UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

JOÃO PAULO KUWANO

DESENVOLVIMENTO DE UM WEBSITE DE ACOMPANHAMENTO E ADOÇÃO DE ANIMAIS DE RUA

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2018

JOÃO PAULO KUWANO

DESENVOLVIMENTO DE UM WEBSITE DE ACOMPANHAMENTO E ADOÇÃO DE ANIMAIS DE RUA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, do Departamento de Informática da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. MSc. Rogério Ranthum

PONTA GROSSA

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE UM WEBSITE DE ACOMPANHAMENTO E ADOÇÃO DE ANIMAIS DE RUA

por

JOÃO PAULO KUWANO

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 06 de Novembro de 2018 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. MSc. Rogério Ranthum
Orientador

Simone de Almeida
Membro titular

Geraldo Ranthum
Membro titular

Prof(a). Dra. Helyane Bronoski Borges
Responsável pelo Trabalho de Conclusão de Curso

Prof. Dr. André Pinz Borges
Coordenador do curso

O dinheiro pode comprar um lindo cão,

mas só o amor pode fazê-lo abanar o rabo.

(Friedman, Kinky)

RESUMO

Kuwano, João Paulo. **Desenvolvimento de um website de acompanhamento e adoção de animais de rua**, 2018. 74 f. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

Segundo a Organização Mundial de Saúde, no Brasil existem cerca de 30 milhões de animais abandonados, em números que variam de 20 a 25% da população humana de cada cidade. Esses animais vivem na rua e dependem da ação voluntária de entidades de proteção animal e centros de zoonoses, além de adoção e doação de alimentos pela população. O objetivo desse trabalho é construir *uma web app* (aplicativo web) para interação desse público (voluntários e adotantes), que terão um sistema completo para publicar fotos do animal, encontrar adotantes, registrar doações, adoções e procedimentos veterinários. substituindo a informalidade e pulverização dessas informações em redes sociais e afins. Inicialmente a aplicação será modelada com foco nos alunos do Campus da UTFPR Ponta Grossa, porém o uso é irrestrito em qualquer lugar do Brasil, aceitando o cadastro de quaisquer novos animais e usuários. A expectativa é incentivar a adoção e cuidados com esses animais por meio da divulgação de seus dados no *app* apresentado.

Palavras-chave: Adoção. Animais de rua. *Web App*.

ABSTRACT

Kuwano, João Paulo. **Development of a website for monitoring and adopting street animals**, 2018. 74 p. Course Completion Work (Technology in Analysis and Development of Systems) - Federal Technology University. Ponta Grossa, 2018.

According to the World Health Organization, there are about 30 million abandoned animals in Brazil, ranging from 20 to 25 percent of the population in each city. These animals live in the street and depend on the voluntary action of animal protection entities and centers of zoonoses, besides adoption and food donation by the population. The purpose of this work is to build a web app for the interaction of this public (volunteers and adopters), who will have a complete system to publish photos of the animal, find adopters, register donations, adoptions and veterinary procedures. replacing the informality and pulverization of this information in social media. Initially the application will be modeled with focus on the students of the Campus of UTFPR Ponta Grossa, however the use is unrestricted anywhere in Brazil, accepting the registration of any new animals and users. It is expected to encourage adoption and care of these animals by disclosing their data in the app presented.

Palavras-chave: Adoption. Street animais. *Web App*

LISTA DE ILUSTRAÇÕES

Figura 1: Requisição Cliente-Servidor 17
Figura 2: Cache em Servidor REST 18
Figura 3: Requisição GET 19
Figura 4: Requisição POST 19
Figura 5: verbos HTTP no REST 20
Figura 6: Resposta de um servidor REST usando HATEOAS 21
Figura 7: Sistema em camadas 22
Figura 8: Níveis de maturidade de Richardson 24
Figura 9: Aplicação Node 27
Figura 10: Event Loop 28
Figura 11: Callback Hell 29
Figura 12: Promise 30
Figura 13: Async / Await 31
Figura 14: Shadow DOM 35
Figura 15: Angular MVC 36
Figura 16: Bootstrap Dashboard 38
Figura 17: Materialize Dashboard 39
Figura 18: Angular Service 40
Figura 19: Modelo Conceitual de Banco de Dados 41
Figura 20: Modelo Lógico de Banco de Dados 42
Figura 21: Diagrama de casos de uso 43
Figura 22: Modelo Lógico do banco de dados 45
Figura 24: Registro de rotas do animal 48
Figura 25: Instância do *Express Router* 48
Figura 26: SQL Model 49
Figura 27: Envio de Fotos no Android 53
Figura 28: Home Page 54
Figura 29: Detalhes do Animal 54
Figura 30: Modal Component 55
Figura 31: Toasts 55
Figura 32: Angular HTTP Service 56
Quadro 1: JavaScript vs JQuery 26
Quadro 2: Funcionalidades gerais do sistema 44

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

ACID	*Atomicidade, Consistência, Isolamento e Durabilidade*
API	*Application Programming Interface*
CRUD	*Create, Read, Update and Delete*
DOM	*Document Object Model*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
JS	*Javascript*
JSON	*Javascript Object Notation*
SPA	*Single Page Application*
SGBD	*Sistema Gerenciador de Banco de Dados*
SSL	*Secure Socket Layer*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
XSS	*Cross-Site Scripting*

SUMÁRIO

1 INTRODUÇÃO 11
1.1 OBJETIVOS 12
1.1.1 Objetivo Geral 12
1.1.2 Objetivos Específicos 12
1.2 JUSTIFICATIVA 13
1.3 METODOLOGIA 14
1.4 ESTUTURA DO TRABALHO 15
2 REFERENCIAL TEÓRICO 16
2.1 REST 16
2.1.1 Arquitetura Cliente-Servidor 16
2.1.2 *Statless* (Sem Estado) 17
2.1.3 *Cache* 17
2.1.4 Interface Uniforme 20
2.1.5 Sistema Em Camadas 22
2.1.6 Código Sob Demanda 22
2.1.7 Modelo De Maturidade 23
2.2 JAVASCRIPT 24
2.3 NODEJS 26
2.3.1 *Callback* 29
2.3.2 *Promise* 29
2.3.3 *Async / Await* 30
2.4 SPA (APLICAÇÃO DE PÁGINA ÚNICA) 31
2.5 WEB COMPONENTS 32
2.5.1 HTML5 33
2.5.2 Custom Elements 34
2.5.3 Shadow Dom 34
2.5.4 HTML Templates 35
2.6 ANGULAR 36
2.6.1 Typescript 37
2.6.2 Components 37
2.6.3 Bootstrap 38
2.6.4 Materialize 39
2.6.5 Serviços 40
2.6.6 Módulos 40
2.7 BANCOS DE DADOS RELACIONAIS 40
2.7.1 Modelagem De Dados 41
3 DESENVOLVIMENTO 43
3.1 VISÃO GERAL DO SISTEMA 43

3.2 BANCO DE DADOS 45
3.3 SERVIDOR 46
3.3.1 Configuração Do *Package.Json* 46
3.3.2 Módulo Principal 47
3.3.3 Funcionalidades Do Servidor 50
3.4 FRONT-END 51
3.4.1 Componentes E Telas Principais 53
3.4.2 Comunicação Com A API 56
3.5 PUBLICAÇÃO DO *APP* EM AMBIENTE WEB 56
4 CONCLUSÃO 57
4.1 CONSIDERAÇÕES FINAIS 57
4.2 TRABALHOS FUTUROS 57
REFERÊNCIAS 58
APÊDICE A – CONFIGURAÇÃO DO EXPRESS 62
APÊNDICE B – DOCUMENTAÇÃO DA API 64
ANEXO A: TERMO DE RESPONSABILIDADE SOBRE ADOÇÃO DE ANIMAIS 73

1 INTRODUÇÃO

No Brasil, cerca de metade dos 65 milhões de domicílios têm um cachorro ou gato, totalizando mais de 70 milhões de animais de estimação. Mais do que bichos domésticos, esses animais podem ser considerados membros da família (FARACO, 2008), proporcionando uma série de benefícios físicos e psicológicos, como redução do estresse, depressão e aumento da autoestima.

Os que não têm um lar acabam vivendo nas ruas, sofrendo uma série de dificuldades diariamente, como risco de atropelamento, agressões, envenenamento, doenças, brigas e desnutrição (SANTOS, 2015). Os principais motivos para abandono de animais são: inadequação às regras do condomínio, ladrar constante (em caso de cães), gravidez do animal, falta de tempo para cuidar, agressividade e nova criança na família (CARDOSO, 2013).

O suporte a esses animais é prestado majoritariamente pelo: (i) poder público, na Figura dos centros de zoonoses e controle de doenças, que os recolhem das ruas e abriga em canis e centros de recolhimento e (ii) por órgãos não governamentais e instituições de caridade que dependem na maioria das vezes de trabalho voluntário para desempenhar as mesmas funções. Em ambos os casos o objetivo desses órgãos é castrar os animais para diminuir sua proliferação nas ruas e disponibiliza-los para adoção, seja nos próprios centros ou feiras de animais (DE PAULA, 2012).

Em locais de aglomeração desses animais, como escolas, universidades, zonas de comércio de alimentos é comum que hajam voluntários que se comprometem em receber doação de ração, levar ao veterinário para realizar castração, vacinação e divulga-los em redes sociais para encontrar adotantes. O problema desse sistema é a falta de divulgação e organização das informações, que são quase sempre anunciadas informalmente em mídias sociais, dificultando a busca de quem procura um animal de perfil específico ou deseja ver fotos e comentários sobre ele, por exemplo.

A motivação dos voluntários que adquirem pra si a responsabilidade de serem cuidadores temporários de bichos abandonados é unicamente o interesse pelo bem-estar destes. A escolha é justificada pela empatia que o ser humano desenvolve por animais sencientes, culminando na legislação de proteção animal, que aumentou

significativamente nos últimos 30 anos e se baseia em protege-los da fome, doenças, dor e estresse (JOFFILY, 2013).

Esse trabalho se propõe a auxiliar esse tipo de público (voluntários) a organizar suas atividades, registrando numa plataforma web suas interações com os animais, como sugestão de nome, fotos, informação de serviços veterinários prestados, dar a descrição deles ao público potencial adotante e um canal de comunicação direta entre os usuários do *site*.

A implementação do sistema em um domínio *web* é uma escolha natural e previsível, uma vez que essa tecnologia disponibiliza conteúdo em um formato simples, poderoso e universal. O uso da *web* atualmente é imprescindível nas relações de trabalho, comércio, serviços e entretenimento (FREIRE, 2008).

1.1 OBJETIVOS

Neste capítulo são especificados os objetivos gerais e específicos deste trabalho

1.1.1 Objetivo Geral

O objetivo é o desenvolvimento de um aplicativo *web* (*web app*), responsivo para *desktop* e *mobile*, que permita cadastrar e localizar animais de rua e adotantes destes, registrando também atividades voluntárias de suporte a esses animais.

1.1.2 Objetivos Específicos

Para tal propósito, especifica-se um sistema atendendo os seguintes requisitos:

- Cadastrar voluntários e adotantes, com perfil (página pessoal).
- Cadastrar animais, com foto e informação de onde vivem, seu tamanho, cor, sexo, temperamento.
- Possibilitar envio de mais fotos a qualquer momento, assim como sugerir um nome a cada animal.

- Registrar visita ao veterinário, e os procedimentos realizados, como castração e desvermifugação.
- Escolha de administradores do sistema para cada local cadastrado.
- Permitir a troca de mensagens entre os usuários do sistema.
- Aprovar a adoção do animal mediante o preenchimento de um formulário e concordância com um termo de responsabilidade.

1.2 JUSTIFICATIVA

Os *websites* disponíveis atualmente para adoção de animais têm, basicamente, um sistema de postagem de conteúdo, em que o usuário realiza o cadastro, *login* e anuncia o animal que está sob seus cuidados fornecendo a descrição mais detalhada possível junto de um telefone ou *email* para contato. O visitante do *site* geralmente pode filtrar os resultados e pesquisar por local, sexo ou tamanho. Os sistemas de acompanhamento da vida do animal, registro de visitas médicas, doações, etc. são privativos de canis, *petshops* ou ONGs, portanto não disponíveis publicamente.

Considerando estes fatores, o trabalho é justificado pela facilitação no processo de adoção e acompanhamento dos animais de rua, principalmente aos cuidadores temporários, que neste *website* terão uma interface própria de registro de suas atividades. Os adotantes contarão com recursos de busca e ferramentas de interação com outros usuários. A motivação principal é a divulgação dos animais para adoção e um sistema unificado para controle dessas adoções, assim como os procedimentos veterinários, doações, fotos com os usuários, escolha de nome, etc.

A escolha desse tema leva em conta a ausência de websites disponíveis na Internet que contemplem os requisitos apresentados no tópico 2. Em resumo, os *websites* disponíveis não têm um canal de troca de mensagens entre os usuários e/ou as informações são apresentadas de forma incompleta ou pulverizada em várias páginas, o que dificulta a busca dos adotantes.

1.3 METODOLOGIA

A metodologia de desenvolvimento utilizada é a Kanban, um método ágil que prioriza o desenvolvimento das tarefas imediatamente necessárias, através da organização e comunicação constante da equipe.

O kanban originalmente é um estratégia de organização de equipes em ambiente industrial existente desde 1940. Foi criada pela empresa automotiva Toyota inspirando-se no sistema de reposição de estoque dos supermercados, em que as prateleiras são sempre carregadas com produtos suficientes para atender a demanda imediata. Essa prática economiza tempo considerável na administração do estoque (ATLASSIAN, 2018), uma vez que elimina os excessos de material disponível.

Kanban era o nome do cartão que transitava entre as equipes no chão de fábrica. A cada caixa de materiais que se esgotava na linha de produção um "kanban" era passado para o depósito com a quantidade exata necessária. O depósito também emitia seu kanban para o fornecimento e assim por diante (ATLASSIAN, 2018).

Sua técnica é conhecida como *Just In Time* (JIT) e, como metodologia de desenvolvimento de software, consiste em sinalizar para a equipe, através de anotações em cartão (uma parede de *post-its* por exemplo), quais tarefas são necessárias para concluir determinada etapa do desenvolvimento e marcá-las de acordo com seu estado, como "a fazer", "trabalhando nisso" ou "finalizado".

O desenvolvimento deste *app* leva em consideração padrões de responsividade providos por *frameworks* específicos e segurança através de *JSON Web Tokens,* o protocolo mais popular e seguro disponível (VELASCO, 2018). Será usado o padrão de arquitetura *REST*, e separação do sistema em *backend* e *frontend.* Todo o tráfego de dados da aplicação é realizado por *JSON* pelo protocolo de Internet HTTP.

O *framework* escolhido para a API é o NodeJS, em linguagem JavaScript. Sua orientação a eventos, assincronismo e possibilidade de lidar com múltiplas conexões simultâneas o tornam altamente escalável (NODEJS, 2018) e a linguagem

JavaScript diminui a curva de aprendizagem em relação às demais tecnologias da web.

O *frontend* da aplicação é desenvolvido em Angular, um *framework* voltado à construção de *Single Page Applications* (SPA). É capaz de lidar com grandes quantidades de dados em alta performance, a mesma aplicação pode ser utilizada em *desktop*, *mobile* e *native mobile* (ANGULAR, 2018). Sua linguagem é o TypeScript, um *superset* do JavaScript que adiciona, principalmente, tipagem estática, classes, interfaces e carregador de módulos próprio, permitindo a verificação de sintaxe em tempo de compilação, *autocomplete* e validação de tipos, recursos não disponíveis em JavaScript puro.

A estilização responsiva da aplicação é designada ao Bootstrap, a mais popular biblioteca de componentes HTML, CSS e JavaScript (BOOTSTRAP, 2018) e a persistência de dados ao MariaDB, SGBD relacional e open-source (MARIADB, 2018).

1.4 ESTUTURA DO TRABALHO

O trabalho é dividido em 3 capítulos: No primeiro é feita uma introdução ao tema, e apresentados seus objetivos, justificativa e metodologia de desenvolvimento.

O capítulo 2 consiste da fundamentação teórica das tecnologias utilizadas na aplicação, ou seja, o estilo arquitetural REST, linguagem de programação JavaScript, e *framework*s NodeJS e Angular.

No terceiro capítulo é apresentado o desenvolvimento do trabalho, sua estrutura, a implementação dos servidores de *back* e *front-end* e do banco de dados, assim como o fluxo de utilização do sistema, e documentação e implantação dos projetos em ambiente Web.

Por fim, o capítulo 4 contém as conclusões e considerações finais do trabalho, e sugestão de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta a fundamentação teórica acerca das tecnologias empregadas no desenvolvimento do trabalho, suas técnicas, linguagens de programação e padrões de projeto.

2.1 REST

Estilo arquitetural proposto por Roy Fielding em sua tese “Architectural Styles and the Design of Network-based Software Architectures” (FIELDING, 2000). O modelo é apresentado com o nome de “*Representational State Transfer* – REST” (em tradução livre, Transferência de Estado Representacional), em que o autor se propõe a apresentar um estilo de arquitetura condizente com a Web e o protocolo HTTP, são enumerados 6 requisitos, ou restrições, que compõem o REST (FRANCISCO, 2016).

2.1.1 Arquitetura Cliente-Servidor

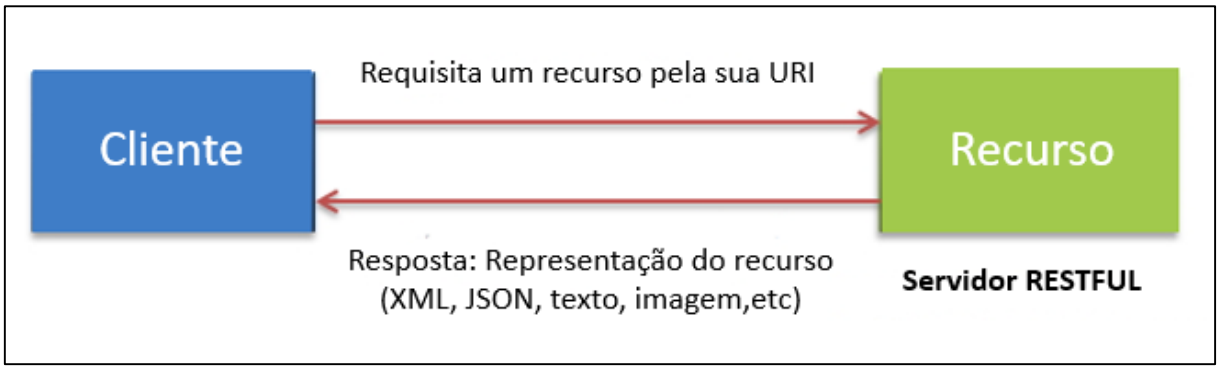
Arquitetura na qual um servidor responde requisições de um cliente usando dados em formato fixo (FRANCISCO, 2016). A natureza real desses dados, que podem ser obtidos de um banco de dados, um servidor externo, etc. fica oculta ao cliente, que recebe apenas a representação destes.

O conceito de representação é importante ao REST pois permite separar responsabilidades das partes da aplicação. Uma vez que o cliente não é responsável por conhecer a estrutura dos dados no servidor, que não é responsável pela utilização e apresentação desses dados, as duas partes podem evoluir e escalar de maneira independente (RODRIGUES, 2009), portando interfaces a diferentes plataformas sem grandes custos.

A arquitetura Cliente-Servidor prevê o encapsulamento destas duas partes, portanto o servidor distribui os mesmos dados formatados em JSON, XML ou outros qualquer seja o cliente, que também pode consumir recursos de outros servidores sem prejuízo ou necessidade de alteração de sua estrutura.

A Figura 1 apresenta uma requisição *web* no modelo cliente-servidor em REST. Neste exemplo, o cliente pode ser uma aplicação *web*, *desktop* ou aplicativo *mobile*.

Figura 1: Requisição Cliente-Servidor



Fonte: Adaptado de (SHAH, 2017)

2.1.2 *Statless* (Sem Estado)

Toda requisição é processada de maneira independente. O servidor não tem conhecimento sobre o estado de qualquer aplicação cliente, seus dados salvos ou suas funções. A requisição deve ter todas as informações necessárias para que seja processada, sem depender de dados previamente ou posteriormente enviados. Essa abordagem permite liberar os recursos computacionais de cada requisição logo após sua finalização, sem a necessidade de salvar qualquer estado (FRANCISCO, 2016).

2.1.3 *Cache*

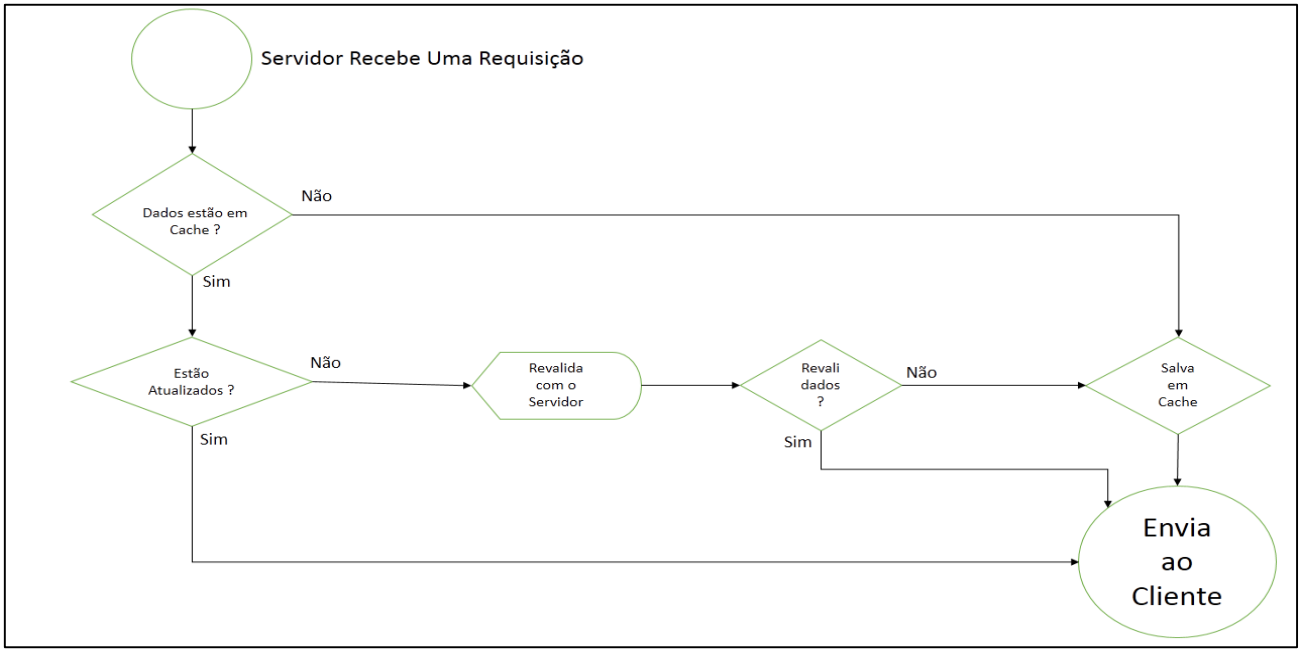
Uma das restrições de uso opcional no REST é o *cache*, que consiste no armazenamento de dados pelo cliente, de forma a reduzir a necessidade de requisições ao servidor, otimizando a rede e consultando recursos de forma mais rápida, em armazenamento local. O *cache* pode ser usado também no servidor que salva recursos frequentemente acessados em memória ou disco, respondendo o cliente em um tempo menor do que se fizesse uma consulta ao banco de dados ou servidor externo (VELASCO, 2018).

A limitação dessa abordagem é que só podem ser ‘cacheados’ métodos idempotentes (que não alteram os dados e retornam respostas idênticas ao longo do tempo). Em termos REST, consultas GET podem ser salvas em *cache* enquanto alteração de dados com POST, PUT, DELETE são exclusivamente processadas pelo servidor, e uma vez que alteram o estado dos dados não podem ser ‘reaproveitadas’ pelo *cache*.

Uma preocupação quanto à adoção de *cache* é a confiabilidade dos dados, que exige verificação constante de sua integridade, já que o servidor pode ter o estado alterado depois de seu salvamento em *cache*.

A Figura 2 apresenta um fluxograma para o uso de *cache* no servidor, que delimita seu uso à dados atualizados e validados.

Figura 2: Cache em Servidor REST

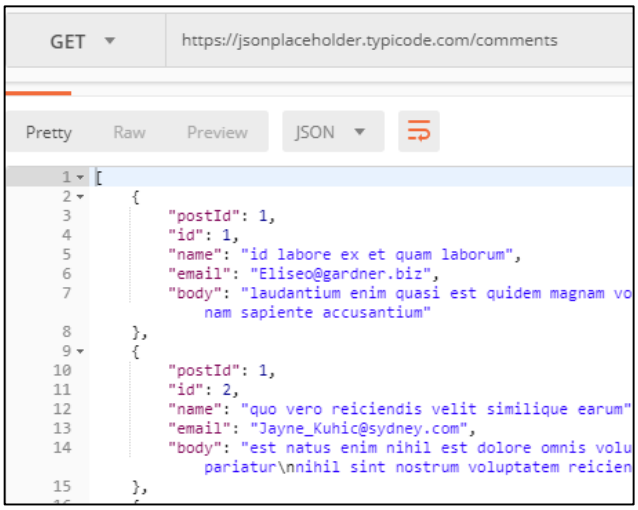


Fonte: Autoria própria

A Figura 3 apresenta um exemplo de requisição do tipo GET a um servidor REST, que retorna uma lista de comentários de um blog. Este resultado poderia ser salvo em *cache*, pois é uma consulta, não importa quantas vezes se faça essa

requisição o mesmo será igual desde que o estado do servidor não tenha sido alterado por outros agentes desse sistema.

Figura 3: Requisição GET



Fonte: Autoria própria

A Figura 4 apresenta uma requisição do tipo POST, que solicita a criação de um novo recurso no servidor. Esta requisição não pode ser salva em *cache*, pois, cada solicitação desde tipo gera uma resposta única do servidor e diferente a cada chamada.

Figura 4: Requisição POST

Fonte: Autoria própria

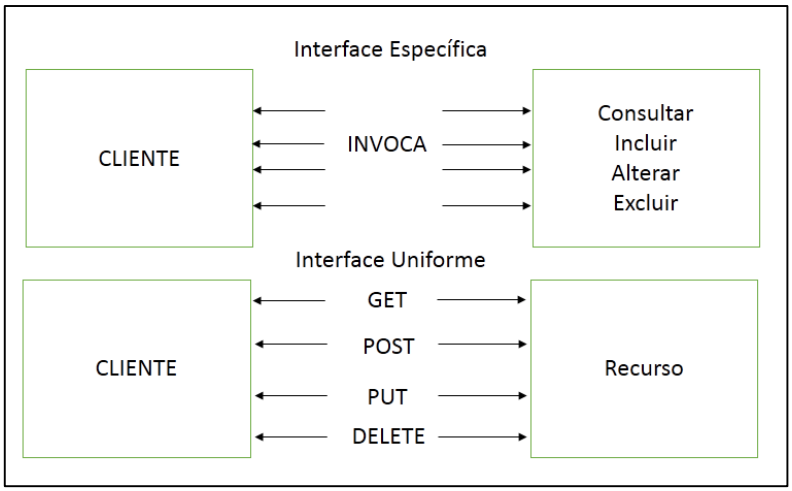
2.1.4 Interface Uniforme

Consiste em aplicar o princípio de engenharia de software da generalidade (MACHADO, 2014) através de 4 restrições: identificação dos recursos, manipulação de recursos pelas suas representações, mensagens auto descritivas e hipermídia como motor de estado (FRANCISCO, 2016). Apesar da aplicação potencialmente perder desempenho por não haver otimização na transferência de informações levando as contas as especificidades de cada projeto, a generalização simplifica a arquitetura do sistema e a exposição dos recursos.

A identificação dos recursos é dada por URL's, que indicam a localização dos recursos, e.g. *utfpr.edu.br* em um domínio *Web* e URI's, que identificam o recurso pelo protocolo de acesso, localização e nome, e.g. *http://utfpr.edu.br/dainf*. As operações do sistema são especificadas através dos verbos HTTP. Obs.: O uso do HTTP não é compulsório, porém a Internet toda é estruturada nesse protocolo, que implementa passagem de metadados por cabeçalhos, envio seguro de dados por camada de criptografia SSL e todas as páginas *Web* trafegam por esse protocolo.

A Figura 5 apresenta os principais verbos HTTP e suas operações correspondentes em um sistema REST.

Figura 5: verbos HTTP no REST



Fonte: Adaptado de (MACHADO, 2014)

Por mensagens auto descritivas entende-se que o conteúdo da mensagem está dissociado dos metadados que a descrevem, através de pares chave-valor

comumente formatados em XML ou JSON (*JavaScript Object Notation*). Não há impedimento quanto ao uso de outros formatos (FRANCISCO, 2016).

Quanto ao gerenciamento de estados da aplicação, o REST usa o padrão HATEOAS (*Hypermedia As The Engine Of Application State*), que funciona como um motor de navegação do sistema. Como exposto na restrição de *statless*, toda requisição para um servidor REST é processada de maneira independente sem qualquer salvamento de estado do cliente, então, para fornecer informações de acesso à API sem que o cliente precise conhecer detalhes de sua implementação são fornecidos *hiperlinks* junto com os dados das respostas indicando ao cliente em quais serviços um recurso consultado pode ser utilizado, quais parâmetros são exigidos em cada um, qual a localização de um recurso criado, qual será a resposta de cada método, entre outros.

A Figura 6 apresenta um exemplo de HATEOAS para o verbo POST, que cria um recurso novo no servidor e informa sua localização ao cliente.

Figura 6: Resposta de um servidor REST usando HATEOAS

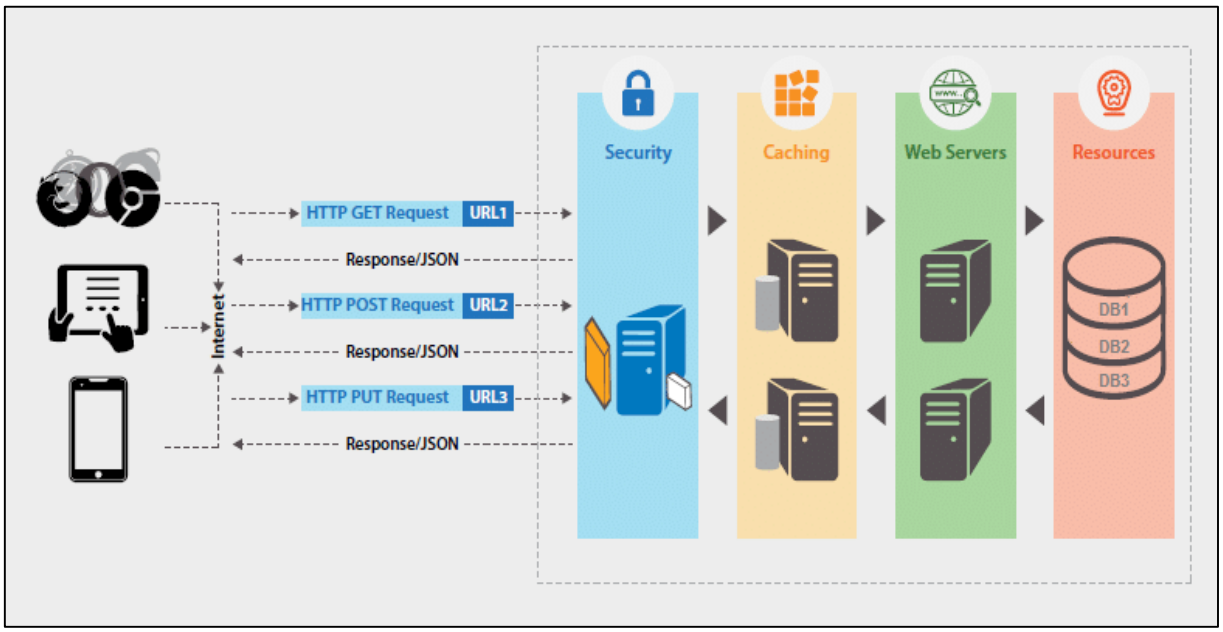
Fonte: Autoria própria

2.1.5 Sistema em camadas

A organização do sistema em uma hierarquia de camadas limita a ação de cada componente às camadas necessárias para sua operação, aumentando a segurança a partir do encapsulamento dos serviços e o desempenho da aplicação a partir do balanceamento de carga de processamento dos componentes (VELASCO, 2018).

A Figura 7 apresenta um exemplo de sistema em camadas composto de um *gateway* de segurança, um módulo de *cache*, banco de dados e o próprio servidor.

Figura 7: Sistema em camadas



Fonte: (DEEPAK, 2015)

2.1.6 Código sob demanda

É uma restrição facultativa, que compreende a disponibilização de um servidor REST diretamente ao cliente, através de um *Java Applet*, programa em Flash, JavaScript, etc. Essa solução proporciona ganhos nos requisitos de implementação e extensibilidade, já que a API será implementada no mesmo ambiente de produção do cliente, porém pode trazer prejuízos em visibilidade e desempenho, com o programa saindo do controle de um servidor específico.

2.1.7 Modelo de Maturidade

Considerando que REST é um estilo de arquitetura, não uma arquitetura em si, é possível implementar *Web Services* que cumprem apenas algumas restrições e ignoram outras. Richardson (2008) propôs em seu artigo "*The Maturity Heuristic*" (Maturidade Heurística) que a classificação dos serviços Web pode ser realizada em 4 níveis de acordo com sua adequação aos requisitos do REST.

No nível 0, chamado "pântano de ROX", está a aplicação que apenas utiliza o protocolo HTTP para tráfego de dados e descumpre todas as restrições, esta não é considerada REST de forma alguma. Neste nível se encontra, por exemplo, o RPC (chamada de procedimento remoto), em que o cliente acessa diretamente os métodos do servidor através de dados enviados por HTTP (FOWLER, 2010).

No nível 1 estão os "Recursos", que é a identificação de cada serviço da aplicação por uma URI específica. Ao invés de usar um simples *endpoint* para todos os pedidos ao servidor, neste nível cada tipo de recurso tem um *endpoint* e cada entidade desse tipo pode ser acessada e manipulada individualmente através de sua URL (FOWLER, 2010). Em um sistema que suporte cadastro de usuário, estes poderiam ser acessados em um *endpoint* "/usuarios" e distinguidos por um identificador, como "/usuarios/1".

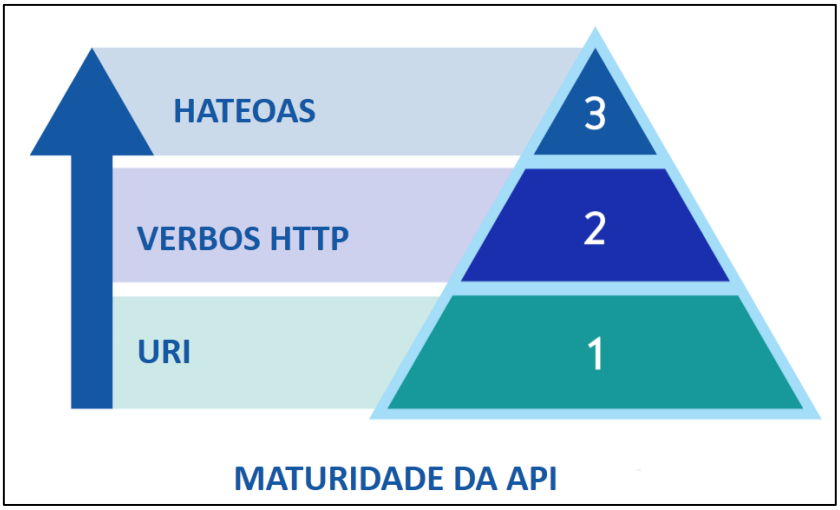
No nível 2, "Verbos HTTP", a ação que se quer realizar com os recursos é informada acessando os *endpoints* dos serviços e identificando a operação desejada através dos verbos HTTP (GET / POST / PUT / DELETE / PATCH, etc.).

GET é um verbo não seguro, uma vez que seus dados são enviados publicamente por parâmetros da URL, e usado no REST para consulta de dados. POST é destinado à inserção de dados e operações não comportadas em nenhum outro verbo. PUT atualiza completamente um recurso, isto é, o cliente envia todos os dados do recurso que quer atualizar, incluindo os que não foram modificados. PATCH trata da atualização parcial de um recurso, em que apenas os dados que se quer modificar são enviados.

O nível 3, "Controles Hipermídia", prevê a implementação do HATEOAS (Hipermídia como Motor de Estado), detalhes sobre esta restrição são apresentados na seção 2.1.4 (Interface Uniforme).

A Figura 8 apresenta uma pirâmide com os níveis de maturidade REST de uma API.

Figura 8: Níveis de maturidade de Richardson



Fonte: Adaptado de (DOERRFELD, 2018)

2.2 JAVASCRIPT

A linguagem de programação JavaScript foi lançada em 1995 com o objetivo de produzir conteúdo dinâmico, matéria inédita à época, para o navegador de Internet Netscape. O projeto é de autoria de Brendan Eich (ex-CEO e cofundador da Mozilla). Seu nome original era Mocha, foi alterado para Livescript e depois Javascript, antes mesmo do final de 1995. Essa escolha definitiva foi influenciada pela parceria da Netscape com a Sun, dona do também recém-lançado Java. As duas linguagens não têm qualquer relação direta. (DUARTE, 2015)

As principais características do JavaScript são (DUARTE, 2015):

- Linguagem Interpretada: Não existe um compilador que traduza o programa para código de máquina, ele é interpretado por programas específicos (geralmente navegador de Internet) que executam cada linha do código.
- Funcional: O principal paradigma de programação do Javascript é o funcional. É possível invocar funções passando outras como parâmetro, atribuir funções à constantes e variáveis. A maioria dos programas desenvolvidos e a própria arquitetura da linguagem são baseados em funções.

- Tipagem Dinâmica: As variáveis podem mudar de tipo em tempo de execução.

Ao longo do tempo a linguagem ganhou várias funcionalidades a fim de melhorar a interatividade com o usuário e com as páginas web, dentre elas o AJAX (*Asynchronous JavaScript and XML*), que permite fazer requisições assíncronas via HTTP, recebendo dados de qualquer servidor externo e atualizando as páginas dinamicamente, sem recarregar. (DUARTE, 2015).

Um dos principais problemas do JavaScript era a falta de um compilador universal, cada *browser* mantinha de forma independente seu interpretador JS e vários “fabricantes” também lançavam de forma independente suas versões da linguagem. Essa incompatibilidade muitas vezes limitava o funcionamento da aplicação web a um único navegador (DUARTE, 2015). A solução foi o uso de bibliotecas compartilhadas entre os *browsers*, a mais famosa e amplamente utilizada até hoje é o JQuery.

JQuery é uma biblioteca JavaScript rápida e concisa que simplifica o tratamento do documento HTML, gerenciamento de *handlers* de eventos, animações e requisições Ajax. Pode ser usada importando seu arquivo JavaScript em um arquivo HTML da aplicação e a biblioteca ficará disponível para aquele HTML. Seu objetivo principal é promover compatibilidade entre os navegadores de Internet e reduzir funções e rotinas complexas em JavaScript em uma sintaxe mais simples e curta, acelerando o desenvolvimento (THOMAS, 2018).

Em resumo, estas são as principais funções do JQuery em uma aplicação *web* (COSTA, 2018):

- Fornecer interatividade com o usuário
- Realizar qualquer tipo de manipulação do DOM
- Realizar chamadas HTTP assíncronas (AJAX)
- Modificar e estilizar o conteúdo da página
- Simplificar e padronizar rotinas JavaScript

Uma comparação dos principais recursos do JavaScript e do JQuery é apresentada no Quadro 1.

Quadro 1: JavaScript vs JQuery

JavaScript	JQuery
Uma linguagem de programação fracamente tipada	Uma biblioteca JavaScript rápida e concisa
Uma linguagem de *script* para interação com interface e manipulação do conteúdo da página	Um *framework* que manipula eventos, animações, e requisições Ajax.
Uma linguagem interpretada	Usa os recursos providos pelo JavaScript para torna-lo mais fácil
Exige que o programador crie todos os seus *scripts*.	Possui uma grande quantidade de *scripts* prontos e disponíveis para serem usados.
Desenvolvedores precisam lidar com a compatibilidade entre navegadores, reescrevendo código ou lançando diferentes versões do programa	A própria biblioteca garante a compatibilidade
Não precisa incluir nenhum código externo para rodar em qualquer navegador que suporta JavaScript	Precisa incluir a URL da biblioteca ou o caminho do arquivo baixado na página HTML
Mais linhas de código	Menos linhas de código
Mais rápido para acessar o DOM	Adequado para operações complexas, diminuindo a propensão a erros e reduzindo o número de linhas de código.

Fonte: Adaptado de (THOMAS, 2018)

O relatório *Web Technologies of the Year 2017* aponta que o JQuery é utilizado em 97.1 % das aplicações JavaScript, totalizando 73.4 % das aplicações web do mundo (W3TECHS, 2018).

2.3 NODEJS

Uma vez que JavaScript é a “linguagem da *Web*”, isto é, usado na quase totalidade dos *sites* da Internet, o NodeJS, ou Node foi lançado por Ryan Dahl (2009) como um projeto para transportar essa linguagem para os servidores de aplicação, consistindo numa plataforma de desenvolvimento de servidores,

interpretada pelo *JavaScript Engine V8* do Google. Seus objetivos são (SOUSA, 2015):

- Produzir aplicações rápidas e escaláveis.
- Ter alta performance e consumir pouca memória, suportando processos de servidor de longa duração.

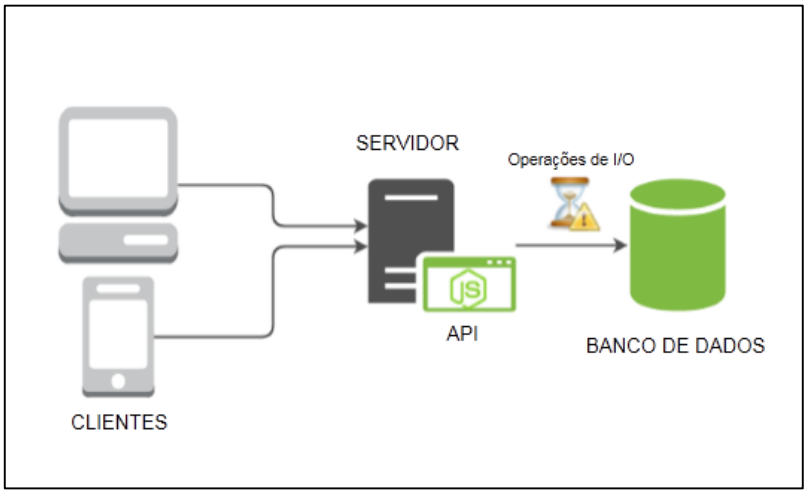
Para tal, a plataforma adota uma arquitetura baseada em 3 conceitos chave:

- I/O não bloqueante
- Programação assíncrona orientada a eventos
- *Single-thread*

Toda aplicação Node opera em *single-thread,* o que alivia o programador das complexidades de um sistema *multithreading* ao mesmo tempo que reduz o consumo de memória por requisição (SOUSA, 2015).

Um exemplo de servidor Node é apresentado na Figura 9:

Figura 9: Aplicação Node



Fonte: Adaptado de (MONOBE, 2017)

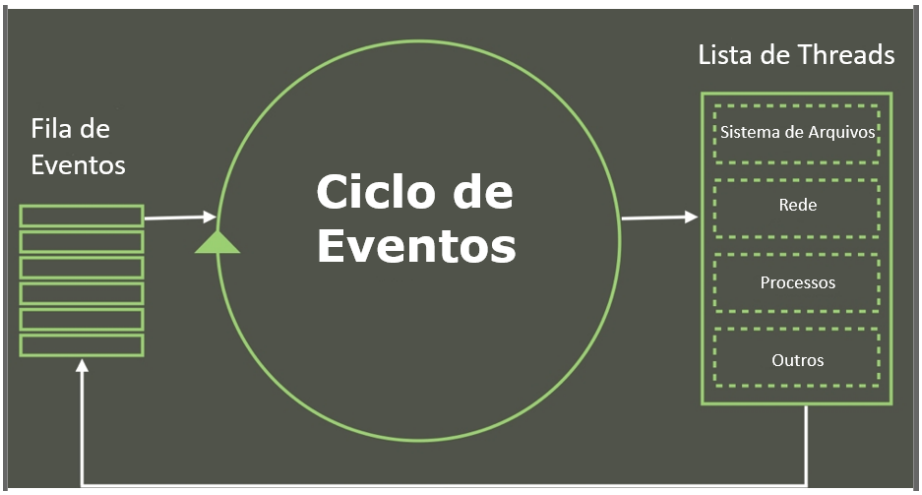
Como I/O (*input*/*output*) não bloqueante entende-se que cada operação de leitura ou escrita de dados não bloqueia o fluxo de execução do programa, que realiza outras operações simultaneamente. Em um sistema distribuído essa característica permite tratar várias requisições ao mesmo tempo sem aumentar o gasto computacional, tornando o Node leve e eficiente (SCHROEDER, 2017).

A implementação *non-blocking* I/O é realizada pelo Ciclo de Eventos, que distribui cada operação de dados para uma de suas 4 *threads* da Lista de *threads* (aqui, *thread* não se refere a módulos executáveis em processadores independentes, mas sim a programas do Node executados dentro de uma mesma *thread* de processamento do *V8 Engine*):

- Sistema de Arquivos: manipula arquivos
- Rede: comunicação na rede
- Processos: abre processos externos
- Outros: Demais operações

As *threads* são organizadas em uma fila chamada Fila de Eventos, que processa as *callbacks* de cada evento, ou o código a ser executado depois de sua conclusão. A Figura 10 apresenta o Ciclo de Eventos e a lista de Threads do Node:

Figura 10: Event Loop



Fonte: Adaptado de (Monobe, 2017)

A utilização de eventos assíncronos garante que várias funções possam ser executadas simultaneamente e de maneira independente, e são usadas em um programa Node sob, basicamente, 3 formas (MONOBE, 2018):

- *Callbacks*
- *Promises*
- *Async / Await*

2.3.1 *Callback*

Callbacks são usados nas chamadas *high-order functions*, que são funções que recebem uma ou mais funções como argumento e retornam estas como resultado de sua operação. Estas funções enviadas por parâmetro, ou argumento, são chamadas de *callbacks* (MONOBE, 2018).

O problema das *callbacks* é que operações complexas, que demandam várias sub-operações em ordem específica podem gerar um código com encadeamento de funções, ilegível e verboso, fenômeno conhecido como *callback hell* e ilustrado na Figura 11:

Figura 11: Callback Hell

```
GetUser(function(user){
    GetProfile(user, function(profile){
        GetAccount(profile, function(acc){
            GetReport(acc, function(report){
                SendStatistics(report, function(e){
                    //código omitido
                })
            })
        })
    })
})
```

Fonte: Adaptado de (DHANANI, 2017)

2.3.2 *Promise*

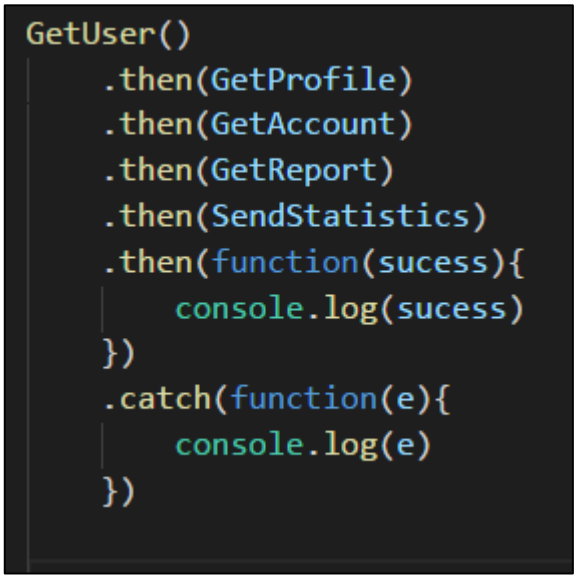
Com o objetivo de gerar códigos mais “limpos” e principalmente encapsular os *callbacks* em um objeto especializado nesse tipo de função foram criadas as *Promises*.

As *promises* são objetos que têm métodos e eventos que permitem ao programador conhecer o estado de execução das suas funções e ter a certeza da resolução bem-sucedida ou malsucedida da *callback*. Antes, esta era uma apenas uma definição lógica criada pelo próprio programador a partir da ordem de

parâmetros retornados ou um tratamento específico que se fazia com eles. Agora, delega-se à *promise* a captura e tratamento de erros e resultados das *callbacks* (MONOBE, 2018).

O código da Figura 12 implementa a mesma funcionalidade da Figura 11, usando *promise* ao invés de *callback*.

Figura 12: Promise



Fonte: Adaptado de (DHANANI, 2017)

2.3.3 *Async / Await*

Simplificando as *promises* e permitindo tratar um código assíncrono de "maneira síncrona", foi implementado no *EcmaScript* 7 o padrão de projeto *async / await,* que substitui a execução do *then* pela palavra *await* antes da chamada da função assíncrona (MOZILLA, 2018). A função é "pausada" até que essa promise seja resolvida, seu resultado pode ser atribuído diretamente a uma varável e a captura de erros feita em blocos *try/catch*.

Toda *promise* pode ser "*awaited"* (esperada) e as principais vantagens dessa sintaxe são a simplificação do código e a possibilidade de intercalar chamadas síncronas com assíncronas, que serão sempre resolvidas na ordem de sua declaração, eliminando o encadeamento de funções.

A Figura 13 apresenta a mesma funcionalidade das Figuras 11 e 12, usando o padrão *async* / *await.*

Figura 13: Async / Await

```
async function sendAsync(){
    let user = await GetUser();
    let profile = await GetProfile(user);
    let account = await GetAccount(profile);
    let report = await GetReport(account);

    let send = SendStatistics(report);

    console.log(send);
}
```

Fonte: Adaptado de (DHANANI, 2017)

2.4 SPA (Aplicação de Página Única)

Uma SPA é uma aplicação web que serve uma única página HTML, renderizada por um servidor JavaScript, que dinamicamente substitui os componentes disponíveis nessa página.

A experiência para o usuário é melhorada pois o carregamento de novas telas é quase sempre instantâneo (FARIA, 2018). Outra vantagem é a redução no tráfego de dados da rede, que o servidor desta aplicação não precisa enviar páginas completas com HTML, CSS, JS, etc. mas apenas dados que servem aos componentes solicitados.

Como desvantagem existe a provável demora no carregamento inicial da aplicação, já que precisa solicitar e renderizar uma grande quantidade de dados que contêm os componentes básicos da *app*. Também as SPA's têm dificuldade na indexação em motores de busca (SEO), por mais que este geralmente não seja um requisito de projeto para SPA's.

O uso recomendado de SPA's é na criação de *Webapps*, uma definição semântica e não necessariamente objetiva que a separa dos *Websites* pelo nível de interatividade. (FARIA, 2018)

Websites são informacionais, como sites de notícia, educacionais, blogs, lojas, etc., nesses casos uma arquitetura tradicional com páginas individualizadas é suficiente na maioria das vezes.

As *Webapps* são totalmente interativas com conteúdo sob controle do usuário, como páginas de e-mail, sistemas empresariais, gerenciador de arquivos em nuvem, etc. *Google Drive, Docs, Maps* são exemplos de *Webapps*, e a organização destes em páginas é inviável, por isso adota-se a SPA como um sistema dividido em partes (componentes).

Em resumo, uma série de fatores contribuem para o sucesso das SPAs e sua adoção em casos cada vez mais amplos:

- Considerável incremento na experiência do usuário, que não precisar esperar o *reload* das páginas pra usar o sistema.
- Evolução dos *frameworks,* que permitem delegar mais responsabilidade à aplicação do cliente (*front-end*), como a geração e renderização completa do HTML, CSS e JavaScript.
- melhorias de *hardware* nos equipamentos médios do mercado que suporta aplicações cada vez mais robustas
- facilidade de manutenção do código, separado em *front-end* e *back-end* e troca de dados em texto JSON ou XML
- melhoria de escalabilidade dos servidores que numa SPA tem o *back-end* como responsabilidade única.

2.5 WEB COMPONENTS

Web Components são um conjunto de tecnologias que permitem criar componentes customizados e reutilizáveis em aplicações *Web*, com suas funcionalidades encapsuladas e isoladas de outras partes do código (MOZILLA, 2018). Um *web component* consiste de 3 tecnologias: *Custom Elements*, *Shadow DOM* e *HTML Templates*.

2.5.1 HTML5

É a principal linguagem declarativa da *web*, utilizada para construção de interfaces de aplicações desktop, IOS, Android, etc. O código de uma linguagem declarativa tem relação direta com o seu resultado (MONTEIRO, 2015). O desenvolvedor especifica seus componentes de interface no código HTML e o interpretador da linguagem é responsável por construí-los em tela.

O uso de uma linguagem imperativa, como o JavaScript, que tem métodos, variáveis e rotinas é imprescindível para o desenvolvimento de funcionalidades mais complexas dentro de uma aplicação que a simples exibição estática de um conteúdo HTML.

A escolha do HTML como a principal linguagem de declaração para interfaces se deve ao fato de não depender da instalação de *plugins* externos para ser interpretada, ter código aberto e livre para qualquer uso, compatibilidade universal com navegadores e plataformas e alta performance. (MONTEIRO, 2015).

As principais vantagens do HTML5 em relação às versões anteriores da linguagem são, em resumo, as seguintes (HERTEL, 2018):

- Suporte à gráficos vetoriais, como *SVG* e *Canvas*, antes, esse recurso era fornecido por terceiros, a maioria softwares proprietários, como *Flash*, *Silverlight* e *VML*.
- Implementação de um banco de dados próprio (*Web SQL Database*) com *cache* de aplicativo, em oposição ao *cache* genérico de navegador.
- API *JS Web Worker*, que interpreta JavaScript em segundo plano ao invés de travar o segmento de interface da página até que o *script* seja finalizado.
- Independência dos padrões de sintaxe SGML, aumentando a portabilidade com interpretador único.
- Elementos depreciados do HTML foram completamente removidos, como *isindex*, *noframe*, *applet*, *basefront* e *frameset*.
- Novos controles de formulário, como *date*, *number*, *tel*, *url*, *search*.
- Tratamento de erros: Não existia um protocolo único para tratamento de erros de sintaxe em versões anteriores do HTML, os desenvolvedores precisavam testar suas aplicações em cada um dos navegadores que quisessem

compatibilizar. No HTML5 existe uma manipulação de erros consistente e padronizada.

- *Tags* semânticas: A designação de responsabilidade dos elementos, que antes era provido apenas por *div's*, no HTML5 dispõem de papéis semânticos definidos em *tags* específicas, como *header*, *article*, *nav*, *body*, etc.

2.5.2 Custom Elements

Cada tag em HTML possui uma função específica, se dedicando à exibição de texto, imagem, vídeo, campo de formulário, etc. A tecnologia de Custom Elements permite ao desenvolvedor criar seus próprios elementos com *tags* customizadas.

A funcionalidade de um *custom element* pode ser definida a partir da junção de várias *tags* nativas e acrescidas de programação em JavaScript. Um elemento customizado é reutilizável e instanciado indefinidamente, simplificando o código fonte e aumentando sua legibilidade, uma vez que minimiza a repetição de código.

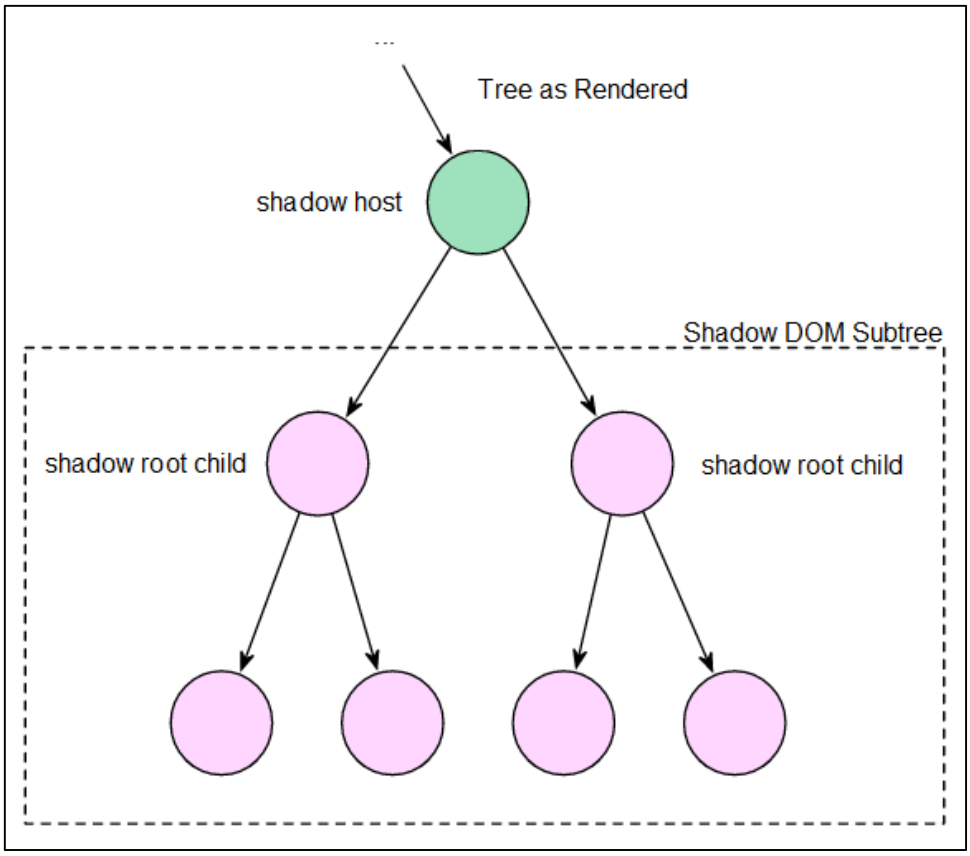
2.5.3 Shadow DOM

O DOM (*Document Object Model*) é uma representação, ou uma interface, dos elementos que compõem uma página HTML. Esse documento é estruturado hierarquicamente em uma árvore (MONTEIRO, 2015). Quando um nó dessa árvore é alterado, seja um elemento nativo ou *custom element* os nós subsequentes, que descendem deste também são.

O *Shadow DOM* encapsula uma parte dessa árvore, ou seja, uma cadeia de elementos em uma estrutura isolada das demais, que pode receber folhas de estilos e *scripts* exclusivos. Essa sub-árvore é chamada de *shadow tree*, seu elemento base é o *shadow root*, ou *shadow host*, e os elementos descendentes são os *shadow child's*.

A Figura 14 ilustra um *Shadow DOM*, contendo um *host* e seus filhos (*child's*)

Figura 14: Shadow DOM



Fonte: (IHRIG, 2012)

2.5.4 HTML Templates

HTML *Templates* são blocos de códigos HTML que podem ser reutilizados e instanciados em outras páginas, seu processamento pode ser realizado pelo navegador (cliente) ou servidor (MONTEIRO, 2015).

Antes dos *Web Components*, as estratégias mais comuns para declarar um template HTML eram: (i) Criar uma *div* e torná-la invisível por CSS. O código do *template* inevitavelmente era carregado independente de ser utilizado ou não; e (ii) Declarar o *template* dentro de um *script* JavaScript, também invisível. Por ser lido como uma *string*, elementos desnecessários não são visíveis em tela, porém são carregados no DOM e criam vulnerabilidades quanto a injeção de scripts *maliciosos* por terceiros (XSS) na aplicação.

Os *Web Components* introduzem a *tag template*, que encapsula um *template* HTML e o renderiza apenas quanto ele é usado, nenhum conteúdo pode ser acessado antes desse momento (MONTEIRO, 2015).

2.6 ANGULAR

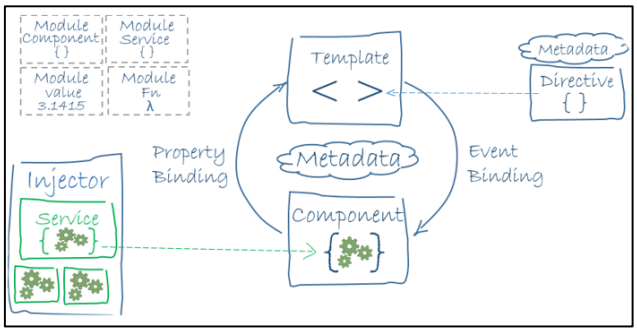
Angular é um *framework* JavaScript MVC (*Model-View-Controller*) para desenvolvimento de interfaces *Web.* Possui os recursos mais modernos das bibliotecas JavaScript, como *data bind*, roteamento e animações. Seu esquema MVC permite separar componentes responsáveis pela apresentação das interfaces, manipulação de dados e lógica de negócio (FARIA, 2018).

A abordagem MVC facilita os testes e promove um desenvolvimento mais rápido e lógicos das aplicações (KORVA, 2008), uma vez que as partes do sistemas não conhecem nem interferem na implementação dos outras partes.

A arquitetura Angular está fundamentada em módulos, que são conjuntos coesos e independentes de componentes (detalhados na seção 2.6.2) e serviços (detalhados na seção 2.6.5), que, basicamente são fontes de dados da aplicação e realizam comunicação à servidores externos (ANGULAR, 2018). Toda aplicação necessita de pelo menos um módulo e todo módulo especifica suas declarações de componentes, importações e exportações de outros módulos ou classes e seus provedores de serviços.

Uma ilustração do funcionamento de um módulo é apresentada na Figura 15.

Figura 15: Angular MVC



Fonte: (KORVA, 2018)

2.6.1 TypeScript

A linguagem de programação usada no Angular é o TypeScript, um *superset* do JavaScript que traz fortes conceitos de orientação a objetos, adicionando tipagem estática, classes, interfaces, módulos e restrição de visibilidade de métodos e variáveis. Seu uso é compulsório no Angular e recomendado na maioria dos ambientes JavaScript, uma vez que permite verificação de sintaxe e semântica a partir da tipagem definida, *autocomplete* e *autoimport* nas *IDE's* e incremento na escalabilidade e facilidade na refatoração. Segundo sua equipe de desenvolvimento, TypeScript é o "JavaScript escalável, para qualquer *browser*, qualquer servidor, qualquer sistema operacional e de código aberto" (TYPESCRIPTLANG, 2018).

As principais funcionalidades da linguagem são as seguintes:

- Tipagem estática, com possibilidade de ser *any* (qualquer tipo) e *void* (nenhum tipo, usado em retorno de métodos)
- *Enum*
- Interface
- Classe
- Visibilidade de métodos e variáveis
- Herança
- *Acessors* (*get* / *set*)
- Métodos estáticos
- Passagem de parâmetros opcionais
- Passagem de *array* de parâmetros

2.6.2 Components

Components, ou componentes, são partes que compõem uma tela, ou *View*. Um componente encapsula um *template* HTML, um conjunto de regras de estilo CSS, uma classe TypeScript que gerencia suas propriedades e comportamentos e opcionalmente, outros componentes.

Nenhum item de interface no Angular pode ser exibido fora de um *component* e são exemplos de componentes os botões, barras de navegação, formulários,

tabelas, etc. O desenvolvedor pode criar seus próprios componentes e instancia-los conforme a necessidade.

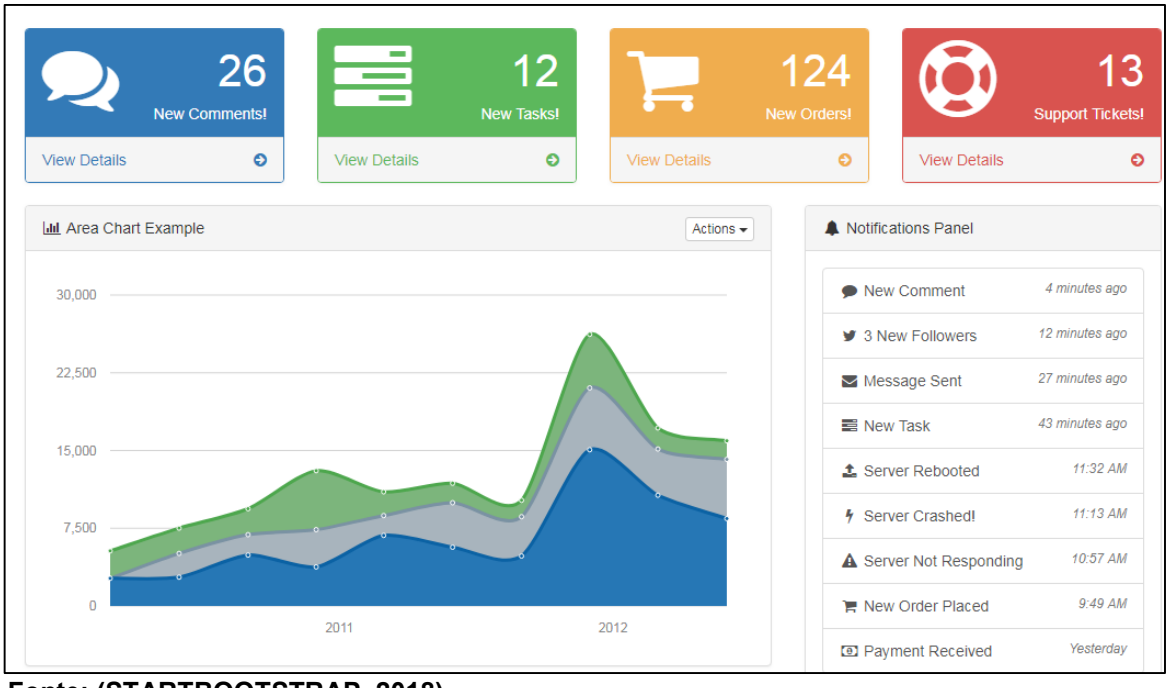
Uma série de *frameworks* HTML, CSS e JavaScript foram criados para fornecer itens de layout e servir à construção de componentes, dentre os quais se destacam *Bootstrap* e *Materialize*.

2.6.3 Bootstrap

Maior *framework* CSS / JS em número de usuários, está presente em quase 18% dos *sites* da Internet (W3TECHS, 2018). Foi criado pela equipe do Twitter e preza pelo layout responsivo, é otimizado para dispositivos móveis (*mobile first*) e foi um grande responsável pela popularização das aplicações *mobile* no navegador de *Internet* (KARLSSON, 2014). Um exemplo de *dashboard* (painel de controle) estilizado com Bootstrap é apresentado na Figura 16.

Sua principal contribuição é o sistema de *grid*, que divide a tela em um conjunto de *containers*, linhas e colunas, adaptáveis e customizáveis em relação ao tamanho da tela em que a página está sendo exibida (BOOTSTRAP, 2018).

Figura 16: Bootstrap Dashboard



Fonte: (STARTBOOTSTRAP, 2018)

2.6.4 Materialize

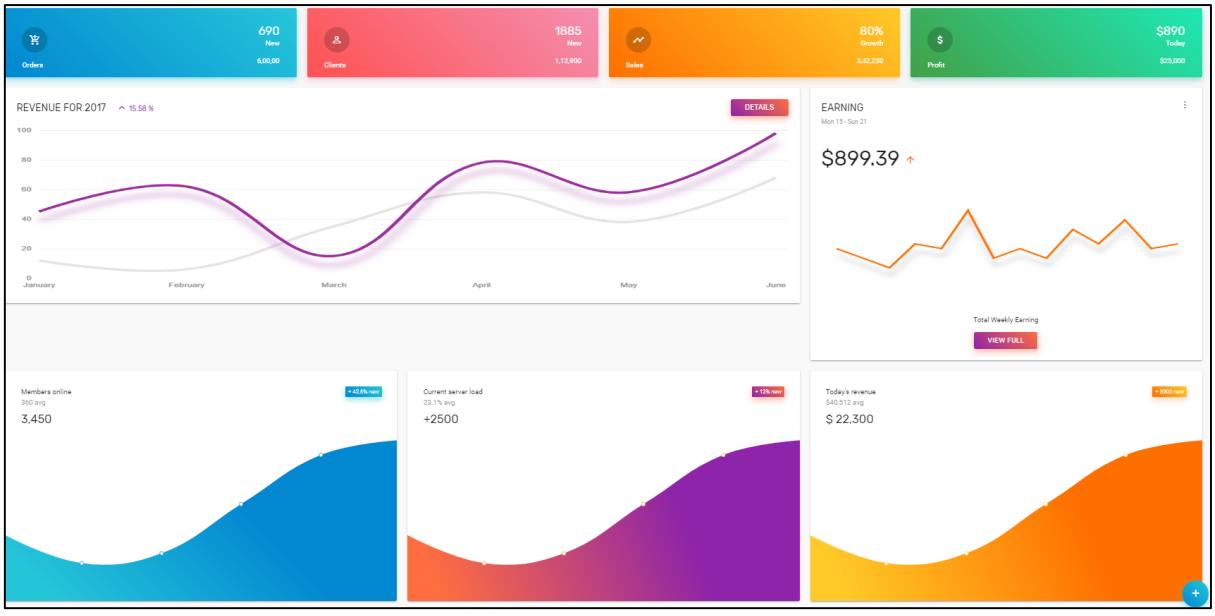
Implementação *open-source* do *material design* do Google, que é uma linguagem e metodologia de design que prioriza, em seus componentes, o realismo, tipografia chamativa, imersão visual, cores claras, animações com física realista, hierarquia de elementos, tridimensionalização, iluminação e sombras individuais (RALLO, 2017).

A primeira versão do *material* foi apresentada em 2014 no sistema operacional Android e sua implementação é fortemente baseada em objetos do mundo real. Seus elementos de design são sólidos, impenetráveis, mutáveis, dobráveis e se movimentam nos 3 eixos cartesianos (RALLO, 2017).

Desde o lançamento, sua metodologia é a que mais se destaca no mercado de *design* para *web* (LATYPOV, 2017)

A Figura 17 apresenta um *dashboard* (painel de controle) construído com Materialize.

Figura 17: Materialize Dashboard



Fonte: (PIXINVENT, 2018)

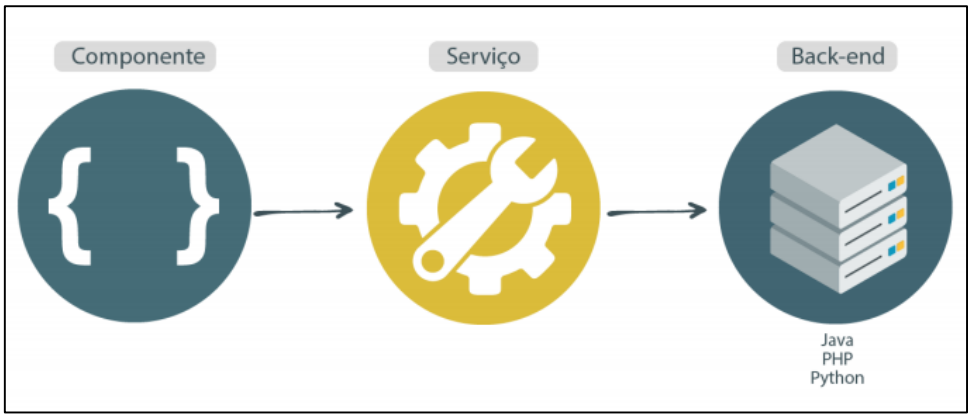
Por ser uma tecnologia mais recente e mais complexa que as demais, uma desvantagem desse *framework* é a comunidade e a quantidade de componentes disponíveis ser menor que dos concorrentes.

2.6.5 Serviços

Serviços, ou *services*, são classes Angular responsáveis por fornecer e processar dados para a aplicação, geralmente fazendo a comunicação com o servidor (*back*-end) e tratando as respostas recebidas. Os serviços são instanciados em cada componente que precise utilizá-lo e seu uso organiza a aplicação e separa a lógica de negócio das classes de interface (FARIA, 2018).

A Figura 18 ilustra a comunicação de um componente com um serviço.

Figura 18: Angular Service



Fonte: (FARIA, 2018)

2.6.6 Módulos

Módulos são *containers*, ou agregadores de *controllers*, sejam eles componentes, serviços, outros módulos Angular e módulos JavaScript. Eles dividem a aplicação em partes, define a ordem de carregamento dessas partes e manipulam o ciclo de vida dos componentes (ANGULAR, 2018).

2.7 BANCOS DE DADOS RELACIONAIS

Bancos de Dados relacionais são repositórios de dados organizados e relacionados entre si (GOMES, 2013). A interação com o usuário é realizada por uma linguagem de programação de alto nível, a SQL (*Structed Query Language*) e o

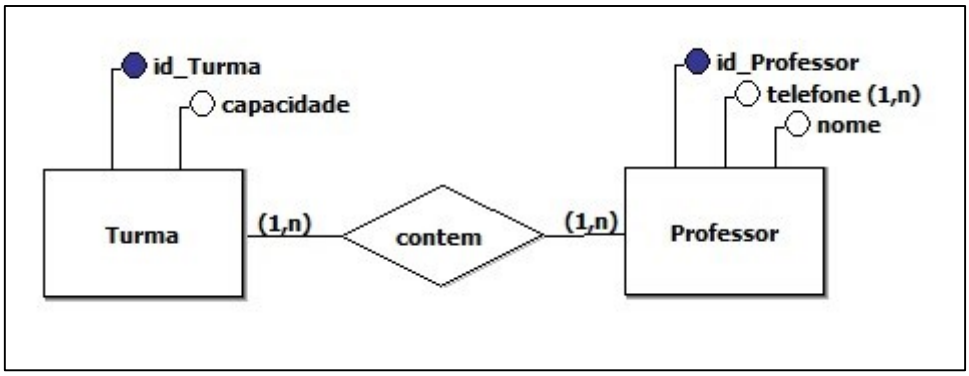
servidor que provê acesso aos dados é chamado SGBD. As operações são realizadas por meio de transações que respeitam uma série de requisitos de segurança e integridade da informação: as propriedades ACID (atomicidade, consistência, isolamento e durabilidade).

2.7.1 Modelagem de Dados

O bom uso de um banco de dados relacional requer a utilização de padrões de modelagem e documentação. Os principais documentos referentes à implementação são:

Modelo Conceitual: Representa os componentes do sistema, por representações do mundo real. O objetivo é apresentar de forma gráfica e simples o conceito do banco a ser implementado. A Figura 19 apresenta um típico modelo conceitual.

Figura 19: Modelo Conceitual de Banco de Dados



Fonte: (SPACEPROGRAMMER, 2016)

Modelo Lógico: É um documento que apresenta de maneira mais detalhada o relacionamento entre as entidades do sistema, dependências funcionais e campos que compõem as tabelas.

Toda tabela de um banco de dados relacional é composta de uma série de campos que contém nome, tipo e identificadores. Os principais identificadores são:

- *Primary key* (PK): Campo que identifica o registro. Cada registro deve conter uma única PK, que não pode ser repetida na tabela e nem pode ser nula.

Geralmente delega-se ao próprio SGBD criar a PK. Este é o campo usado para referenciar o registro em outras tabelas.

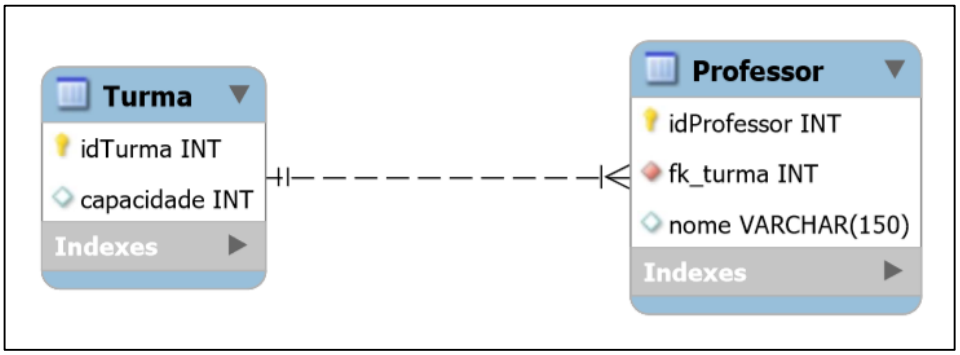
- *Foreign Key* (FK): Campo que identifica o relacionamento entre tabelas. Cada tabela pode ter múltiplas PK's, que sempre assumem o valor da PK da tabela referenciada. Na figura 20 o campo *fk_turma* referencia a tabela Turma à qual o Professor está relacionado.

- *Not Null*: Indica que o campo não pode ser vazio (nulo).

- *Unique*: Indica que o valor do campo não pode ser repetido.

O papel do modelo lógico, ilustrado na figura 20 é representar as tabelas com todos os seus campos, tipos, relacionamentos e cardinalidades (implementados com PK e FK) e demais identificadores.

Figura 20: Modelo Lógico de Banco de Dados



Fonte: Autoria própria

3 DESENVOLVIMENTO

Neste capítulo é apresentado o desenvolvimento da aplicação, uso das metodologias de desenvolvimento e funcionalidades da partes componentes do sistema.

3.1 VISÃO GERAL DO SISTEMA

A aplicação é dividida em 3 partes: (i) o banco de dados MariaDB / MySQL, (ii) a API, ou servidor Node, e (iii) o servidor de front-end, em Angular. A Figura 21 apresenta os principais casos de uso da aplicação e o quadro 2 descreve com mais detalhes as funcionalidades implementadas no *backend* e *frontend*, a fim de atender os objetivos propostos.

Figura 21: Diagrama de casos de uso

Fonte: Autoria própria

O quadro 2 apresenta as funcionalidades do sistema, divididas entre *back-end* (API) e *front-end*.

Na descrição, entende-se por CRUD o conjunto de operações de criação, leitura, exclusão e atualização de recursos.

Quadro 2: Funcionalidades gerais do sistema

API	FRONT-END
Persistência de dados com SGBD MySQL	Listagem geral de animais na *home page*
ConFiguração de Rotas com *Express*	Busca por Nome e uso de filtros de nome, tamanho, sexo, etc.
CRUD de usuários (Inserção, Remoção e Atualização)	Página de *Login* / Logout
CRUD de animais	Página de cadastro de usuário e animais
CRUD de fotos	Página de detalhes sobre o animal
CRUD de nomes (sugestões de nomes vindas dos usuários)	Componentes de envio de foto, sugestão de nome, doações, etc.
CRUD de adoção	Página pessoal do usuário
CRUD de doações	Responsividade para diversos navegadores e dispositivos
CRUD de visitas veterinárias	Validação de dados em formulários
Envio de *e-mail* para os usuários	*Toasts* de confirmação e erro em solicitações ao servidor
Login com geração de *token* JWT	Submissão de formulário para adoção mediante concordância com termo de responsabilidade
Proteção de rotas com *middlewares* de autenticação	
Segurança de dados com criptografia de senhas e armazenamento de *secret keys* fora do servidor	

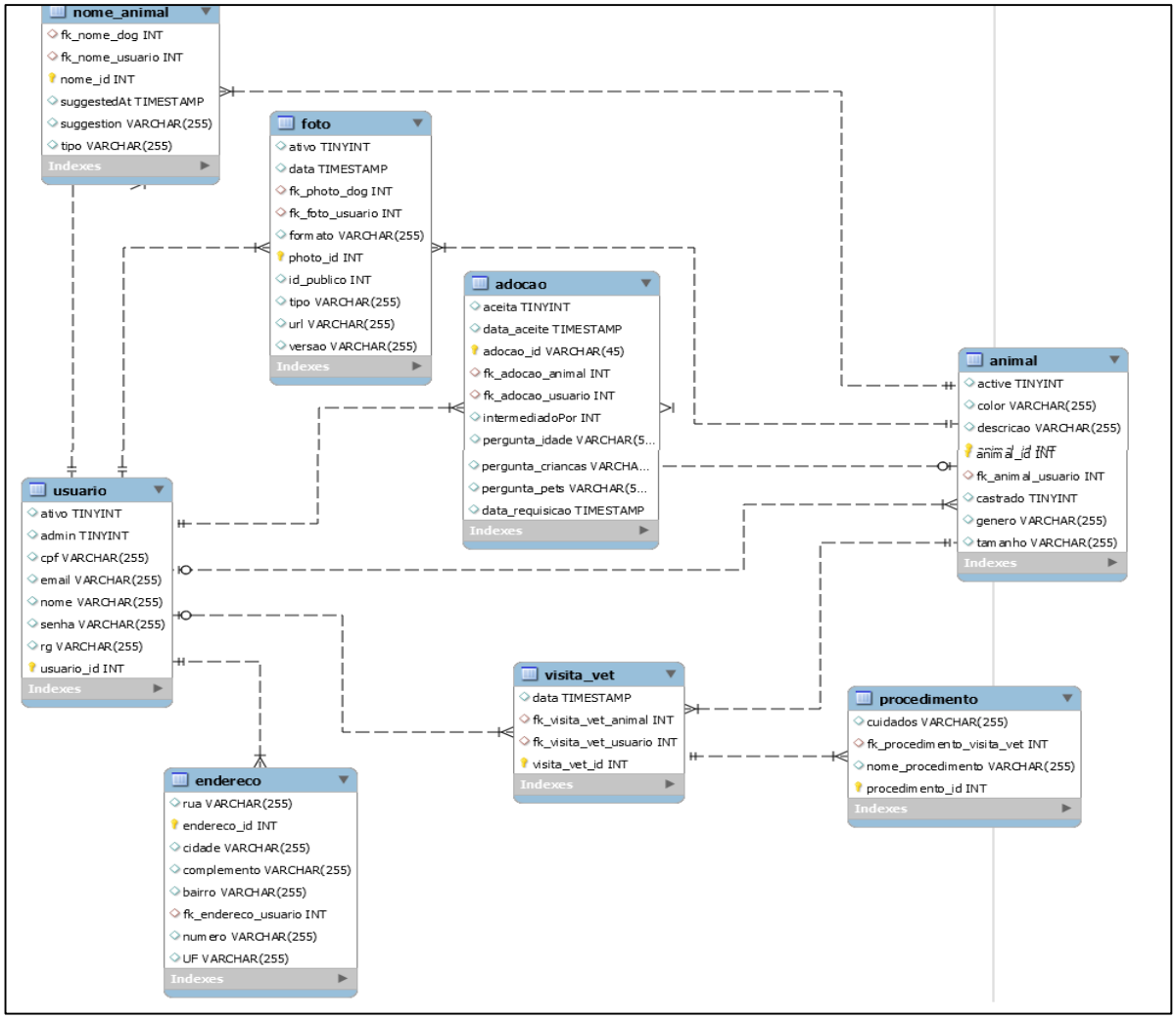
Fonte: Autoria própria

3.2 BANCO DE DADOS

O banco de dados escolhido foi o MariaDB, na versão 10.1.35. Este DB é um *fork* do MySQL, *open-source* e distribuído sob licença GNU / GPL. A escolha é justificada pela farta documentação, popularidade (é o mais usado no mundo), velocidade, escalabilidade e robustez (MARIADB, 2018).

No contexto deste trabalho, as principais vantagens do MariaDB são a compatibilidade nativa com JSON e alto grau de maturidade de seus módulos no NPM. Na Figura 22 o modelo lógico do banco de dados é apresentado.

Figura 22: Modelo Lógico do banco de dados



Fonte: Autoria própria

A estratégia de exclusão adotada nesse sistema é a de *soft delete*, que consiste em somente atualizar o registro com um campo que indica que se ele está ativo ou inativo, portanto todas as entidades que podem ser 'deletadas' têm um campo chamado *active* para esse propósito. Essa técnica possibilita maior auditoria dos dados e previne problemas de perda de informação.

Na nomenclatura dos campos *foreign key* foi adotado o padrão *fk_campo1_campo2*, onde *campo1* é a tabela que recebe a FK e *campo2* é a tabela referenciada.

No diagrama foram omitidos campos de controle / auditoria, como o *timestamp* de criação de cada registro, bem como o usuário do DB que realizou a transação e a origem da requisição.

3.3 SERVIDOR

O servidor de aplicação, ou API desse sistema segue a padronização *Restful* (segue todas as restrições obrigatórias do REST). Toda transferência de dados entre API e *front-end* é realizada em formato JSON e protocolo de comunicação HTTP. O *framework* utilizado é o Node 8.11.4

3.3.1 Configuração do *package.json*

A API é iniciada pelo *npm*, com o comando de console *npm init*, que gera um arquivo denominado *package.json*, no qual se encontram todas as dependências do Node Package Manager, os *scripts* de execução, testes, a descrição do projeto, com informações de nome, autor, repositório remoto, etc.

Os módulos importados ao *package.json* são os seguintes:

- *axios*: Faz requisições AJAX / HTTP à servidores externos
- *body-parser*: Converte o corpo das requisições recebidas ao formato JSON.
- *express*: *Framework* web de tratamento de requisições, criação de rotas e middlewares.
- *jsonwebtoken*: Implementação do protocolo de geração de *tokens* JWT
- *moment*: Biblioteca de manipulação de data e hora.

- *morgan*: Faz um breve relatório das requisições recebidas e respondidas, com *status code*, origem, tempo de resposta, etc.
- *mysql2*: Módulo de acesso ao SGBD Mysql e MariaDB
- *dontenv*: Expõe ao escopo global do programa as variáveis de ambiente disponíveis no arquivo. *env*.
- *crypto*: Pacote depreciado no *npm*, agora ele se encontra no *core* (*built-in*) do Node e é responsável por encriptar / decriptar texto em diversos protocolos disponíveis.

3.3.2 Módulo Principal

O servidor pode ser iniciado pelo CLI do *node* ou através de *script* do *package.json*. O arquivo de entrada é chamado *index.js*, e ele é responsável por:

- criar a instância do módulo *express*, que define as rotas e *middlewares* da aplicação
- testar a conexão com o banco de dados
- habilitar o módulo de *log* (*morgan*) e JSON *parser* das requisições com *body-parser*.

O *express*, após instanciado no arquivo de entrada, é encaminhado aos 3 *routers* da aplicação, para ser carregado com as rotas do servidor: a do animal, do usuário e dos middlewares, a função do *router* é criar as rotas e validar a entrada e saída de dados, transferindo o processamento da informação e interação com banco de dados aos módulos da pasta *models*. O *express*, após instanciado no arquivo de entrada, é encaminhado aos 3 *routers* da aplicação, para ser carregado com as rotas: a do animal, do usuário e dos middlewares, a função do *router* é criar as rotas e validar a entrada e saída de dados, transferindo o processamento da informação e interação com banco de dados aos módulos da pasta *models*.

Este arquivo (*index.js*) está disponível no apêndice A.

A Figura 24 apresenta o *router* e algumas operações do servidor referentes ao animal.

Figura 24: Registro de rotas do animal

```
router.post('/', (req,res) => addDog(req,res))
router.post('/photo', (req,res) => addPhoto(req,res))
router.post('/name', (req,res) => suggestName(req,res))
router.post('/addopt', (req,res) => addoptDog(req,res))
router.get('/:id', async (req,res) => getDogByid(req,res))
router.get('/', async (req,res) => getAllDogs(req,res))
router.get('/query/addopted', async (req,res) => getAddoptedDogs(req,res))
router.get('/mydogs/:userId', async (req,res) => getDogsByUser(req,res))
router.delete ('/:id', (req,res) => deleteDogById(req,res))
router.delete ('/photo/:id', (req,res) => deletePhotoById(req,res))
```

Fonte: Autoria própria

A Figura 25 apresenta a função completa de adição de sugestão de nome para um animal.

Figura 25: Instância do *Express Router*

```
router.post('/name', (req,res) => {

    let {user, dog, name} = req.body;

    if (user && dog && name){

        dogHandler.suggestName(req.body, (status, result) => {

            status = typeof(status) == 'number' ? status : 500;
            result = typeof(result) == 'object' ? result : {};

            if (status == 201){
                res.setHeader('id', result.insertId);
            }

            res.status(status).json(result);
        })

    }else{
        res.status(400).json({error: 'Missing required fields'})
    }

})
```

Fonte: Autoria Própria

Na Figura 25, a tarefa do *router* é verificar se os campos obrigatórios estão presentes na requisição, caso negativo, a operação é cancelada com o *status code*

400, indicando erro do usuário e enviando uma mensagem de erro. Caso positivo, os dados são enviados ao *model* correspondente para serem processados, geralmente lendo ou gravando no banco de dados.

Validando a saída desses dados ao cliente o *router* atribui o *status code* 500, de erro interno do servidor à resposta caso não haja algum vindo do *model* e caso não exista um resultado JSON, um objeto vazio é enviado.

O *status code* de sucesso desta operação apresentada na Figura 26 é o 201 (criado), já que se trata de uma requisição POST, e um *header location* é adicionado seguindo a restrição de HATEOAS do REST.

Esta estratégia de tratamento de requisições é usada em todo o sistema.

A Figura 26 apresenta a operação de manipulação do banco de dados responsável pela sugestão de nome.

Figura 26: SQL Model

```
dogHandler.suggestName = function(dogName, callback) {

  let sql = `INSERT INTO
  name (suggestion, suggestedAt, type, fk_name_dog, fk_name_user)
  VALUES ('${dogName.name}', '${moment().format("YYYY-MM-DD HH:mm:ss")}',
  'suggestion', '${dogName.dog}', '${dogName.user}')`;

  mysql.queryDB(sql, function(status, result) {
    if (status == 200) {
      callback(201, result);
    } else {
      callback(status, result);
    }
  });
};
```

Fonte: Autoria Própria

O que esse *handler* faz é criar uma *string* com o comando SQL de inserção na tabela *name* dos dados recebidos por parâmetro, a seguir envia ao módulo do *mysql*, e devolve a resposta do banco de dados ao *router* que o invocou.

3.3.3 Funcionalidades do servidor

Resumidamente, o servidor realiza as seguintes operações (uma descrição mais detalhada da API em formato JSON, está disponível no apêndice B):

- CORS: Permite acesso de qualquer domínio externo pelos métodos GET, PUT, POST, DELETE e PATCH.
- Criar Usuário: Usa nome, e-mail e senha para criar um usuário. A senha é criptografada antes de ser armazenada no banco de dados.
- Buscar Lista de Usuários: Retorna uma lista de todos os usuários.
- Buscar Usuário por E-mail: Retorna um usuário a partir de seu e-mail.
- *Login*: A partir do e-mail e senha, a API retorna um *token* de acesso, válido por 14 dias.
- Recuperar Senha: O usuário pode redefinir sua senha solicitar um *link* por email
- Para acesso às rotas protegidas, recebe o *token* de autorização pelo *header x-access-token*.
- Atualizar / Deletar Usuário: A partir do e-mail informado, atualiza ou exclui o usuário.
- Cadastrar Animal: O usuário pode cadastrar um animal, informando nome, sexo, tamanho, cor, local em que vive, fazer um comentário e adicionar foto.
- Buscar lista de animais: Retornar a lista de todos os animais.
- Buscar animal: Retorna um animal a partir de seu identificador (*id*).
- Buscar animais por sexo, tamanho ou local: Retorna uma lista de animais a partir do critério definido.
- Adicionar Foto: O usuário pode adicionar novas fotos para cada animal
- Excluir Foto: O usuário pode excluir as fotos que tenha adicionado.
- Sugerir Nome: O usuário pode sugerir um novo nome para o animal.
- Votar em uma Sugestão de Nome: O usuário pode votar em um dos nomes sugeridos para cada animal.
- Registrar Visita Veterinária: O usuário pode registrar uma visita veterinária em que tenha acompanhado o animal, informando qual clínica visitou e quais foram os procedimentos realizados, assim como os cuidados necessários.

- Adotar Animal: O usuário pode solicitar a adoção do animal, preenchendo um formulário com seus documentos pessoais (*rg* e *cpf*), endereço e perguntas que auxiliam o cuidador a entender o contexto dessa adoção, como a presença de crianças na casa e outros animais.
- Conversar com Outros Usuários: O usuário pode enviar mensagens a outros usuários ou administradores, que vão recebe-las na interface do site e também por e-mail.

3.4 FRONT-END

O front-end da aplicação foi desenvolvido em Angular 6.1.5, e seu fluxo de utilização é o seguinte:

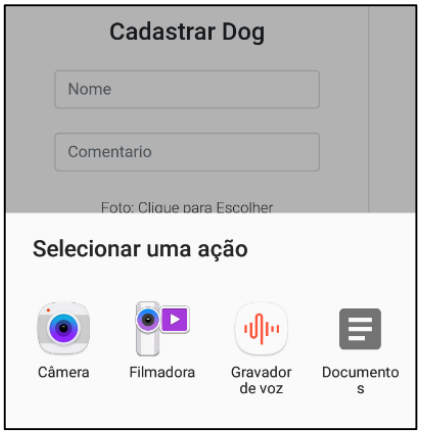
- Usuário (logado ou não logado) acessa a *url* principal e visualiza uma lista geral de animais cadastrados, com nome e foto.
- O usuário pode filtrar os resultados dessa listagem escolhendo o botão *filtrar dogs* da barra de navegação. Os filtros disponíveis são: tamanho, sexo e local.
- Através da barra de navegação ou ao acessar qualquer rota protegida e não estar logado o usuário é redirecionado à pagina de *login*.
- Ao acessar a página de *login* o usuário pode informar seu email e senha ou navegar à página de cadastro de usuário, disponível em um link.
- Para concluir o cadastro o usuário informa seu nome, email e senha, então é automaticamente logado.
- Ao realizar o *login* um token de acesso é enviado pela API e armazenado em *localStorage*.
- Ao acessar a página de logn o usuário pode solicitar um link de redefinição de senha, enviado por email e válido por 24 horas.

- Na lista de animais da home page, o usuário pode clicar em um itens e é direcionado à página de detalhes sobre esse animal, que contém suas fotos e informações cadastradas.
- Na lista de detalhes sobre o animal o usuário tem diversos *links* disponíveis que abrem em forma de modal, são eles: Sugerir Nome, Registrar Consulta Veterinária, Adicionar Foto e Adotar Animal.
- Na sugestão de nome, o usuário informa o novo nome que ele sugere ao animal, este nome fica disponível para votação dos demais usuários.
- A cada 10 dias o nome mais votado de cada animal substitui seu nome atual.
- Caso o usuário acione o botão de adicionar foto na versão de *desktop* do site, o explorador de arquivos de seu sistema operacional exibe um popup para escolha do arquivo. Caso esteja usando um dispositivo mobile, além do explorador de arquivo existe a opção de usar as câmeras do aparelho, como ilustrado na Figura 27. Após tirar a foto, ela é automaticamente enviada ao sistema. Em qualquer caso, uma pré-visualização da imagem é apresentada ao usuário, que pode alterá-la antes de submeter definitivamente.
- A submissão de qualquer formulário do sistema sem o preenchimento de todos os campos obrigatórios é bloqueada através da desativação do botão de ação.
- Toda comunicação com a API possui um *feedback* visual para o usuário por meio de *toasts* de erro / sucesso.
- No componente de registro de visita veterinária o usuário informa a clínica que levou o animal, os procedimentos realizados e os cuidados necessários.
- No componente de adoção o usuário preenche seu endereço completo, RG, CPF e responde se tem outros animais, se tem crianças em casa e como seria a convivência deles com o *pet* novo. Essas perguntas são realizadas a fim dos administradores minimizarem o risco de entregarem o animal a alguém que pode ficar insatisfeito com a adoção ou que porventura venha a abandonar o animal. Por fim, o usuário informa se concorda com o termo de

responsabilidade disponível no site e reproduzido no anexo A e seu pedido é encaminhado aos administradores.

- O usuário tem um perfil pessoal, em que constam os animais por ele cadastrados, as fotos e mensagens.
- O usuário visualiza o nome de quem cadastrou o animal e pode enviar mensagens pra ele, assim como pra qualquer outro usuário.

Figura 27: Envio de Fotos no Android



Fonte: Autoria própria

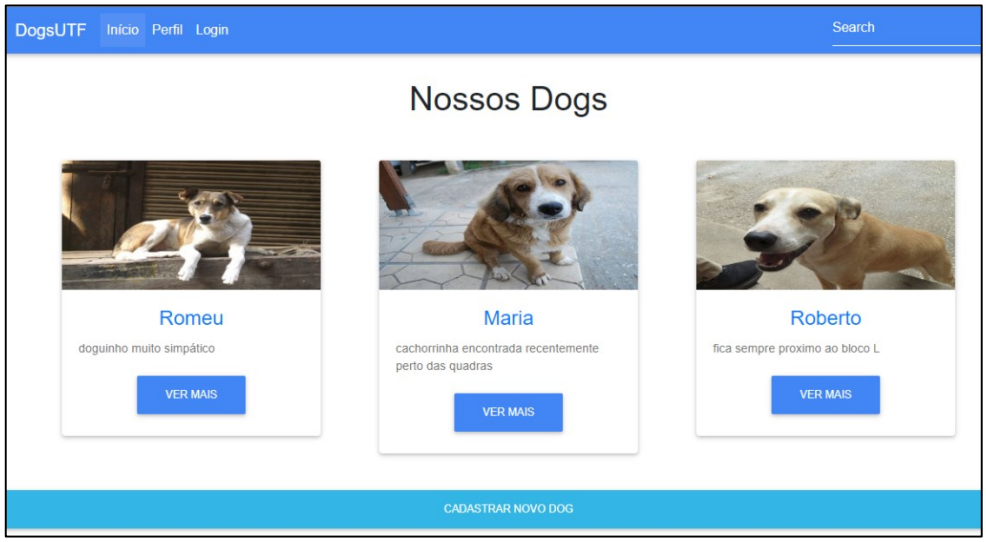
3.4.1 Componentes e Telas Principais

A aplicação está estruturada nos seguintes componentes:

- Telas de Boas Vindas
- HomePage
- Cadastro de Usuários
- Cadastro de Animais
- Detalhes do Animal
- *Login*
- Perfil de Usuário
- Termo de Responsabilidade

A Figura 28 mostra a *home page* da aplicação, com a lista geral de animais cadastrados e uma barra de navegação.

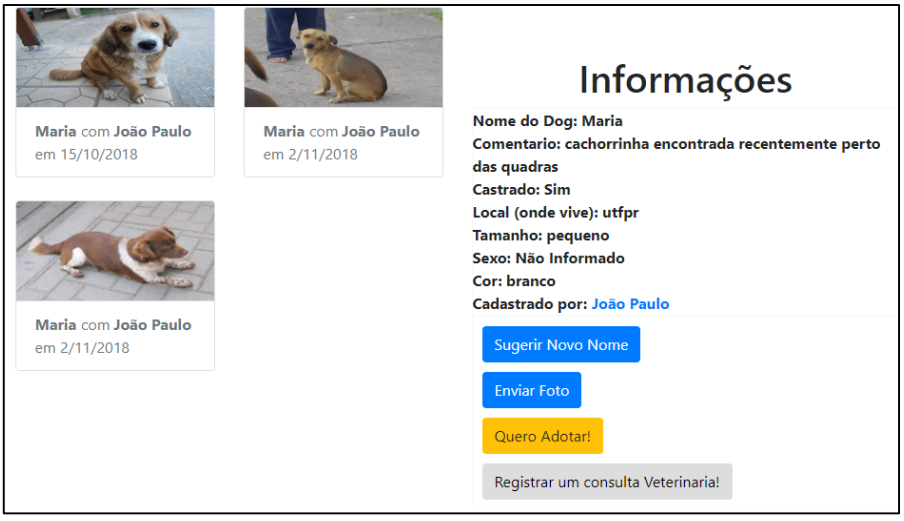
Figura 28: Home Page



Fonte: Autoria própria

Os detalhes e botões de ações sobre os animais são apresentados na Figura 29.

Figura 29: Detalhes do Animal

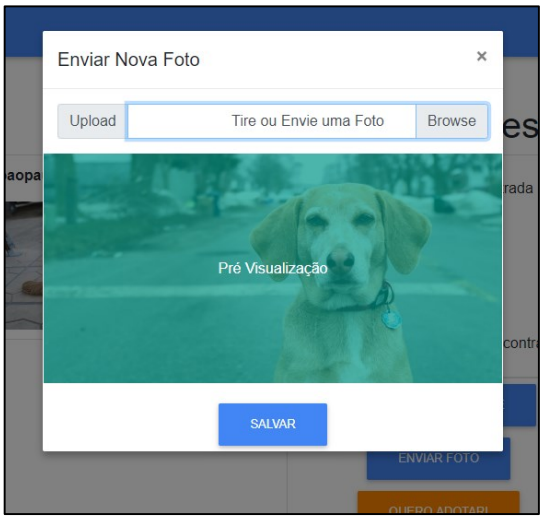


Fonte: Autoria própria

Na Figura 29 é possível visualizar detalhes sobre o animal, como porte, sexo, nome do usuário que realizou o cadastro, etc. A interação que se pode realizar com o animal e com os demais usuários do sistema é apresentada nos botões de ação, além das barras de navegação e componentes ocultados da Figura 29.

Ao clicar em qualquer ação, um componente modal (que aparece sobre a tela atual) com um formulário é mostrado, como apresentado na Figura 30.

Figura 30: Modal Component



Fonte: Autoria própria

Para o *feedback* das ações realizadas, *toasts* do módulo npm *ngx-toasta* são mostrados, como apresentado na Figura 31.

Figura 31: Toasts

Fonte: Autoria própria

Os demais componentes e telas do sistema são apresentados nos domínios e repositórios da sessão 3.4 (Publicação do *app* em Ambiente Web).

3.4.2 Comunicação com a API

A comunicação com a API é realizada pelo módulo *HTTPClient* do Angular e encapsulado em um serviço. Todo o tráfego de dados desta aplicação é realizado em JSON, portanto o header *application/json* é adicionado em todas as requisições. A Figura 32 apresenta um exemplo de requisição

Figura 32: Angular HTTP Service

```
signUpUser(user: User) {

  let headers = new HttpHeaders({
      'Content-Type': 'application/json'
  })
  return new Promise((resolve, reject) => {
    this.http.post('http://localhost:3000/user', user, { observe: 'response', headers: headers })
      .subscribe((res: any) => {
        console.log(res.status);
        if (res.status == 201) {
          resolve(res.body);
        }
        else {
          console.log('subscribe com erros ' + JSON.stringify(res));
          reject({ error: res.body, statusCode: res.status }); //status diferente de 201 created
        }
      }, err => {
        //console.log('errors '+JSON.stringify(err));
        reject({ error: err.error, statusCode: err.status }); //api raw response
        reject(err);
      })
  })
}
```

Fonte: Autoria própria

3.5 Publicação do *app* em Ambiente Web

O código fonte da API e da aplicação Angular estão disponíveis em repositórios públicos na plataforma de hospedagem Github [1].

O website foi publicado pelo serviço HerokuApp [2].

A hospedagem de imagens é realizada pelo serviço Cloudnary [3].

O banco de dados é hospedado pelo serviço Amazon RDS, em domínio privado.

4 CONCLUSÃO

Neste capítulo são apresentadas as considerações finais sobre o desenvolvimento do projeto e sugestões de trabalhos futuros.

4.1 CONSIDERAÇÕES FINAIS

A proposta desse app foi implementar um sistema completo de acompanhamento e controle de adoção de animais de rua, serviço que qualquer pessoa pode utilizar e beneficiar enormemente um animal abandonado que pode encontrar um lar e uma família que adquire um novo *pet*.

Os objetivos propostos foram atingidos, tendo como resultado uma aplicação de interface com o usuário e um servidor que provê seus dados. Os mais atuais requisitos de segurança, como o *token JWT* e de usabilidade, como o *Material Design* e *Bootstrap*, foram observados e o app se comporta de maneira responsiva e intuitiva em *desktops* e *mobiles*.

As aplicações são documentadas, pelos diagramas expostos neste trabalho e instruções técnicas para uso do servidor. Suficiente para que desenvolvedores consumam os serviços implementados, melhore-os e incremente-os conforme lhe for conveniente.

4.2 TRABALHOS FUTUROS

Sugere-se como trabalho futuro a integração desse *app* com alguma(s) base de dados de animais abandonados / perdidos e também a integração com um sistema especializado em monitoramento de animais que suporte rastreio por chip, por exemplo, e seja operado por uma equipe profissional.

REFERÊNCIAS

ANGULAR, c2018. Página Inicial. Disponível em: < https://angular.io >. Acesso em: 02 set. 2018

BONFIM, Felipe L. **APLICAÇÕES ESCALÁVEIS COM MEAN STACK**. 2014, 48 f. Monografia (Bacharelado em Ciência da Computação) – Universidade Federal do Paraná, 2014. Disponível em: < http://www.inf.ufpr.br/bmuller/TG/TG-FilipeMichael.pdf>. Acesso em: 15 out. 2018

BOOTSTRAP: The Most Popular HTML, CSS, and JS library in the world, c2018. Página Inicial. Disponivel em: < http://getbootstrap.com>. Acesso em: 02 set. 2018.

CARDOSO, Sandra P. D. **Causas de Renúncia de Cães e Gatos Nos Concelhos de Cascais e Sintra**. 2013. 88 f. Dissertação (Mestrado em Medicina Veterinária) - Universidade Lusófona de Humanidades e Tecnologias, Lisboa, 2013. Disponível em: < http://recil.grupolusofona.pt/bitstream/handle/10437/5353/Tese%20-%20Sandra% 20Cardoso .pdf? sequence=1 >. Acesso em: 02 set. 2018.

DEEPAK, K. **Best Practices for Building RESTful Web Services**. Infosys Limited, 2015. Disponível em: < https://www.infosys.com/digital/insights/Documents/restful-web-services.pdf >. Acesso em: 15 out. 2018

DOEREFELD, B. **REST-API-maturity-richardson-maturity-model**, 2018. Disponível em: < https://nordicapis.com/what-is-the-richardson-maturity-model/rest-api-maturity-richardson-maturity-model/ >. Acesso em: 15 out. 2018

DUARTE, Nuno C. **Frameworks e Bibliotecas JavaScript**. 2015. 70 f. Dissertação (Mestrado em Engenharia Informática) – Instituto Superior de Engenharia do Porto, Porto, 2015. Disponível em: < http://recipp.ipp.pt/bitstream/10400.22/8223/1/DM_NunoDuarte_2015_MEI.pdf> Acesso em: 15 out. 2018.

FARACO, Ceres B. **Interação Humano-Cão: O social constituído pela relação interespécie**. 2008. 109 f. Tese (Doutorado em Psicologia) - Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2008. Disponível em: < http://repositorio.pucrs.br/dspace/bitstream/10923/4831/1/000400810-Texto%2bCompleto-0.pdf>. Acesso em: 24 ago. 2018.

FARIA, Thiago; Afonso, Alexandre. **Fullstack Angular e Spring - Guia para se tornar um desenvolvedor moderno**. 1 ed. Algaworks Software, 2018. Disponível em: < https://s3.amazonaws.com/algaworks-assets/ebooks/algaworks-livro-fullstack-angular-e-spring-v1.1.pdf >. Acesso em: 15 out. 2018

FRANCISCO, Rodrigo E. **Web Services REST Conceitos, análise e implementação**. 2016. Disponível em: < https://www.researchgate.net/publication/312116988/download >. Acesso em: 15 out. 2018

FREIRE, André P. **Acessibilidade no desenvolvimento de sistemas web: um estudo sobre o cenário brasileiro**. 2008, 154 f. Dissertação (Mestrado em Ciências da Computação e Matemática Computacional) - Universidade de São Paulo, São Carlos, 2008. Disponível em: < http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06052008-101644/publico/Dissertacao_Andre_Freire.pdf>. Acesso em: 02 set. 2018.

HOSTINGER. **Diferença entre HTML e HTML5**, c2018. Disponível em: < https://www.hostinger.com.br/tutoriais/diferenca-entre-html-e-html5/#gref >. Acesso em: 15 out. 2018

IHRIG, Collin. **The Basics of the Shadow DOM**, c2018. Disponível em: < https://www.sitepoint.com/the-basics-of-the-shadow-dom/ >. Acesso em: 15 out. 2018

JOFFILY, Diogo, et al. Medidas para o controle de animais errantes desenvolvidas pelo grupo Pet Medicina Veterinária da Universidade Federal Rural do Rio de Janeiro. **Em Extensão.,** Uberlândia, v. 12 n. 1 p. 197-211, jan - jun. 2013. Disponível em: < http://www.seer.ufu.br/index.php/revextensao/article/viewFile/20847/12670 >. Acesso em: 02 set. 2018

MONGODB, **What is MongoDB?**, c2018. Disponível em: < https://www.mongodb.com/what-is-mongodb >. Acesso em: 02 set. 2018

MONOBE, Felipe. **A Evolução do assíncrono no Node.js**. Disponível em: < https://medium.com/engenharia-noalvo/evolucao-assincrono-nodejs-p2-e7634d76218f >. Acesso em: 15 out. 2018

MONTEIRO, Rui T. **Arquitetura Front-end: WebComponents, Single-Page, Responsive e Offline-first**. 2015, 68 f. Dissertação (Mestrado em Engenharia Informática) – Universidade do Porto, Porto, 2015. Disponível em: < https://paginas.fe.up.pt/~ei10086/rui.monteiro/wiki/lib/exe/fetch.php?media=dissertacao:relato_rio_de_pdis_-_rui_monteiro.pdf >. Acesso em: 15 out. 2018

MOZILLA. **Funções Assíncronas**, c2018. Disponível em: < https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/funcoes_assincronas >. Acesso em: 15 out. 2018

MOZILLA. **Web Components**, c2018. Disponível em: < https://developer.mozilla.org/en-US/docs/Web/Web_Components >. Acesso em: 15 out. 2018

NODEJS. **About Node.js**, c2018. Disponível em: < https://nodejs.org/en/about/ >. Acesso em: 02 set. 2018.

PAULA, Silvana A. de. **Política Pública de esterilização cirúrgica de animais domésticos, como estratégia de saúde e de educação.** 2012. 43 f. Monografia de Especialização - Universidade Tecnológica Federal do Paraná, Curitiba, 2008.

Disponível em: < http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/1495/4/CT_GPM_II_2012_32.pdf >. Acesso em: 24 ago. 2018.

PIXINVENT. Materialize – **Material Design Admin Template**. Disponível em: < https://pixinvent.com/materialize-material-design-admin-template/html/semi-dark-menu/ >. Acesso em: 15 out. 2018

RALLO, Rafael. **Material Design: aprenda tudo sobre o design do Google**!. 2017. Disponível em: < https://marketingdeconteudo.com/material-design/ >. Acesso em: 15 out. 2018

SANTOS, Pedro. Algumas questões relativas ao encaminhamento de cães e gatos para adoção. **R@ u-Revista de Antropologia da UFSCar**., São Carlos, v. 7 n. 1 p. 230-247, jan-jun. 2015. Disponível em: < http://www.rau.ufscar.br/wp-content/uploads/2016/09/12_rau07104.pdf>. Acesso em: 24 ago. 2018.

SCHMITZ, Daniel. **Diga olá ao TypeScript e adeus ao JavaScript**. 2015. Disponível em: < https://tableless.com.br/diga-ola-ao-typescript-e-adeus-ao-javascript/ >. Acesso em: 15 out. 2018

SCHROEDER, Ricardo; dos Santos, Fernando. **ARQUITETURA E TESTES DE SERVIÇOS WEB DE ALTO DESEMPENHO COM NODE.JS E MONGODB**. Disponível em: < http://www.ceavi.udesc.br/arquivos/id_submenu/787/ricardo_schroeder_versao_final_.pdf >. Acesso em: 15 out. 2018

SOUSA, Filipe P. **Criação de framework REST/HATEOAS Open Source para desenvolvimento de APIs em Node.js**, 2015, 125f. Dissertação (Mestrado em Engenharia Informática). Universidade do Porto, Porto, 2015. Disponível em: < https://repositorio-aberto.up.pt/bitstream/10216/83505/2/35403.pdf >. Acesso em: 15 out. 2018

SPACEPROGRAMMER**. Introdução ao Modelo de Dados e seus níveis de abstração**. Disponível em: < http://spaceprogrammer.com/bd/introducao-ao-modelo-de-dados-e-seus-niveis-de-abstracao/ >. Acesso em: 15 out. 2018

STARTBOOTSTRAP. **SB Admin 2**. Disponível em: < https://startbootstrap.com/template-overviews/sb-admin-2/ >. Acesso em: 15 out. 2018

THOMAS, Dennis. **JavaScript Vs jQuery - Difference Between JavaScript And jQuery**. Disponível em: < https://www.c-sharpcorner.com/article/javascript-vs-jquery-difference-between-javascript-and-jquery/ > Acesso em: 15 out. 2018

TypeScript. Pagina Inicial, c2018. Disponível em: < https://www.typescriptlang.org/ >. Acesso em: 15 out. 2018.

W3TECHS. **Usage statistics and market share of Bootstrap for websites**. Disponível em: < https://w3techs.com/technologies/details/js-bootstrap/all/all >. Acesso em: 15 out. 2018

W3TECHS. **Usage statistics and market share of jQuery for websites**. Disponível em: < https://w3techs.com/technologies/details/js-jquery/all/all >. Acesso em: 15 out. 2018

APÊNDICE A – CONFIGURAÇÃO DO EXPRESS

```
// acess .env keys
require('dotenv').config();

// dependencies
const express = require('express');
const morgan = require('morgan');
const bodyParser = require('body-parser')

// setup the app
const app = express();
app.use(morgan('dev'));
app.use(bodyParser.json());

// test database connection
require('./database/mysql').queryDB('use dogsapi',function(status, result){
    if (status == 200) console.log('local mysql connected');
    else console.log(status + result);
});

// setup routers
require('./routers/dogRouter').setRouter(app);
require('./routers/userRouter').setRouter(app);

// start server
app.listen(3000, () => {
    console.log('listening on 3000')
})
```

APÊNDICE B – DOCUMENTAÇÃO DA API

```
FORMAT: 1A
HOST: http://localhost:3000

# dogsapi
TODO: Add Description

# Group Misc

## User 15 [/user/15]

### get user by id [GET]

+ Request
    + Headers

            x-access-
token:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsImVtYWlsIjoiZW1haWxAbWFpbC5jb20iLCJuYW1lIjoiam9hb3BhdWxvIiwiYWRtaW4iOm51bGwsImlhdCI6MTUzODI4MTc1NiwiZXhwIjoxNTM4MjgyMDU2fQ.lq25ymaDFlYJJr97KA1oZ5LVRJJlBzZaKmE1Wmjp1hc



+ Response 200




## User Login [/user/login]

### login [POST]

+ Request (application/json)
    + Attributes (loginRequest)

    + Body

            {
                "email": "usuario@usuario.com",
                "password": "eldorado"
            }


+ Response 200




## Dog Photo [/dog/photo]

### add photo [POST]

+ Request (application/json)
    + Attributes (add photoRequest)

    + Body

            {
                "dogId": "5babe0179454a831d48990c1",
                "user": "tomas",
```

```
                "url": "commais"
            }


+ Response 200



### add photo1 [POST]

+ Request (application/json)
    + Attributes (add photoRequest8)

    + Body

            {
                "user": 26,
                "dog": 16,
                "url": "cloudnary.com/"
            }


+ Response 200




## User [/user]

### add user [POST]

+ Request (application/json)
    + Attributes (add userRequest)

    + Body

            {
                "name": "joaopaulo",
                "email": "usuario@usuario.com",
                "password": "eldorado"
            }


+ Response 200



### get all users [GET]

+ Response 200




## Dog [/dog]

### add dog [POST]

+ Request (application/json)
    + Attributes (add dogRequest)
```

```
118
119     + Body
120
121             {
122                 "description": "fds",
123                 "gelded": true,
124                 "user": "nbnbv",
125                 "photo": "123",
126                 "name": "abcde"
127             }
128
129
130 + Response 200
131
132
133
134 ### create dog [POST]
135
136 + Request (application/json)
137     + Attributes (create dogRequest)
138
139     + Body
140
141             {
142                 "gelded": false,
143                 "description": "cachorrinho",
144                 "place": "utfpr",
145                 "size": "medium",
146                 "gender": "male",
147                 "color": "white",
148                 "user": 15,
149                 "photo": {
150                     "url": "cloudnary"
151                 }
152             }
153
154
155 + Response 200
156
157
158
159 ### get all dogs [GET]
160
161 + Request
162     + Headers
163
164             x-access-
    token:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsImVtYWlsIjoiZW1haWxAbWFpbC5jb2
    0iLCJuYW1lIjoiam9hb3BhdWxvIiwiYWRtaW4iOm51bGwsImlhdCI6MTUzODMxMDYyMSwiZXhwIjoxNTM4MzE
    wOTIxfQ.AO_03sZRe_ae5bck_8-hbGiQXjpleNJ3qiRbANQjkSM
165
166
167
168
169 + Response 200
170
171
172
173
174 ## User 5 [/user/5]
```

```
175
176 ### delete user by id [DELETE]
177
178 + Response 200
179
180
181
182
183 ## User Address [/user/address]
184
185 ### add address [POST]
186
187 + Request (application/json)
188     + Attributes (add addressRequest)
189
190     + Body
191
192             {
193                 "address": "rua osorio",
194                 "number": 45,
195                 "complement": "condominio",
196                 "district": "centro",
197                 "city": "pg",
198                 "state": "parana",
199                 "user": 15
200             }
201
202
203 + Response 200
204
205
206
207 ### update address [PUT]
208
209 + Request (application/json)
210     + Attributes (update addressRequest)
211
212     + Body
213
214             {
215                 "address": "rua general",
216                 "number": 45,
217                 "complement": "condominio",
218                 "district": "centro",
219                 "city": "pg",
220                 "state": "parana",
221                 "user": 15
222             }
223
224
225 + Response 200
226
227
228
229
230 ## Dog 33 [/dog/33]
231
232 ### get dog by id [GET]
233
234 + Request
```

```
    + Headers

            x-access-token:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsImVtYWlsIjoiZW1haWxAbWFpbC5jb20iLCJuYW1lIjoiam9hb3BhdWxvIiwiYWRtaW4iOm51bGwsImlhdCI6MTUzODMxMDYyMSwiZXhwIjoxNTM4MzEwOTIxfQ.AO_03sZRe_ae5bck_8-hbGiQXjpleNJ3qiRbANQjkSM



+ Response 200




## Dog 12 [/dog/12]

### delete dog [DELETE]

+ Request
    + Headers

            x-access-token:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsImVtYWlsIjoiZW1haWxAbWFpbC5jb20iLCJuYW1lIjoiam9hb3BhdWxvIiwiYWRtaW4iOm51bGwsImlhdCI6MTUzODMxMDYyMSwiZXhwIjoxNTM4MzEwOTIxfQ.AO_03sZRe_ae5bck_8-hbGiQXjpleNJ3qiRbANQjkSM



+ Response 200




## Dog Photo 12 [/dog/photo/12]

### delete photo [DELETE]

+ Request
    + Headers

            x-access-token:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsImVtYWlsIjoiZW1haWxAbWFpbC5jb20iLCJuYW1lIjoiam9hb3BhdWxvIiwiYWRtaW4iOm51bGwsImlhdCI6MTUzODMxMDYyMSwiZXhwIjoxNTM4MzEwOTIxfQ.AO_03sZRe_ae5bck_8-hbGiQXjpleNJ3qiRbANQjkSM



+ Response 200




## Dog Name [/dog/name]

### add name [POST]

+ Request (application/json)
```

```
    + Attributes (add nameRequest)

    + Body

            {
                "dog": "5babe0179454a831d48990c1",
                "user": "tomas",
                "name": "commais"
            }


+ Response 200




## Dog Query Addopted [/dog/query/addopted]

### get all dogs addopted [GET]

+ Request
    + Headers

            x-access-
token:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsImVtYWlsIjoiZW1haWxAbWFpbC5jb20iLCJuYW1lIjoiam9hb3BhdWxvIiwiYWRtaW4iOm51bGwsImlhdCI6MTUzODMxMDYyMSwiZXhwIjoxNTM4MzEwOTIxfQ.AO_03sZRe_ae5bck_8-hbGiQXjpleNJ3qiRbANQjkSM



+ Response 200




# Data Structures

## loginRequest (object)


### Properties
+ `email` (string, required)
+ `password` (string, required)


## add photoRequest (object)


### Properties
+ `dogId` (string, required)
+ `user` (string, required)
+ `url` (string, required)


## add userRequest (object)


### Properties
```

```
+ `name` (string, required)
+ `email` (string, required)
+ `password` (string, required)


## add dogRequest (object)


### Properties
+ `description` (string, required)
+ `gelded` (boolean, required)
+ `user` (string, required)
+ `photo` (string, required)
+ `name` (string, required)


## add addressRequest (object)


### Properties
+ `address` (string, required)
+ `number` (number, required)
+ `complement` (string, required)
+ `district` (string, required)
+ `city` (string, required)
+ `state` (string, required)
+ `user` (number, required)


## update addressRequest (object)


### Properties
+ `address` (string, required)
+ `number` (number, required)
+ `complement` (string, required)
+ `district` (string, required)
+ `city` (string, required)
+ `state` (string, required)
+ `user` (number, required)


## create dogRequest (object)


### Properties
+ `gelded` (boolean, required)
+ `description` (string, required)
+ `place` (string, required)
+ `size` (string, required)
+ `gender` (string, required)
+ `color` (string, required)
+ `user` (number, required)
+ `photo` (Photo, required)


## Photo (object)


### Properties
```

```
+ `url` (string, required)


## add photoRequest8 (object)


### Properties
+ `user` (number, required)
+ `dog` (number, required)
+ `url` (string, required)


## add nameRequest (object)


### Properties
+ `dog` (string, required)
+ `user` (string, required)
+ `name` (string, required)


```

ANEXO A: Termo de Responsabilidade Sobre Adoção de Animais

Termo de Responsabilidade

Ao adotar o animal descrito declaro-me apto para assumir a guarda e a responsabilidade sobre este animal, eximindo o doador de toda e qualquer responsabilidade por quaisquer atos praticados pelo animal a partir desta data.

Declaro ainda estar ciente de todos os cuidados que este animal exige no que se refere à sua guarda e manutenção, além de conhecer todos os riscos inerentes à espécie e raça no convívio com humanos, estando apto a guardá-lo e vigiá-lo, comprometendo-me a proporcionar boas condições de alojamento e alimentação, assim como, espaço físico que possibilite o animal se exercitar. Responsabilizo-me por preservar a saúde e integridade do animal e a submetê-lo aos cuidados médico-veterinários sempre que necessário para este fim.

Comprometo-me a não transmitir a posse deste animal a outrem sem o conhecimento do doador. Comprometo-me também, a permitir o acesso do doador ao local onde se encontra o animal para averiguação de suas condições.

Tenho conhecimento de que caso seja constatado por parte do doador situação inadequada para o bem estar do animal, perderei a sua guarda, sem prejuízo das penalidades legais. Comprometo-me ainda em ESTERILIZAR o animal adotado se o doador já não o tiver feito, contribuindo assim para o controle da população de animais domésticos.

Comprometo-me a cumprir toda a legislação vigente, municipal, estadual e federal, relativa à posse de animais.