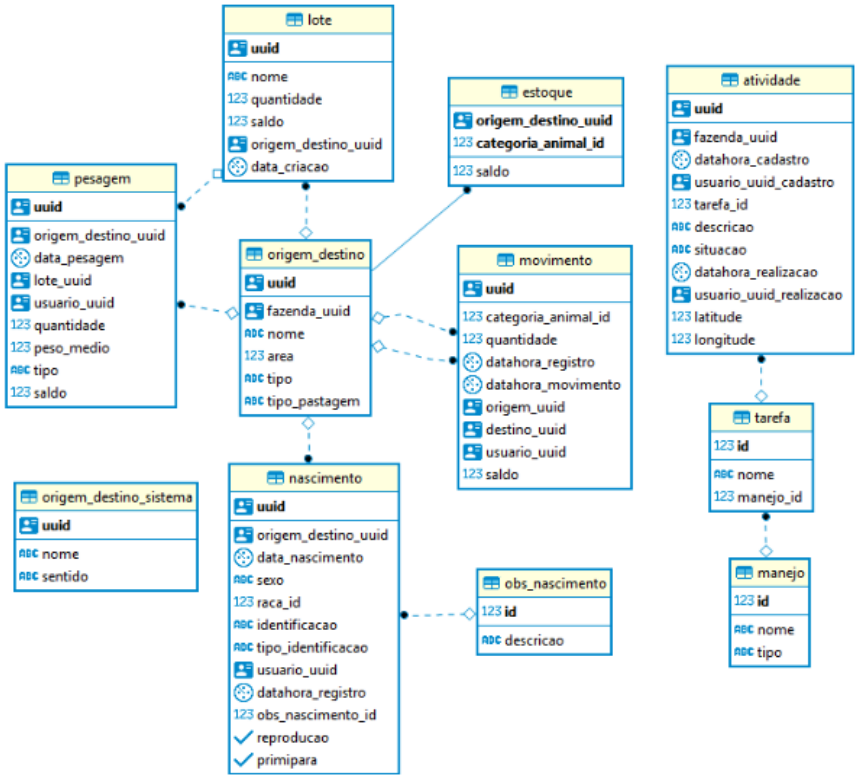
Figura 13 – Diagrama ER do esquema público.



Fonte: Autor (2019)

O esquema *agenda_campeira* é o esquema responsável por manter os dados do protótipo de um aplicativo que está sendo desenvolvido no projeto *MyBeef*. Com ele, é possível armazenar dados para a gestão de uma propriedade de um produtor pecuário. Para isso, o esquema conta com tabelas para armazenamento de dados de subdivisões da propriedade, nascimentos, lotes, estoques e pesagem de animais. Também disponibiliza o armazenamento de dados mais dinâmicos, como movimentação de animais entre subdivisões e atividades realizadas dentro de uma propriedade. O modelo e a forma como estão associadas as tabelas podem ser visualizados na Figura 14:

Figura 14 – Diagrama ER do esquema da agenda campeira.



Fonte: Autor (2019)

O último modelo apresentado contém dependências do esquema *public*, para sua total funcionalidade são exigidos registros das tabelas referentes a usuários e fazendas. Os lançamentos de dados neste esquema só poderão ser feitos após a criação de usuários e fazendas. Como o modelo idealizado será colocado em prática em um aplicativo, o usuário utilizador precisará realizar login para se autenticar na aplicação e, no mínimo, terá que conter uma fazenda cadastrada e associada ao seu identificador único na tabela *fazenda_usuario*, presente na Figura 13.

Após a análise do banco de dados, foi escrito um documento formalizando as

necessidades que devem ser atendidas pelo *Web Service*. A documentação servirá para identificar as exigências do projeto em forma de requisitos funcionais e não funcionais. Além da identificação das exigências, o documento servirá como registro de acordo das partes interessadas no desenvolvimento do trabalho. A documentação descrita nesse parágrafo se encontra no Apêndice A.

4.1.2 Documentação dos Recursos com Swagger

A ferramenta Open API Swagger, apresentada na seção 2.8.1, tornou-se uma opção viável para realizar a modelagem do *Web Service* por uma série de motivos, como podemos citar: é uma ferramenta *open-source*, intuitiva, de sintaxe simples e amigável, e por ser uma ferramenta que possibilita que a modelagem sirva como documentação do WS quando finalizada. Para realizar a modelagem, utilizou-se o editor online na versão 3.0. Existem duas linguagens que podem ser compreendidas pelo compilador do Swagger: JSON e yaml. Neste caso, a linguagem escolhida foi yaml pelo simples fato de ter um formato que deixará o código resultante com maior carga útil. Para elucidação, na Figura 15 as duas linguagens são comparadas.

Figura 15 – Comparação visual entre JSON e yaml.

yaml	JSON

schemes:
- https
paths:
/atividades:
post:
summary: 'Inclui uma atividade.'
description: 'Método HTTP para incluir informações no BD.'
	"schemes": [
	"https"
	],
	"paths": {
	"/atividades": {
	"post": {
	"summary": "Inclui uma atividade.",
	"description": "Método HTTP para incluir informações no BD.",
	"consumes": [
	"application/json"
	],

Fonte: Autor (2019)

Por convenção, algumas restrições foram impostas na modelagem para que o *Web Service* fique restrito apenas na função de realizar a comunicação entre o banco de dados e o aplicativo da agenda campeira. A seguir, a lista de restrições empregadas:

1. Por haver a documentação para auxiliar os desenvolvedores do aplicativo na utilização dos recursos do WS, somente os seguintes métodos foram implementados: GET, POST, PUT e DELETE;

2. Para não haver redundâncias, cada recurso só poderá ser acessado por um endereço;

3. Variáveis referentes ao endereço único do recurso (URI) devem ser *lowercase*. Em caso de um caminho precisar ser indicado por duas ou mais palavras, deve ser utilizado hífen para separar cada palavra. Exemplo: */origens-destinos*;

4. Tabelas que armazenam dados que se referem a cidades, estados, países e categorias de animais, que têm o objetivo de servir como padrão de registros só poderão ser acessadas pelo método GET.

5. O único formato de mídia aceito para requisições e utilizado nas respostas do WS será "*application/json*".

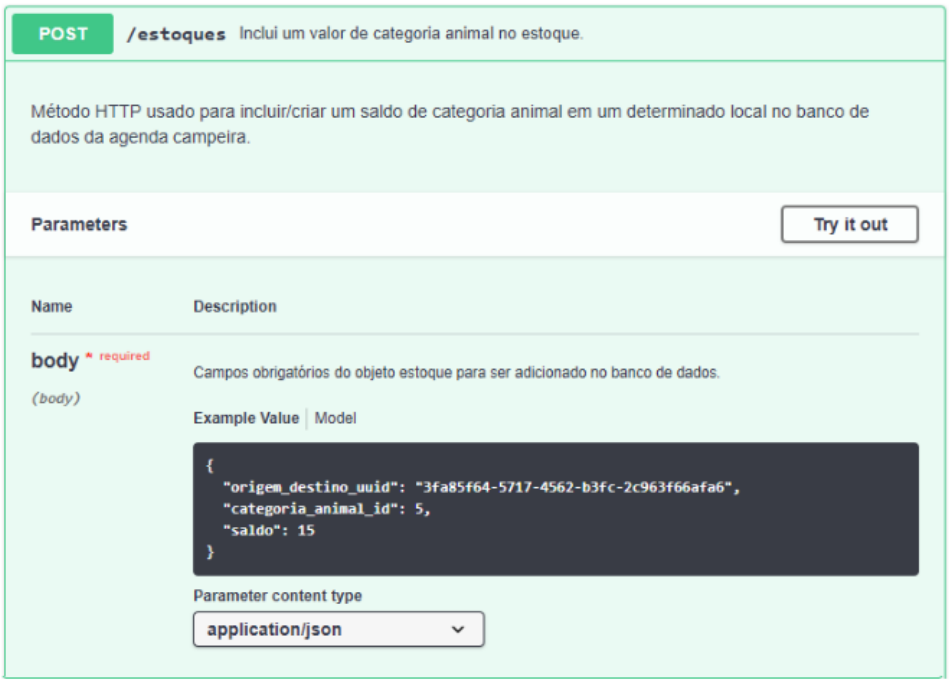
6. Por segurança, as solicitações de dados de usuários serão autenticadas por meio de *tokens*, que serão enviados para o aplicativo no momento de login ou criação de um usuário.

O objetivo principal da documentação é demonstrar ao desenvolvedor utilizador da API, de forma clara e direta como seus recursos devem ser usados. Para isso, na documentação é possível indicar endereços, parâmetros, tipos das variáveis (*int, string, float*) e as possíveis respostas de uma solicitação. Com o editor, podemos definir parâmetros nos recursos, declarar as tabelas e seus atributos, e exemplificar respostas de acordo com o seu código HTTP (como mostrado na Tabela 3). Por exemplo, a Figura 16 e a Figura 17 mostram, respectivamente, um exemplo de indicação de parâmetros e um exemplo de resposta a uma solicitação.

Para complementar a modelagem, foram reproduzidas todas as tabelas do banco de dados manipuladas pelos recursos do *Web Service*. Através da criação de componentes no código, é possível criar objetos que podem ser referenciados em exemplos de corpo de requisições e resposta no documento de modelagem. Um exemplo de componente utilizado para exemplificação no Swagger pode ser visto na Figura 18.

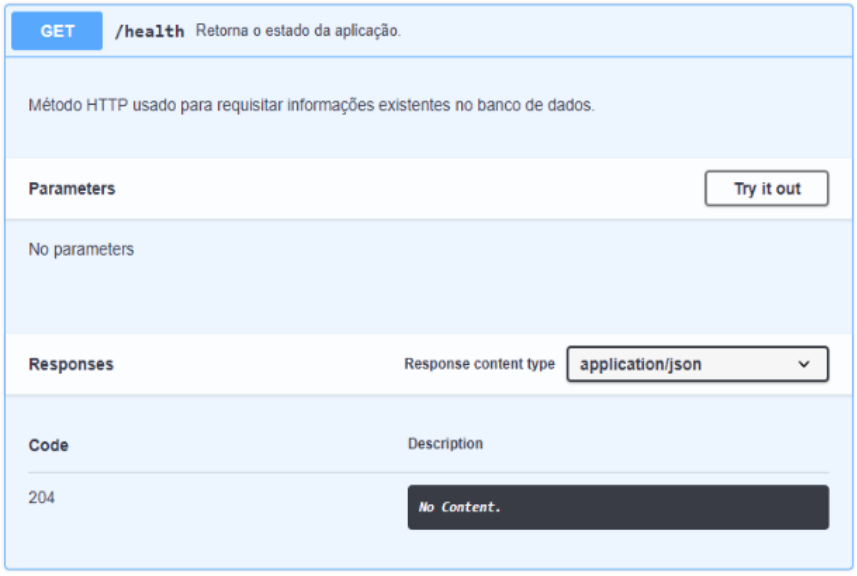
A análise dos dados e a modelagem do *Web Service* com o Swagger resultou na versão atual documentação, as tabelas que mostram todos os recursos modelados estão no Apêndice B, contendo todos os recursos disponibilizados e quais os métodos HTTP que são aceitos para cada recurso. A estratégia em disponibilizar recursos de listagem de registros padronizados é fazer com que o desenvolvedor possa ter conhecimento de como os dados estão armazenados e podendo solicita-los em qualquer momento pelo aplicativo, e que isso possa influenciá-lo em melhores decisões no desenvolvimento do aplicativo.

Figura 16 – Exemplo de indicação de parâmetros *body* no Swagger.



Fonte: Autor (2019)

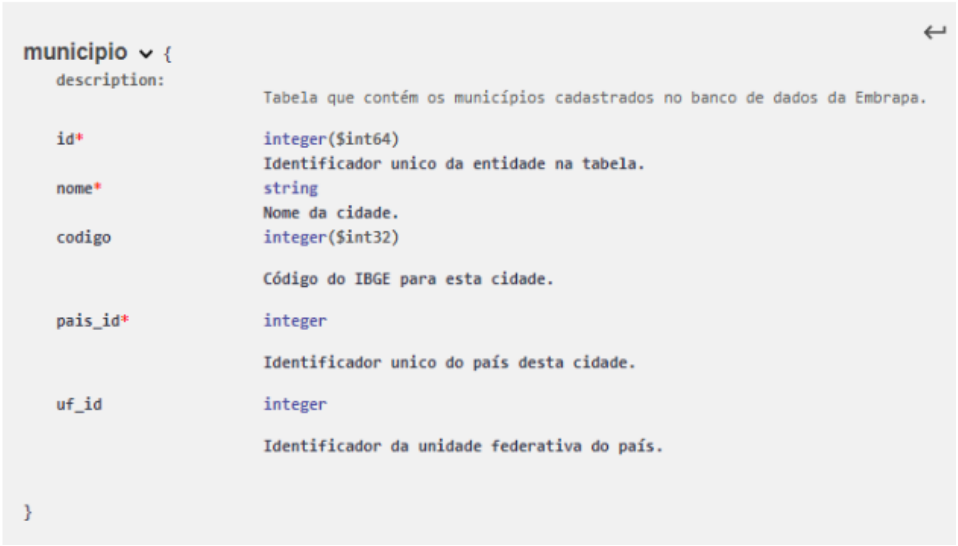
Figura 17 – Exemplo de uma resposta no Swagger.



Fonte: Autor (2019)

Concluindo, nessa etapa foram modelados 12 recursos principais e aproximadamente 20 sub-recursos, contando com mais de 60 chamadas distintas com exemplos de requisições e respostas.

Figura 18 – Componente de exemplificação no Swagger.



Fonte: Autor (2019)

4.2 Desenvolvimento do Web Service

Para assumir o gerenciamento de construção de projeto, foi requisitado pelos supervisores do projeto o uso da ferramenta Apache Maven. Maven é uma ferramenta de apoio a projetos Java baseada em um arquivo POM (*Project Object Model*), que é responsável por manter as declarações da estrutura e dependências de um projeto. A utilização desta ferramenta possibilita automatizar e padronizar a construção e publicação de um projeto Java. Ela é responsável por organizar todos os *plugins* necessários para incluir na pasta do arquivo Java executável. A compilação é baseada no conceito de ciclo de vida, entende-se como ciclo de vida as etapas necessárias para uma compilação completa da aplicação. Dentro das etapas utilizadas neste projeto podemos citar:

1. *compile*: Compilação do código fonte;
2. *package*: Compila e executa o empacotamento, de acordo com o POM, no formato escolhido. Normalmente, o empacotamento é direcionado a uma pasta chamada *target*. Exemplo de formatos: .war e/ou .jar;
3. *install*: Instala os pacotes descritos no POM no repositório local;
4. *clean*: Este comando não faz parte do ciclo de vida comum, é um comando adicionado por *plugin* ao Maven. O *clean* é responsável por excluir a pasta *target* para que numa segunda compilação não ocorra nenhuma inconsistência na pasta utilizada.

Um comando pode ser executado pelo terminal Linux no diretório que contenha o arquivo POM ou pelo ambiente de desenvolvimento utilizado, sua sintaxe tem o seguinte formato: mvn [comando]. A Figura 19 demonstra o cabeçalho do arquivo deste projeto, o mesmo indica que o comando *package* deve informar a geração de um .war na compilação.

Figura 19 – Cabeçalho POM e suas propriedades.

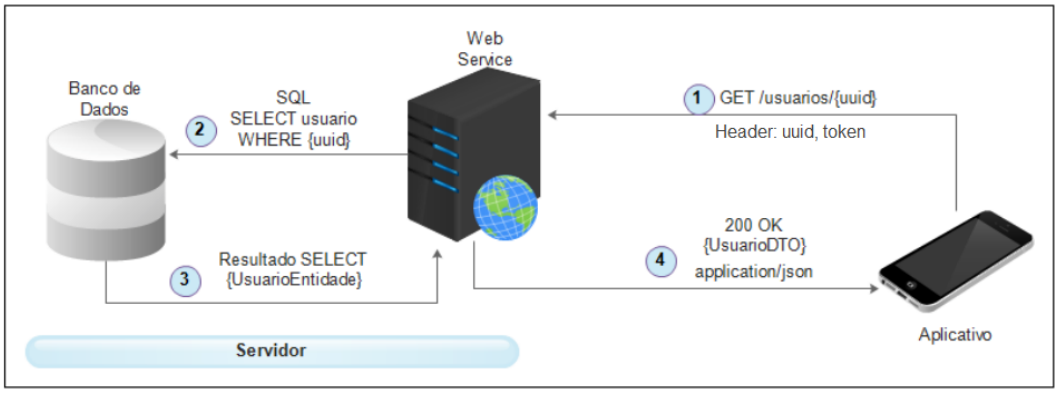
```
<modelVersion>4.0.0</modelVersion>
<groupId>br.embrapa.cppsul.agendacampeira</groupId>
<artifactId>agenda-campeira-backend</artifactId>
<name>Agenda Campeira Backend</name>
<version>1.0.0-SNAPSHOT</version>
<packaging>war</packaging>

<properties>
    <version.wildfly.swarm>2018.3.3</version.wildfly.swarm>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <skipTests>true</skipTests>
</properties>
```

Fonte: Autor (2019)

O funcionamento básico do *Web Service* pode ser visto na Figura 20, ela retrata em etapas como deve acontecer a comunicação do aplicativo e o banco de dados. No exemplo, é representada uma requisição de leitura pelo aplicativo, onde o WS é responsável por executar uma clausula SQL na tabela de usuários a partir do identificador enviado como parâmetro no endereço e responder ao aplicativo o resultado dessa leitura.

Figura 20 – Funcionamento básico do *Web Service*.



Fonte: Autor (2019)

1. O aplicativo solicita via HTTP uma leitura no endereço *"/usuarios/{uuid}"* enviando o identificador deste usuário como parâmetro no endereço.

2. Recebendo a solicitação, o WS é responsável por reconhecer e validar a chamada. Após isso, ela utiliza o identificador enviado para executar uma leitura no banco de dados no servidor. Se a chamada conter inconsistências, seja no corpo de requisição ou parâmetros enviados, a resposta contendo os erros são enviados nesta etapa para o aplicativo, não sendo propagada a requisição para a instância do banco de dados.

3. O banco de dados retorna para o WS o resultado da leitura. O WS armazena os dados do usuário em uma classe denominada *"UsuarioEntity"*.

4. O WS cria outra classe *"UsuarioDTO"* a partir da classe da etapa anterior e a envia ao aplicativo como resposta da solicitação HTTP da etapa 1. O formato dos dados utilizado na resposta é JSON.

Para obter maior controle sobre os dados recebidos pelo aplicativo o *Web Service* utilizará um padrão chamado DTO (Objeto de Transferência de Dados), seu objetivo no código é mapear, organizar e validar os corpos das mensagens recebidas e enviadas para o aplicativo. Sendo assim, a implementação deste padrão facilitará que as informações recebidas pelo WS sejam validadas antes que sejam encaminhadas ao banco de dados.

Para que as classes possam ser reconhecidas pela aplicação, devem possuir as notações referentes aos formatos utilizados nas mensagens. Como padronizado: Nas classes DTO, as notações se referem ao JSON e as classes de entidades devem conter referências das tabelas do banco. As classes que representam as entidades do banco, tem em sua implementação as notações ORM que realizam o mapeamento das variáveis da linguagem Java para os campos do SGBD PostgreSQL. A Figura 21 mostra a implementação das variáveis da classe *UsuarioDTO* e na Figura 22 mostra-se a implementação da classe *UsuarioEntity*.

A implementação do padrão DTO traz uma confiança maior no formato dos dados enviados pelo aplicativo devido a camada de validação feita pelo *Web Service*, entretanto, no contexto do projeto o padrão obriga que seja construído uma classe auxiliar para cada DTO, que faça a transformação para uma classe que representa uma entidade e de uma entidade para um DTO. Dentro das pastas, criou-se um diretório para estas classes auxiliares denominado *Mapper*. Basicamente, as classes de mapeamento contém três métodos que são necessários para um mapeamento completo das classes de entidades para as classes DTO. Como pode ser visto na Figura 23, a relação de classes de procedimentos do recurso "lotes". Os principais métodos da classe LoteMapper são *toDTO*, *toCreateEntity* e *toUpdateEntity*. Com eles, são possíveis qualquer transformação necessária dentro dos

Figura 21 – Atributos da classe *UsuarioDTO*.

```
public class UsuarioDTO {

    @JsonProperty("uuid")
    private String uuid;          // 1

    @JsonProperty("email")
    private String email;         // 2

    @JsonProperty("nome")
    private String nome;          // 3

    @JsonProperty("fone_ddi")
    private String foneDDI;       // 4

    @JsonProperty("fone_area")
    private String foneArea;      // 5

    @JsonProperty("fone_numero")
    private String fone;          // 6

    @JsonProperty("inativo")
    private Object inativo;       // 7

    @JsonProperty("criacao")
    @JsonFormat(shape=JsonFormat.Shape.STRING, timezone="America/Sao_Paulo")
    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private Date criacao;         // 8

    @JsonProperty("modificacao")
    @JsonFormat(shape=JsonFormat.Shape.STRING )
    @JsonInclude(JsonInclude.Include.NON_EMPTY
    private Date modificacao;     // 9

    @JsonProperty("cpf")
    private Object cpf;           // 10

    @JsonProperty("senha")
    private String senha;         // 11

    @JsonProperty("confirmacao")
    private String confirmacao; // 12

    @JsonProperty("novo_email")
    private String novoEmail;     // 13

    @JsonProperty("pais_id")
    private Object paisId;        // 14

    @JsonProperty("municipio_id")
    private Object municipioId; // 15
```

Fonte: Autor (2019)

Figura 22 – Atributos parciais da classe *UsuarioEntity*.

```
@Entity
@Table(name = "public.usuario")
public class UsuarioEntity {
    @Id
    @Column(name = "uuid", nullable = false, length = 36)
    private String uuid;              // 1
    @Column(name = "email", nullable = false, length = 50, unique = true)
    private String email;             // 2
    @Column(name = "nome", nullable = false, length = 50)
    private String nome;              // 3
    @Column(name = "fone_ddi", length = 2)
    private String foneDDI;           // 4
    @Column(name = "fone_area", length = 2)
    private String foneArea;          // 5
    @Column(name = "fone_numero", length = 9)
    private String fone;              // 6
    @Column(name = "inativo")
    private Boolean inativo;          // 7
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "criacao", nullable = false)
    private Date criacao;             // 8
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "modificacao", nullable = false)
    private Date modificacao;         // 9
    @Column(name = "cpf")
    private Long cpf;                 // 10
    @Column(name = "senha", length = 40)
```

Fonte: Autor (2019)

procedimentos programados no WS.

A implementação do gerenciamento dos dados e toda organização das pastas principais do projeto podem ser divididas em: Modelos (*Models*), Recursos (*Resources*) e Serviços (*Services*).

1. Modelos: A pasta modelos contém três subdivisões, onde cada uma contém um formato de classe necessária para a troca de dados com o aplicativo e o banco

Figura 23 – Classes *Resource*, *Service* e *Mapper* para o recurso Lotes.

Fonte: Autor (2019)

de dados. A primeira, denominada DTO, contém as classes que reconhecem o formato de mídia JSON e, recebem e enviam dados diretamente com o aplicativo. A segunda, denominada *Entity*, contém todas as classes que serão manipuladas com operações CRUD no banco de dados. A última, *Mapper*, contém as classes que contém os métodos que geram uma classe DTO a partir de uma entidade e vice-versa.

2. Serviços: São as classes que implementam uma *Entity Manager* para controle da lógica das operações CRUD. Essas classes tem a função de validar as informações das classes DTO e realizar de forma persistente as manipulações de dados no banco de acordo com cada chamada realizada pelas classes de recursos. Por organização, existe uma interface que expõe os métodos e outra classe que as implementa. Para cada recurso modelado foi criada uma classe *Service*.

3. Recursos: São as classes que gerenciam um serviço, elas utilizam a notação @*Inject* para criar uma instância de uma classe Serviço e os utilizam para criar as respostas de cada método. Nela, estão contidas também as notações referentes aos métodos, endereços e formato de dados, como mostrado na Figura 24 um exemplo de método GET na classe *UsuarioResource*. Cada recurso modelado contém uma classe nesse formato.

Figura 24 – Método GET do recurso de usuários.

```
@GET
@Path("/{uuid}")
@Produces(MediaType.APPLICATION_JSON + ";charset=utf-8")
public Response getUsuarioById(@PathParam("uuid") UUID uuid, @Context HttpServletRequest request){

    String token = request.getHeader("token");

    if(token == null)
        return Response.status(Status.PRECONDITION_FAILED).type("token").build();

    String tokenBD = this.usuarioService.findToken(uuid);

    if(tokenBD.equals(token)) {
        ResponseDTO<UsuarioDTO> responseUsuarioDTO = new ResponseDTO<UsuarioDTO>();
        responseUsuarioDTO.setDTO(this.usuarioService.findByUuid(uuid));
        this.usuarioService.createLog(uuid, "usuarios", "r", new Date() , uuid.toString(),
                request.getRemoteAddr(), request.getRequestURL().toString());
        return Response.ok(responseUsuarioDTO).build();
    }
    else {
        return Response.status(Status.UNAUTHORIZED).build();
    }
}
```

Fonte: Autor (2019)

4.2.1 Autenticação e Validação

Pelo fato de cada requisição REST ser independente, existe a necessidade de enviar informações que identifiquem o usuário utilizador do aplicativo em cada requisição. Esse tipo de informação deve estar contido no cabeçalho da requisição HTTP. Um usuário pode ser reconhecido pelo *Web Service* através de seu identificador único universal (uuid), entretanto ainda precisamos ter um controle maior para validar uma requisição.

Um dos formatos mais simples possíveis para validar um usuário seria enviar em cada requisição o identificador e a senha do usuário, isso afetaria diretamente a segurança do *Web Service* pela necessidade de enviar a senha em cada requisição. Além do mais, quando falamos em HTTP, qualquer pessoa que tenha acesso a requisição na arquitetura da rede poderá visualizar as informações de acesso deste usuário e poderá replicá-las. Também pode-se trabalhar com criptografia, como o HTTPS, para evitar que usuários maliciosos da rede tenham acesso direto aos dados da requisição.

No *Web Service*, a autenticação de usuários é feita através de *tokens*. Sempre que um novo usuário for criado no banco de dados ou houver uma chamada no recurso de usuários para autenticação será retornado um *token* de acesso para o aplicativo. Nestas chamadas, o *token* é criado e salvo no banco de dados na tabela de usuários, e em cada requisição feita pelo aplicativo deve conter o mesmo *token* e o identificador do usuário no cabeçalho HTTP da requisição. Sendo assim, a partir da primeira chamada para criação

de um usuário, só será enviada a senha para o WS novamente quando um usuário efetuar um login no aplicativo.

Em cada requisição recebida pelo *Web Service*, exceto na criação e autenticação de usuários, os dados do identificador e *token* do usuário são lidos da requisição e um procedimento de verificação é realizado. Primeiramente, é testado se os valores não são nulos e posteriormente é verificado se o usuário existe, e se o *token* recebido é semelhante ao que se encontra no banco de dados. Um exemplo de como é feita a autenticação em uma requisição pode ser visto na Figura 25. O identificador de usuário não é necessário no cabeçalho quando já consta como parâmetro da requisição (Figura 24).

Figura 25 – Procedimento padrão para validar uma requisição.

```
String token = request.getHeader("token");
UUID uuidUsuario = UUID.fromString(request.getHeader("uuid"));

if(token == null)
    return Response.status(Status.PRECONDITION_FAILED).type("token").build();
if(uuidUsuario == null)
    return Response.status(Status.PRECONDITION_FAILED).type("Header: uuid").build();

String tokenBD = this.usuarioService.findToken(uuidUsuario);

if(tokenBD.equals(token)) {
        //executa a requisição
}
else {
    return Response.status(Status.UNAUTHORIZED).build();
}
```

Fonte: Autor (2019)

Um ponto importante para validação das requisições é que um usuário só poderá editar os dados de uma fazenda na qual ele está vinculado. Então, para cada requisição de qualquer recurso das tabelas do esquema da *agenda_campeira* deverá ser avaliado se o usuário tem autorização para realizar qualquer manipulação desses dados. Para isso, em cada classe que não contém diretamente o identificador da fazenda (estoques, movimentos, pesagens, nascimentos...) foi necessária a criação de uma função que encontrasse uma fazenda a partir dos identificadores únicos dessas tabelas, para posteriormente realizar o teste e validar a associação.

4.2.2 Logs

Como requisito do *Web Service*, ele precisa registrar no banco de dados um histórico de cada requisição válida que efetuou uma manipulação CRUD no banco. Portanto,

sempre que houver uma requisição válida de um usuário autenticado, antes de enviar a resposta ao aplicativo os dados dessa requisição devem ser salvos no banco. Uma chamada para a criação de um *log* pode ser visualizada na Figura 24.

No desenvolvimento desta funcionalidade, não conseguiu-se através da ferramenta *Hibernate* mapear diretamente uma *string* com o endereço de IP do usuário para o formato *inet* presente no PostgreSQL. Para resolver tal problema, criou-se uma clausula SQL exclusiva para esta tabela, implementada na *EntityManager* para que todas classes que utilizem herança tenham acesso. O procedimento de inserir um *log* no banco de dados utilizou a função *CAST* de SQL para realizar a tipagem correta.

4.2.3 Wildfly Swarm

O *Wildfly Swarm*, versão inferior do atual Thorntail, oferece uma abordagem para empacotar e executar aplicativos Java EE (*Java Enterprise Edition*). É o *framework* utilizado para controlar toda a execução do *Web Service*. Com o *Wildfly Swarm* é possível declarar as especificações de Java EE que serão necessárias para a execução, o *framework* fornece suporte a desenvolvimento de microsserviços em conjunto com um servidor de aplicação para execução de testes.

Como o *Swarm* será o responsável pela execução da aplicação, foi necessário incluir algumas configurações para que a ferramenta tenha acesso ao banco de dados da Embrapa. Propriedades como nome de usuário, senha, endereço do banco e as configurações estão contidas no arquivo POM. Por padrão, para execução o *Swarm* concede acesso à aplicação na porta 8080.

Concluindo, para a obtenção de um arquivo .jar e a execução da aplicação desenvolvida deve ser necessário seguir o seguinte fluxo via terminal linux:

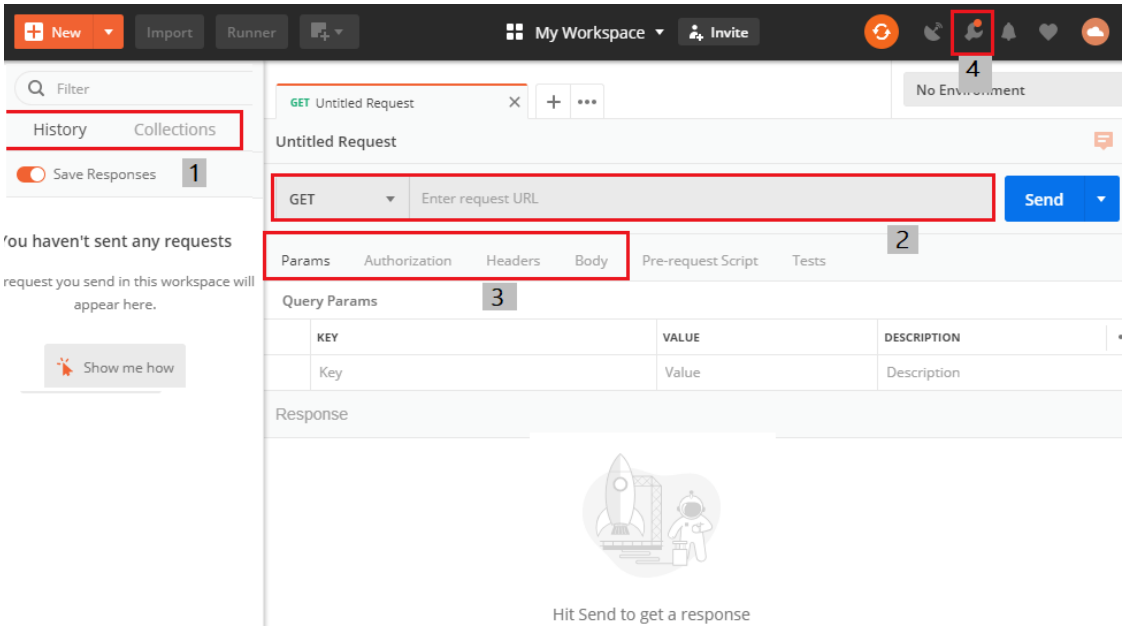
1. mvn *clean*
2. mvn *package*
3. Redirecionar o terminal até a pasta *target*: cd /target
4. -jar agenda-campeira-backend-swarm.jar

5 TESTES DOS RECURSOS

Neste capítulo serão mostrados os testes realizados com o *Web Service*. Primeiramente, é apresentada a principal aplicação utilizada para executar e armazenar requisições e suas respostas em um ambiente de testes. Posteriormente, são apresentados alguns exemplos de respostas de requisições executadas na aplicação. Os testes estão ligados apenas nas funcionalidades do WS, sendo assim, nenhum teste de exaustão dos serviços ou de tempo de resposta foi executado durante este trabalho.

Postman[1] é uma aplicação gratuita para realizar requisições HTTP através de uma interface compartilhada, que facilita os testes de serviços de APIs em equipes. A aplicação pode ser utilizada em sistemas operacionais como Linux, Windows e Mac, podendo também ser utilizada como uma extensão no navegador Web Google Chrome. No contexto deste trabalho, Postman foi a principal aplicação para executar testes do *Web Service* durante o desenvolvimento, por possibilitar o armazenamento de requisições para posteriormente serem reutilizadas. Na Figura 26, é possível visualizar sua tela inicial.

Figura 26 – Tela inicial do Postman.



Fonte: Autor (2019)

1. Na primeira aba da aplicação é possível verificar um histórico das últimas chamadas realizadas. Em *Collections* são listadas pastas de requisições criadas pelo usuário com o objetivo de organizar chamadas em coleções.

[1] https://www.getpostman.com/

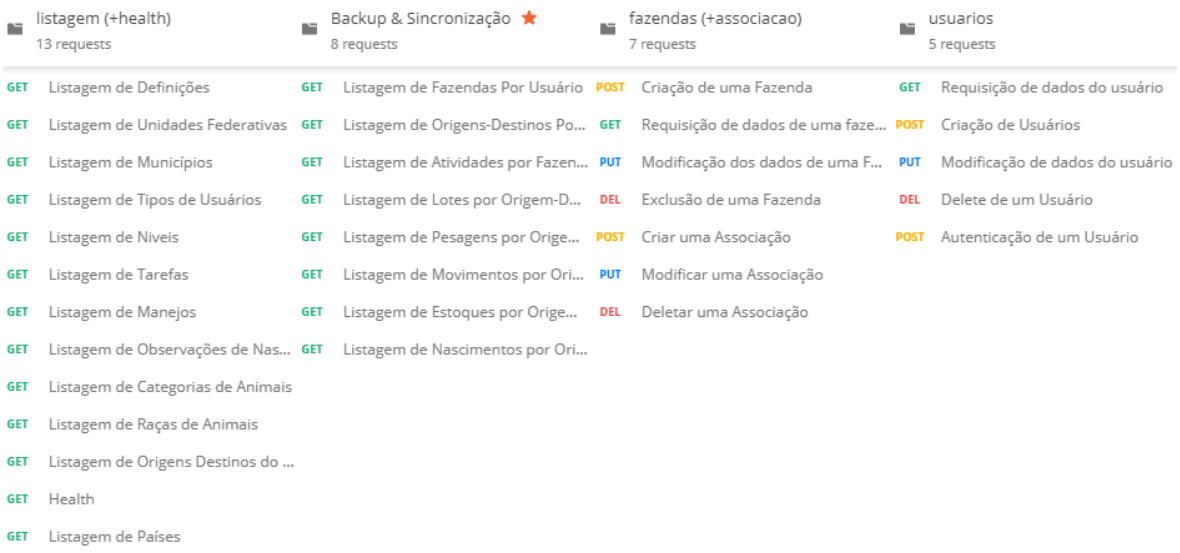
2. Local onde deve ser indicado o método HTTP na requisição e o endereço do recurso a ser acessado.

3. Locais onde devem ser inseridas as informações adicionais da requisição, como parâmetros no endereço (*Query Params*), modo de autorização e cabeçalho (*Header Params*). E por último, o corpo da requisição (*Body*) que será utilizado somente nas requisições POST e PUT.

4. Local onde estão contidas as configurações gerais do ambiente e do *proxy* utilizado pela aplicação.

Os testes foram realizados com o objetivo de verificar se a modelagem está de acordo com os recursos disponibilizados pelo *Web Service*. Para isso, criou-se coleções de chamadas organizadas pelos recursos modelados, totalizando 11 coleções, onde em cada coleção foram geradas requisições para os testes dos recursos implementados. Cada requisição dentro das coleções tem no mínimo um exemplo de resposta válida. Na Figura 27 e na Figura 28 apresentam-se as coleções com as requisições. A primeira contém as coleções que contêm sub-recursos, a segunda mostra 4 coleções que seguem a mesma linha de modelagem dos recursos restantes, onde há uma requisição por método HTTP e nenhum sub-recurso.

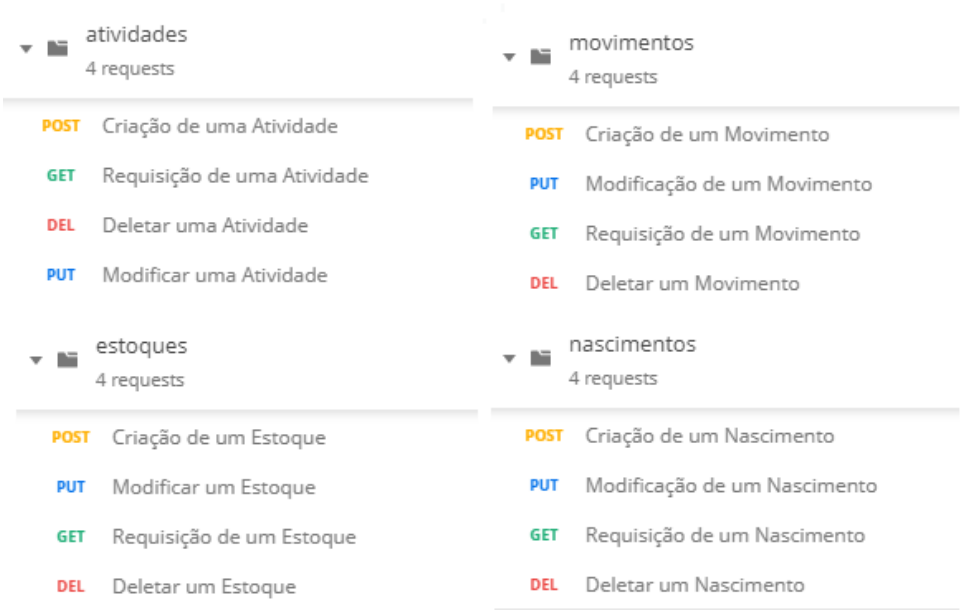
Figura 27 – Coleções de requisições específicas.



Fonte: Autor (2019)

Todas as chamadas para o *Web Service* necessitam de dados para autenticação de usuário, com exceção das requisições para criação e autenticação de um usuário. Um

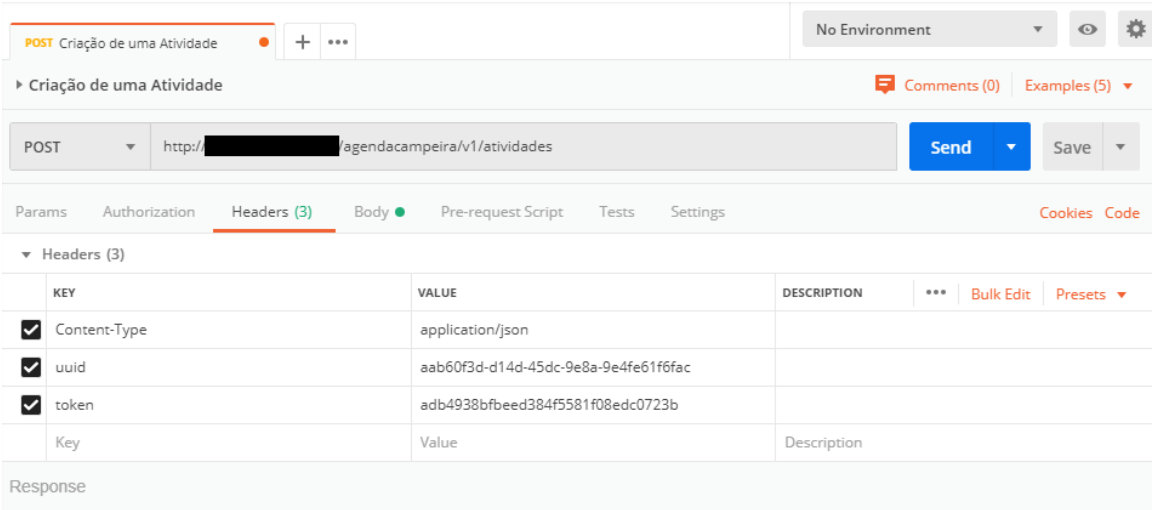
Figura 28 – Exemplos de coleções de requisições comuns.



Fonte: Autor (2019)

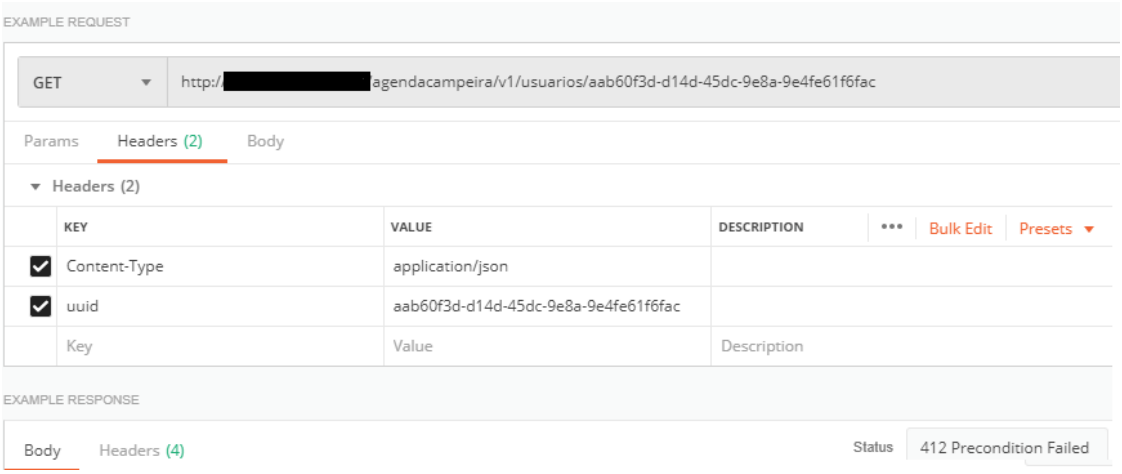
exemplo de cabeçalho de requisição pode ser visualizado na Figura 29. Nele, constam os campos com o identificador do usuário e o seu *token*, necessários para identificar o usuário e o validar antes de executar cada manipulação no banco de dados. Na falta de qualquer um desses campos de autenticação o WS retornará o código 412 *Precondition Failed*, como pode ser visualizado na Figura 30, onde consta um exemplo de execução de um GET no endereço do recurso de usuários para retornar os dados através de um identificador.

Figura 29 – Exemplo de cabeçalho HTTP para os testes.



Fonte: Autor (2019)

Figura 30 – Exemplo de *Precondidition Failed*.

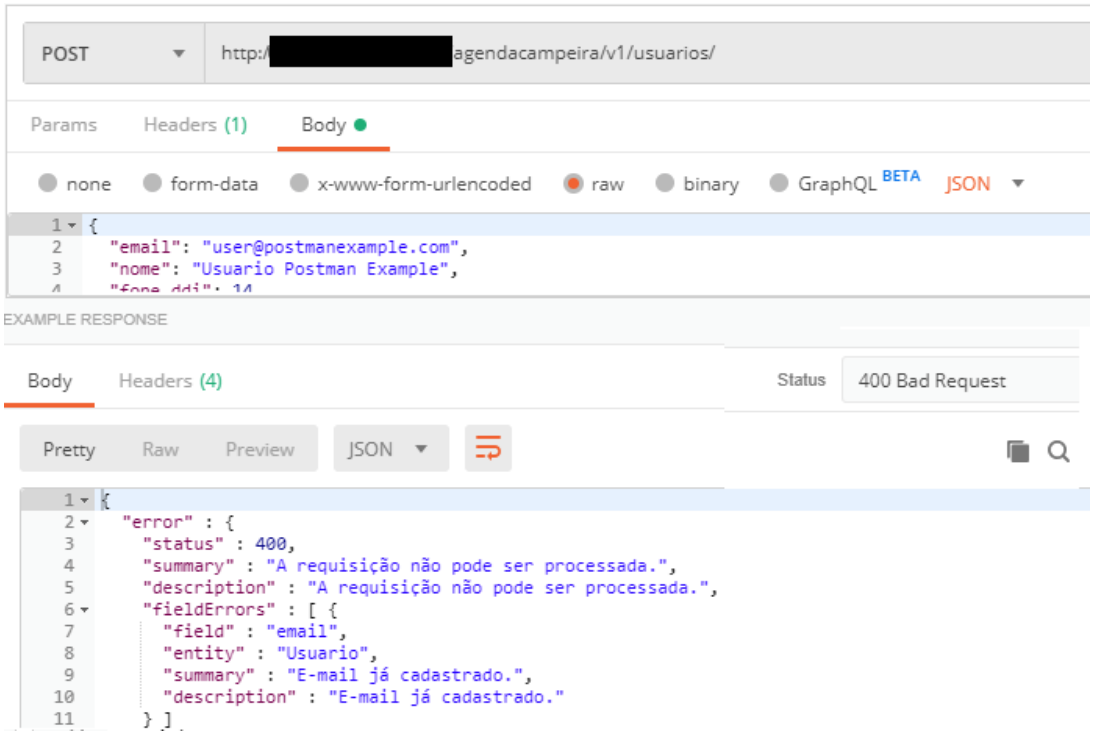


Fonte: Autor (2019)

Um determinado recurso acessado por um método HTTP pode ter diferentes respostas de acordo com os dados enviados em uma requisição. O *Web Service* realiza uma validação dos dados enviados no corpo da requisição para retornar o código de resposta mais adequado para o aplicativo, para facilitar a depuração de erros ou inconsistências no decorrer de sua utilização. Por exemplo, o campo para armazenar um e-mail na tabela referente a usuários no banco de dados é identificado como único, implicando que não podem ter dois usuários utilizando o mesmo e-mail. Se for enviado para a API um formulário de cadastro de um usuário com um e-mail já utilizado, será retornado o código 400 *Bad Request* com uma descrição indicando qual foi o erro, tal situação pode ser visualizada na Figura 31. O mesmo código e uma mensagem similar são enviados quando há falta de campos obrigatórios nos corpos das requisições nos métodos POST e PUT.

Após passar as validações dos campos de uma requisição, há outros erros que podem ser encontrados durante a execução do código. Como por exemplo, um usuário manipular dados de uma fazenda na qual ele não está associado ou enviar um campo de chave estrangeira com um identificador de uma tabela que não existe no banco de dados. Para o primeiro caso, existe uma rotina no código para verificar se todas as chaves manipuladas fazem parte da fazenda referida na requisição e também se o usuário é associado a esta fazenda. No exemplo da Figura 32, uma requisição POST foi montada para criar uma atividade, onde o valor do campo da chave estrangeira da fazenda é o identificador de uma fazenda existente no banco de dados mas que o usuário não tem uma associação. A resposta indica que não foi encontrada a fazenda, para que o requisitante do *Web Service* não tenha conhecimento de que o identificador utilizado de fato existe no banco de dados.

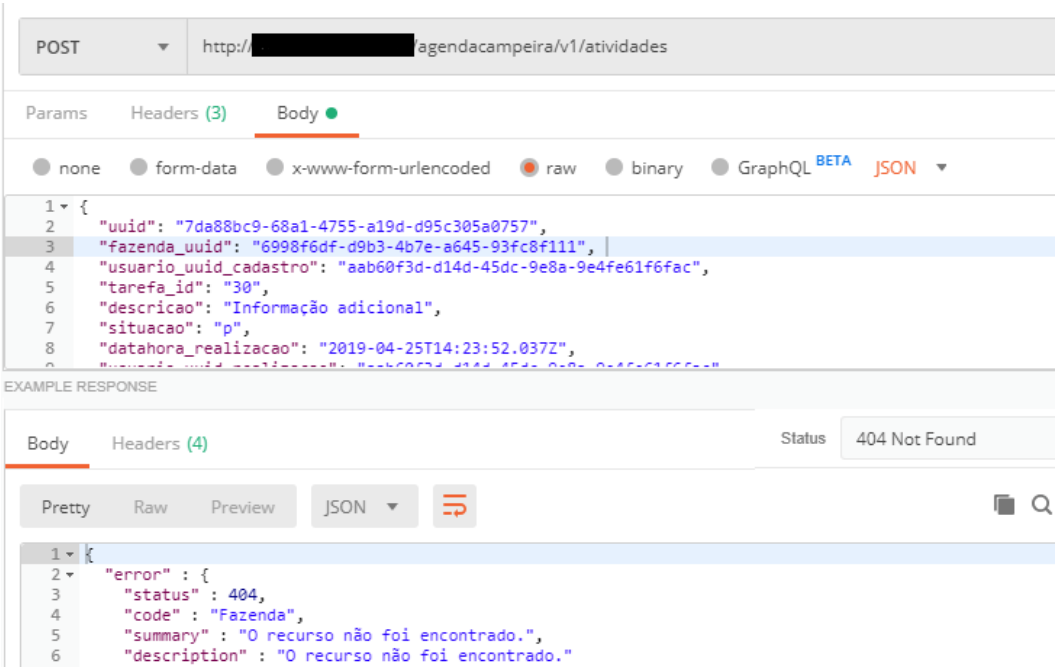
Figura 31 – Exemplo de *Bad Request*.



Fonte: Autor (2019)

Então, é enviado 404 *Not Found* como código de resposta.

Figura 32 – Exemplo de *Not Found*.

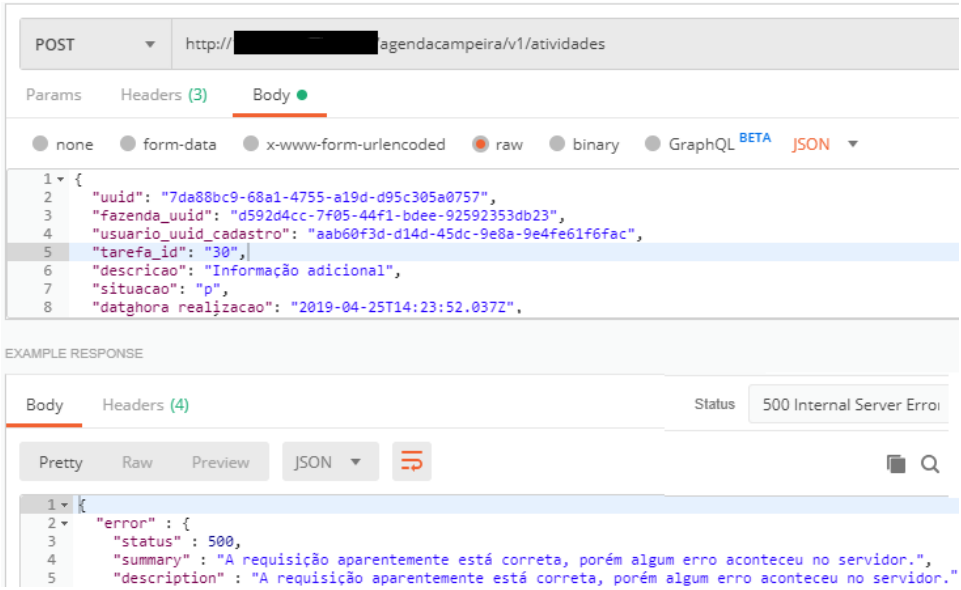


Fonte: Autor (2019)

Quando for enviada uma requisição em que o corpo da mensagem utilize uma

chave estrangeira inválida de uma das tabelas do recurso de listagem, o *Web Service* retornará o código 500 *Internal Server Error*. Isso acontece por que tal erro não foi mapeado para resolver no WS, sendo assim, quando um valor inválido de chave estrangeira é utilizado para realizar uma criação ou alteração de dados é lançado uma exceção no *framework* Hibernate que finaliza a requisição. No terminal de execução do WS, é lançado um erro indicando que a cláusula SQL não pode ser finalizada por que está sendo utilizada uma referencia de uma tabela que não existe no banco de dados. Na Figura 33, apresenta-se uma requisição onde ocorre este tipo de erro com o campo que referencia uma tarefa que não existe no banco. A mesma mensagem de resposta é utilizada quando acontece um erro na execução de uma exclusão, onde não é possível excluir um registro cujo identificador é utilizado como referencia em outra tabela no banco de dados.

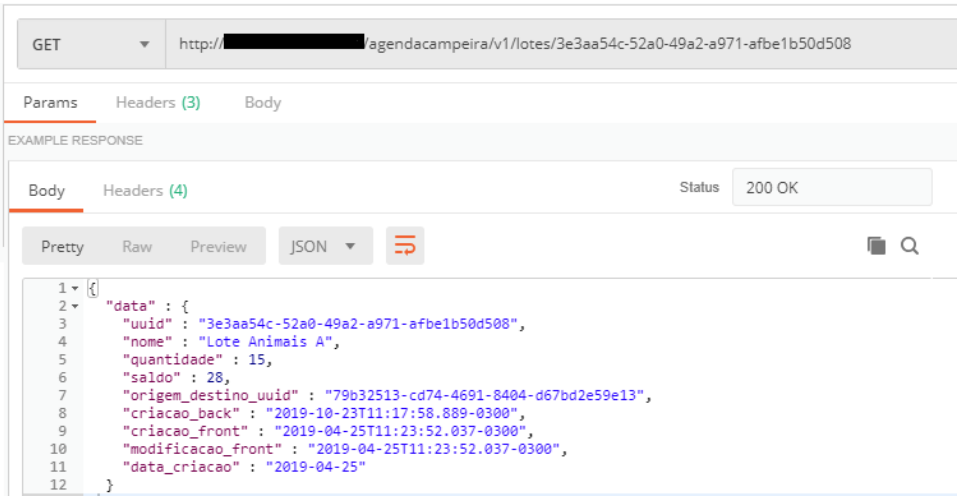
Figura 33 – Exemplo de *Internal Server Error*.



Fonte: Autor (2019)

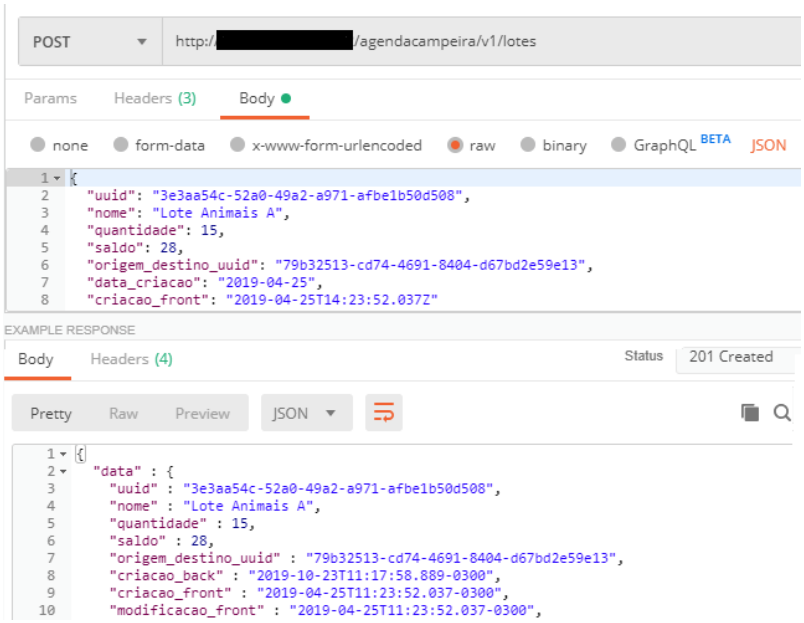
Quando a requisição não passar pelas exceções apresentadas acima, ela é considerada válida e terá uma mensagem de sucesso retornada ao usuário. Logo, as respostas para GET e PUT serão 200 *OK*, POST receberá 201 *Created* e DELETE receberá o código 204 *No Content*. Na Figura 34, temos um exemplo de requisição GET, onde é retornado o valor de um lote de animais em uma subdivisão de uma fazenda associada ao usuário. E na Figura 35, apresenta-se um exemplo de requisição POST para a criação de um lote de animais que foi retornado no exemplo anterior. Nas respostas de sucesso para os métodos POST e PUT sempre será enviado os dados da tabela que foi criada ou modificada no corpo da resposta.

Figura 34 – Exemplo de 200 *OK*.



Fonte: Autor (2019)

Figura 35 – Exemplo de 201 *Created*.



Fonte: Autor (2019)

Este capítulo apresentou os testes de utilização da API com a aplicação Postman, onde foram testadas e documentadas as repostas de todos os recursos da API. No total, foram criadas 11 coleções com mais de 60 requisições no total e no mínimo um exemplo de resposta de sucesso para cada requisição. Os testes também mostraram uma falha pela possibilidade de utilizar chaves estrangeiras não existentes no recurso de listagens.

6 CONCLUSÕES

A utilização de um *Web Service* dentro do contexto de uma empresa pode influenciar o desenvolvimento de aplicações positivamente. Neste trabalho, o WS foi criado para realizar a comunicação de um aplicativo produzido pela Embrapa Pecuária Sul com o banco de dados da empresa, possibilitando assim, o armazenamento e leitura de dados lançados no aplicativo que será utilizado por produtores rurais pecuários. O WS possibilitou que o aplicativo fosse desenvolvido com maior foco em suas funcionalidades. Por realizar a comunicação com o banco de dados da Embrapa, o aplicativo poderá ter um número maior de utilidades disponíveis para o usuário e também para a própria empresa. Como por exemplo:

1. Utilizando um banco de dados para receber registros de propriedades pecuárias, a empresa poderá armazenar dados pertinentes sobre os produtores rurais, como o perfil do produtor e características gerais do seu sistema produtivo.

2. Através de autenticação, o aplicativo poderá recuperar lançamentos de fazendas de um usuário através dos recursos de sincronização. Também, através da validação feita pelo WS, usuários associados a uma mesma fazenda poderão realizar os lançamentos de registros no aplicativo de forma colaborativa.

3. Mudanças no banco de dados da Embrapa, dependendo da prioridade, poderão somente serem implementadas no WS. Podendo evitar que o aplicativo necessite de atualizações.

A utilização de REST foi adequada para o problema apresentado pela empresa. A utilização do protocolo HTTP para indicar qual manipulação deve ser realizada no banco de dados conseguiu ser facilmente mapeada e modelada. Através das ferramentas apresentadas no trabalho, foi possível desenvolver uma solução que cumpre sua função de conectar duas aplicações. REST e JSON foram boas escolhas para realizar uma comunicação do aplicativo com o *Web Service* pelo motivo das requisições serem eficientes em tempo de leitura.

Os testes apresentados nesse trabalho foram realizados utilizando ferramentas que simulam chamadas HTTP. Através da aplicação Postman, foi possível armazenar respostas do *Web Service* para que sirvam de exemplos para possíveis erros da utilização do WS pelos desenvolvedores do aplicativo. Além das mensagens de resposta enviadas pela próprio WS na resposta HTTP.

A principal dificuldade encontrada durante a execução deste trabalho foi a de manter todos os documentos (modelagem, projeto e testes) atualizados durante o desenvolvimento do *Web Service*. O banco de dados teve várias mudanças no decorrer do trabalho e para cada mudança, foi necessário atualizar a modelagem, realizar as mudanças no código e realizar os testes novamente para adequação dos documentos. O ambiente ideal para a criação do WS seria, no mínimo, com o banco de dados consolidado.

Apesar de adequada a solução ao problema apresentado, alguns pontos do trabalho podem ser melhorados e outras tecnologias ou modelos de programação podem ser acrescentados. Sobre continuação e aprimoramento da API nesse trabalho, pode-se citar:

1. Estudo da implementação de DAOs (*Data Access Objects*) para substituir o padrão DTO em algumas classes. Um DAO é um objeto que pode ser mapeado diretamente via *frameworks* para a comunicação entre o banco de dados e outra aplicação. Por mais que seja requisito da API a utilização de DTOs, em algumas classes sua utilização é obsoleta, porque todos os dados da entidade são enviados para a resposta sem nenhuma restrição. Logo, nesses casos, não há necessidade de ter uma classe DTO e executar os procedimentos necessários para utilizá-la.

2. Implementação de recursos adicionais para criação de usuários a partir de dados de autenticação da Google e do Facebook;

3. Aprimorar o sistema de autenticação, utilizando alguma forma de serviço secundário para desvalidar *tokens* através de um limite de tempo ou por número de requisições.

4. Após o aplicativo que utilizará o *Web Service* ser finalizado e distribuído para utilização, pode-se realizar análises das informações armazenadas no banco de dados da Embrapa.

5. Manutenção do *Web Service*.

REFERÊNCIAS

ABITEBOUL, S.; BUNEMAN, P.; SUCIU, D. **Data on the web: from relations to semistructured data and XML**. São Francisco, Califórnia: Morgan Kaufmann, 2000.

ANACLETO, J. A. C. **Desenvolvimento de uma aplicação web para dispositivos móveis - Monitorização e controlo de uma rede de digital signage**. Tese (Doutorado) — Universidade do Minho Escola de Engenharia, 2012. Disponível em: <http://wiki.di.uminho.pt/twiki/pub/Research/APEX/Publications/tese.pdf>.

ANDRY, L. W. F.; NICHOLSON, D. A MOBILE APPLICATION ACCESSING PATIENTS' HEALTH RECORDS THROUGH A REST API - How REST-Style Architecture can Help Speed up the Development of Mobile Health Care Applications. In: . [S.l.]: Scitepress, 2011. p. 27–32.

BIEHL, M. **RESTful API design : APIs your consumers will love**. 1ª. ed. [S.l.]: API-University Press, 2016. 287 p. ISBN 9781514735169.

BIH, J. Service oriented architecture (soa) a new paradigm to implement dynamic e-business solutions. **Ubiquity**, ACM, Nova York, NY, USA, v. 2006, n. August, p. 4:1–4:1, ago. 2006. ISSN 1530-2180. Disponível em: <http://doi.acm.org/10.1145/1162511.1159403>.

BRAY, T.; ED. **The JavaScript Object Notation (JSON) Data Interchange Format**. 2017. Disponível em: <https://tools.ietf.org/html/rfc8259>. Acesso em: 8 mai. 2019.

BRUM, L. **Aplicação de Técnicas de Business Intelligence em Sistemas de Apoio à Tomada de Decisão de Produtores Rurais.** Dissertação (Mestrado) — UNIPAMPA - Universidade Federal do Pampa, Bagé, 2019.

CLARK, D.; TENNENHOUSE, D. L. Architectural considerations for a new generation of protocols. In: **In Proceedings ACM SIGCOMM90 Symposium**. Filadélfia, PA: [s.n.], 1990. p. 200–208.

DANTAS, D. C. T. **Simple Object Access Protocol (SOAP)**. 2007. Disponível em: <https://www.gta.ufrj.br/grad/07_2/daniel/index.html>. Acesso em: 13 mai. 2019.

EMBRAPA. **Desenvolvimento de sistemas de apoio à decisão e de métodos de coleta, análise de dados e monitoramento da pecuária na região Sul do Brasil.** Empresa Brasileira de Pesquisa Agropecuária, 2015. Disponível em: <https://www.embrapa.br/busca-de-projetos/-/projeto/210797/desenvolvimento-de-sistemas-de-apoio-a-decisao-e-de-metodos-de-coleta-analise>. Acesso em: 25 mar. 2019.

EMBRAPA. **Gestão de custos é a principal preocupação do pecuarista brasileiro. Brasília**. Empresa Brasileira de Pesquisa Agropecuária, 2018. Disponível em: <https://www.embrapa.br/busca-de-noticias/-/noticia/36645433/gestao-de-custos-e-a-principal-preocupacao-do-pecuaristabrasileiro>. Acesso em: 25 mar. 2019.

FERREIRA, P. B. V. **Arquitetura REST em smartphones Android**. Dissertação (Mestrado) — Instituto Superior de Engenharia do Porto, 2015.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) — Dept. of Information and Computer Science, University of California, Irvine, 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern Web architecture. **ACM Transactions on Internet Technology**, ASM - Association for Computing Machinery, v. 2, n. 2, p. 115–150, jul 2002. ISSN 15335399.

FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding code mobility. **IEEE Transactions on Software Engineering**, v. 24, n. 5, p. 342–361, May 1998. ISSN 0098-5589.

GUILLAUD, H. **Comprendre les interfaces de programmation**. 2011. Disponível em: <http://www.internetactu.net/2011/06/24/comprendre-les-interfaces-de-programmation/>. Acesso em: 5 jun. 2019.

HEFFNER, R. The forrester wave TM: API management solutions, Q3 2014. **Cambridge: Forrester Research**, 2014. Disponível em: <https://www.forrester.com/report/The+Forrester+Wave+API+Management+Solutions+Q3+2014/-/E-RES119266>.

HEUSER, C. A. **Projeto de banco de dados : Volume 4 da Série Livros didáticos informática UFRGS.** 6. ed. [S.l.]: Bookman, 2009. ISBN 978-85-7780-452-8.

HOFER, E. et al. A relevância do controle contábil para o desenvolvimento do agronegócio em pequenas e médias propriedades rurais. v. 3, n. 1, p. 27–42, 2011. ISSN 1984-6266. Disponível em: <https://revistas.ufpr.br/rcc/article/view/21490>.

HOLLIFIELD, C.; DONNERMEYER, J. Creating demand: influencing information technology diffusion in rural communities. **Government Information Quarterly**, v. 20, n. 2, p. 135–150, 2003. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0740624X03000352>.

IBGE. **Acesso à internet e à televisão e posse de telefone móvel celular para uso pessoal : 2016**. IBGE - Instituto Brasileiro de Geografia e Estatística, 2018. Disponível em: <https://biblioteca.ibge.gov.br/index.php/biblioteca-catalogo?view=detalhes&id=2101543>. Acesso em: 25 abr. 2019.

KREGER, H. **Web Services Conceptual Architecture (WSCA 1.0)**. 2001. Disponível em: <https://www.researchgate.net/publication/235720479_Web_Services_Conceptual_Architecture_WSCA_10>.

KUHN, E. C. **ELABORAÇÃO DE UM PROTÓTIPO DE APLICATIVO PARA ACOMPANHAMENTO DE REQUISIÇÕES DE TÁXI**. Dissertação (Mestrado) — UTFPR - Universidade Tecnológica Federal do Paraná, Curitiba, 2012.

LUCCHI, R.; MILLOT, M. Resource Oriented Architecture and REST. Assessment of impact and advantages on INSPIRE. **Communities**, 2008. Disponível em: <https://core.ac.uk/download/pdf/38617810.pdf>.

MACHADO, C. F.; NANTES, F. D. Adoção da tecnologia da informação em organizações rurais: o caso da pecuária de corte. **Gest. Prod.**, v. 18, n. 3, p. 555– 570, 2011. Disponível em: <http://www.scielo.br/pdf/gp/v18n3/09.pdf>.

NEWCOMER, E. **Understanding Web Services: XML, WSDL, SOAP, and UDDI**. Addison-Wesley, 2002. (Independent technology guides). ISBN 9780201750812. Disponível em: <https://books.google.com.br/books?id=SHSBri-rMyQC>.

PAUTASSO, C. RESTful Web service composition with BPEL for REST. **Data and Knowledge Engineering**, v. 68, n. 9, p. 851–866, sep 2009. ISSN 0169023X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0169023X09000366>.

RODRIGUES, R. M. **CONSTRUINDO UMA API PARA EXIBIR HORÁRIOS DE TRANSPORTE**. Dissertação (Mestrado) — UTFPR - Universidade Tecnológica Federal do Paraná, Londrina, 2015.

SOMMERVILE, I. **Engenharia de Software.** 8. ed. São Paulo: Pearson Addison Wesley, 2007.

VAQQAS, M. **RESTful Web Services: A Tutorial**. 2014. Disponível em: <http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069>. Acesso em: 01 jun. 2019.

VAZ, G. J. et al. AgroAPI: criação de valor para a Agricultura Digital por meio de APIs. v. 2, n. SBIAgro, p. 59–68, 2017. Disponível em: <https://ainfo.cnptia.embrapa.br/digital/bitstream/item/169596/1/AgroAPI-SBIAgro.pdf>.

VUKOVIC, M. et al. Riding and thriving on the api hype cycle. **Commun. ACM**, ACM, Nova York, NY, USA, v. 59, n. 3, p. 35–37, fev. 2016. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2816812>.

W3C. **Simple Object Access Protocol (SOAP) 1.1**. 2000. Disponível em: <https://www.researchgate.net/profile/Satish_Thatte/publication/239553871_Simple_object_access_protocol_SOAP_11/links/54489e4e0cf2f14fb8142a59/Simple-object-access-protocol-SOAP-11.pdf>. Acesso em: 04 jun. 2019.

W3C. **W3C Web Services Glossary**. Working Group Note, 2004. Disponível em: <http://www.w3.org/TR/ws-gloss/>. Acesso em: 04 jun. 2019.

W3C. **Extensible Markup Language (XML) 1.0 (Fifth Edition)**. 2008. Disponível em: <https://www.w3.org/TR/xml/>. Acesso em: 01 jun. 2019.

WAGH, K. S.; THOOL, R. A Comparative study of SOAP vs REST web services provisioning techniques for mobile host Person Re-Identification View project MANET topology View project A Comparative Study of SOAP Vs REST Web Services Provisioning Techniques for Mobile Host. v. 2, n. 5, 2012. ISSN 2225-0506.

WALDO, J. et al. **A Note on Distributed Computing. IEEE Micro**. [S.l.], 1994. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7628>. Acesso em: 04 jun. 2019.

APÊNDICE A – DOCUMENTO DE REQUISITOS DO WEB SERVICE

Documento de Requisitos

Web Service REST

Responsável pela Elaboração

Eduardo Klein Pandolfo – Estudante do Curso de
Engenharia de Computação - Unipampa

Público Alvo

Este documento está destinado aos orientadores do trabalho, supervisores da Embrapa Pecuária Sul e para os desenvolvedores envolvidos no aplicativo da agenda campeira. Com este documento, espera-se dar subsídios para o desenvolvimento da proposta do *Web Service* para integração entre o aplicativo da agenda campeira e o banco de dados da Embrapa.

Histórico de Alterações

Data	Versão	Autor	Descrição
12/06/2019	1.0	Eduardo Klein Pandolfo	Criação do documento.
25/06/2019	1.0	Eduardo Klein Pandolfo	Elaborado uma descrição geral do sistema com imagem.

1 Introdução

Este documento especifica o *Web Service* REST produzido para o aplicativo da agenda campeira dentro do projeto *MyBeef,* fornecendo aos desenvolvedores as informações necessárias para o planejamento e implementação, assim como para a realização dos testes e homologação do sistema.

Visão geral deste documento

Esta introdução fornece as informações necessárias para fazer um bom uso deste documento, explicitando as convenções que foram adotadas no texto. A primeira seção do documento é sua própria introdução, o restante do documento é organizado da seguinte forma:

- **Seção 2** – Descrição geral do sistema: Apresenta uma visão geral do sistema, caracterizando qual é o seu escopo.
- **Seção 3** – Requisitos funcionais: Especifica todos os requisitos funcionais do sistema.
- **Seção 4** – Requisitos não funcionais: Especifica todos os requisitos não funcionais do sistema.

Convenções, termos e abreviações

1. Identificação dos Requisitos

Por convenção, a referência a requisitos é feita através do identificador do requisito, de acordo com o esquema abaixo: [Identificador do requisito]

Por exemplo, o requisito [RF016] está descrito em um bloco identificado pelo número [RF016]. RF significa requisito funcional e RN significa requisito não funcional.

2. Prioridades dos Requisitos

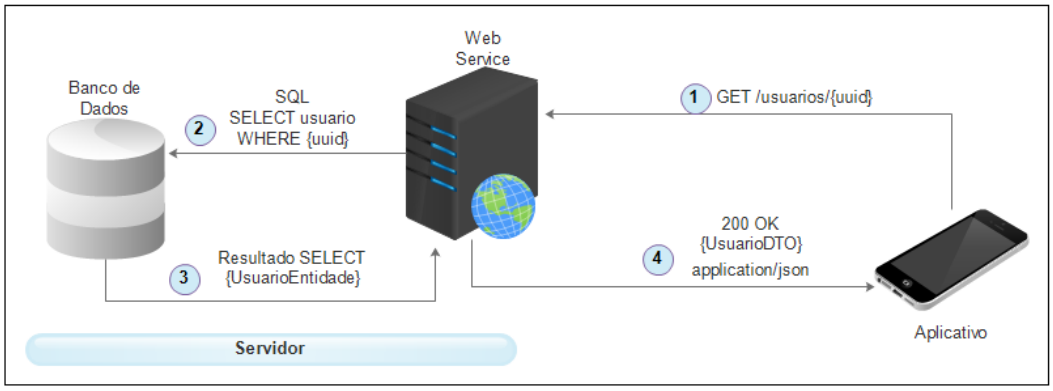
Para estabelecer a prioridade dos requisitos foram adotadas as denominações “essencial”, “importante” e “desejável”.

- **Essencial** é o requisito sem o qual o sistema não entra em funcionamento. Requisitos essenciais são requisitos imprescindíveis, que devem ser necessariamente implementados.
- **Importante** é o requisito sem o qual o sistema entra em funcionamento, mas de forma não satisfatória. Requisitos importantes devem ser implementados, mas, se não forem, o sistema poderá ser implantado e usado mesmo assim.
- **Desejável** é o requisito que não compromete as funcionalidades básicas do sistema. Logo, o sistema pode funcionar de forma satisfatória sem ele. Requisitos desejáveis são requisitos que podem ser deixados para versões posteriores do sistema, caso não haja tempo hábil para implementá-los na versão que está sendo especificada.

Seção 2

Descrição geral do sistema

O Web Service contribuirá para que os desenvolvedores do aplicativo da agenda campeira tenham um acesso padronizado ao banco de dados utilizado pelo aplicativo, tendo o acesso e a manipulação de dados facilitados pelo Web Service. Um exemplo da arquitetura contendo o *Web Service*, banco de dados e aplicativo podem ser vistos na Figura a seguir:



Para o desenvolvimento do Web Service foi utilizado o modelo arquitetural REST para definir os padrões de comunicação entre aplicativo e o *Web Service*. REST restringe que a comunicação seja feita com o protocolo de camada de aplicação HTTP. Isso quer dizer que, existem duas formas de mensagem: requisição e resposta. Requisição é como o aplicativo solicitará manipulações de dados no banco para o *Web Service*. Resposta será o resultado da solicitação feita pelo usuário, sendo enviado pelo Web Service ao aplicativo. Com isso, como padronizado pelo HTTP: será utilizado os verbos HTTP nas requisições para identificar a manipulação no banco solicitado e os códigos HTTP nas respostas para indicar o estado da solicitação. REST também restringe que as comunicações não tenham estado, logo, cada requisição é independente de requisições anteriores.

Foram planejadas classes para diferenciar as comunicações entre BD x *Web Service* e *Web Service* x aplicativo: Para a primeira serão usadas classes entidades (mapeadas com ORM) e para a segunda serão usadas classes DTO (mapeadas com propriedades JSON).

Abrangência e sistemas relacionados

O sistema de gerenciamento de banco de dados utilizado pela Embrapa é o PostgreSQL (esquemas "*public"* e "*agenda_campeira"*) e o aplicativo no qual o Web Service é focado será o da agenda campeira. O objetivo do Web Service é possibilitar uma comunicação entre esses dois sistemas de informação interoperáveis, efetuando uma troca de dados entre os mesmos.

Descrição dos usuários

O usuário do Web Service em si será o desenvolvedor do aplicativo da agenda campeira. Será ele que utilizará dos recursos do *Web Service* para prover ao usuário do aplicativo (considerado aqui como usuário final) suas funcionalidades.

1. Usuário - Desenvolvedor

O desenvolvedor utilizará o documento da modelagem dos recursos para compreender como são as chamadas para o *Web Service* e quais as respostas esperadas. Também poderá ver exemplos de requisições para testar o funcionamento de cada recurso. O documento de modelagem dos recursos também trará representações das tabelas no banco de dados manipuladas por cada recurso.

2. Usuário Final – Usuário do aplicativo

O usuário final será quem desfrutará das utilidades providas pelo Web Service. Como o *Web Service* será utilizado por um aplicativo, para obter seu acesso será necessário um cadastro de usuário (final) válido no banco de dados. Cada usuário final do aplicativo poderá realizar cadastros de fazendas e fazer os demais lançamentos para gestão de sua(s) propriedade(s) no aplicativo.

Seção 3

Requisitos funcionais

Nesta seção, serão apresentados todos os requisitos funcionais do sistema. Cada um dos requisitos é descrito em um bloco. O identificador do bloco contém seu nome e o número do requisito funcional.

Entrada no Aplicativo

Usuário final utilizará o login para poder realizar os lançamentos dos dados no aplicativo.

[RF001] Sistema de Login

O usuário deve iniciar sua sessão no aplicativo, utilizando seu usuário e senha. O propósito desta parte é manter o sistema seguro e evitar que alguma pessoa sem autorização consiga acessar. O *Web Service* terá que disponibilizar um recurso ou sub-recurso para poder criar um usuário e posteriormente o validar a partir dos dados enviados pelo aplicativo.

Usuários envolvidos: Desenvolvedor e usuário final.

Prioridade: ■ Essencial □ Importante □ Desejável

Backup

[RF002] Backup

Caso o usuário utilize o aplicativo em outro celular ou perca os dados das últimas sessões, o Web Service terá que prover uma maneira de enviar todos os dados dos lançamentos de um usuário com login válido para o aplicativo.

Usuários envolvidos: Desenvolvedor.

Prioridade: ■ Essencial □ Importante □ Desejável

Usuários envolvidos: Desenvolvedor.

Autenticação de Usuários

[RF003] Autenticação utilizando Tokens

Por REST utilizar comunicação sem estado, para se comunicar com o Web Service com maior segurança o aplicativo deverá enviar um token do usuário no cabeçalho de cada solicitação HTTP. O token deve ser gerado na criação do usuário ou no login, com o padrão disponibilizado pelo OAuth 2.

Usuários envolvidos: Desenvolvedor.

Prioridade: ■ Essencial □ Importante □ Desejável

Manipulação de dados

[RF004] Métodos HTTP Disponibilizados

Para simplificar os acessos, serão utilizados os seguintes métodos HTTP com os seguintes significados:

Para incluir uma entidade no banco será utilizado POST;

Para ler uma entidade do banco será utilizado GET;

Para modificar uma entidade no banco será utilizado PUT;

Para deletar uma entidade no banco será utilizado DELETE.

Usuários envolvidos: Desenvolvedor.

Prioridade: □ Essencial ■ Importante □ Desejável

[RF005] Respostas ao aplicativo após operações no banco

Sempre que houver uma inserção, leitura e modificação de alguma entidade no banco, o *Web Service* deverá enviar ao aplicativo a entidade com os dados criados/solicitados/alterados no corpo da resposta.

Usuários envolvidos: Desenvolvedor.

Prioridade: □ Essencial ■ Importante □ Desejável

4 Capítulo

Requisitos não funcionais

Esta seção contém os requisitos não funcionais do *Web Service*. Segue o mesmo esquema da seção 2 para nomenclaturas, com diferença apenas na sigla RF que agora será RN e com o nome dos tópicos que estão envolvidos estes requisitos não funcionais.

Requisitos não funcionais – Tópico

[RN001] Treinamento dos usuários – Treinamento/Usabilidade

Após a validação do Web Service, será realizado uma apresentação para os desenvolvedores do aplicativo, explicando as funcionalidades do *Web Service* e suas restrições, quando houver alguma.

Usuários envolvidos: Desenvolvedor.

Prioridade: □ Essencial □ Importante ■ Desejável

[RN002] Adequação do Web Service e documentação – Usabilidade/Confiança

A documentação deve estar atualizada com os recursos disponibilizados pelo Web Service.

Usuários envolvidos: Desenvolvedor.

Prioridade: ■ Essencial □ Importante □ Desejável

[RN003] Comunicação otimizada – Desempenho

A comunicação entre aplicativo e Web Service deve ser otimizada, utilizar o formato JSON para influenciar em menores tamanhos de mensagens.

Usuários envolvidos: Desenvolvedor e usuário final.

Prioridade: □ Essencial ■ Importante □ Desejável

[RN004] Segurança dos dados – Segurança

O Web Service deve criar e gerenciar os tokens para cada usuário. Com isso, cada usuário poderá alterar apenas as tabelas referentes às suas fazendas. Requisito atrelado ao **[RF003].**

Usuários envolvidos: Usuário final.

Prioridade: □ Essencial ■ Importante □ Desejável

[RN005] Ferramenta para modelagem dos Recursos – Usabilidade/Manutenção

Os recursos disponibilizados pelo *Web Service* devem estar documentados. No documento deve constar os endereços para os recursos, exemplos de requisições e de respostas. Para esse requisito será utilizado a ferramenta Swagger (https://editor.swagger.io/), a escolha desta ferramenta foi solicitada pela Embrapa.

Usuários envolvidos: Desenvolvedor.

Prioridade: □ Essencial ■ Importante □ Desejável

[RN006] Tecnologias para o desenvolvimento – Usabilidade/Manutenção

Os recursos disponibilizados devem contemplar todos as tabelas que se referem ao aplicativo. Para o desenvolvimento do Web Service serão utilizadas as seguintes ferramentas: Maven, Hibernate e WildFly Swarm. Também solicitadas pela Embrapa.

Usuários envolvidos: Desenvolvedor.

Prioridade: ■ Essencial □ Importante □ Desejável

APÊNDICE B – RECURSOS MODELADOS COM SWAGGER

Este apêndice apresenta os recursos modelados que compreendem a manipulação das tabelas dos diagramas ER apresentados no capítulo 4 deste trabalho. A Tabela 6 apresenta os recursos com operações além de escrita em conjunto com os recurso para sincronização e *backup*. E a Tabela 7, apresenta os recursos que aceitam apenas leitura que padronizam registros, descritos aqui como recursos de listagens.

Tabela 6 – Recursos Modelados com Swagger.

Método	**URI**	**Descrição**
Health		
GET	/health	Retorna o estado da aplicação
Atividades		
POST	/atividades	Inclui uma atividade.
GET	/atividades/{uuid}	Retorna uma atividade.
PUT	/atividades/{uuid}	Atualiza uma atividade.
DELETE	/atividades/{uuid}	Exclui uma atividade.
Backup e Sincronização		
GET	/backup/fazendas/{uuid}	Lista de fazendas por usuário.
GET	/backup/origens-destinos/{uuid}	Lista de ODs por fazenda.
GET	/backup/atividades/{uuid}	Lista de atividades por fazenda.
GET	/backup/lotes/{uuid}	Lista de lotes por ODs.
GET	/backup/pesagens/{uuid}	Lista de pesagens por ODs.
GET	/backup/movimentos/{uuid}	Lista de movimentos por ODs.
GET	/backup/estoques/{uuid}	Lista de estoques por ODs.
GET	/backup/nascimentos/{uuid}	Lista de nascimentos por ODs.
Estoques		
POST	/estoques	Inclui um estoque.
GET	/estoques/{uuid}	Retorna um estoque.
PUT	/estoques/{uuid}	Atualiza um estoque.
DELETE	/estoques/{uuid}	Exclui um estoque.
Fazendas		

Continua na próxima página

Tabela 6 – Recursos Modelados com Swagger. Continuação da página anterior

Método	URI	Descrição
POST	/fazendas	Inclui uma fazenda.
GET	/fazendas/{uuid}	Retorna uma fazenda.
PUT	/fazendas/{uuid}	Atualiza uma fazenda.
DELETE	/fazendas/{uuid}	Exclui uma fazenda.
POST	/fazendas/associacao	Inclui uma associação.
PUT	/fazendas/associacao/{uuid}	Atualiza uma associação.
DELETE	/fazendas/associacao/{uuid}	Exclui uma associação.
Lotes		
POST	/lote	Inclui um lote.
GET	/lote/{uuid}	Retorna um lote.
PUT	/lote/{uuid}	Atualiza um lote.
DELETE	/lotes/{uuid}	Exclui um lote.
Movimentos		
POST	/movimentos	Inclui um movimento.
GET	/movimentos/{uuid}	Retorna um movimento.
PUT	/movimentos/{uuid}	Atualiza um movimento.
DELETE	/movimentos/{uuid}	Exclui um movimento.
Nascimentos		
POST	/nascimentos	Inclui um nascimento.
GET	/nascimentos/{uuid}	Retorna um nascimento.
PUT	/nascimentos/{uuid}	Atualiza um nascimento.
DELETE	/nascimentos/{uuid}	Exclui um nascimento.
Origens-Destinos		
POST	/origens-destinos	Inclui uma origem ou destino.
GET	/origens-destinos/{uuid}	Retorna uma origem ou destino.
PUT	/origens-destinos/{uuid}	Atualiza uma origem ou destino.
DELETE	/origens-destinos/{uuid}	Exclui uma origem ou destino.
Pesagens		
POST	/pesagens	Inclui uma pesagem.
GET	/pesagens/{uuid}	Retorna uma pesagem.

Continua na próxima página

Tabela 6 – Recursos Modelados com Swagger. Continuação da página anterior

Método	URI	Descrição
PUT	/pesagens/{uuid}	Atualiza uma pesagem.
DELETE	/pesagens/{uuid}	Exclui uma pesagem.
Usuários		
POST	/usuarios	Inclui um usuário, retorna um token.
GET	/usuarios/{uuid}	Retorna um usuário.
PUT	/usuarios/{uuid}	Atualiza um usuário.
DELETE	/usuarios/{uuid}	Exclui um usuário.
POST	/usuarios/autenticacoes	Autentica um usuário, retorna um token.

Fonte: Autor (2019).

Tabela 7 – Recursos de Listagem.

Método	URI	Descrição
Países		
GET	/listagem/paises	Retorna uma lista de países.
Estados-Ufs		
GET	/listagem/ufs/{paisId}	Retorna uma lista de estados por país.
Municípios		
GET	/listagem/municipios/{ufId}	Retorna uma lista de municípios de um estado.
Definições		
GET	/listagem/definicoes	Retorna uma lista de definições.
Tipos-usuarios		
GET	/listagem/tipos-usuarios	Retorna uma lista de tipos de usuários.
Níveis		
GET	/listagem/niveis	Retorna uma lista de níveis de usuários.
Tarefas		
GET	/listagem/tarefas	Retorna uma lista de tarefas.
Manejos		

Continua na próxima página

Tabela 7 – Recursos de Listagem. Continuação da página anterior

Método	URI	Descrição
GET	/listagem/manejos	Retorna uma lista de manejos.
Obs-Nascimentos		
GET	/listagem/obs-nascimento	Retorna uma lista de observações de nascimento.
Categorias-Animais		
GET	/listagem/categorias-animais	Retorna uma lista de categorias de animais.
Raças-Mapa		
GET	/listagem/racas-mapa	Retorna uma lista de raças de animais.
Origens-Destino-Sistema		
GET	/listagem/origens-destino-sistema	Retorna uma lista de origens ou destinos padronizados pelo sistema.

Fonte: Autor (2019)