

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME OLIVEIRA DOS SANTOS

**Desenvolvimento de um Gerenciador de
Smart Devices Usando o Protocolo MQTT e
uma Stack em Javascript para Modelar um
Cenário de IoT**

Monografia apresentada como requisito parcial
para a obtenção do grau de Bacharel em Ciência
da Computação

Orientador: Prof. Dr. Leandro Krug Wives

Porto Alegre
2017

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Rui Vicente Oppermann

Vice-Reitora: Prof. Jane Fraga Tutikian

Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Carla Maria Dal Sasso Freitas

Coordenador do Curso de Ciência de Computação: Prof. Raul Fernando Weber

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Queria agradecer ao meu pai, minha mãe e meu irmão, que sempre me deram apoio em toda esta empreitada. Aos meus colegas de faculdade e hoje amigos que me ajudaram diversas vezes nas disciplinas que cursamos juntos, fazendo trabalhos e estudando para provas. Aos colegas das empresas que trabalhei que contribuíram muito com o meu desenvolvimento técnico e pessoal. Aos meus orientadores Christopher Heinz - que me ajudou muito com esse trabalho no período em que estive na Alemanha - e, também, ao Prof. Leandro Wives pela disponibilidade, ajuda, incentivo e ter aceitado me orientar neste projeto.

RESUMO

O objetivo deste trabalho é propor uma solução escalável para o cenário de IoT, utilizando o protocolo MQTT, um banco de dados NoSQL e apenas a linguagem Javascript para realizar toda essa integração. O sistema permite controlar dispositivos inteligentes hipotéticos através de uma interface comum, podendo ser ligados, desligados, receber ou enviar mensagens específicas. As funcionalidades foram desenvolvidas com base em modelos já existentes para o controle desses dispositivos. Além disso, o foco desta aplicação também é testar um protocolo de comunicação específico para IoT, provando que é possível criar de forma simples, prática e moderna um cenário, usando tecnologias disponíveis no mercado.

Palavras-chave: AngularJS. Node.js. NoSQL. MQTT. IoT. Raspberry.

ABSTRACT

The goal of this work is purpose a escalable solution for a real IoT scenario. Using the MQTT protocol, a NoSQL database and only the Javascript language to create the whole integration. The application allows the user to control hypothetical smart devices through an interface, so they could be turn on, turn off or even receive or send specific messages. The functionalities were developed based on already existent models to control those devices. Besides that, one of the main objectives were test a specific communication protocol for IoT, proving it might be possible create in a easy way an IoT scenario using modern technologies which are already in use.

Keywords: AngularJS, Node.js, NoSQL, MQTT, IoT, Raspberry.

LISTA DE FIGURAS

Figura 2.1	Esquema de funcionamento do MQTT	16
Figura 2.2	Esquema de funcionamento do MQTT	16
Figura 2.3	Esquema de funcionamento do MQTT	17
Figura 3.1	Arquitetura da Aplicação	19
Figura 3.2	Estrutura de Pastas do Back-End	19
Figura 3.3	Simulador MQTT.fx	32
Figura 4.1	Diagrama de casos de uso	34
Figura 4.2	Tela Inicial	35
Figura 4.3	Tabela que lista os registros da entidade Node	36
Figura 4.4	Adicionando um registro da entidade Node	37
Figura 4.5	Adicionando um registro da entidade Prop	37
Figura 4.6	Publicando e Inscrevendo-se em tópicos	38

LISTA DE TABELAS

Tabela 3.1 Tabela de rotas da entidade Node	23
---	----

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
DOM	Document Object Model
NPM	Node Package Manager
HTML	HyperText Markup Language
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
MQTT	Message Queue Telemetry Transport
MVC	Model-View-Controller
NoSQL	Not Only Structured Query Language
CRUD	Create-Read-Update-Delete
REST	Representational State Transfer

SUMÁRIO

1 INTRODUÇÃO	10
1.1 Objetivo.....	11
1.2 Estrutura do texto	11
2 FUNDAMENTOS E TECNOLOGIAS	12
2.1 A Internet das Coisas.....	12
2.2 JavaScript	12
2.3 Cliente / Servidor	13
2.4 AngularJS	13
2.5 Bootstrap.....	14
2.6 Node.js.....	14
2.7 Express	15
2.8 MongoDB.....	15
2.9 MQTT	15
3 DESENVOLVIMENTO DA APLICAÇÃO	18
3.1 Metodologia de Desenvolvimento	18
3.2 Arquitetura	18
3.3 Banco de Dados e Back-End	19
3.3.1 Modelos.....	21
3.3.2 Rotas	23
3.3.3 Controladores	24
3.4 Desenvolvimento do Front-End	25
3.4.1 Rotas	26
3.4.2 Serviços.....	27
3.4.3 Diretivas	29
3.4.4 NgMQTT	31
3.5 Broker e Clientes.....	32
4 GUIA DE USO DA APLICAÇÃO	34
4.1 Casos de Uso	34
4.2 Navegação	35
4.2.1 Início	35
4.2.2 CRUDs e Publish/Subscribe	36
5 CONCLUSÃO	39
REFERÊNCIAS	41

1 INTRODUÇÃO

Muitas tecnologias surgiram na Web desde o aparecimento dela. Os sites, antes estáticos, tornaram-se mais dinâmicos por meio de novos navegadores. Esses, devido a novas linguagens de programação, começaram a oferecer variadas possibilidades. Aplicativos dinâmicos começaram a ser criados fazendo uso dessas tecnologias.

Esses novos aplicativos passaram a comunicar-se por meio de serviços e a trocar dados, tanto os gerados por eles quanto fora deles. Além disso, passou a existir uma maior preocupação com a usabilidade e o design dessas aplicações. Mais tarde, esses aplicativos que funcionavam passivamente, ou seja, quando em interação com um usuário, passaram também a trocar informações pró-ativamente.

Nesse contexto, ocorre o surgimento da *Internet of Things* (IoT). Segundo a (AMAZON, 2017), IoT foi um termo criado por Kevin Ashton, que a define como um conjunto de sistemas onipresentes, conectando um mundo físico à Internet. Embora as "coisas", a Internet e a conectividade sejam os três principais componentes da Internet, o diferencial da IoT é a aproximação do mundo digital com o mundo físico. Estima-se que, no ano de 2020, a IoT será composta de aproximadamente 20 bilhões de dispositivos.

Existem várias questões ainda em aberto em relação a IoT no que tange a escalabilidade, conectividade, privacidade e também no quesito de segurança. As oportunidades, como os desafios, também são muitos. Prevê-se o surgimento de novos modelos de negócio, novas formas de receita, melhoramento de produtividade em diversos setores, menor consumo de energia, entre outros.

O foco deste trabalho é mostrar um cenário de IoT usando tecnologias disponíveis no mercado. A aplicação mostra como gerenciar dispositivos hipotéticos, por meio de um aplicativo Web com uma interface gráfica com um design moderno e uma boa usabilidade. Também é utilizado o protocolo MQTT para comunicação dos dispositivos com a aplicação que os gerencia. Por fim, foi utilizado um banco de dados não-relacional, por esse garantir uma maior flexibilidade na modelação dos dados, não definindo nada no banco e sim na aplicação.

Atualmente existem algumas aplicações que realizam o gerenciamento de dispositivos distribuídos, no entanto um dos focos deste trabalho é testar um protocolo diferente do HTTP para fazer esse controle.

1.1 Objetivo

O objetivo deste trabalho é o desenvolvimento de uma aplicação Web para gerenciar dispositivos inteligentes em uma rede, simulando um cenário de uma casa inteligente. O sistema provê uma interface web para o usuário, sendo assim acessível por dispositivos móveis e computadores tradicionais. Nela, o usuário pode adicionar, atualizar ou obter informações dos dispositivos conectados na rede.

Além disso, o trabalho faz o uso de um protocolo específico para IoT, o MQTT, bem como de frameworks modernos como o Angular JS, o que facilita bastante o trabalho do desenvolvimento *Front-End*. Para o *Back-End*, o framework Express foi usado. Ele é responsável por enviar os dados para o banco de dados não relacional Mongo, e também por disponibilizar uma REST API em formato JSON para o lado cliente da aplicação, o *Front-End*.

1.2 Estrutura do texto

A abordagem seguida para estruturar este trabalho resultou em um texto da seguinte forma: primeiramente serão explicados os fundamentos e tecnologias usados, após isso será descrita a metodologia usada para o desenvolvimento da aplicação bem como trechos importantes e específicos da aplicação. A terceira parte é um guia de uso, juntamente com telas e instruções, e finalmente é apresentada a conclusão, discutindo problemas enfrentados e implementações futuras.

2 FUNDAMENTOS E TECNOLOGIAS

Neste trabalho, a maior parte das tecnologias empregadas foram tecnologias fortemente indicadas para Web, salvo algumas exceções. Essas tecnologias serão conceitualmente aqui descritas e alguns exemplos serão dados para o melhor entendimento.

2.1 A Internet das Coisas

Com a evolução da comunicação em dispositivos como microcontroladores, sistemas embarcados e sensores, esses ganharam a capacidade de se conectar à Internet. De forma bem resumida, a Internet das Coisas é a extensão da Internet atual (antes praticamente restrita a computadores comuns) às coisas, que são os outros dispositivos que adquiriram a capacidade de também se conectar à Internet.

A Internet das Coisas tem recebido bastante atenção, tanto no ramo acadêmico como no segmento industrial, visto que diversas novas possibilidades foram criadas por objetos que antes não tinham muita funcionalidade e hoje já são definidos como objetos inteligentes como Smartphones, Smart TVs, SmartWatches etc.

Uma das definições de Internet das Coisas (MATTERN; FLOERKEMEIER, 2010) é que as "coisas", terão habilidades de comunicação, umas com as outras, provendo dados, consumindo dados e ainda reagindo a eventos. Outra definição (SANTOS LUCAS A. M. SILVA, 2016) é a de que as "coisas" são objetos inteligentes, usando diferentes tipos de protocolos para a intercomunicação entre essas.

2.2 JavaScript

A Mozilla (MOZILLA, 2015) descreve Javascript como uma linguagem leve, interpretada, baseada em eventos, objetos e em funções de primeira classe. Ela é majoritariamente utilizada na Web, porém com o surgimento do Node.js (um interpretador de Javascript baseado na *engine* usada pelo navegador Google Chrome), passou a ser usada também fora dos navegadores. Ela é uma linguagem de *script* multi-paradigma, suportando assim o estilo funcional e também orientado a objetos.

A opção de usar Javascript como linguagem principal para este trabalho, foi muito pelo fato de poder programar em somente uma linguagem, tanto no lado do cliente, como

no lado do servidor. Além disso, há uma enorme gama e variedade de bibliotecas que tornam a prototipação rápida e prática.

O fato de a comunidade ser muito ativa na linguagem também foi fator determinante para a escolha, pois existem muitas opções de bibliotecas e frameworks. A grande quantidade dessas, porém, também é um dos contras da linguagem, pelo fato de dificultar as escolhas até para casos bem simples.

2.3 Cliente / Servidor

Como um dos objetivos do trabalho era criar um API REST disponibilizando uma interface de comunicação para outras aplicações consumirem, a arquitetura ficou dividida em duas partes: cliente e servidor.

O cliente tinha a responsabilidade de criar uma interface para o usuário manipular os dados. Ele que ficou encarregado de enviar as requisições de salvar, atualizar, deletar ou mostrar dados para o servidor, valendo-se da API REST criada pelo servidor. As tecnologias usadas no cliente foram os frameworks Angular e Bootstrap, assim como a biblioteca NgMQTT, para integrar o protocolo MQTT com essa camada.

O servidor ficou responsável por fazer a comunicação com o banco de dados não relacional e também por disponibilizar uma API REST para o cliente, para isso o Express Framework foi usado, em conjunto com o driver NoSQL Mongoose.

Os gerenciadores de pacote foram usados tanto no cliente como no servidor.

2.4 AngularJS

O AngularJS (ANGULARJS, 2010) é um framework MVC desenvolvido pela Google que tem por objetivo básico estender o HTML. Ele é baseado no conceito de diretivas, essas diretivas podem ser Controladores, Modelos, Serviços, injeção de dependências.

Ele foi fundamental para o design do *client-side*. Pelo fato de seguir uma arquitetura *Model-View-Controller*, a lógica de negócio ficou nos controladores. Esse tipo de arquitetura isola as partes da aplicação, assim não se tem a lógica de negócio no lado da *view* (HTML) e também a definição de entidades fica isolada nos modelos, e o controlador, como já dito, ficou encarregado da lógica de negócio.

Como ele facilita estender o HTML, *tags* específicas foram criadas com propósitos

específicos, assim tornando a integração do Javascript com o HTML uma tarefa mais agradável e também possibilita o reaproveitamento de código e a legibilidade.

A injeção de dependências foi também outro mecanismo amplamente usado, dessa formação, serviços puderam ser injetados dentro de controladores, de forma que o controlador pudesse usar sem preocupar-se se o serviço estava ou não instanciado.

Outros frameworks como o React também poderiam ser usados para o desenvolvimento deste projeto, no entanto optou-se pelo Angular por ele oferecer muitas "funções prontas", diferentemente do React, o qual muito código precisa ser escrito no início.

2.5 Bootstrap

Desenvolvedores de Software em geral não são bons com diagramação de interfaces. Por isso, optou-se por usar o Framework CSS Bootstrap (BOOTSTRAP, 2017). Ele facilitou a criação de layouts em HTML5 com o uso de classes CSS pré-definidas e de propósito geral, ou seja, tanto para Desktop como para dispositivos móveis.

Ele possui classes com diferentes propósitos como: sistema de grids, barra de navegação e formulários. Com elas, não precisei escrever código algum em CSS.

2.6 Node.js

Node.js (NODE.JS, 2017) é um interpretador de código Javascript baseado na *engine* do V8 do Chrome que permite rodar Javascript no lado do servidor. Tem por característica ser orientado a eventos, não-bloqueante, bastante leve sem perder eficiência.

Ele foi peça chave para a escolha da linguagem Javascript pelo fato de não precisar trabalhar com a sincronização de *threads*.

Possui uma um ecossistema grande como uma enorme variedade de bibliotecas. Além disso, possui um gerenciador de pacotes NPM (*Node Package Manager*) que torna o processo de instalação de novas bibliotecas muito simples e fácil.

O NPM foi usado tanto no cliente quanto no servidor, em ambos os casos ele serviu para a instalação dos frameworks e das bibliotecas usadas no desenvolvimento do projeto.

2.7 Express

A documentação oficial da biblioteca (EXPRESS, 2017) define o Express como um framework minimalista e flexível que, por sua vez, oferece um conjunto rico de recursos para criação de uma API em cima do protocolo HTTP.

O Express é um framework muito utilizado na comunidade Javascript. Basicamente com ele é muito fácil de criar Web APIs.

Era muito importante ter uma API para que outros clientes pudessem consumir os dados. Neste trabalho, temos somente um cliente, entretanto pelo uso do Express, muitos poderiam fazer uso dessa API.

2.8 MongoDB

MonogDB (MONGODB, 2017) é um banco de dados não estruturado e orientado a documentos. Num cenário de IoT, o uso desse tipo de banco de dados é de suma importância, pois os dispositivos que se conectam à rede têm diferentes atributos, e num banco de dados estruturado essa modelagem seria bastante complexa. Essa é a principal razão para a escolha do Mongo: a heterogeneidade dos dispositivos.

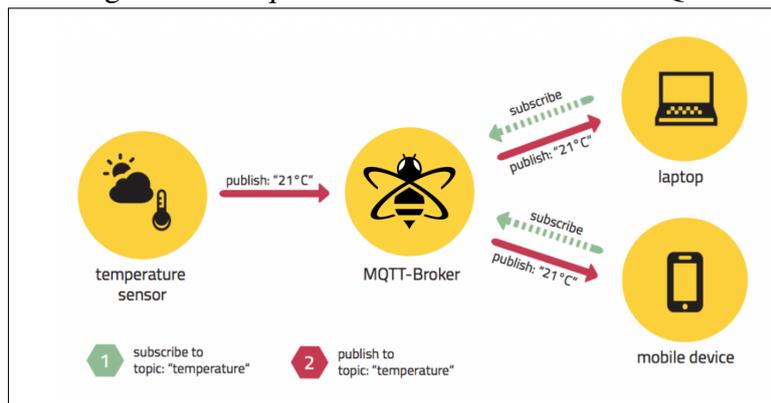
O MongoDB também é facilmente integrável com o ambiente Javascript. Neste projeto, fez-se uso da biblioteca Mongoose. Esse, atuou como um *driver* para as operações com o banco de dados.

2.9 MQTT

MQTT (MQTT, 2017) (*Message Queue Telemetry Transport*) é um protocolo de mensagens do tipo *publish-subscribe* criado para ser usado em *devices* limitados, com baixa *bandwidth*, alta latência e em redes não confiáveis. Essas características configuram o MQTT como um protocolo excelente para o cenário de IoT.

Para usar o protocolo MQTT foi necessário o uso de um *Broker*. Um *Broker* é um padrão de arquitetura que é responsável por coordenar a comunicação na rede, encaminhando requisições e mensagens. No caso desse projeto, foi usado o (MOSQUITTO, 2015), um *Broker open-source* que implementa o protocolo MQTT.

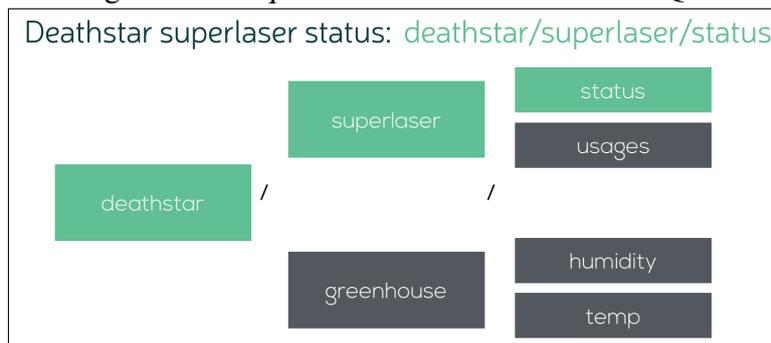
Figura 2.1: Esquema de funcionamento do MQTT



Fonte: (HIVEMQ, 2015)

O funcionamento do MQTT (HIVEMQ, 2015) é baseado em dois fatores chave: *publish-subscribe* e tópicos. Quando alguém quer emitir uma mensagem, deve publicar (*publish*) essa informação para um tópico. Quem tiver inscrito no determinado tópico (*subscribed*), irá receber a informação.

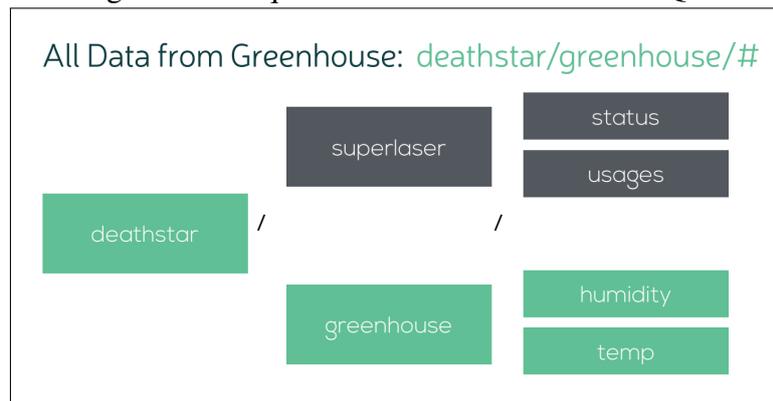
Figura 2.2: Esquema de funcionamento do MQTT



Fonte: (HIVEMQ, 2015)

Uma informação também pode ser publicada para mais de um tópico. O protocolo também provê um esquema de hierarquias, o que facilita a propagação da informação de forma mais otimizada, sem precisar especificar todos os tópicos abaixo daquela hierarquia.

Figura 2.3: Esquema de funcionamento do MQTT



Fonte: (HIVEMQ, 2015)

No próximo capítulo serão abordados os detalhes de implementação e como foi feita a integração do MQTT com o framework Angular. Também será demonstrada a arquitetura do projeto, explicando cada camada separadamente.

3 DESENVOLVIMENTO DA APLICAÇÃO

Este capítulo descreve o processo, a metodologia de desenvolvimento da aplicação, uma descrição da arquitetura e os detalhes de implementação de cada camada.

Como a aplicação foi dividida da seguinte forma: *Back-End*, *Front-End*, *Broker* e *Clients*, optou-se por descrever cada camada separadamente, assim tornando o entendimento das responsabilidades de cada uma.

3.1 Metodologia de Desenvolvimento

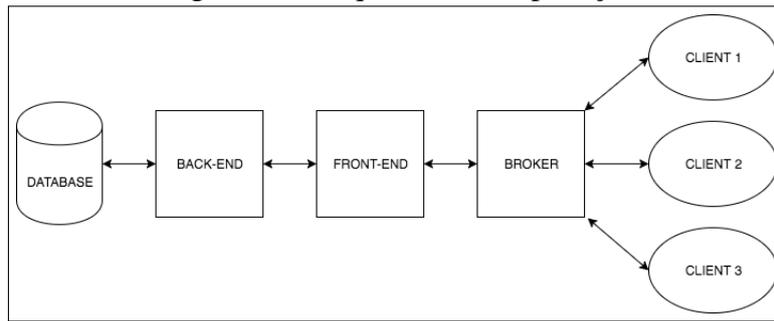
A metodologia de desenvolvimento foi feita de forma experimental, ou seja, muitas ferramentas e bibliotecas foram testadas para as decisões serem tomadas. A partir da ideia inicial que era simular um cenário de IoT, algumas decisões foram tomadas. A primeira foi em escolher um protocolo já existente e que provesse uma boa documentação. O MQTT foi escolhido. Depois decidimos que uma aplicação Web seria o ideal para comunicar-se com outros *Devices* através de uma API. Optou-se por escrever uma aplicação Web em Javascript por justamente fornecer uma grande opção de bibliotecas, boa documentação e agilidade de prototipação. Por fim, definimos que o banco deveria ser não-relacional, justamente pela heterogeneidade de propriedades que poderíamos ter nos *Devices*, nesses casos, um banco relacional demandaria bastante esforço para fazer as mudanças, enquanto com a opção que escolhemos requereu apenas mudanças no lado da aplicação.

3.2 Arquitetura

A arquitetura da aplicação foi dividida da forma como mostra a figura seguinte. Nas próximas sessões, serão mostrados os detalhes das implementações de cada camada.

A decisão de dividir a aplicação com essa arquitetura foi em razão de isolá-las e criar uma interdependência entre elas. Tendo uma camada só para o *Back-End* e provendo uma API para ela, torna a tarefa de mudar o *Front-End* mais fácil. Além disso, os clientes poderiam usar a API para cadastrar-se na rede num primeiro momento, implementação que não foi feita, porém influenciou na decisão.

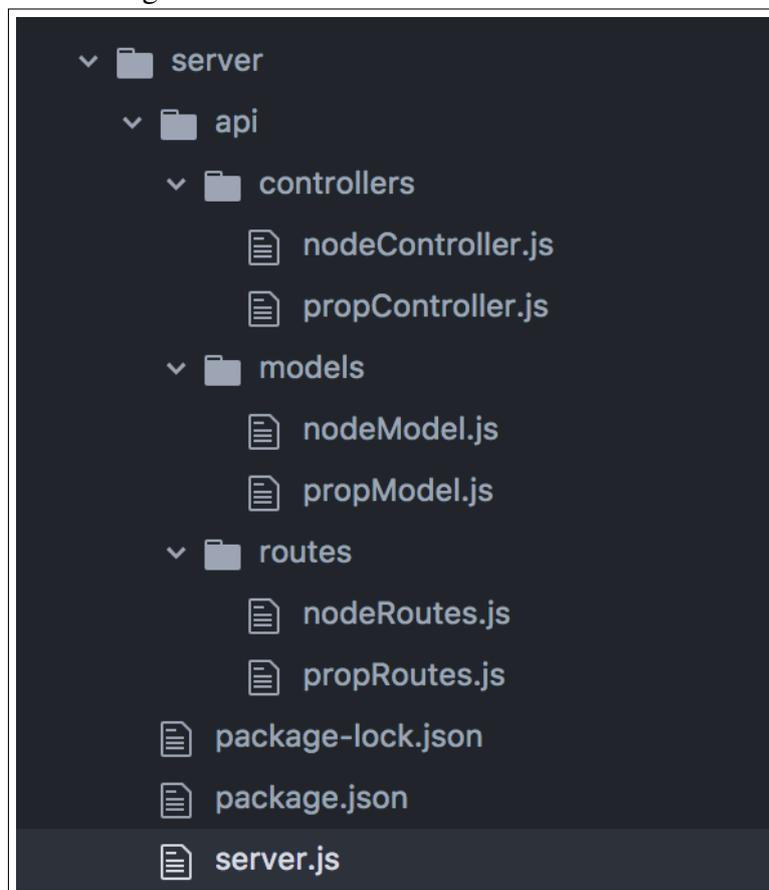
Figura 3.1: Arquitetura da Aplicação



3.3 Banco de Dados e Back-End

As camadas do banco de dados e do *Back-End* serão explicadas em conjunto, visto que a única parte que se comunicava com o banco de dados era o *Back-End*, e como o *Back-End* é quem contém a modelagem das entidades que refletem o banco de dados, optou-se por não separá-los um em cada sessão.

Figura 3.2: Estrutura de Pastas do Back-End



Para o desenvolvimento do *Back-End*, as principais bibliotecas usadas foram: Ex-

press, Mongoose e o banco de dados usado foi o MongoDB. Para a instalação dessas bibliotecas, o NPM foi utilizado.

Listing 3.1: Dependências do Back-End

```
"dependencies": {
  "body-parser": "^1.17.2",
  "cors": "^2.8.3",
  "express": "^4.15.3",
  "mongoose": "^4.10.5"
}
```

Foi criada uma API REST, responsável por fazer operações de leitura, escrita, atualização e deleção no banco de dados MongoDB. O arquivo *server.js* (Listing 3.2) conecta-se ao banco e também inicializa a aplicação *backend*.

Listing 3.2: Arquivo de entrada do Back-End

```
const express = require('express'),
  cors = require('cors'),
  app = express(),
  port = process.env.PORT || 3000,
  mongoose = require('mongoose'),
  Node = require('./api/models/nodeModel'),
  Prop = require('./api/models/propModel'),
  bodyParser = require('body-parser');

mongoose.Promise = global.Promise;
mongoose.connect('mongodb://localhost/bacon-lab');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(cors());

const nodeRoutes = require('./api/routes/nodeRoutes');
const propRoutes = require('./api/routes/propRoutes');

nodeRoutes(app);
propRoutes(app);

app.use(function(req, res) {
```

```
res.status(404).send({url: req.originalUrl + ' not found'})
});

app.listen(port);

console.log('RESTful API server started on: ' + port);
```

3.3.1 Modelos

Como o banco *MongoDB* é um *NoSQL*, as entidades foram definidas pela biblioteca *Mongoose*. No arquivo Javascript que segue, podemos ver a descrição da entidade *Node* (Listing 3.3). Ao importar a biblioteca *Mongoose*, um objeto do tipo *Schema* pode ser instanciado, descrevendo a estrutura da entidade que será usada para escrever informações dela no banco.

Listing 3.3: Modelo da entidade Node

```
'use strict';

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const NodeSchema = new Schema({
  iface: {
    type: String,
    required: true
  },
  name: {
    type: String,
    required: true
  },
  address: {
    type: String,
    required: true
  },
  last_heartbeat: {
    type: Date,
    default: Date.now,
    required: true
  }
});
```

```

    },
    props : [{
      type: Schema.Types.ObjectId,
      ref: 'Prop'
    }]
  });

module.exports = mongoose.model('Node', NodeSchema);

```

O modelo Prop (Listing 3.4), referente às propriedades de cada Node também foi estruturado usando a mesma biblioteca, e de forma bastante parecida. Porém é importante salientar que ela contém uma referência para a entidade Node na sua estrutura, ou seja, a entidade *Node* tem *Props*, estabelecendo assim, uma relação 1:N.

Listing 3.4: Modelo da entidade Prop

```

'use strict';

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const PropSchema = new Schema({
  _creator: {
    type: Schema.Types.ObjectId,
    ref: 'Node'
  },
  name: {
    type: String,
    required: true
  },
  value: {
    type: String,
    required: true
  },
  type: {
    type: String,
    required: true
  }
});

module.exports = mongoose.model('Prop', PropSchema);

```

3.3.2 Rotas

Com o suporte da biblioteca Express, uma API REST foi disponibilizada com as rotas que seguem. As *rotas* juntamente com os *métodos HTTP* usados foram:

Tabela 3.1: Tabela de rotas da entidade Node

Método	URI	
GET	/nodes	Busca todos os resources do tipo Node
GET	/nodes/:id	Busca pelo id um resource
POST	/nodes	Cria um novo resource Node
PUT	/nodes/:id	Atualiza pelo id um resource Node
DELETE	/nodes/:id	Remove do banco pelo id um resource Node

A Tabela 3.1 é representada no código da aplicação por um arquivo de rotas. Esse arquivo recebe uma instância da biblioteca Express que se encarrega de encaminhar para a rota certa, usando o verbo HTTP, e os parâmetros enviados, como por exemplo o id, para isso.

Listing 3.5: Arquivo de rotas da entidade Node

```
'use strict';

module.exports = function(app) {
  var nodeController = require('../controllers/nodeController');

  app.route('/api/nodes')
    .get(nodeController.getAll)
    .post(nodeController.create);

  app.route('/api/nodes/:id')
    .get(nodeController.get)
    .put(nodeController.update)
    .delete(nodeController.remove);
};
```

3.3.3 Controladores

Definido os modelos e as rotas, a orquestração das chamadas fica por conta dos controladores. Dado que a entidade já foi definida pela biblioteca Mongoose, e o Express também já se encarregou de encaminhar a requisição para a devida rota. Falta fazer a chamada para escrever, ler, deletar ou atualizar o registro no banco de dados. O controlador ficou com essa função.

Listing 3.6: Controlador da entidade Node

```
'use strict';

const mongoose = require('mongoose');
const Node = mongoose.model('Node');

exports.getAll = function(req, res) {
  Node.find(req.query, function(err, nodes){
    if (err)
      return res.status(500).send(err)

    return res.send(nodes);
  });
};

exports.create = function(req, res) {
  let node = new Node(req.body);

  node.save(function(err, node){
    if (err)
      return res.status(500).send(err)

    return res.json(node);
  });
};

exports.get = function(req, res) {
  Node.findById(req.params.id, function(err, node){
    if (err)
      return res.status(500).send(err)
```

```
    return res.json(node);
  });
};

exports.update = function(req, res) {
  Node.findOneAndUpdate(req.params.id, req.body, {new: true}, function(
    err, node) {
    if (err)
      return res.status(500).send(err)

    return res.json(node);
  });
};

exports.remove = function(req, res) {
  Node.findOneAndRemove(req.params.id, function(err, node) {
    if (err)
      return res.status(500).send(err)

    return res.json({message: "Record successfully deleted."});
  });
};
};
```

3.4 Desenvolvimento do Front-End

Para a criação do *Front-End*, uma série de bibliotecas foram usadas. As principais foram: AngularJS para extensão do HTML, Gulp para a automação de tarefas, Bootstrap para a criação de estilos (CSS) e a biblioteca ngmqtt, biblioteca possibilitou a integração do MQTT com essa camada.

Para a instalação dessas bibliotecas, dois gerenciadores de pacote foram usados: Bower e NPM. Com essas ferramentas de gerenciamento de bibliotecas por meio de um JSON, esses gerenciadores conseguem identificar e instalar as bibliotecas listadas.

A escolha de dois gerenciadores de aplicação para essa aplicação se deve ao fato de que o Bower é mais usado para dependências de CSS como Bootstrap, o NPM foi mais usado para o gerenciamento do Framework AngularJS.

Listing 3.7: Dependencias do Front-End

```
{
  "name": "client",
  "version": "0.0.0",
  "dependencies": {
    "angular": "^1.4.0",
    "bootstrap": "^3.2.0",
    "angular-resource": "^1.4.0",
    "angular-route": "^1.4.0",
    "ngmqtt": "*"
  }
}
```

3.4.1 Rotas

Assim como no *Back-End*, o *Front-End* também precisava de um gerenciador de rotas para a navegação entre as páginas. Para isso, uma diretiva de rotas foi usada. Essa é uma funcionalidade do Framework Angular e como pode ser visto na figura 3.10.

Listing 3.8: Rotas Front-End

```
'use strict';

/**
 * @ngdoc overview
 * @name clientApp
 * @description
 * # clientApp
 *
 * Main module of the application.
 */
angular
  .module('clientApp', [
    'ngResource',
    'ngRoute',
    'ngmqtt'
  ])
  .config(function ($routeProvider, $locationProvider) {
    $routeProvider
      .when('/', {
```

```

    templateUrl: 'views/main.html',
  })
  .when('/nodes', {
    templateUrl: 'views/nodes.html',
    controller: 'NodesmanagerCtrl',
  })
  .when('/nodes/create/', {
    templateUrl: 'views/node-create.html',
    controller: 'NodescreateCtrl',
  })
  .when('/nodes/update/:id', {
    templateUrl: 'views/node-update.html',
    controller: 'NodesupdateCtrl',
  })
  .when('/nodes/props/create/:id', {
    templateUrl: 'views/prop-create.html',
    controller: 'PropscreateCtrl',
  })
  .otherwise({
    redirectTo: '/'
  });

  $locationProvider.hashPrefix('');

}).run(function($rootScope, $location){
  $rootScope.isActive = function (viewLocation) {
    return viewLocation === $location.path();
  };
});

```

3.4.2 Serviços

O Angular também oferece uma forma bastante simples de usar serviços, ele implementa o padrão de projeto: Factory. Usando esse padrão, podemos criar objetos sem expor a lógica para outras partes da aplicação (PATTERN, 2017).

Listing 3.9: Serviço da entidade Node

```
'use strict';
```

```

/**
 * @ngdoc service
 * @name clientApp.nodeService
 * @description
 * # nodeService
 * Factory in the clientApp.
 */
angular.module('clientApp')
  .factory('NodeService', function ($http, $resource) {
    return $resource('http://localhost:3000/api/nodes/:id/', { id: '@_id' }, {
      update: {
        method: 'PUT'
      }
    });
  });
});

```

Assim, nos controladores, os métodos de salvar, obter, deletar ou atualizar, podiam ser chamados diretamente, apenas um objeto de serviço precisava ser instanciado e lógica alguma precisava ser implementada.

Listing 3.10: Usando o serviço criado

```

'use strict';

/**
 * @ngdoc function
 * @name clientApp.controller:NodescreateCtrl
 * @description
 * # NodescreateCtrl
 * Controller of the clientApp
 */
angular.module('clientApp')
  .controller('NodescreateCtrl', function ($scope, $location,
    NodeService) {
    $scope.title = 'add';

    $scope.node = new NodeService();

    $scope.submitNode = function () {

```

```

$scope.node.$save(function (node) {
    $location.url(`#/nodes/${$scope.node._id}`);
});

};

});

```

3.4.3 Diretivas

Um dos componentes mais poderosos do framework Angular é a criação de diretivas, com essas podemos estender o HTML de forma que novas *tags* podem ser criadas.

Listing 3.11: Definindo uma diretiva

```

'use strict';

/**
 * @ngdoc directive
 * @name clientApp.directive:nodeForm
 * @description
 * # nodeForm
 */
angular.module('clientApp')
  .directive('nodeForm', function () {
    return {
      templateUrl: '../views/node-form.html',
      restrict: 'E'
    };
  });

```

No código a cima, mostra a criação de uma nova *tag* HTML para um formulário. Os detalhes da implementação desse formulário ficaram isolados em um arquivo HTML, como pode ser visto no trecho de código seguinte.

Listing 3.12: Implementando a Diretiva

```

<h2 class="text-capitalize">{{title}} node </h2>
<form>

```

```

<div class="form-group">
  <label for="name">Name:</label>
  <input type="text" ng-model="node.name" class="form-control" id="
    name">
</div>
<div class="form-group">
  <label for="interface">Interface:</label>
  <input type="text" ng-model="node.iface" class="form-control" id="
    interface">
</div>
<div class="form-group">
  <label for="address">Address:</label>
  <input type="text" ng-model="node.address" class="form-control" id=
    "address">
</div>
<div class="form-group">
  <label for="last-heartbeat">Last Heartbeat:</label>
  <input type="text" ng-model="node.last_heartbeat" ng-readonly="true
    " class="form-control" id="last-heartbeat">
</div>
<div class="form-group">
  <button type="button" class='btn btn-primary text-capitalize'
    ng-click="submitNode()" >{{title}} Node</button>
</div>
</form>

```

Por fim, o uso dessa nova *tag*, pode ser visto no *Listing 3.13*. A grande vantagem do uso de diretivas, é que uma vez criada, ela pode ser reutilizada em diversas partes do código. Como o formulário de criar um Node e atualizar um Node era o mesmo. Essa diretiva pode ser reutilizada, assim, menos código precisou ser escrito.

Listing 3.13: Usando a diretiva criada

```

<node-form>
</node-form>

```

3.4.4 NgMQTT

A integração do MQTT com o *Front-End* foi feita com a biblioteca NgMQTT. Por meio dela, o usuário poderia publicar informações para o Broker, como mostra a Figura 3.16.

Listing 3.14: Publicando mensagens

```
$scope.publish = function(prop) {
  const topic = prop.name;
  const message = prop.value;
  ngmqtt.listenConnection("NodesupdateCtrl", function() {
    ngmqtt.publish(`//${$routeParams.id}/${topic}`, message.toString());
  });
};
```

A API dela também disponibilizava uma forma de inscrever-se em tópicos, como mostrado abaixo.

Listing 3.15: Inscrevendo- em tópicos

```
$scope.subscribe = function(topic) {
  ngmqtt.listenConnection("NodesupdateCtrl", function() {
    ngmqtt.subscribe(`//${$routeParams.id}/${topic}`);
  });
};
```

Por fim, para manter esses dados atualizados na página, era preciso que a biblioteca sempre estivesse procurando por eventos relacionados aos tópicos. Isso era feito no controlador dos Nodes. Esse controlador, usando a característica do Javascript de ser orientado a eventos, estava sempre verificando se novas informações tinham sido publicadas para os tópicos em que o determinado Node estava inscrito.

Listing 3.16: Capturando as mensagens publicadas

```
ngmqtt.listenMessage("NodesupdateCtrl", function(topic, message) {
  const [ , _creator, name ] = topic.split('/');

  PropService.query({ _creator: _creator, name: name }, function(props) {
    let [ prop ] = props;
```

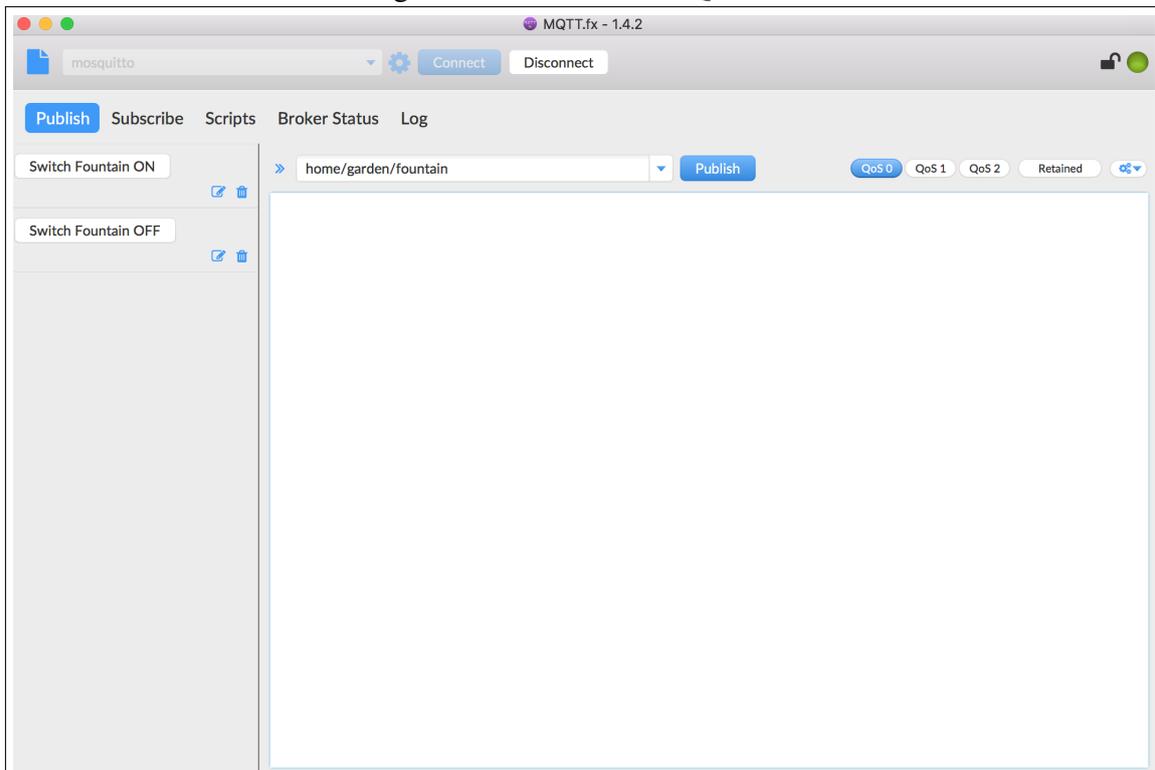
```
prop.value = message.toString();
prop.$update(function () {
  $scope.props = PropService.query({ _creator: $routeParams.id });
});
});
});
```

3.5 Broker e Clientes

Dois clientes foram usados para testar a aplicação: um Raspberry conectado na rede em que estava o *Broker* e o simulador MQTT.fx.

Primeiramente os testes foram realizados por meio do software MQTT.fx, ele provê uma interface para publicar mensagens e também inscrever-se em tópicos.

Figura 3.3: Simulador MQTT.fx



Com o MQTT.fx, eu pude utilizar das funcionalidades do protocolo, sem a necessidade de escrever código para os testes. Pela interface do simulador, eu publicava informações para os devidos tópicos com as respectivas mensagens e podia verificar no

aplicativo se essas tinham sido recebidas. Dessa forma eu pude prosseguir para a próxima etapa, que se caracterizou pelo uso de um microcontrolador.

Após o sucesso com o teste no simulador, um Raspberry foi conectado com sensores de umidade e temperatura. Para o envio da informação desses sensores, um *script* em linguagem Python foi escrito para desempenhar tal função.

Listing 3.17: Script em python do sensor de movimento

```
import RPi.GPIO as GPIO
import time
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)
GPIO.setup(11, GPIO.IN)           #Read output from PIR motion sensor
GPIO.setup(3, GPIO.OUT)          #LED output pin
while True:
    i=GPIO.input(11)
    if i==0:                      #When output from motion sensor is LOW
        print "No intruders",i
        GPIO.output(3, 0)        #Turn OFF LED
        time.sleep(0.1)
    elif i==1:                   #When output from motion sensor is HIGH
        print "Intruder detected",i
        GPIO.output(3, 1)        #Turn ON LED
        time.sleep(0.1)
```

Tanto o simulador como o Raspberry enviavam suas informações para o *Broker*, esse, por sua vez, encaminhava as mensagens para a aplicação que estava também conectada no *Broker*, assim, cada *Node* podia ter sempre os dados das suas respectivas Props atualizadas.

É importante salientar um aspecto que diz respeito a como os tópicos foram utilizados neste projeto. A primeira parte do tópico correspondia ao endereço MAC Address do Node, a segunda parte era composta pelo nome da Prop e por último a mensagem dessa Prop, ou seja, MACAddress/PropName/PropValue.

4 GUIA DE USO DA APLICAÇÃO

Neste capítulo é feita uma análise da aplicação desenvolvida e também é ilustrada a forma como o usuário pode navegar no aplicativo, bem como as funcionalidades oferecidas para ele.

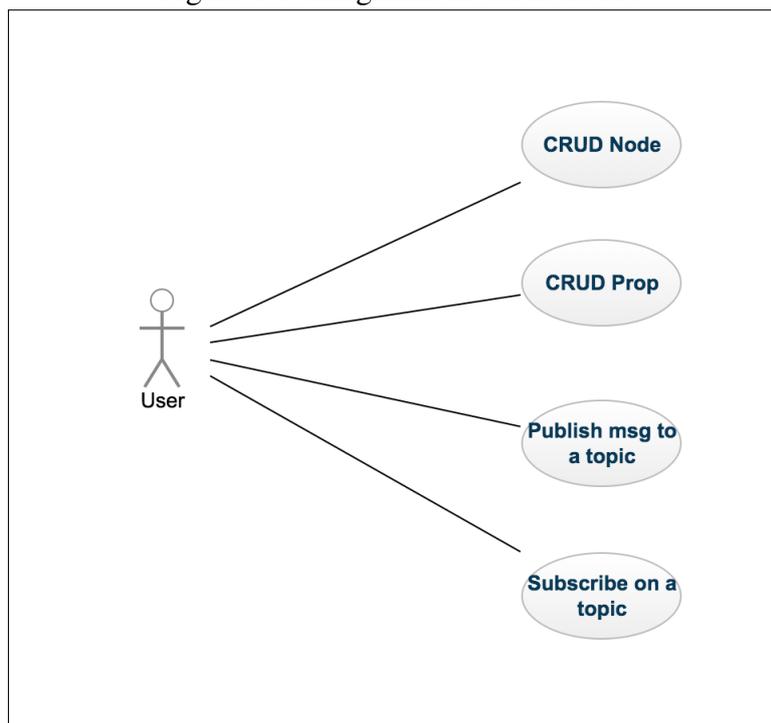
4.1 Casos de Uso

De forma sucinta as operações que podem ser feitas na aplicação são ilustradas pelo diagrama de casos de uso da Figura 4.1.

Os casos de uso descritos como CRUD (Create, Read, Update, Delete), incluem as operações de criar, ler, atualizar e deletar um registro da entidade Node, bem como da entidade Prop.

As ações de publicar e inscrever-se em um tópico são representadas por *Publish* e *Subscribe*. Essas ações são as principais funcionalidades do sistema, elas enviam informações para o *Broker* que as encaminha para o cliente correspondente.

Figura 4.1: Diagrama de casos de uso



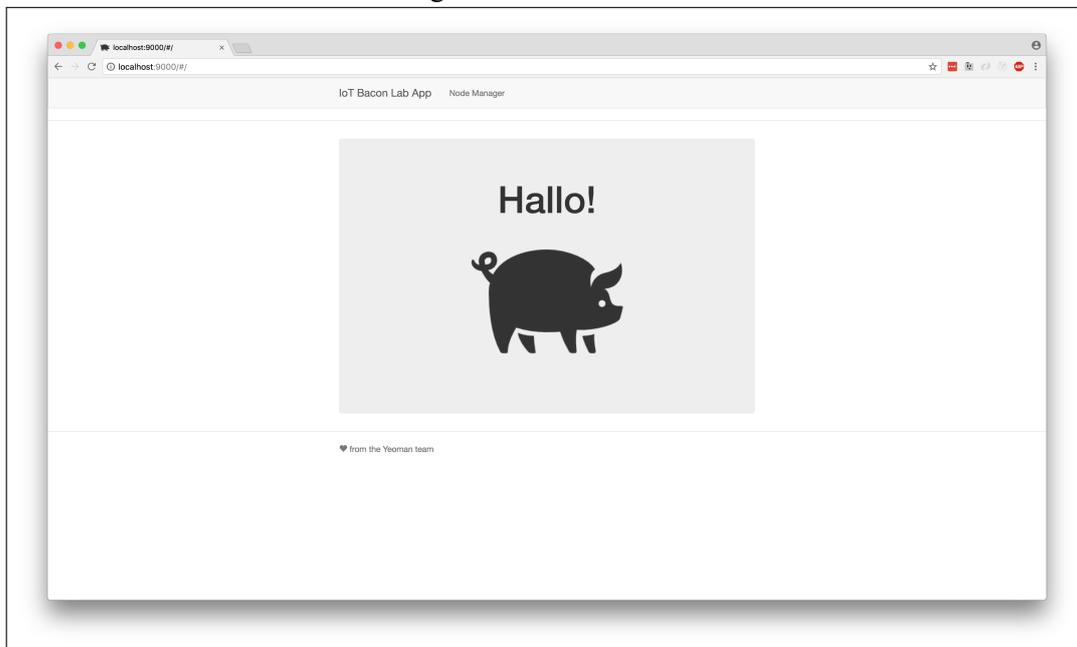
4.2 Navegação

Nesta seção são mostradas como a aplicação foi organizada, ilustrando com o uso de imagens e descrevendo melhor os casos de uso descritos na sessão anterior.

4.2.1 Início

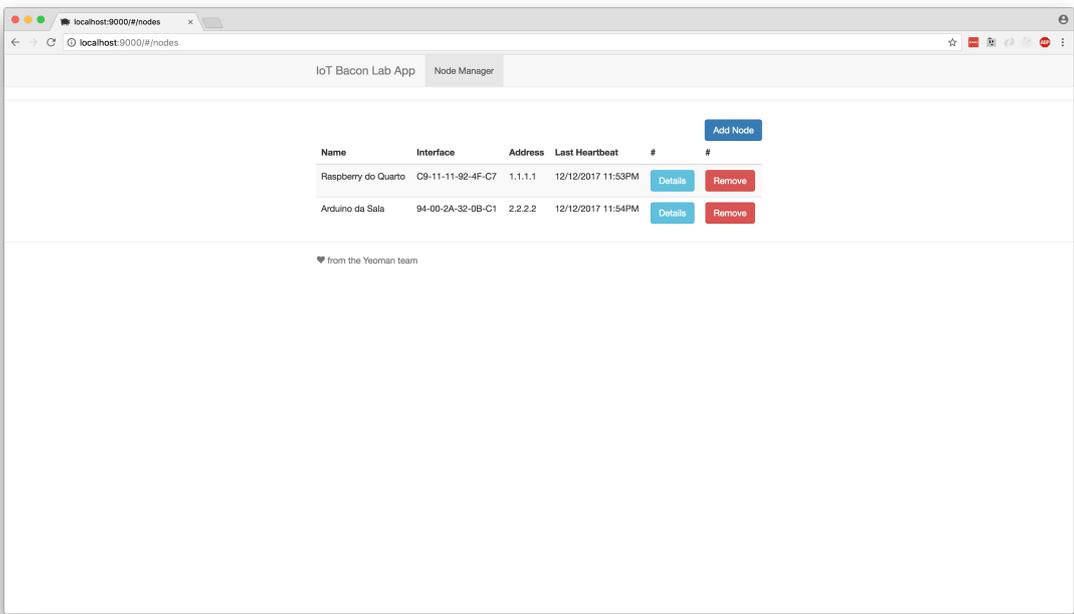
A tela de início, como mostra a Figura 4.2 tem apenas algumas informações sobre a aplicação e a barra de navegação na qual o usuário é redirecionado para a página onde os *Nodes* podem ser gerenciados.

Figura 4.2: Tela Inicial



Como ilustrado na Figura 4.3, algumas ações do diagrama de casos de uso já podem ser mostradas, são elas: Criação de um *Node*, atualização de um *Node*, listagem dos *Nodes* e também a deleção desses.

Figura 4.3: Tabela que lista os registros da entidade Node



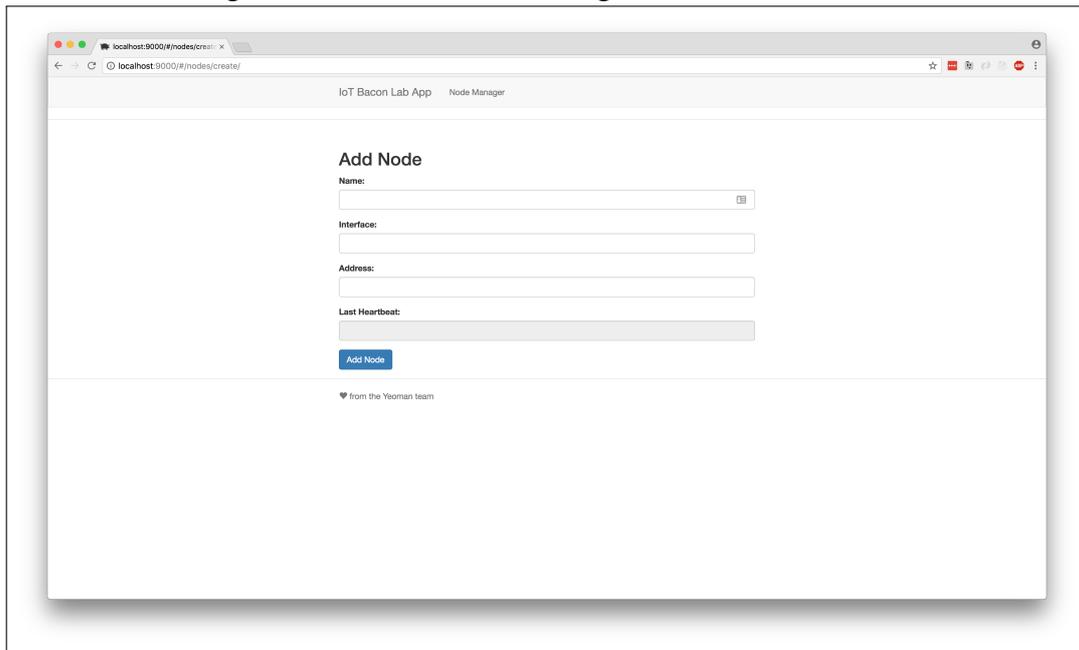
Name	Interface	Address	Last Heartbeat	#	#	
Raspberry do Quarto	C9-11-11-92-4F-C7	1.1.1.1	12/12/2017 11:53PM			Details Remove
Arduino da Sala	94-00-2A-32-0B-C1	2.2.2.2	12/12/2017 11:54PM			Details Remove

from the Yeoman team

4.2.2 CRUDs e Publish/Subscribe

Para um usuário poder inscrever-se em tópicos, ele precisa ter algum Node cadastrado. A Figura 4.4 mostra a funcionalidade de criação de um Node. Importante notar que o campo referente a última vez que o Node publicou ou inscreveu-se em algum tópico é autogerenciado pela aplicação, ou seja, o usuário não precisa entrar esse campo, ele é apenas de leitura.

Figura 4.4: Adicionando um registro da entidade Node



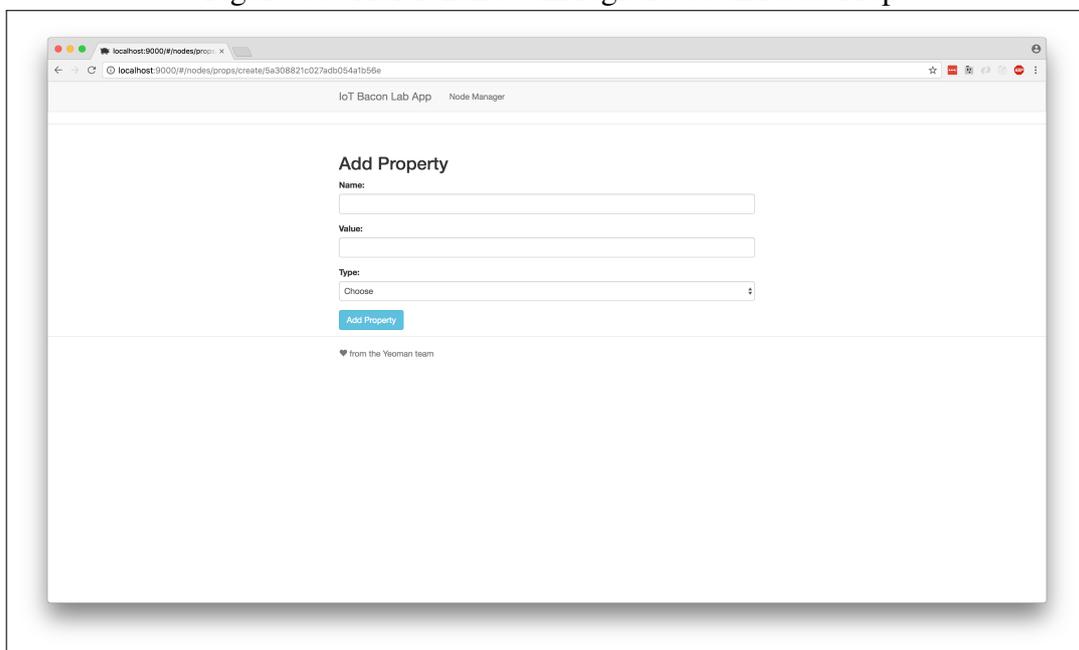
The screenshot shows a web browser window with the URL `localhost:9000/#/nodes/create/`. The page title is "IoT Bacon Lab App Node Manager". The main content is a form titled "Add Node" with the following fields:

- Name:
- Interface:
- Address:
- Last Heartbeat:

Below the fields is a blue "Add Node" button. At the bottom of the page, there is a small logo and the text "from the Yeoman team".

Além de ter além de um Node, pelo menos uma Prop cadastrada referente a esse Node. A Figura 4.5 descreve o cenário de uso de criação de uma Prop, essa tela também é a mesma em caso de atualização de um registro dessa entidade.

Figura 4.5: Adicionando um registro da entidade Prop



The screenshot shows a web browser window with the URL `localhost:9000/#/nodes/props/create/5a308821c027adb054a7b056e`. The page title is "IoT Bacon Lab App Node Manager". The main content is a form titled "Add Property" with the following fields:

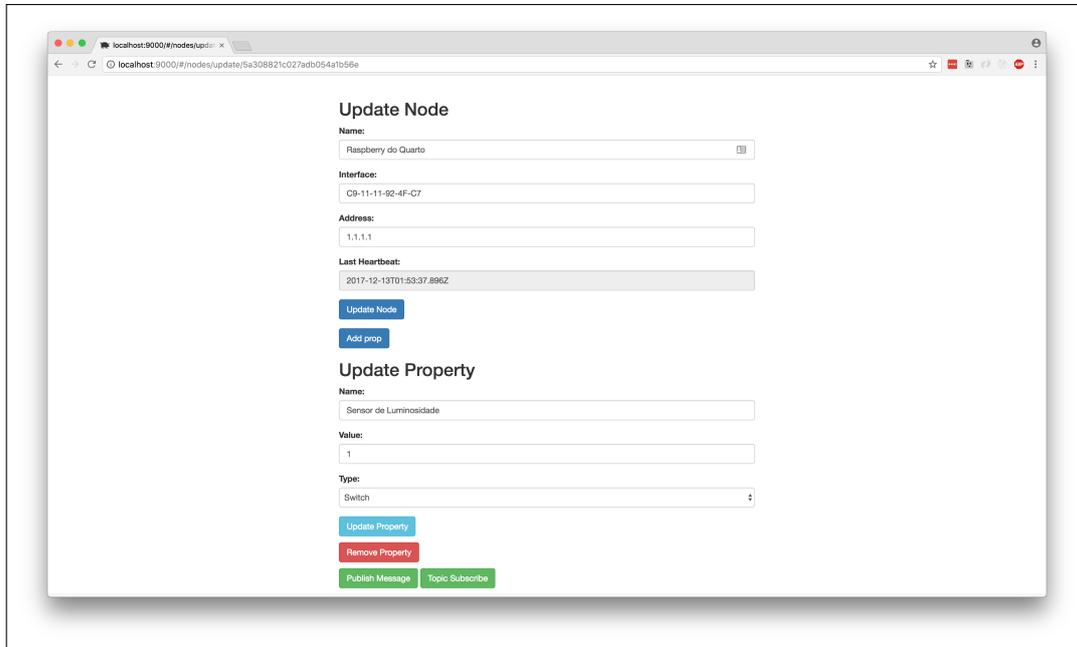
- Name:
- Value:
- Type:

Below the fields is a blue "Add Property" button. At the bottom of the page, there is a small logo and the text "from the Yeoman team".

Uma vez que um *Node* já foi cadastrado na base de dados e esse Node possui também possui atributos (Props), o usuário pode enfim publicar e inscrever-se em tópicos. Os botões *publish* e *subscribe* têm esse propósito, eles enviam as informações referentes ao campo *Name* e também ao campo *Value* no caso de um *publish*, ou simplesmente

inscrevem-se em um tópico com o valor contido no campo *Name*, no caso de um *subscribe*.

Figura 4.6: Publicando e Inscrevendo-se em tópicos



5 CONCLUSÃO

Neste trabalho foi apresentada uma aplicação de gerenciamento, simulando um cenário de IoT. Neste capítulo, comentarei brevemente sobre as impressões que tive referente as decisões técnicas que foram tomadas, ao passo que também explicitarei quais seriam alguns dos possíveis trabalhos futuros para o projeto.

No que tange as tecnologias utilizadas neste projeto, Javascript foi com certeza uma decisão acertada. A forte comunidade contribuiu em diminuir a curva de aprendizado necessária para a construção de conhecimento sobre da linguagem. No entanto, algumas tecnologias durante o desenvolvimento da aplicação foram deprecadas, dentre elas o framework Angular e o gerenciador de pacotes Bower.

Isso visivelmente expõe um dos problemas da linguagem: a forte segmentação criada pela grande diversidade de opções de bibliotecas, e constante mudança e rápida evolução dessas. Exemplificando: três novas versões sem retrocompatibilidade do framework angular foram disponibilizadas durante o desenvolvimento deste trabalho (ALURA, 2017). Não obstante, a biblioteca que realizava a integração do Angular com o MQTT, também foi descontinuada.

O uso do MQTT também se mostrou uma ótima decisão, visto que foi relativamente simples usá-lo, tanto no lado do cliente, quanto nos *devices*. As inúmeras opções de informações disponíveis na Web sobre ele, alavancam a escrita de aplicações usando essa tecnologia.

Em relação a trabalhos futuros, devido ao fato de o gerenciador de pacotes Bower ter sido descontinuado, juntamente com o framework Angular e a biblioteca que realizava a integração do MQTT com esse, a aplicação precisaria ser reescrita em uma tecnologia nova e mais estável como, por exemplo, React ou o novo Angular que promete retrocompatibilidade de versões a partir de agora.

Como o foco do projeto era criar um gerenciador de dispositivos inteligentes hipotéticos e simular um cenário de IoT, a autenticação de usuário e também a segurança foram negligenciadas em detrimento das funcionalidades principais que eram o gerenciamento e a simulação. Porém o próximo passo seria construir uma tela de login e cadastro de usuários, assim podendo isolar as permissões e dados de cada um.

Outra implementação futura seria o uso de uma solução para resolver o problema da interoperabilidade entre os dispositivos e suas propriedades. Uma das possíveis implementações para a resolução disso, seria o uso do Hypercat. Ele (HYPERCAT, 2015) é

uma ferramenta que propõe um padrão, assim tornando viável clientes descobrirem informações sobre os dispositivos por meio de um catálogo de metadados, de forma a mitigar a criação de silos de informação.

REFERÊNCIAS

- ALURA. **Alura - AngularJS, Angular 1, Angular 2 ou Angular 4? Que confusão!** 2017. <blog.alura.com.br/angularjs-angular-1-angular-2-ou-angular-4-que-confusao/>. [Online: acessado 01 de Janeiro de 2018].
- AMAZON. **What is the internet of things.** 2017. <<https://aws.amazon.com/pt/iot/what-is-the-internet-of-things/>>. [Online: Acesso em: 06 Nov 2017].
- ANGULARJS. **Concepts.** 2010. <<https://docs.angularjs.org/guide/concepts>>. [Online: Acessado 12 de Setembro de 2017].
- BOOTSTRAP. **Bootstrap - About Bootstrap.** 2017. <<https://getbootstrap.com/docs/4.0/about/overview/>>. [Online: Acessado 29 de Agosto de 2017].
- EXPRESS. **Express — Fast, unopinionated, minimalist web framework for Node.js.** 2017. <<http://expressjs.com/>>. [Online: acessado 13 de Outubro de 2017].
- HIVEMQ. **BMQTT Essentials Part 2: Publish & Subscribe.** 2015. <<https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe>>. [Online: acessado 10 de Dezembro de 2017].
- HYPERCAT. **Hypercat - Hypercat is a Global Alliance and standard (PAS 212) driving secure and interoperable Internet of Things (IoT) for Industry and cities.** 2015. <<http://www.hypercat.io/>>. [Online: acessado 10 de Dezembro de 2017].
- MATTERN, F.; FLOERKEMEIER, C. From the internet of computers to the internet of things. **Paper**, 2010.
- MONGODB. **MongoDB - Introduction to MongoDB.** 2017. <<https://docs.mongodb.com/manual/introduction/>>. [Online: acessado 1 de Setembro de 2017].
- MOSQUITTO. **Mosquitto - An Open Source MQTT Broker.** 2015. <<https://mosquitto.org/>>. [Online: acessado 10 de Dezembro de 2017].
- MOZILLA. **JavaScript.** 2015. <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>>. [Online: Acessado 20 de Setembro de 2015].
- MQTT. **MongoDB - Concepts.** 2017. <<https://github.com/mqtt/mqtt.github.io/wiki>>. [Online: acessado 10 de Dezembro de 2017].
- NODE.JS. **Node.js — As an asynchronous event driven JavaScript runtime.** 2017. <<https://nodejs.org/en/about/>>. [Online: acessado 13 de Outubro de 2017].
- PATTERN, F. **Factory.** 2017. <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>. [Online: acessado 01 de Janeiro de 2018].
- SANTOS LUCAS A. M. SILVA, C. S. F. S. C. J. B. B. N. B. S. P. M. A. M. V. L. F. M. V. O. N. G. e. A. A. F. L. B. P. Internet das coisas: da teoria à prática. **Paper**, 2016.