

Gabriel Marques de Melo

## **Central de controle em domótica via Internet**

Brasil

2018



Gabriel Marques de Melo

## **Central de controle em domótica via Internet**

Relatório técnico com o objetivo de documentação e apresentação do projeto desenvolvido na Universidade Federal de Lavras, no curso de Ciência da Computação.

Universidade Federal de Lavras  
Departamento de Engenharia  
Graduação em Ciência da Computação

Brasil  
2018

# Resumo

O aprimoramento das comunicações sem fio e o custo cada vez menor de dispositivos que se comunicam desta forma, impulsionaram o movimento da Internet das coisas bem como as técnicas de acesso remoto via Internet. Neste cenário, a domótica, que oferece praticidade, conforto e segurança aos lares, ganhou popularidade e foco de pesquisas e projetos. Residências automatizadas fornecem controle remoto de dispositivos, como lâmpadas, climatizadores e portas, além de informações de monitoramento do lar por sensores e câmeras. Nossa projeto, de finalidade aplicada, tem como objetivo geral discutir e criar um sistema de controle de automação residencial seguro, por acesso remoto via internet, com quantidade de nós atuadores escalável e de custo reduzido. A análise quantitativa da escalabilidade e desempenho dos nós fornece um parâmetro de comparação em relação às outras modelagens empregadas para o problema. Como protótipo, foi desenvolvido um servidor web HTTP, para acesso do usuário, utilizando um Raspberry Pi Model B, com imagem ubuntu 16.04.1, implementado na linguagem de programação PHP 7.0.3 e o framework web Laravel 5.5.40. Os nós foram projetados em uma placa de circuito impresso que tem como controlador um esp8266-01 e alguns atuadores e sensores genéricos. Além da escalabilidade dos nós, estes estabelecem uma conexão bidirecional por Web Socket com um servidor em Node.js integrado ao mesmo servidor web, comunicando com o sistema em tempo real.

**Palavras-chaves:** domótica. acesso remoto. servidor web.

# Listas de ilustrações

Figura 1 – Arquitetura do projeto . . . . .	50
Figura 2 – Nô <i>light</i> . . . . .	52
Figura 3 – Esquemático do circuito de um nô <i>light</i> . . . . .	53
Figura 4 – Pack de baterias . . . . .	54
Figura 5 – Regulador de tensão ams11173V3 . . . . .	54



# **Lista de tabelas**

Tabela 1 – Modos de boot . . . . .	20
Tabela 2 – Pinagem I2C . . . . .	21
Tabela 3 – Pinagem UART . . . . .	21
Tabela 4 – Pinagem PWM . . . . .	22
Tabela 5 – Modos de Consumo Energético . . . . .	24
Tabela 6 – Especificações ESP8266 vs ATMEGA-328p . . . . .	24
Tabela 7 – Funções básicas . . . . .	26
Tabela 8 – Funções de contagem de tempo . . . . .	27



# Listings

3.1	Instalando Apache e PHP 7.2 . . . . .	36
3.2	Instalando o Composer (gerenciador de pacotes PHP) . . . . .	36
3.3	Criando novo projeto Laravel . . . . .	36
3.4	Permissões de acesso no apache . . . . .	36
3.5	Arquivo de configuração do projeto . . . . .	36
3.6	<i>laravel.conf</i> . . . . .	36
3.7	Desabilitando virtualhost padrão e habilitando o novo . . . . .	36
3.8	Gerando chave de aplicação . . . . .	37
3.9	Criando uma model . . . . .	38
3.10	<i>app/Pessoa.php</i> . . . . .	39
3.11	Criando uma migration . . . . .	39
3.12	Arquivo de migration exemplo . . . . .	39
3.13	Executando uma migração . . . . .	40
3.14	<i>Arquivo de migration exemplo 2</i> . . . . .	40
3.15	<i>app/Pessoa.php</i> . . . . .	40
3.16	<i>app/Telefone.php</i> . . . . .	41
3.17	Ecoando uma variável . . . . .	41
3.18	Ecoando uma variável em PHP puro . . . . .	41
3.19	Correspondente ao @if em PHP puro . . . . .	41
3.20	Variação do @if . . . . .	42
3.21	Correspondente ao <i>isset()</i> em PHP puro . . . . .	42
3.22	Valor default usando @yield . . . . .	42
3.23	@yield sem valor default . . . . .	42
3.24	Controlando sobrescrita . . . . .	43
3.25	Passagem de parâmetro por @include . . . . .	43
3.26	Correspondente ao <i>isset()</i> em PHP puro . . . . .	43
3.27	<i>/routes/web.php</i> . . . . .	44
3.28	<i>/routes/web.php 2</i> . . . . .	44
3.29	<i>/routes/web.php 3</i> . . . . .	44
3.30	<i>/routes/web.php 4</i> . . . . .	45
3.31	<i>/routes/web.php 5</i> . . . . .	45
3.32	Helper <i>route()</i> . . . . .	45
3.33	Helper <i>url()</i> . . . . .	45
3.34	Grupo de middleware . . . . .	46
3.35	Prefixos de caminho . . . . .	46
3.36	Criando um controller . . . . .	47

3.37 Prefixos de caminho . . . . .	47
6.1 Executando comando make:module . . . . .	55

# Lista de abreviaturas e siglas

IoT	<i>Internet of Things</i>
VCS	<i>Version control system</i>
OS	<i>Operational system</i>
RPI	<i>Raspberry pi</i>
HTTP	<i>Hypertext transfer protocol</i>
PC	<i>Personal computer</i>
JSON	<i>Javascript object notation</i>
IR	<i>Infrared</i>



# Sumário

<b>Listings</b> . . . . .	<b>7</b>
<b>Introdução</b> . . . . .	<b>15</b>
<b>I PROJETO</b> . . . . .	<b>16</b>
<b>0.1 Objetivo</b> . . . . .	<b>17</b>
<b>II REFERENCIAL TEÓRICO</b> . . . . .	<b>18</b>
<b>1 ESP8266</b> . . . . .	<b>19</b>
<b>1.1 Especificações</b> . . . . .	<b>19</b>
1.1.1 CPU . . . . .	19
1.1.2 Modem Wi-Fi . . . . .	19
1.1.3 Alimentação . . . . .	20
1.1.4 Modos de operação . . . . .	20
1.1.5 GPIOs . . . . .	20
1.1.6 I2C . . . . .	21
1.1.7 UART . . . . .	21
1.1.8 Watchdog . . . . .	21
1.1.9 PWM . . . . .	22
1.1.10 Interrupção externa . . . . .	22
1.1.11 Consumo de Energia . . . . .	23
1.1.12 Comparação com Arduino . . . . .	23
<b>1.2 Firmware</b> . . . . .	<b>23</b>
1.2.1 Conversores USB para UART . . . . .	25
1.2.2 IDE Arduino . . . . .	25
1.2.3 Monitor Serial . . . . .	25
1.2.4 A Linguagem Arduino . . . . .	26
1.2.4.1 Funções Estruturais . . . . .	26
1.2.4.2 Timer: <i>delay()</i> VS <i>millis()</i> . . . . .	27
1.2.4.3 Função <i>yield()</i> . . . . .	27
1.2.5 SPIFFS . . . . .	28
1.2.6 PROGMEM . . . . .	29
<b>2 RASPBERRY PI 3</b> . . . . .	<b>31</b>

2.0.1	Uso no projeto . . . . .	31
<b>2.1</b>	<b>Linux</b> . . . . .	<b>31</b>
<b>2.2</b>	<b>Git</b> . . . . .	<b>32</b>
2.2.1	Uso no projeto . . . . .	32
2.2.2	Instalação . . . . .	32
2.2.2.1	Linux . . . . .	32
2.2.2.2	Windows . . . . .	32
2.2.3	Principais comandos básicos . . . . .	33
<b>2.3</b>	<b>Github</b> . . . . .	<b>33</b>
2.3.1	Integração de repositórios locais e remotos . . . . .	33
2.3.2	Principais comandos básicos . . . . .	34
<b>3</b>	<b>LARAVEL</b> . . . . .	<b>35</b>
<b>3.1</b>	<b>Introdução</b> . . . . .	<b>35</b>
3.1.1	MVC (model-view-controller) . . . . .	35
3.1.1.1	Model . . . . .	35
3.1.1.2	View . . . . .	35
3.1.1.3	Controller . . . . .	35
<b>3.2</b>	<b>Preparação do ambiente</b> . . . . .	<b>36</b>
3.2.1	Linux . . . . .	36
<b>3.3</b>	<b>Configurações iniciais do projeto</b> . . . . .	<b>37</b>
3.3.1	Chave de aplicação . . . . .	37
3.3.2	Banco de dados . . . . .	37
<b>3.4</b>	<b>Estrutura do projeto</b> . . . . .	<b>37</b>
3.4.1	Diretórios . . . . .	37
3.4.2	Arquivos 'soltos' . . . . .	38
<b>3.5</b>	<b>Banco de Dados</b> . . . . .	<b>38</b>
3.5.1	Models . . . . .	38
3.5.2	Migrations . . . . .	39
3.5.3	Relacionamentos entre Models . . . . .	40
<b>3.6</b>	<b>Templates com Blade</b> . . . . .	<b>41</b>
3.6.1	Ecoando dados . . . . .	41
3.6.2	Estruturas de Controle . . . . .	41
3.6.2.1	Condicionais . . . . .	41
3.6.2.2	Repetições . . . . .	42
3.6.2.3	Herança . . . . .	42
<b>3.7</b>	<b>Roteamento e Controladores</b> . . . . .	<b>43</b>
3.7.1	Rotas . . . . .	43
3.7.1.1	Verbos de rota . . . . .	44
3.7.1.2	Parâmetros de rotas . . . . .	44

3.7.1.3	Restrições com regexp . . . . .	45
3.7.1.4	Nome de rotas . . . . .	45
3.7.1.5	Helper <i>route()</i> . . . . .	45
3.7.1.6	Helper <i>url()</i> . . . . .	45
3.7.1.7	Grupos de rotas . . . . .	45
3.7.1.7.1	Middleware . . . . .	46
3.7.1.7.2	Prefixos de caminho . . . . .	46
3.7.1.8	Chamando views com parâmetros . . . . .	46
3.7.2	Controladores . . . . .	46
<b>4</b>	<b>SOCKET.IO . . . . .</b>	<b>48</b>
<b>4.1</b>	<b>Uso no projeto . . . . .</b>	<b>48</b>
<b>4.2</b>	<b>Exemplo . . . . .</b>	<b>48</b>
<b>III</b>	<b>DESENVOLVIMENTO . . . . .</b>	<b>49</b>
<b>5</b>	<b>ARQUITETURA . . . . .</b>	<b>50</b>
<b>5.1</b>	<b>Camada 1 . . . . .</b>	<b>50</b>
<b>5.2</b>	<b>Camada 2 . . . . .</b>	<b>51</b>
<b>5.3</b>	<b>Camada 3 . . . . .</b>	<b>51</b>
<b>6</b>	<b>NÓS . . . . .</b>	<b>52</b>
<b>6.1</b>	<b>Nó <i>light</i> . . . . .</b>	<b>52</b>
6.1.1	Circuito . . . . .	53
6.1.2	Alimentação . . . . .	53
6.1.2.1	Regulação . . . . .	53
<b>6.2</b>	<b>Nó <i>Tv</i> . . . . .</b>	<b>53</b>
6.2.1	Circuito . . . . .	55
<b>6.3</b>	<b>Nó <i>node</i> . . . . .</b>	<b>55</b>
<b>6.4</b>	<b>Criação de nós pela plataforma . . . . .</b>	<b>55</b>
6.4.1	make:module . . . . .	55
<b>IV</b>	<b>TRABALHOS FUTUROS . . . . .</b>	<b>57</b>
<b>7</b>	<b>CIRCUITO . . . . .</b>	<b>58</b>
<b>7.1</b>	<b>ESP8266 . . . . .</b>	<b>58</b>
<b>7.2</b>	<b>PCB . . . . .</b>	<b>58</b>
<b>7.3</b>	<b>Alimentação . . . . .</b>	<b>58</b>
<b>7.4</b>	<b>Raspberry pi 3 . . . . .</b>	<b>58</b>

<b>7.5</b>	<b>Contratação de host</b>	<b>58</b>
<b>8</b>	<b>SOFTWARE</b>	<b>59</b>
<b>8.1</b>	<b>Nós</b>	<b>59</b>
<b>8.2</b>	<b>ESP8266</b>	<b>59</b>

# Introdução

Este documento surgiu com o propósito de documentar as atividades e progressos realizados durante o período de orientado voluntário, pelo PIVIC, sob orientação do professor Fabio Domingues de Jesus, do Departamento de Engenharia da UFLA.

Os conteúdos baseados para redação deste estão devidamente referenciados ao final da obra. Considerável parcela das obras consultadas são de materiais de minha autoria, disponíveis em meu [github](#).

A documentação assume um teor didático, de forma a introduzir novos orientados ao projeto e que estes, por sua vez, incrementarão o mesmo, construindo uma sólida referência do projeto.

Na primeira parte do documento, **Projeto**, serão abordados os objetivos e ideias iniciais do projeto.

O **referencial teórico** possui um capítulo no estilo *getting started* para cada uma das tecnologias utilizadas. Neste momento, são abordados pontos sobre preparação do ambiente, funcionalidades e exemplos.

Em **desenvolvimento**, será apresentado o projeto em si. A arquitetura do projeto, bem como os principais trechos de código serão exaltados.

E por fim, são sugeridos **trabalhos futuros** que servirão como incremento, otimização e correção de componentes atuais.

O documento pode servir como ponto de partida para novos projetos correlatos e sua distribuição deve ser estimulada.

Este documento foi redigido em L<sup>A</sup>T<sub>E</sub>Xusando o pacote *abnTeX2*.

Parte I

Projeto

## 0.1 Objetivo

O objetivo do projeto é o desenvolvimento de uma plataforma de controle em *IoT* que se posicionasse como uma alternativa de custo mais acessível do que as soluções disponíveis atualmente no mercado. A projeto deve fornecer acesso remoto e seguro à aplicação, sem impactar negativamente na experiência do usuário, uma vez que o sistema deve ser de fácil acesso e manipulação.

Outro ponto pertinente diz respeito a escalabilidade da solução, sendo possível controlar vários dispositivos pelo mesmo sistema.

A autonomia energética do circuito tem significativo peso no desenvolvimento e é quisto que haja uma recarga suficiente da fonte de alimentação (bateria), preferencialmente, por uma energia limpa, como a solar.

# Parte II

## Referencial teórico

# 1 Esp8266

É um microcontrolador de 32-bits com modem *Wi-Fi* integrado desenvolvido pela *Espressif Systems* surgido, em meados de 2014, para suprir a contínua demanda por uma plataforma que fosse de **baixo consumo** energético, **compacta** e de **desempenho confiável** na industria de *IoT*.

Assim como a maioria dos modelos Arduino, o ESP possui GPIOs (Pinos de entrada e saída de propósito geral) e suporte a PWM (Modulação por largura de pulso). O upload do firmware é feito também pela UART (RX/TX), porém o ESP8266 conta com upload OTA (over-the-air), que é a gravação através de uma rede.

Seguir pontos do curso

## 1.1 Especificações

### 1.1.1 CPU

O ESP8266 utiliza o processador Tensilica L106 32-bit, RISC e que possui velocidade de clock de 80MHz.



### Na Prática...

Facilmente, podemos alterar - *overclock* - da *CPU* para 160MHz! Para isso, basta adicionarmos o seguinte código no escopo geral do código:

```
extern "C" {
    bool system_update_cpu_req(int);
}
```

E na função *setup()*:

```
void setup() {
    ...
    system_update_cpu_freq(160);
    ...
}
```

### 1.1.2 Modem *Wi-Fi*

Com um completo e autônomo sistema de conexão *Wi-Fi* (suporte a *802.11b/g/n/e/i*), o *ESP8266* pode tanto estabelecer uma rede própria, em modo **AP**, ou um se conectar a

um *host*, em modo **STA**. Possui ainda o modo **AP\_STA**, que cria uma rede, mas também acessa uma *station*, simultaneamente.

Pode ser implementado como um "módulo WiFi" em conjunto com qualquer microcontrolador e comunicá-lo através de interfaces **SPI**, **I2C** e **UART** disponíveis na placa.

### 1.1.3 Alimentação

A faixa de tensão de operação do *ESP8266* é de **2.5V** à **3.6V**, sendo necessário o fornecimento de, ao menos, **500mA** de corrente.



#### CUIDADO!

O *ESP8266* **não** é tolerante à 5V. Alimentá-lo com essa tensão pode - *e vai* - danificar seu dispositivo.

### 1.1.4 Modos de operação

O *ESP8266* possui **três** modos de boot:

Tabela 1 – Modos de boot

<b>UART mode</b>	Modo deve ser selecionado para realizar o upload de um novo <i>firmware</i> no MCU. A <b>GPIO0</b> deve estar em <b>GND / GPIO2 em VCC</b> .
<b>Flash mode</b>	Neste modo, o boot é realizado da memória <i>flash</i> . O último firmware gravado será executado. A <b>GPIO0</b> deve estar em <b>VCC / GPIO2 em VCC</b>
<b>SDIO mode</b>	Quando habilitado, o MCU realiza o boot não da memória <i>flash</i> , mas sim de um cartão SD, usando protocolo <i>SPI</i> . A <b>GPIO15</b> deve estar em <b>VCC</b> .

### 1.1.5 GPIOs

O *ESP8266* possui 17 pinos GPIO que podem assumir varias funções através da devida programação de seus registradores. Cada um dos pinos pode ser configurado com **pull-up** ou **pull-down** interno e também alta impedância. Possui um pino **ADC**, TOUT ⊥ 9, de resolução de 1024 bits, porém sua alimentação deve ser limitada de 0 a 1V.

### 1.1.6 I<sub>2</sub>C

*I<sub>2</sub>C* (*Inter-integrated Circuit*) é um protocolo de comunicação *master/slave* entre dispositivos que se baseia em um barramento de apenas duas vias: **SDA** (Serial Data) por onde são transmitidos e recebidos os dados e **SCL** (Serial Clock) que dita a temporização do tráfego das informações. O grande diferencial do protocolo é que são permitidos, teoricamente, até 127 dispositivos distintos comunicando através de um mesmo barramento.

O *ESP8266* possui suporte a interface *I<sub>2</sub>C*, tanto como Master quanto Slave, para comunicação com outros microcontroladores e outros equipamentos periféricos, como sensores. A pinagem do barramento é vista pela Tabela 2 abaixo:

Tabela 2 – Pinagem I<sub>2</sub>C

Pino	Função	Descrição
<i>GPIO02</i> $\perp$ 14	SDA	Pino de dados
<i>MTMS</i> $\perp$ 9	SCL	Pino de Clock

### 1.1.7 UART

O *ESP8266* possui **duas** interfaces *UART* (Universal Asynchronous Receiver/-Transmitter) que podem alcançar **4.5 Mbps** de velocidade de transferência: a **UART0**, que pode ser usada para comunicação bidirecional; e a **UART1** que apenas envia dados, por seu pino *TX* (geralmente usado como *debugger*).

Tabela 3 – Pinagem UART

Pino	Função	Descrição
<i>GPIO1</i> $\perp$ 26	TX0	Transmissão Serial
<i>GPIO3</i> $\perp$ 25	RX0	Recepção Serial
<i>GPIO2</i> $\perp$ 23	TX1	Transmissão Serial

### 1.1.8 Watchdog

*Watchdog timer* é um circuito contador independente do clock principal e que serve como um recurso de segurança adicional.

Uma das maiores diferenças entre o *ESP8266* e um MCU utilizado em alguns modelos *Arduino*, é o fato de que o microcontrolador da *Espressif* executa varias tarefas em segundo plano: mantém a conexão do modem Wi-Fi, gerencia a pilha do protocolo TCP/IP, etc. Para manter a execução destas tarefas, ele possui dois watchdog timers: **SW\_WDT**, que é via software; e **HW\_WDT**, via Hardware. Caso uma função gere um loop longo (mais de 3 segundos) onde não haja alimentação do watchdog, o microcontrolador irá reiniciar, justamente para manter a execução das tarefas de segundo plano.

Consulte a subseção 1.2.4.3 para melhor entendimento de como o software controla o *timer* do *Watchdog*. Este tópico tratará da função *yield()*, que realiza a alimentação do **SW\_WDT**.

### 1.1.9 PWM

*PWM* (ou *Modulação por Largura de Pulso*) se refere a um tipo de sinal digital que permite a variação do tempo, de forma analógica, em que um sinal digital assume nível lógico alto. Desta forma, é possível modular a largura do pulso (duração do nível alto) e gerar sinais de tensões intermediárias.



#### Na Prática...

Para usarmos um pino como saída de PWM:

```
analogWrite(pino, valor);
```

Alterar a frequência do sinal:

```
analogWriteFreq(novaFrequencia); //100 Hz and 1 kHz
```

Alterar o *range* do sinal:

```
analogWriteRange(novoRange); // 1024 por padrão -> 2^10
```



#### Você sabia?

Uma clássica experiência do modelo pode ser realizada ao chavear, rapidamente, o interruptor da iluminação do seu quarto, por exemplo. A sensação será a de uma iluminação intermediária entre lâmpada desligada e normalmente acesa.

O ESP8266 possui 4 pinos de interface *PWM*, mas podem ser estendidos pelo usuário. A definição padrão dos pinos é dada pela Tabela 4 abaixo.

Tabela 4 – Pinagem PWM

Pino	Função
<i>MTDI</i> ± 10	PWM0
<i>MTDO</i> ± 13	PWM1
<i>MTMS</i> ± 9	PWM2
<i>GPIO4</i> ± 16	PWM4

### 1.1.10 Interrupção externa

Interrupções são eventos ou condições que levam o microcontrolador a pausar a execução de uma tarefa em andamento, executar outra temporariamente e, então, retornar

para a tarefa inicial.

Com exceção do pino *GPIO16*  $\perp$  8, todas as demais GPIOs do ESP8266 possuem funcionalidade de interrupção externa.

### Na Prática...

Podemos utilizar uma interrupção que será deflagrada ao estado do sinal do pino apresentar uma curva de subida - *rising* - e chamará a função *resolveInterrupcao()* da seguinte forma:

```
void setup() {  
    ...  
    pinMode(pinoDeInterrupcao, INPUT);  
    attachInterrupt(pinoDeInterrupcao, resolveInterrupcao, "  
    RISING");  
    // Além de "RISING", pode assumir "CHANGE" ou "FALLING"  
    ...  
}  
  
void resolveInterrupcao() {  
    ...  
}
```

#### 1.1.11 Consumo de Energia

O ESP8266 já possui modos de consumo definidos, que gerenciam o funcionamento de algumas funcionalidades visando economia energética de acordo com cada tipo de aplicação.

A Tabela ?? apresenta estes modos, suas funções ativas e consumo médio.

#### 1.1.12 Comparação com Arduino

Uma comparação direta pode ser assumida com seu famoso - *querido* - concorrente: o Arduino. A Tabela ?? ilustra uma comparação de hardware e desempenho entre o ESP8266 e o Arduino Uno, a plataforma mais utilizada do Arduino.

## 1.2 Firmware

O firmware padrão traz um conjunto de instruções AT, porém é possível realizar o upload de outros firmwares para programação em linguagens mais comuns como o nodeMCU

Tabela 5 – Modos de Consumo Energético

<b>modem-sleep</b>	Modo usado em aplicações que requerem a CPU funcionando, como em aplicações com PWM e I2S. "Desliga" o circuito do Modem Wi-Fi enquanto mantiver uma conexão Wi-Fi sem transmissão de dados. <i>Consumo médio: 15mA.</i>
<b>light-sleep</b>	Durante o modo, a CPU pode ser suspensa em aplicações como uma interruptor Wi-Fi. Sem haver transmissão de dados, a o circuito do Modem Wi-Fi pode ser desligado e a CPU suspensa para economia de consumo de energia. <i>Consumo médio: 0.9mA.</i>
<b>deep-light</b>	Durante o modo, o modem Wi-Fi é totalmente desligado. Para aplicações onde existam longos intervalos de tempo sem transmissão de dados. Por exemplo, o monitoramento da temperatura ambiente, lendo dados durante um período, dormindo por outro período e acordando para reconectar a um ponto de acesso. <i>Consumo médio: 20µA.</i>

	ESP-12	ATMEGA-328p
Arquitetura	32-bits	8-bits
Frequência	80 ~160 MHz	16 ~20 MHz
Tensão de operação	2.5 ~3.6V	1.8 ~5.5V
Memória Flash	1Mb ~4Mb	32Kb
Memória RAM	15 ~26Kb*	2Kb
GPIOs	18 (17/d e 1/a)	20 (14/d e 6/a)
Preço	\$8,79	\$9,99

Tabela 6 – Especificações ESP8266 vs ATMEGA-328p

(Lua) e o [micropython](#)<sup>1</sup> (Python). Ambos os firmwares baseam em um sistema interno de arquivos, com um arquivo "main" executado no boot e também possuem sua própria biblioteca que é atualizada por suas comunidades. Entretanto, a Arduino IDE possui, atualmente, suporte ao ESP8266, tornando possível a programação em Arduino (C++),

<sup>1</sup> Possuo um tutorial detalhado para preparação do ambiente micropython. Acesse-o [aqui](#).

utilizando, inclusive, algumas bibliotecas do mesmo sem realizar quaisquer alterações.

### 1.2.1 Conversores *USB para UART*

O *ESP8266* não possui, nativamente, interface para comunicação *USB*, porém existem conversores que facilitam a comunicação entre o microcontrolador e um PC para realização do upload do programa, por exemplo.

É possível ainda utilizar de dispositivos que já possuam um conversor interno para realizar o upload, como o *Arduino UNO*. Interligando os pinos de *UART* dos dois dispositivos (*TX(ESP) → RX(Arduino)* e *RX(ESP) → TX(Arduino)*) é possível realizar a comunicação do *ESP8266* com um PC através do conversor *USB* do *Arduino*.



#### ATENÇÃO!

A faixa de tensão de operação do *ESP8266* é de 2.5 a 3.6V, sendo necessário, então, um divisor de tensão do *TX* (*Arduino*), que possui 5V, para o *RX* (*ESP8266*).

### 1.2.2 *IDE Arduino*

Implementada em *Java*, a *IDE Arduino* possui uma interface **simples** e **objetiva**.

Apesar de não apresentar algumas funções e macros presentes em outros editores de texto, como *auto-complete* e snippets, a *IDE* possui recursos interessantes ao usuário, como o **Monitor Serial**.

### 1.2.3 Monitor Serial

É uma interface de interação com o usuário e depuração do código compilado fornecida pela *IDE Arduino*, onde é possível visualizar dados enviados, via comunicação serial, do MCU para um PC. É possível usá-lo, também, para envio de dados em tempo real do computador para o microcontrolador. Todos os projetos exemplificados neste livro utilizam o **Monitor Serial** para identificarmos as etapas de execução do programa no *ESP8266*.



### Na Prática...

Antes de utilizarmos o **Monitor Serial**, devemos nos certificar de ter inicializado a comunicação serial com seu *baudrate* (taxa de transferência de dados), via software:

```
void setup() {
    ...
    Serial.begin(115200);
    ...
}
```

Para escrevermos alguma mensagem no monitor, usamos a função *print()*:

```
Serial.print("MENSAGEM PARA MONITOR");
```

## 1.2.4 A Linguagem *Arduino*

Baseada em **C++**, a linguagem de domínio específico *Arduino* traz facilidades ao programador de microcontroladores com funções de leitura, escrita e depuração de pinos de I/O implementadas. Além das funções estruturais *loop()* e *setup()*, que guiam o desenvolvimento do software.

A DSL Arduino possui suporte à orientação a objetos, alocação dinâmica e manipulação de ponteiros, bem como os tipos de dados aceitos por sua linguagem "mãe".

### 1.2.4.1 Funções Estruturais

Um código em Arduino possui duas funções essenciais: a *loop()* e a *setup()*:

Tabela 7 – Funções básicas

<b>void setup()</b>	Função executada apenas uma vez após o MCU dar o boot ou reset. Nela são inicializadas variáveis e indicadas classes, setados modos de entrada ou saída dos pinos, etc.
<b>void loop()</b>	Função chamada após executar a void <i>setup()</i> , que rege a execução do código, repetindo infinitas vezes seu conteúdo. Ao final de um ciclo de execução da função, é chamada a função <i>yield()</i> para dar feed ao <i>SW WDT</i> .



**Na Prática...**

```

void setup() {
    ...
}

void loop() {
    ...
}

```

#### 1.2.4.2 Timer: *delay()* VS *millis()*

A linguagem possui duas funções principais para contagem de tempo durante execução do programa:

Tabela 8 – Funções de contagem de tempo

<b>delay(<i>tempo</i>)</b>	Função que recebe em <i>tempo</i> um valor, em milisegundos, em que o MCU irá pausar a maioria das tasks e retornar a execução normal após este período. A função, porém não desabilita interrupções e mantém os valores da porta serial (recebidos em <i>RX</i> ), PWM e estados dos pinos. <i>A função delay() chama, internamente, a função yield().</i>
<b>millis()</b>	Função retorna o tempo, em milisegundos, desde o início do programa atual. O valor retornado tem tipo <i>unsigned long</i> e sofrerá overflow em cerca de 50 dias de execução contínua do MCU. <i>A função millis() não chama, internamente, a função yield().</i>

#### 1.2.4.3 Função *yield()*

Uma das situações mais comuns aos novos usuários do ESP8266, é o reset inesperado - e misterioso - do MCU. Como visto anteriormente, o 8266 possui 2 circuitos de *Watchdog* sendo um via software (SW\_WDT) e outro via Hardware (HW\_WDT). O SW\_WDT é alimentado justamente pela função *yield()*, que faz a contagem de seu timer (cerca de 3 segundos) reiniciar.



### Na Prática...

No trecho de código abaixo podemos analisar o uso do `yield()` para evitar o reset do microcontrolador:

```
void loop() {
    ...
    piscaLed(led,10000); // recebe o tempo em milissegundos
    ...
}

int piscaLed(int led,int tempo) {
    int tempoInicial = millis();
    int contador = tempoInicial;
    while (contador - tempoInicial < tempo) {
        digitalWrite(led, HIGH);
        delayMicroseconds(500000);
        digitalWrite(led, LOW);
        delayMicroseconds(500000);
        contador = millis();
        yield();
    }
}
```

Como a função `piscaLed()` possui um loop interno que dura, no caso, 10 segundos, caso comentássemos a chamada da `yield()`, o *SW\_WDT* não seria alimentado e, em cerca de 3 segundos, o microcontrolador se reiniciaria. Podemos, analogamente, utilizar as seguintes funções que chamam internamente a `yield()` para evitar o *reset*:

```
delay();
```

ou

```
ESP.wdtFeed(); // Esta, porém, alimenta também o HW_WDT
```



### ATENÇÃO!

As funções `delayMicroseconds()` e `millis()` **não** chamam, internamente, a função `yield()`.

## 1.2.5 SPIFFS

Um dos problemas enfrentados pelos programadores de microcontroladores é a limitação física da memória, principalmente, na memória flash. Pensando nisso, o *ESP8266* traz, nativamente, um sistema interno de arquivos baseado em um sistema *SPI NOR Flash*. Não suporta criação e manipulação de diretórios, porém é possível nomear um arquivo

utilizando "/", tornando possível, ao menos, fantasiar um diretório.



### Na Prática...

Inicia o SPIFFS

```
SPIFFS.begin();
```

Abre um arquivo

```
SPIFFS.open(path, mode); // mode recebe "r", "w", "a", "r+", "w+"  
ou "a+".
```

Remove um arquivo

```
SPIFFS.remove(path)
```

Fecha um arquivo

```
file.close()
```

## 1.2.6 PROGMEM

Ainda pesando na limitação de hardware da memória, o ESP importa da biblioteca do Arduino uma função que possibilita a alocação de strings e variáveis na memória flash (que é bem maior) ao invés da RAM: a **PROGMEM**.



## Na Prática...

Existem duas formas de declarar uma variável para ser armazenada em **flash**:

```
const tipoDado nomeVariavel [] PROGMEM = {};
```

ou

```
const PROGMEM tipoDado nomeVariavel [] = {};
```

Em ambas, após a alocação na memória, para acessar o seu conteúdo são necessários funções especiais de leitura. Por exemplo, para recuperar o valor da variável *mensagem* declarada como:

```
const char mensagem [] PROGMEM = {"ESTE LIVRO EH UMA INTRODUCAO  
AO IOT"};
```

Devemos fazê-la da seguinte forma:

```
char mensagemAux;  
for (k = 0; k < strlen_P(mensagem); k++)  
{  
    mensagemAux = pgm_read_byte_near(mensagem + k);  
    Serial.print(mensagemAux);  
}
```

Ao utilizar o **Monitor Serial** como debugger ou uma interface de numerosas mensagens, estamos propícios a, facilmente, consumir muita memória RAM durante as chamadas funções de impressão como:

```
Serial.print("ESTE LIVRO EH UMA INTRODUCAO AO IOT");
```

Nestas chamadas, todas os dados são armazenados, por padrão, na RAM do dispositivo. Para contornar isso, podemos modificá-las adicionando a macro *F()*, que altera este armazenamento para a flash:

```
Serial.print(F("ESTE LIVRO EH UMA INTRODUCAO AO IOT"));
```



### ATENÇÃO!

Todas as variáveis devem ser definidas globalmente ou definidas como estáticas, **caso contrário** o **PROGMEM** não funcionará.

## 2 Raspberry pi 3

O Raspberry pi é um microcomputador que fornece funcionalidades completas de um PC através da conexão de periféricos de entrada/saída, tais como monitores, teclados, mouses, microfones, etc.

Os modelos Raspberry pi causaram grande impacto, principalmente na comunidade maker de sistemas embarcados, devido ao seu baixo consumo energético, quando comparado a um PC de mesa ou notebook, e também por sua portabilidade, devido às suas dimensões reduzidas. A aplicabilidade de microcomputadores em projetos IoT é praticamente infinita e representa uma evolução no sentido de processamento, comparado aos microcontroladores utilizados até então.

A comunidade raspberry cresceu abruptamente desde seu lançamento e existem os mais diversos tutoriais disponibilizados, gratuitamente, na Internet.

### 2.0.1 Uso no projeto

O modelo Raspberry pi 3 foi utilizado no projeto devido ao seu processamento superior aos demais microcomputadores disponíveis pela mesma faixa de preço, até a data de aquisição<sup>1</sup>. O dispositivo será utilizado como um **Web server**, recebendo e tratando requisições de um (ou mais) clientes e se comunicando de forma assíncrona com alguns ESP8266, via Internet. Como sistema operacional, optamos pela utilização do [Raspbian](#), uma distribuição Linux disponibilizada pela Raspberry Pi e otimizada para processadores ARM.

Como veremos na parte de **Trabalhos futuros**, é sugerida a alteração do microcomputador pela contratação de uma máquina distribuída.

## 2.1 Linux

**Linux**, ou GNU/Linux, é o termo utilizado para a referência ao sistema operacional (OS) que utiliza o Kernel Linux. O projeto Linux possui código aberto é sua utilização, bem como o estudo, modificação e distribuição é livre sob a licença GPL (versão 2). O código-fonte do kernel linux pode ser acessado neste [repositório](#).

As distribuições Linux são, em sua maioria, focadas no uso do terminal de comando e na liberdade do usuário em personalizar (com acesso mais baixo nível) as características e comportamento de sua máquina.

---

<sup>1</sup> Julho de 2017

Pelo tamanho em disco da imagem do SO e, também, pela quantidade de memória principal (RAM) necessária para, respectivamente, armazenamento e execução de um sistema Linux serem consideravelmente inferiores aos sistemas Windows, o primeiro é amplamente utilizado em sistemas embarcados.

## 2.2 Git

O git é o sistema gerenciador de versão (VCS) de software mais utilizado no mundo, tanto por pessoas físicas (estudantes, profissionais autônomos e entusiastas), quanto pessoas jurídicas.

Um VCS é indispensável na produção de software pela segurança e organização provida. Conforme o código-fonte vai aumentando, novas funcionalidades vão sendo inseridas e, desta forma, a integridade do projeto é um desafio maior. O Git armazena cópias do estado do projeto (commits) em determinado tempo do desenvolvimento, possibilitando o rollback (retrocesso) do software para um estado quisto e seguro.

### 2.2.1 Uso no projeto

O projeto foi construído, desde o início, utilizando o git como controlador de versão. Todos os códigos gerados podem ser visualizados neste [repositório](#).

### 2.2.2 Instalação

#### 2.2.2.1 Linux

As distribuições Linux atuais já contam com o git instalado, portanto não é necessária nenhuma instalação adicional.

Caso você precise instalar o git em uma máquina que não o possua, o faça pelo repositório padrão (em distribuições derivadas do Debian):

```
1 apt-get install git  
2
```

#### 2.2.2.2 Windows

A estratégia mais adotada para utilização do git em um ambiente Windows, é a instalação do [Git bash](#).

### 2.2.3 Principais comandos básicos

Criação de um repositório.

```
1 git init  
2
```

Listagem do estado do repositório.

```
1 git status  
2
```

Adição do arquivo *file1.txt* ao repositório.

```
1 git add file1 .txt  
2
```

Commit dos arquivos adicionados no repositório.

```
1 git commit  
2
```

Listagem dos commits.

```
1 git log  
2
```

## 2.3 Github

O Github é uma plataforma online que disponibiliza repositórios remotos baseados em Git para exposição de projetos e desenvolvimento de software em equipe.

### 2.3.1 Integração de repositórios locais e remotos

Após criar o repositório remoto no github, faça a linkagem do mesmo em seu repositório local com o comando:

```
1 git remote add origin https://github.com/SEUUSURIO/NOMEDOREPOSITORIO.git  
2
```

### 2.3.2 Principais comandos básicos

Envio do repositório local para repositório remoto.

```
1 git push origin master  
2
```

Envio do repositório remoto para repositório local.

```
1 git pull origin master  
2
```

# 3 Laravel

## 3.1 Introdução

O Laravel é um framework web PHP, baseado na arquitetura MVC, que busca agilizar e facilitar a criação de aplicações, evitando a codificação repetitiva e prezando por boas práticas e seguindo padrões de design. Seus diferenciais estão na intuitividade da sua estrutura de arquivos e diretórios, na sua rica documentação e, principalmente, na sua grande e participativa comunidade – não me deparei com nenhuma dúvida em que já não existisse um tópico respondido nos Laracasts 1 ou mesmo no professor Stack overflow. A ideia deste documento é introduzir, superficialmente, os aspectos e funcionamentos do framework, bem como conduzir os leitores à demais documentos e referências.

### 3.1.1 MVC (model-view-controller)

O MVC é um padrão de arquitetura de software que tem como objetivo isolar a representação da informação da interação do usuário.

#### 3.1.1.1 Model

As models são classes que modelam uma entidade e a torna acessível para a aplicação de uma forma dinâmica. Podemos dizer que são usadas como uma interface entre o banco de dados e a aplicação.

#### 3.1.1.2 View

As views são representações - visuais - dos dados de uma aplicação, voltados para um ou mais usuários.

#### 3.1.1.3 Controller

Geralmente, os componentes mais complexos, em termos de codificação, em uma aplicação de arquitetura MVC são os controladores<sup>1</sup>. Estes componentes são classes que organizam a lógica de uma ou mais rotas. As rotas encaminham a aplicação para o controlador, que, por sua vez, computa as operações necessárias à lógica da aplicação, podendo acessar as models para obter dados persistentes, e direciona uma view, passando os parâmetros necessários para sua renderização.

---

<sup>1</sup> Na verdade, uma boa prática de desenvolvimento é utilizar os controladores como responsáveis apenas das requisições HTTP, uma vez que existem outras maneiras de uma aplicação receber solicitações, como cren jobs, queue jobs, etc.

## 3.2 Preparação do ambiente

### 3.2.1 Linux

```

1 sudo add-apt-repository ppa:ondrej/php
2 sudo apt-get update
3 sudo apt-get install apache2 libapache2-mod-php7.2 php7.2 \
4                               php7.2-xml php7.2-gd php7.2-opcache php7.2-mbstring

```

Listing 3.1 – Instalando Apache e PHP 7.2

```

1 cd /tmp
2 curl -sS https://getcomposer.org/installer | php
3 sudo mv composer.phar /usr/local/bin/composer

```

Listing 3.2 – Instalando o Composer (gerenciador de pacotes PHP)

```
1 composer create-project --prefer-dist laravel/laravel Nome_Projeto
```

Listing 3.3 – Criando novo projeto Laravel

```

1 sudo chgrp -R www-data /var/www/html/Nome-projeto
2 sudo chmod -R 775 /var/www/html/Nome-projeto/storage

```

Listing 3.4 – Permissões de acesso no apache

```

1 cd /etc/apache2/sites-available
2 sudo nano laravel.conf

```

Listing 3.5 – Arquivo de configuração do projeto

```

1 <VirtualHost *:80>
2     ServerName seuDominio.app
3     ServerAdmin webmaster@localhost
4     DocumentRoot /var/www/html/Nome-projeto/public
5
6     <Directory /var/www/html/Nome-projeto>
7         AllowOverride All
8     </Directory>
9
10    ErrorLog ${APACHE_LOG_DIR}/error.log
11    CustomLog ${APACHE_LOG_DIR}/access.log combined
12 </VirtualHost>

```

Listing 3.6 – *laravel.conf*

```

1 sudo a2dissite 000-default.conf
2 sudo a2ensite laravel.conf
3 sudo service apache2 reload

```

Listing 3.7 – Desabilitando virtualhost padrão e habilitando o novo

## 3.3 Configurações iniciais do projeto

### 3.3.1 Chave de aplicação

```
1 | php artisan key:generate
```

Listing 3.8 – Gerando chave de aplicação

Se a chave **não é declarada**, suas sessões de usuário e outros dados criptografados **não estarão seguros!**

### 3.3.2 Banco de dados

As configurações do banco de dados são feitas no arquivo config/database.php . O arquivo faz referências a variáveis de ambiente da aplicação, definidas no arquivo oculto /.env.

## 3.4 Estrutura do projeto

### 3.4.1 Diretórios

Todos projetos Laravel possuem a mesma estrutura de diretórios padrão:

/app → Maior parte do aplicativo. Contém os models, controllers, definições de rota, comandos e código de domínio PHP;

/bootstrap → Contém os arquivos usados na inicialização do Laravel;

/config → Contém os arquivos de configuração da aplicação;

/database → Contém as migrations (operações nas models) e seeds;

/public → É o diretório que o servidor aponta. Contém o index.php, que é o controlador frontal que inicia o processo de bootstrapping e roteia as solicitações adequadamente. Contém arquivos voltados ao cliente, como imagens, css, scripts ou downloads;

/resources → Contém arquivos não PHP necessários para a realização de outros scripts. Views e arquivos opcionais, como Sass/Less e javascript;

/routes → Contém as definições de rotas. Tanto para rotas HTTP, quanto para "rotas de console", por exemplo, o comando artisan;

/storage → Contém cache, logs e arquivos compilados;

/tests → Contém testes de unidade e integração;

/vendor → É onde o composer instala suas dependências. É, por padrão, incluída no .gitignore.

### 3.4.2 Arquivos 'soltos'

.env → Possui as variáveis de ambiente (informações da aplicação, banco de dados, broadcast, session, queue, smtp). É, por padrão, incluído no .gitignore. Suas variáveis são acessadas internamente na aplicação pelo helper env();

.env.example → Arquivo padrão que serve como base para montagem do seu .env;

package.json → Arquivo que lista os pacotes javascript a serem instalados na aplicação ao executar npm install na raiz do projeto.

## 3.5 Banco de Dados

O Laravel possibilita o usuário a criar, consultar e manipular banco de dados de forma ágil e fácil.

### 3.5.1 Models

As models são classes que modelam uma entidade e a torna acessível para a aplicação de uma forma dinâmica. Podemos dizer que são usadas como uma interface entre o banco de dados e a aplicação.

O seguinte comando, executado na raiz do projeto, criará uma model de nome nome\_model em App/nome\_model.php.

```
1 php artisan make:model nome_model
```

Listing 3.9 – Criando uma model

Acesse-o e realize as modificações conforme os campos de sua entidade. Os atributos definidos no campo **\$fillable**, serão de atribuição em massa, ou seja, não existirá proteção no caso de um usuário passar um parâmetro inesperado por uma requisição HTTP que alterará uma coluna no banco de dados, por exemplo. Para contornar isso, existe o campo **\$guarded** que garante que seus atributos sejam protegidos quanto a atribuição em massa. Por default, todos modelos Eloquent são protegidos.

O campo **\$table** definirá o nome da model. Por default, assume o plural (em inglês) do nome do nome da classe.

Exemplo abaixo:

```

1 class Pessoa extends Model
2 {
3     protected $fillable = [
4         'nome',
5         'cpf'
6     ];
7     protected $table = 'Pessoas';
8 }

```

Listing 3.10 – *app/Pessoa.php*

Deve ser usado **\$fillable** ou **\$guarded** - não os dois. Todos não assinalados como **\$fillable** serão **\$guarded** e vice-versa.

### 3.5.2 Migrations

Uma migration é responsável por modelar, criar e deletar um banco de dados referente a uma model.

O exemplo abaixo criará um arquivo default de migration em */database/migrations*.

```
1 php artisan make:migration nome_migration --create nome_model
```

Listing 3.11 – Criando uma migration

```

1 public function up()
2 {
3     Schema::create('_log', function (Blueprint $table) {
4         $table->increments('id');
5         $table->string('MAC');
6         $table->string('status');
7         $table->string('error');
8         $table->string('time');
9         $table->timestamps();
10    });
11 }

```

Listing 3.12 – Arquivo de migration exemplo

Os campos **timestamps()** e **increments()** são padrões em Laravel.

Os campos comuns de Blueprint estão listados no apêndice I.

O construtor de esquemas do Laravel permite apenas comandos do padrão **ANSI SQL**. Para executar demais comandos, use *DB::statement()*.

O comando abaixo executa as migrações pendentes, criando, editando ou dropando as tabelas no banco de dados configurado em *.end/*.

```
1  php artisan migrate
```

Listing 3.13 – Executando uma migração

Demais opções de comandos de migração estão no apêndice II.

Não delete uma migration sem antes executar `php artisan migrate:reset`. Caso contrário, ocorrerá inconsistência em seu banco de dados!

### 3.5.3 Relacionamentos entre Models

Podemos facilmente relacionar duas models através de uma **FOREIGN KEY** em Laravel.

Suponha uma nova model **Telefone** que terá um relacionamento muitos-para-um com **Pessoa**. Uma pessoa pode conter vários telefones, mas cada telefone pertence a somente uma pessoa.

```
1  public function up() {
2      Schema::create('Telefones', function (Blueprint $table) {
3          $table->increments('id');
4          $table->string('DDD');
5          $table->string('numero');
6          $table->integer('id_pessoa')->unsigned();
7          $table->foreign('id_pessoa')->references('id')->on('Pessoas')->onDelete('cascade');
8          $table->timestamps();
9      });
10 }
```

Listing 3.14 – Arquivo de migration exemplo 2

Tenha certeza de que os tipos da foreign key e a referência **são iguais**. Certifique, também que a referência é uma *primary key*.

Criada a model e sua migration, podemos editar a classe **Pessoa** para acessarmos os telefones referentes a cada instância:

```
1  public function telefone()
2  {
3      return $this->hasMany(Telefone::class, 'id_pessoa');
4  }
```

Listing 3.15 – app/Pessoa.php

O mesmo pode ser feito na classe **Telefone** para acessar os dados do dono do telefone:

```

1  public function pessoa()
2  {
3      return $this->belongsTo(Pessoa::class, 'id_pessoa');
4  }

```

Listing 3.16 – app/Telefone.php

## 3.6 Templates com Blade

Inspirado na **Razor** da plataforma .NET, o **Blade** é a elegante -e limpa- engine de templates do Laravel. O componente fornece rapidez em atividades corriqueiras e também acessibilidade e facilidade em requisitos complexos, como herança aninhada e recursão.

### 3.6.1 Ecoando dados

As chaves duplas `{{ e }}` são usadas para delimitar as seções de PHP que serão ecoadas.

```
{{ $variavel }}
```

Listing 3.17 – Ecoando uma variável

É o equivalente a:

```
<?php htmlentities($variavel) ?>
```

Listing 3.18 – Ecoando uma variável em PHP puro

### 3.6.2 Estruturas de Controle

As tags em Blade são prefixadas com a diretiva `@`. Suas estruturas de controle têm aparência mais clean quando comparadas com o PHP puro.

#### 3.6.2.1 Condicionais

`@if, @endif`

Transcrito para:

```
<?php if($condicao): ?>
```

Listing 3.19 – Correspondente ao @if em PHP puro

O mesmo serve para `@else` e `@elseif`.

`@unless, @endunless`

É o mesmo que:

```
@if (!$variavel)
```

Listing 3.20 – Variação do @if

**or**

Helper que substitui o *isset()*. Confere se a variável foi declarada **ou** retorna um valor padrão definido. Como, por exemplo:

```
{{ $variavel or "default" }}
```

Listing 3.21 – Correspondente ao *isset()* em PHP puro

### 3.6.2.2 Repetições

**@for, @endfor**

**For** tradicional.

**@foreach, @endforeach**

**For** iterando sobre itens/objetos.

**@forelse, @endforelse**

**For** com um *fallback* adicional quando o objeto de iteração se "esvaziar".

### 3.6.2.3 Herança

O Laravel possui diretivas que definem seções que podem ser estendidas e reutilizadas por templates-filhos:

**@yield**

Define uma seção com nome específico. Pode ter um valor default, caso a seção não seja estendida. O primeiro parâmetro de uma seção é sempre seu nome. Veja os exemplos:

```
<title> @yield('titulo', 'Meu titulo') </title>
```

Listing 3.22 – Valor default usando @yield

Neste caso, a tag `<title>` conterá '*Meu titulo*' caso o template-pai não seja estendido, ou conterá o valor atribuído à seção '*titulo*' no template que o estender.

Observe outra situação:

```
<p> @yield('conteudo') </p>
```

Listing 3.23 – @yield sem valor default

Já neste caso, a tag <p> só conterá valores caso seja estendida.

Podemos controlar se uma seção possui uma sobrescrita (ou pegar seu valor) de um template-filho através de:

```
@if ($env->yieldContent('conteudo'))
```

Listing 3.24 – Controlando sobreescrita

**@section, @show** Seção do template-pai que pode definir um bloco inteiro como fallback padrão. Seu conteúdo pode ser sobreescrito por templates-filhos (como no caso da diretiva **@yield**), mas também pode ser disponível pela diretiva **@parent** nos filhos.

Nos filhos as seções são iniciadas também por **@section**, porém são finalizadas por **@endsection**.

**@include** Seção que permite abrir uma view a partir de outra. É possível passar dados de maneira explícita, através do segundo parâmetro da diretiva **@include**, mas também podemos referenciar variáveis do arquivo que incluiu essa view.

Veja os exemplos:

```
@include ('botao-login', ['texto' => 'Faa seu login'])
```

Listing 3.25 – Passagem de parâmetro por @include

Neste caso, a diretiva incluirá a view localizada em *resources/views/botao-login.blade.php*, passando a variável **texto** como parâmetro. Essa view poderia, de uma forma bem simples, ser desta forma:

```
<a class="button"> {{ $texto }} </a>
```

Listing 3.26 – Correspondente ao *isset()* em PHP puro

## 3.7 Roteamento e Controladores

As rotas apontam para determinado código quando receber determinada requisição do usuário. Os controladores, vêm como uma forma inteligente de ligar as **models** com as **views**, como vimos na primeira seção deste material, e nos ajudarão muito na construção de aplicações enxutas e inteligentes.

### 3.7.1 Rotas

Em Laravel, as rotas são definidas em */routes*. As rotas web, que têm retorno em html, são definidas em */routes/web.php*.

Um simples exemplo de rota:

```

1 Route::get('/', function() {
2     return view('layouts.welcome');
3 });

```

Listing 3.27 – /routes/web.php

Ao receber uma solicitação GET sem nenhum parâmetro, “/”, (e.g. acessar localhost/), supondo que seu localhost seja definido na pasta /public da aplicação), a rota chamará a view em *resources/views/layouts/welcome.blade.php*.

O mesmo funcionamento ocorre no exemplo abaixo, ao acessar sua aplicação com a URL / contato:

```

1 Route::get('/contato', function() {
2     return view('layouts.contato');
3 });

```

Listing 3.28 – /routes/web.php 2

A chamada de views diretamente das rotas é uma prática ruim. Veremos, mais adiante, que utilizar controllers para chamada das views é uma alternativa melhor e coerente com o padrão MVC.

### 3.7.1.1 Verbos de rota

#### **Route::get(), Route::post()**

Recebe e trata as requisições padrões do protocolo HTTP GET/POST

#### **Route::put(), Route::delete(), Route::any(), Route::match()**

Recebe e trata demais requisições HTTP, geralmente usadas em APIs REST.

### 3.7.1.2 Parâmetros de rotas

Rotas que tiverem segmento(s) da estrutura URL variável(is) podem ser acessadas por:

```

1 Route::get('/post/{id}', function($id) {
2     // alguma coisa aqui
3 });

```

Listing 3.29 – /routes/web.php 3

Desta forma, a closure receberá como parâmetro o próprio valor de id na variável \$id.

Note que os parâmetros da closure e os segmentos da URL não devem possuir a mesma nomenclatura. A relação entre eles é dada pela ordem em que são dispostos.

### 3.7.1.3 Restrições com regexp

Podemos restringir os parâmetros aceitos por uma rota utilizando expressões regulares (regexes), como no exemplo abaixo:

```
1 Route::get('/post/{id}', function($id) {
2     // alguma coisa aqui
3 })->where('id', '[0-9]+');
```

Listing 3.30 – /routes/web.php 4

### 3.7.1.4 Nome de rotas

Podemos definir nome às nossas rotas de forma a acessá-las internamente em nossa aplicação de forma mais fácil.

```
1 Route::get('/', function() {
2     // alguma coisa aqui
3 })->name('welcome');
```

Listing 3.31 – /routes/web.php 5

### 3.7.1.5 Helper *route()*

Podemos, então, utilizar o helper *route()* para referenciar uma rota utilizando seu nome.

```
1 <a href="{{ route('welcome') }}>
```

Listing 3.32 – Helper *route()*

### 3.7.1.6 Helper *url()*

De forma alternativa, podemos utilizar o helper *url()* para referenciar uma rota sem fazê-la explicitamente ou usando um nome.

```
1 <a href="{{ url('/') }}>
```

Listing 3.33 – Helper *url()*

### 3.7.1.7 Grupos de rotas

Podemos agrupar algumas rotas que compatilhem alguma característica, como mesmo prefixo, requisitos de autenticação e namespaces de controladores.

### 3.7.1.7.1 Middleware

Camada usada, dentre alguns objetivos, para autenticar usuários e impedir acesso de usuários não autorizados em determinadas partes da aplicação. Por exemplo:

```

1 Route::group([ middleware => auth ], function()
2 {
3     Route::get( member , function() {
4         Return view( layout . member );
5     });
6
7     Route::get( admin , function() {
8         Return view( layout . admin );
9     });
10 });

```

Listing 3.34 – Grupo de middleware

Nesta situação, o usuário só receberá a view esperada - member ou admin - caso seja autenticado no sistema.

### 3.7.1.7.2 Prefixos de caminho

Para simplificar a estruturação e legibilidade do código, podemos usar o agrupamento por prefixos de caminho, como exemplificados abaixo:

```

1 Route::group([ prefix => post ], function() {
2     Route::get( / , function() {
3         Return view( layout . posts );
4     });
5     Route::get( /{id} , function($id) {
6         Return view( layout . postId ) ->with( id , $id );
7     });
8 });

```

Listing 3.35 – Prefixos de caminho

### 3.7.1.8 Chamando views com parâmetros

Ainda utilizando o exemplo acima, podemos notar que utilizamos o método adicional *with()*. Esse método injeta variáveis às views.

## 3.7.2 Controladores

Os controladores residem, por padrão, em *app/Http/Controller*. Para criar um novo controller, utilizamos, mais uma vez, o Artisan:

```
1 |     php artisan make:Controller nome_controller
```

Listing 3.36 – Criando um controller

Os controladores são chamados nas rotas da seguinte forma:

```
1 | Route::get(    / home , 'HomeController@index')->name('home');
```

Listing 3.37 – Prefixos de caminho

Este exemplo direcionará, pelo namespace padrão, a aplicação quando receber a rota para execução do método **index()** no arquivo *app/Http/Controller/HomeController.php*.

# 4 Socket.io

**Socket.io** é uma aplicação que roda em Node.js e possibilita uma solução simples para comunicação assíncrona de servidores para clientes. Desta forma, é possível que um servidor realize uma ação para o usuário quando estourar um evento server-side.

## 4.1 Uso no projeto

A utilização do Socket.io se deu da necessidade de que o web server (Raspberry pi 3) atualizasse, em tempo real, ao usuário o estado (ligado/desligado) dos periféricos controlados. Isso ficará mais claro na parte **Desenvolvimento**.

## 4.2 Exemplo

O próprio [website](#) do projeto disponibiliza um passo-a-passo para iniciantes. Existe também um [repositório](#) que contém um projeto de chat básico que ilustra bem a utilização do Socket.io.

# Parte III

## Desenvolvimento

# 5 Arquitetura

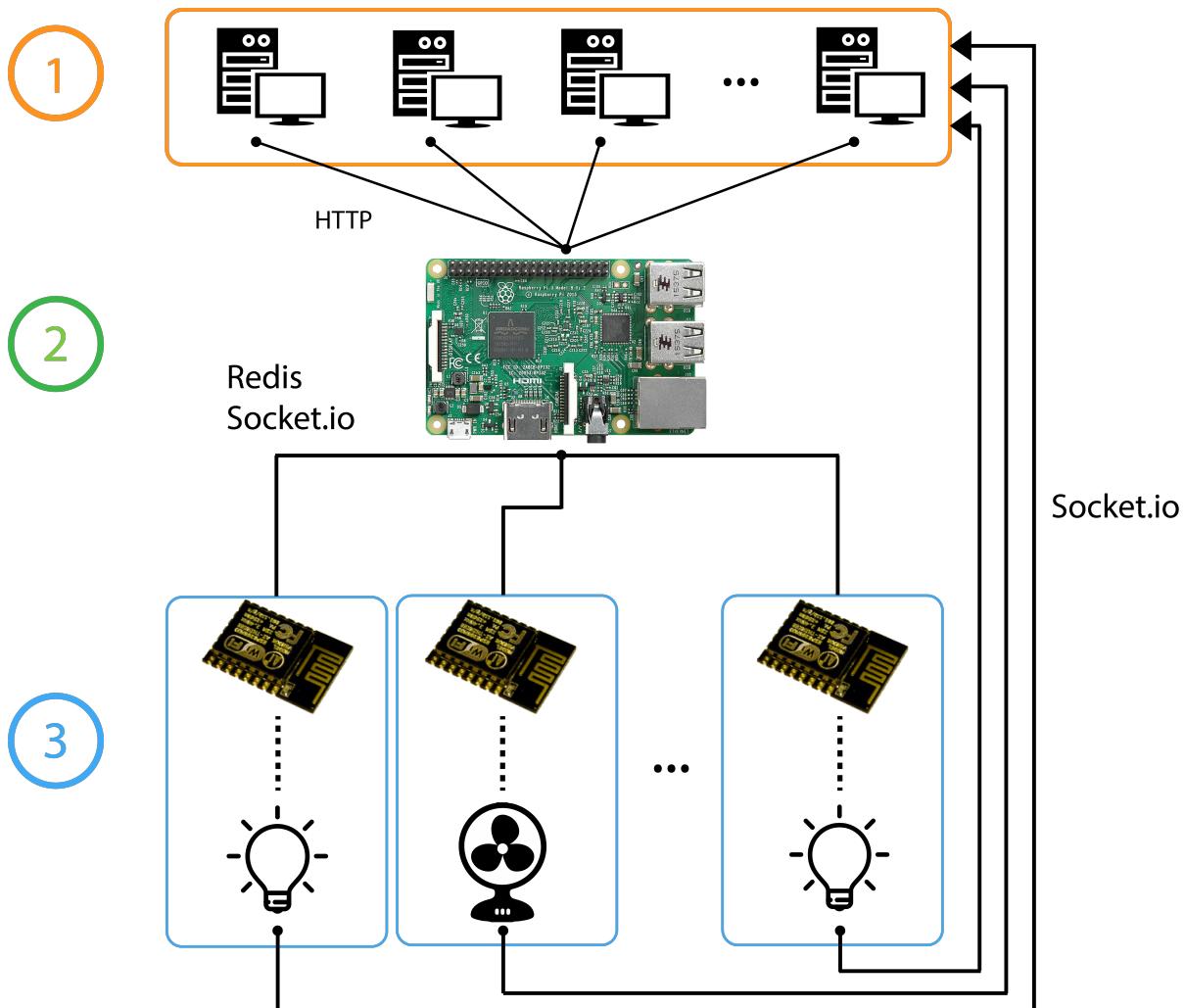


Figura 1 – Arquitetura do projeto

A arquitetura do projeto pode ser dividida em 3 camadas. O código-fonte dos arquivos citados neste capítulo se encontram no [repositório](#) do projeto e estão devidamente comentados de forma a completarem as explicações aqui expostas.

## 5.1 Camada 1

Esta camada contempla os clientes que consumirão o serviço, via Internet. Qualquer dispositivo que emita requisições HTTP pode ser considerado um cliente válido (PCs, smartphones, tablets).

Existe uma camada de autenticação de forma a melhorar a segurança do projeto e restringir seu acesso. Ao acessar o sistema, uma tela de login solicita os dados de

autenticação.

As rotas podem ser acessadas no arquivo *iot\_server/routes/web.php*, disponível no repositório do projeto.

## 5.2 Camada 2

É na segunda camada que o **webserver** é implementado no Raspberry pi. Ele recebe às requisições HTTP advindas dos clientes, as roteia devidamente e retorna a view correta para o usuário. Também no microcomputador são mais dois serviços: um caching com **Redis** e um server **Socket.io**. O Redis é utilizado para capturar eventos disparados pelo Laravel. Esse nível se comunica com de baixo através do servidor Socket.io, por mensagens em broadcasting.

A configuração do Redis é feita no arquivo *iot\_server/.env* e depende de informações particulares da maquina. O evento de clique no botão da Dashboard está implementado no arquivo *iot\_server/app/Events/Button.php*, disponível no repositório do projeto.

O arquivo onde é realizada a inscrição no canal do server Redis e a criação do servidor e emissão de dados pelo Socket.io é o *iot\_server/Socket.js*.

## 5.3 Camada 3

A última camada suporta os denominados **nós**. Cada nó é composto por um ESP8266 e, pelo menos, um periférico conectado. Os nós se inscrevem em um canal específico do servidor Socket.io, por onde a Raspberry enviará o broadcasting. As mensagens enviadas são objeto JSON e contém um identificador que indica qual nó é o real destinatário. Os nós que não são o destino da mensagem, simplesmente a ignoram. O restante da mensagem indica um valor de estado (ligado, desligado ou alguma outra ação) para o microcontrolador escrever em seus periféricos.

Quando houver alteração do estado de um periférico (seja por sinal do microcontrolador ou por alguma alteração externa) é enviado no mesmo canal subscrito do Socket.io, uma mensagem contendo as informações de alteração que serão capturadas por um módulo javascript (*Socket.io.js*) rodando em cada um dos clientes. Desta forma, os estados dos periféricos são atualizados em tempo real para os usuários.

O código principal (que contém a função *loop()*) executado pelos ESP8266 está em *iot\_server/esp8266/socket.io/socket.io.ino*.

A view enviada para o cliente pelo Raspberry pi e que executa o módulo *Socket.io.js* está em *iot\_server/resources/views/dashboard.blade.php*.

# 6 Nós

No projeto, cada nó é constituído de um circuito com o ESP8266 e um (ou mais) periféricos de saída. Os periféricos de saída podem ser: luzes (lámpadas pisca-pisca), Tvs e dispositivos gerais de comportamento binário (ligado/desligado).

Para cada tipo de periférico, foi desenvolvido um tipo de nó. Ao todo são implementados 3 nós, com circuitos distintos, porém com softwares idênticos.

## 6.1 Nô *light*



Figura 2 – Nô *light*

Foi o primeiro nó desenvolvido e dimensionado para um periférico específico: dois conjuntos de lâmpadas pisca-pisca (como as usadas no natal); cada um com 4 sequências paralelas de 3 LEDs cada. O software de acionamento reconhece apenas dois comandos: 1 (para ligar) e 0 (para desligar). Quando ligado, o ESP8266 piscará os conjuntos de LEDs de forma alternada em um período de 500ms.

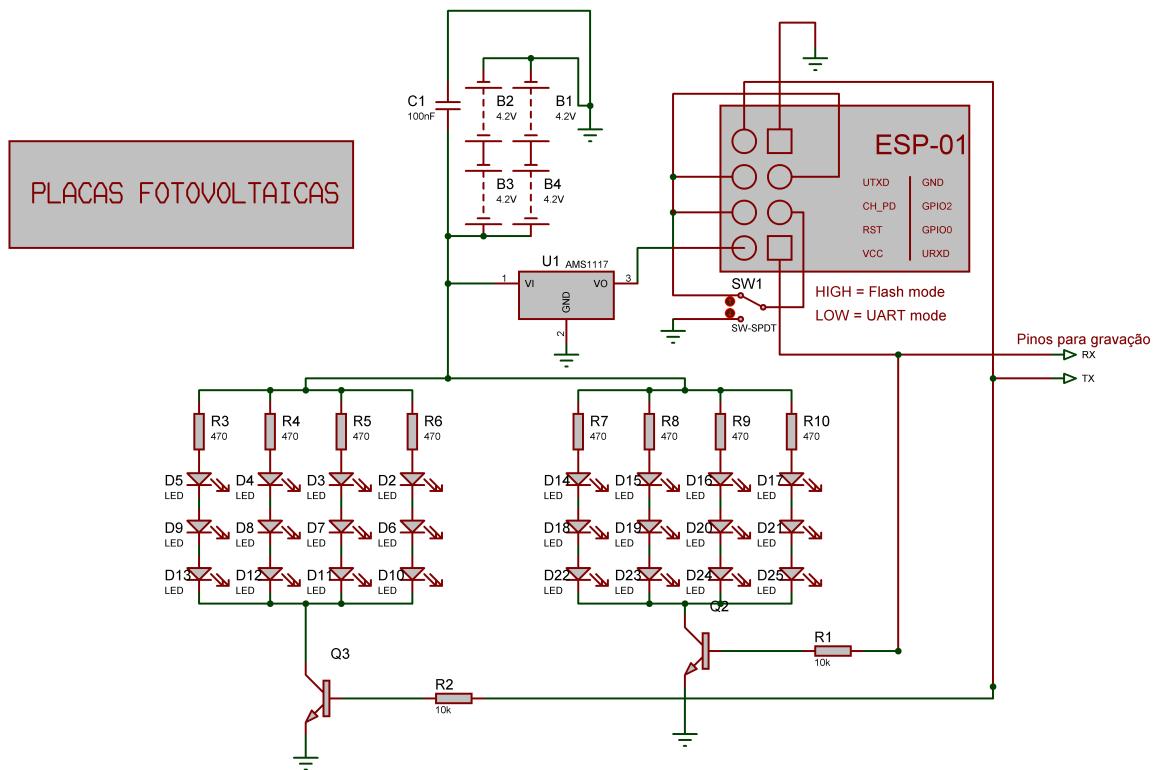


Figura 3 – Esquemático do circuito de um nó *light*

### 6.1.1 Circuito

### 6.1.2 Alimentação

Para alimentação do circuito, foi construído um pack de baterias para adequar à tensão necessária para ativação das lâmpadas e corrente suficiente para garantir uma autonomia de 0 dias. O pack possui 4 pilhas (2 em paralelo - 2 em série). Cada pilha possui 4.2V e 6800mAh. A Figura 5 ilustra o pack.

#### 6.1.2.1 Regulação

A adequação da tensão para alimentação do ESP8266 foi obtida com um regulador de tensão *ams11173V3*. A Figura 6 ilustra o regulador.

## 6.2 Nô *Tv*

O nô *Tv* foi projetado para controlar uma tv remotamente, funcionando como um tradicional controle remoto. O periférico de saída também é específico e simples: um LED emissor infravermelho (IR). O software de acionamento do nô reconhece os seguintes botões mapeados de um controle remote da marca SAMSUNG (funciona para qualquer tv da marca):



Figura 4 – Pack de baterias

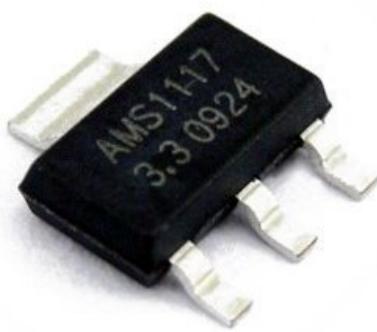


Figura 5 – Regulador de tensão ams1117V3

**power:** liga/desliga a tv;

**source:** altera a fonte de dados da tv;

**vol+** e **vol-**: aumenta/diminui o volume da tv;

**ch+** e **ch-**: avan a/retrocede o canal atual da tv;

**direcionais**: movimenta o seletor da tv para as quatro dire es cardinais e

**enter**: seleciona/confirma.

### 6.2.1 Circuito

O Circuito mais simples que o do n o anterior e possui apenas uma alimenta o adequada para o ESP8266 e um LED emissor infravermelho ligado a um pino digital do microcontrolador.

## 6.3 N o node

Por  ltimo, o n o *node* n o possui um perif rico de sa a definido. Quaisquer dispositivos que tenham comportamento digital, podem ser adicionados no circuito.  um n o vers til de utiliza o gen rica.

## 6.4 Cria o de n os pela plataforma

A implementa o ainda n o permite que o usu rio crie seus pr prios m dulos diretamente no sistema web, por m foi desenvolvido um comando no *Artisan* no Laravel para faz -los: o **make:module**.

### 6.4.1 make:module

Atrav s do comando **make:module**  poss vel criar n os, incluindo-os automaticamente no sistema web. Para execut -lo, basta estar no diret rio root do projeto e execut -lo pelo artisan. A sintaxe  a seguinte:

```
1 | php artisan make:module {type} {id?} {local?} {owner?} {--d} {--M} {--C} {--D}
```

Listing 6.1 – Executando comando **make:module**

Apresentando cada um dos argumentos:

**type**: Tipo do n o (node ou tv). *Light* e *node* s o considerados o mesmo tipo aqui;

**id (opcional)**: O n mero identificador do ESP.  setado em *iot\_server/esp8266/socket.io/config*;

**local (opcional)**: Nome do c modo onde o n o estar ;

**owner (opcional)**: Nome do dono do n o;

**flag -d:** Define o nó como sendo *light*;

**flag -M:** Define o modelo da Tv. Aceitos: SAMSUNG ou LG;

**flag -C:** Define o contador do nó (define a ordem de exposição no sistema web);

**flag -D:** Deleta um módulo pelo número do contador (-C).

A implementação do comando pode ser encontrada no repositório do projeto em *iot\_server/app/Console/Commands/makeModule.php*.

# Parte IV

## Trabalhos futuros

# 7 Circuito

## 7.1 ESP8266

O modelo do ESP8266 utilizado é ESP-01, o pioneiro da série. Apesar de estável, ele possui menos pinos digitais e menos memória flash e RAM. A sugestão é a utilização do ESP-12X, que possui características físicas vantajosas e preço comparável ao modelo já utilizado.

## 7.2 PCB

Outra sugestão é o desenho e confecção de uma pcd, utilizando impressão das trilhas na placa de fenolite. A placa atual foi, manualmente, soldada em uma placa de fenolite furada e, por isso, seu aspecto é grosseiro. O desenho do layout, com disposição otimizada das trilhas auxiliará na montagem do circuito.

## 7.3 Alimentação

Como fonte de recarga da bateria foi pensado o uso de **células fotovoltaicas** no circuito, porém o prazo para aquisição das células não foi suficiente.

## 7.4 Raspberry pi 3

Como uma alternativa ao Raspberry pi, é sugerido o uso de novos microcomputadores significativamente mais baratos, como os modelos da *Orange pi*.

## 7.5 Contratação de host

Como última etapa do projeto, deve ser levado em consideração a contratação de um DNS, um certificado SSL e um serviço de host para a servir a aplicação, de forma à torná-la mais barata (quando comparada à compra de um Raspberry), tercerizando o processamento e fazendo-a mais acessível.

# 8 Software

## 8.1 Nós

Deve ser considerada a implementação de novos tipos de nós que englobem periféricos de comportamento analógico.

Outra sugestão é a ampliação dos modelos de Tv suportados pelo projeto.

## 8.2 ESP8266

Existem outros firmwares, utilizando outras linguagens, compatíveis com o ESP8266. Um deles é o [micropython](#), que traz como vantagens à versatilidade da linguagem Python e uma maneira interessante de modularização do código. Outro aspecto relevante deste firmware é o fato da linguagem utilizada ser interpretada, facilitando mudanças e testes em trechos do código rapidamente.