UNIVERSIDADE
AUTÓNOMA
DE LISBOA

Departamento de Ciências e Tecnologia

Curso de Engenharia Informática

Rede de Controladores de Casa

Autor:

Guilherme Costa Lopes

Vaz Beja

Coordenadores :

Prof. Raul Dionísio

16 de Julho de 2018

Resumo

Na atualidade, sempre existiu a necessidade de controlar e monitorizar, isto é visível tanto nas grandes industrias, como nas casa de habitação. No primeiro caso, os sistemas de manufaturização, distribuição de *stock* e futuramente venda de produtos serão automatizados ou semi-automatizados. Já nas casa de habitação, um realidade mais próxima do utilizador, sendo composta por diversos *gadgets*, que tem como objectivo de tornar certas tarefas rotineiras mais rápidas e com menos intervenção do homem

Tendo esta premissa em vista, o projeto foca-se na criação de um sistema centralizado, utilizado uma quantidade indefinida de dispositivos modulares, para gerar um ambiente controlado e posteriormente monitorizado. Estes dispositivos aos serem modelares, criam a possibilidade de serem configurados, tudo depende do seu contexto. Existem vários aspectos que devem de ser previamente analisados antes da implementação, na escolha do local poderá haver ser factores que possam definir a instalação do dispositivo.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

O presente projeto foi desenvolvido no âmbito da cadeia de Laboratório de Projeto, orientado pelo professor Raul Dionísio. Este passa por criar um sistemas de controlo e monitorização para casa, mais concretamente criar uma aplicação web e uns dispositivos eletrônicos. Os objectivo trassado passam por demonstrar competências ao nível de programação, base de dados, comunicaçoes, organizaçao e planeamento de projetos. Este documento está dividido em duas partes. A primeira parte aborda o problema gerado pelo mercado de dispositivos, que tornam as casas mais automatizadas, e ainda se apresenta um sistema que soluciona o mesmo. Na segunda parte estuda-se a implementação e todos os pormenores que se tem que ter em conta na criação da solução.

2 Visão Geral

2.1 Problema

Com a última revolução tecnológica a automatização dos espaços que nos rodeiam começa a ser uma necessidade. Como resposta, aparece a área da domótica[6], que é definida pela integração de mecanismos automáticos dentro de casas para simplificar o quotidiano das pessoas. Muito ligado ao conceito da domótica estão os dispositivos Smart Home [5]. Estes passam por permitir o controlo e monitorização da iluminação, climatização, segurança, som de ambiente, entre outras. No entanto, as soluções disponíveis (Fig.**??**) focam-se, na maioria, apenas num aspecto ou aplicação, sendo que para obter um controlo total sobre uma área, é necessário recorrer a diversas marcas e dispositivos, cada um com o seu protocolo, diferentes interfaces físicos, entre outros.

2.2 Solução Proposta

A solução que se propõe contem duas partes. Primeiro, a criação de um protocolo (HomeController Protocol) que unifica diferentes sensores e atuadores(*input, output*) com diferentes protocolos e interfaces. A segunda parte passa pela criação de um software de controlo e monitorização que permite aceder aos componentes através do protocolo anteriormente referido. Desta forma cria-se um sistema de domótica versátil, modular e expansível a novos componentes. De seguida observa exemplos de alguns componentes potencialmente suportados:

- Sensor de temperatura;
  - Digital: I^2C, SPI, 1-Wire;
  - Analógico, lido por um ADC.
- Sensor de luminosidade;
- Relé;
  - Solid State;
  - Solenoide.
- Motores;
  - Servo: PWM;
  - Steppers.
- Outros.

De forma a interagir com os componentes, estes organizaram-se em grupos no que se chama **HomeController**. O HomeController é uma placa física, desenhada e montada no contexto do projecto, que permite interligar múltiplos componentes desde exista suporte para os mesmos. Na Fig. 1 observa-se como os componentes, os HomeControllers e o controlo formam o sistema.

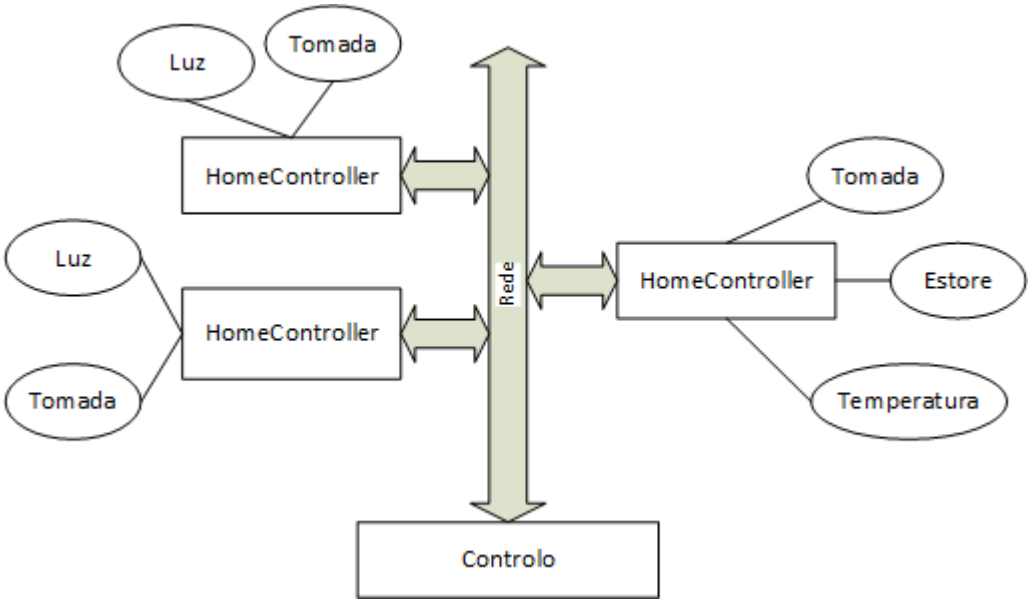


Figura 1: Vista geral da solução proposta.

A Fig. 1 evidencia as três grandes características do sistema: Controlo, Rede e HomeController.

2.2.1 Controlo

O controlo do sistema tem como objetivo fornecer acesso a todos os HomeControllers na rede através do *HomeController Protocol* e disponibilizar ao utilizador uma forma de interagir com os dispositivos. O controlo é um sistema central (topologia em estrela) que viabiliza o acesso aos HomeControllers via *HomeController Protocol* construído por cima do protocolo HTTP e estabelece a interação com o utilizador usando uma *Web Application*.

2.2.2 Rede

Segundo a Fig. 1 os HomeControllers interagem com o controlo através de um acesso a um meio comum. Após alguma consideração o protocolo escolhido para implementar as comunicações é o *IEEE 802.11*, ou na sua forma mais conhecida *Wifi*. Esta escolha deve-se ao facto de existirem muitos dispositivos com capacidades *Wifi*, bastante documentação e fácil acesso.

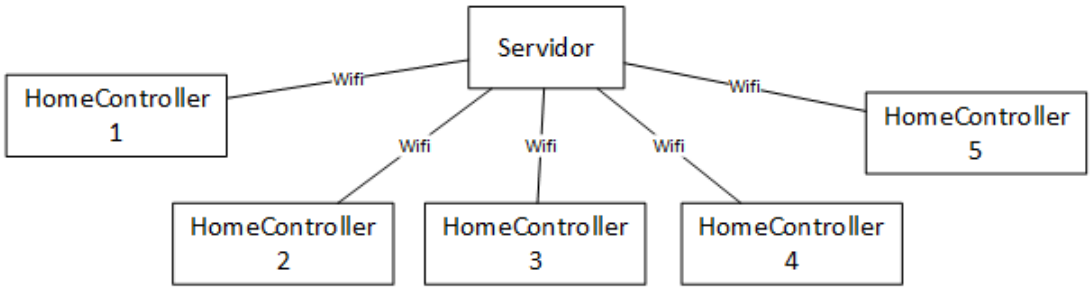


Figura 2: Diagrama de um sistema em estrela.

Segundo a Fig. 2 o sistema projetado necessita de uma estrutura centralizada com uma topologia em estrela. Define-se por uma componente central, um servidor, contendo periféricos nas suas extremidades, dispositivos *HomeController*, que recebem pedidos e retornam respostas (protocolo HTTP).

2.2.3 HomeController

O HomeController é um dispositivo que faz interface com vários componentes de *input/output*: sensor de temperatura, LED, etc. Nestes dispositivos encontra-se um pequeno servidor HTTP com objetivo de responder a pedidos do controlo. O conjunto de diferentes possíveis pedidos HTTP constitui o **HomeController Protocol**. Estes dispositivos encontram-se implementados em Hardware recorrendo ao microcontrolador *ESP-8266* e diversos componentes de eletrónica.

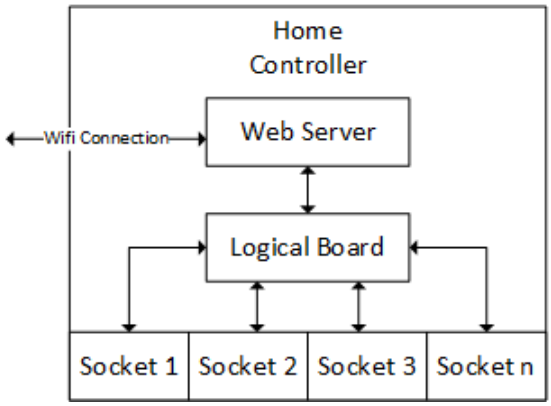


Figura 3: Diagrama de blocos de um HomeController.

Tal como se pode observar na Fig.3 um HomeController suporta várias componentes de entrada/saída ao mesmo tempo através de **Sockets**. Um **Socket** é um conjunto de GPIOs que é capaz de suportar diferentes protocolos e interfaces. Um HomeController para utilizar um componente necessita de ter suporte (*drivers*) que especifiquem o comportamento desse. O mesmo **Socket** pode ser usado, por exemplo, tanto para um regulador de temperatura com para um estore controlado por um

motor. O número de componentes suportados pelos sockets é definido na *firmware* que corre nos HomeControllers.

3 Implementação

Neste capitulo observa-se em detalhe a implementação da solução proposta, explorando a construção do sistema de controlo, as diferentes arquiteturas e protocolos utilizados, tecnologias e razões pelas escolhas das mesmas.

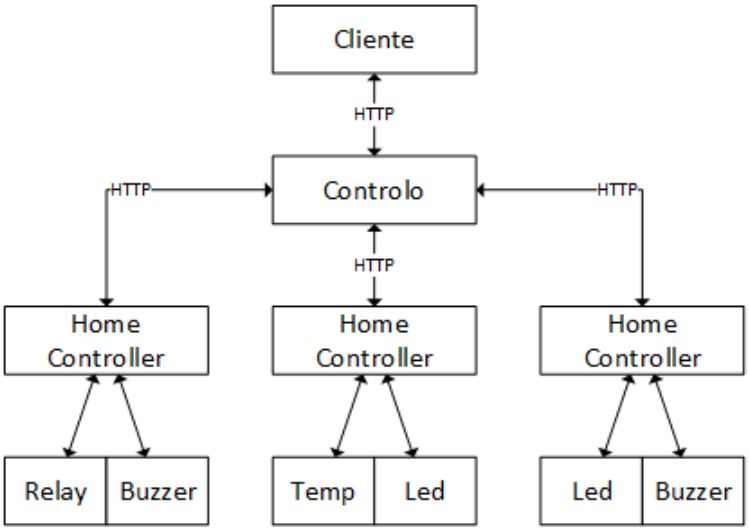


Figura 4: Diagrama de blocos do sistema.

A Fig. 4 representa o objetivo final do projeto em que um sistema de controlo fornece uma via para que um utilizador possa configura e aceder aos HomeControllers e também gere a rede dos mesmos.

3.1 Controlo

Esta componente tem como objetivo controlar todos os aspetos relacionados com o projeto desde os HomeControllers até à forma como o utilizador interage com os mesmos. O sistema de controlo encontra-se separado em duas partes: HomeController API e Web Framework. A primeira constitui um conjunto de comandos para interagir com os HomeControllers e a segunda passar por criar um ambiente central onde se controla todos os comandos que passam pelo sistema.

A Fig. 5 demonstra com algum detalhe as diversas componentes que constituem o sistema de controlo. A web framework segue uma arquitetura MVC (*Model View Controller*) [10]. Os modelos (*Model*) implementam-se usando o módulo de javascript ***Mongoose***[1], que interage com uma base de dados do tipo MongoDB[3]. As vistas (*View*) são geradas usando *Handlebars*[4]. Finalmente, a componente controlo é implementada usando a ferramenta ***Express***[2] para javascript.

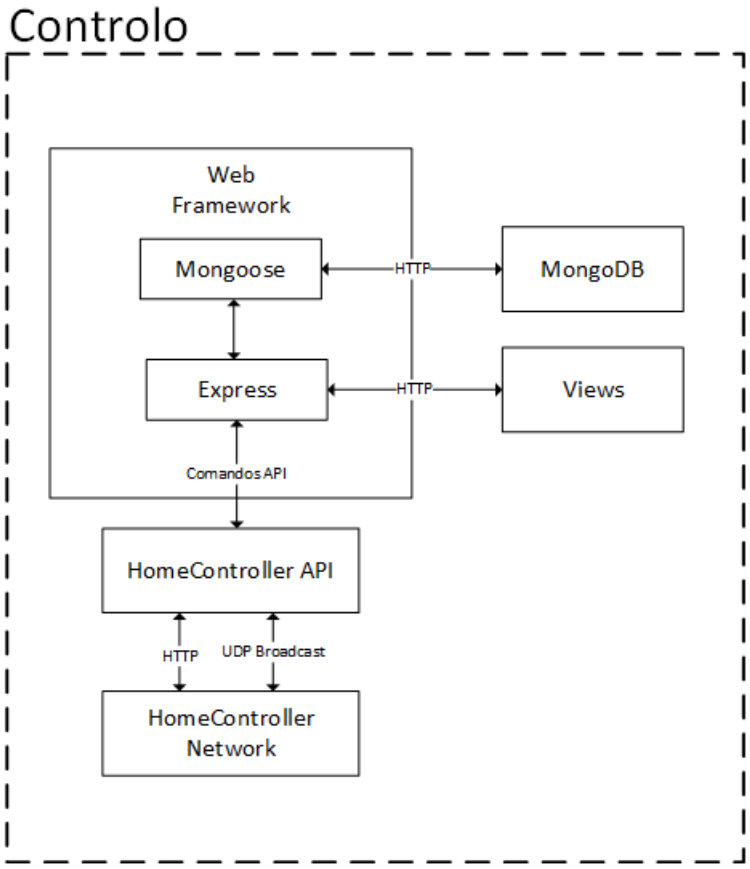


Figura 5: Diagrama de blocos da Home Controller API.

Para interagir com um HomeController o utilizador faz um pedido, através das vistas (*Views*), ao Express (*Controller*) que por si, através da HomeController API, envia o comando para a rede.

3.1.1 HomeController API

A componente HomeController API fornece uma forma de controlar os dispositivos HomeController, através de um conjunto de comandos implementados em HTTP, ou seja, um utilizador ao mudar o estado de um componente, irá enviar um comando HTTP para o HomeController.

Comandos API

Existem três tipos de comandos na API: Config, Update e Execute. Todos eles contem um pedido como uma resposta. Os pedidos têm vários tipos de métodos[12] e as respostas enviem vários códigos de estado[11].

Config

O comando **Config** serve para obter uma "imagem" do estado do HomeController. Este contém: o número de Sockets, os componentes que cada Socket suporta, qual componente que os Socket estão a suportar no momento e estado do HomeController.

Tabela 1: Pedido HTTP de Config.

Propriedade HTTP	Valor
Method	GET
URL	http://[ip]/board/config

Tabela 2: Resposta HTTP de Config.

Propriedade HTTP	Valor
Status Code	200, 400 ou 404
Content-Type	JSON
Body	Descrito adiante

O pedido HTTP correspondente ao comando de Config da API encontra-se na tabela 1. Este suporta apenas o método GET. Um pedido para este URL com o método POST resultaria numa resposta com *status code* de 400 (*bad request*). A resposta para este pedido observa-se na tabela 2. O corpo que este retorna apresenta o seguinte formato:

```
[
  {
    "id": "1",
    "SupportedDevices": [
      "3",
      "0"
    ],
    "CurrentDevice": "0"
  },
  {
    "id": "2",
    "SupportedDevices": [
      "3",
```

```
      "2",
      "0"
    ],
    "CurrentDevice": "0"
  }
]
```

Este segmento de código representa a resposta a um pedido de Config. Este é um objecto JSON que contém um *array* com as carateristas que todos os Sockets de um HomeController contêm. O primeiro elemento é o ID do Socket, note-se que este começa em 1 pois o 0 foi reservado para a própria placa HomeController. A segunda caraterística é um outro *array* que contem os IDs correspondentes ao dispositivos que são suportados pelo determinado Socket (neste exemplo, 3 para relé, 2 para o sensor de temperatura e o 0 serve para indicar que o Socket não tem nenhum dispositivo). Por fim a última caraterística refere o componente que está instalado no momento.

Update

O comando **Update** serve para mudar o componente que se encontra suportado por um Socket. Este comando indica de qual o Socket que se pretende alterar e o tipo de componente que se deseja mudar.

Tabela 3: Pedido HTTP de Update.

Propriedade HTTP	Valor
Method	GET
URL	http://[ip]/board/interface/device/change
Parâmetro 1	interfaceNumber
Parâmetro 2	newDevice

Tabela 4: Resposta HTTP de Update.

Propriedade HTTP	Valor
Status Code	200, 400, 404, 403 ou 500
Content-Type	...
Body	...

O pedido HTTP correspondente ao comando de Update da API encontra-se na tabela 3. Este suporta apenas o método GET, contudo tem dois parâmetros: um serve para indicar um código, que é dado aos componentes, e o outro para indicar qual a nova localização do componente. A

resdposta a este pedido, vista na tabela4, tem como objectivo retornar se este foi sucedido, através do *status code*.

Execute

O comando **Execute** serve para interagir com um determinado Socket de um HomeController. Este comando contém parâmetros para interagir com o *Driver* do componente que se encontra no Socket.

Tabela 5: Pedido HTTP de Execute.

Propriedade HTTP	Valor
Method	POST
URL	http://[ip]/board/interface/device/call
HTTP Body	
Parâmetro 1	<valor>
Parâmetro 2	<valor>
Parâmetro n	<valor>

Tabela 6: Resposta HTTP de Execute.

Propriedade HTTP	Valor
Status Code	200 ou 400
Content-Type	octet-stream
Body	byte array

O pedido HTTP correspondente ao comando de Execute da API encontra-

se na tabela 5. Este suporta apenas o método POST, com ***n*** parâmetros, que são enviados no corpo. A quantidade variável de parâmetros deve-se ao facto de os Drivers receberem associados aos Sockets receberem um número de parâmetros variáveis. A reposta associada a este pedido, tabela 6, tem dois tipos de *status code*. O código 200 (OK) significa que o pedido foi bem sucedido e o Driver executou a função com sucesso. Já o código 400 (Bad Request) significa que houve erros nos parâmetros, por exemplo número de parâmetros insuficiente.

3.1.2 Web Framework

A Web Framework foi criada com o objetivo estabelecer uma ligação entre o utilizador e os HomeControllers. Fornece uma forma de interface gráfico através o utilizador interage com o sistema e comunica com a rede de HomeControllers através da HomeController API.

Arquitetura MVC

A aplicação web construída baseia-se na arquitetura padrão MVC (Model, View, Controller) [10]. Esta separa-se em três partes (Figura 6): Model, View, Controller. O Model define o tipo de dados que a aplicação tem, como se armazenam e se relacionam entre si. Já a componente View, é responsável pela apresentação gráfica dos dados da aplicação. Por fim, o Controller contém a parte logica do programa, ou seja, é responsável por manipular a parte do Modelo e dar resposta as interações que o utilizador faz com a View.

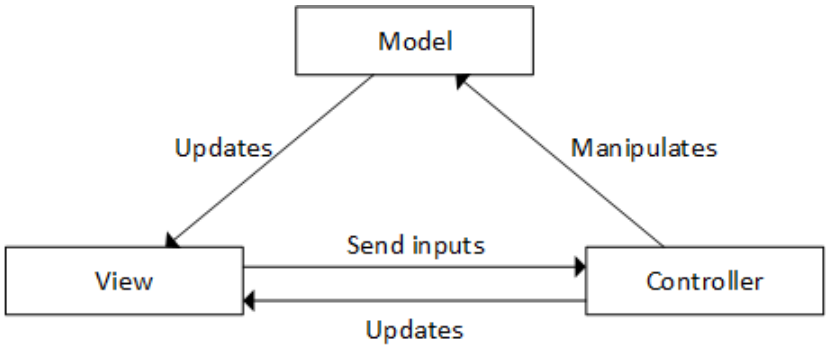


Figura 6: Diagrama de blocos de MVC.

A figura 7 representa como a arquitetura MVC se aplica à Web Fra-

mework.

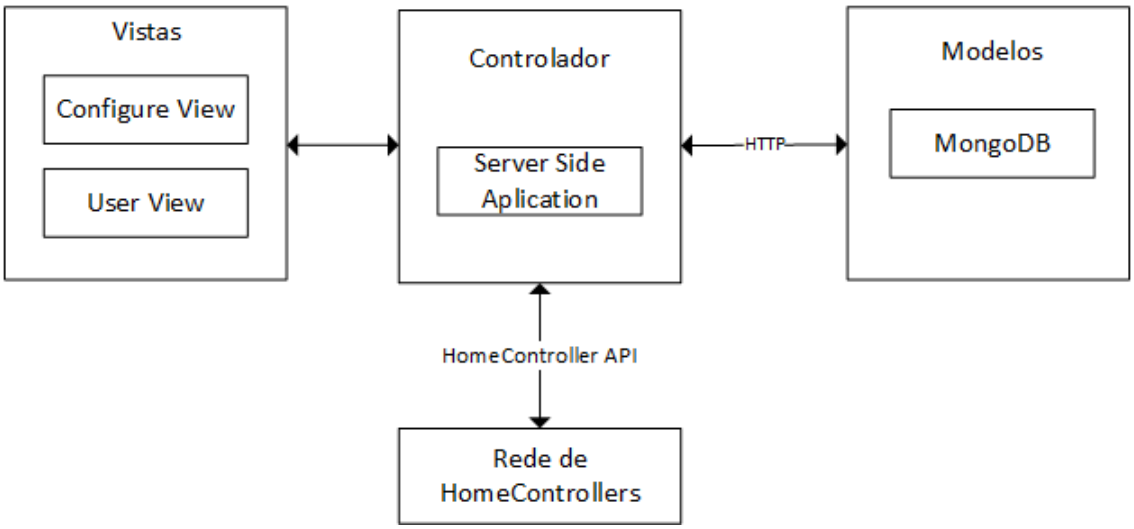


Figura 7: Representação gráfica do funcionamento do sistema.

A componente base de dados utiliza-se o sistema MongoDB para guardar os dados necessários. Com a componente Vistas, criada em HTML e Express, proporciona ao utilizador uma parte gráfica para interagir com os Home Controller. Por fim, a componente Home Controller API fornece uma forma de controlar os dispositivos Home Controller, através de um conjunto de comandos implementados em HTTP. Esta corre num servidor instalado no local onde se pretende utilizar o sistema.

Modelo

O software de base de dados usado para guardar as informação sobre os HomeControllers é o MongoDB. Este é uma base de dados é do tipo NoSQL, não relacional, orientada a documentos. Contudo, para interagir com a base de dados é preciso utilizar o Mongoose[1]:um módulo para javascript que implementa na totalidade a componente de Model em MVC. Com o Mongoose, torna-se possível e fácil efetuar procuras na base de dados, alterar e guardar documentos. Este módulo permite também a criação de esquemas (conjunto de regras que definem documentos) facilitando assim a criação e integração de novos modelos na aplicação. O excerto de código seguinte representa a criação do modelo dos HomeControllers.

```
1  var mongoose = require('mongoose');
2  var Schema = mongoose.Schema;
3
4  var HomeController = new Schema({
5    IP : String,
6    Name: String,
7    Interfaces : [{
8      Name: String,
9      Supported : [Number],
10     Current : Number
11   }],
12 });
13
```

```
14 module.exports = {
15   HomeController : mongoose.model('HomeControllers', HomeController)
16 };
```

As constituintes do modelo HomeController são as seguintes:

- **IP**: identificador na rede (camada *networking* TCP/IP) do HomeController;
- **Name**: nome que o utilizador queira dar ao dispositivo. Este, por exemplo, é usado para substituir o IP no Interface grafico com o qual o utilizador interage;
- **Interface**: *array* de comprimento definido pela quantidade de Sockets no HomeController. Este tem as seguintes características:
  - **Name**: nome que o utilizador pode dar ao Socket. Este é usado para substituir o ID do componente, por exemplo: "Temperatura", "Tomada", "Luz", etc.
  - **Supported**: *array* que contem os IDs dos componentes que o Socket suporta;
  - **Current**: ID do componente que o Socket tem ativo no momento.

Na figura 8 representa como o modelo HomeController se encontra armazenado na base de dados.

Figura 8: Objeto dentro da base de dados.

Nesta estrutura de dados, note-se que existe uma adição ainda não falada, o campo **_id**. Este campo é introduzido pelo Mongoose e tem a mesma funcionalidade que uma chave primário numa base de dados relacional.

O excerto de código seguinte demonstra como se pode aceder a um modelo que se encontre na base de dados, usando Mongoose. Quando o objeto relacionado ao modelo é encontrado, a função de *callback* passada como segundo parâmetro é executada.

```
1 HomeController.findById(2, function (err, c) {
2 });
```

Esta função, acede, através de um ID (neste exemplo, 2) um modelo que se encontre na base de dados. Caso exista algum erro no acesso, o parâmetro ***err*** da função de *callback* retorna um objeto com a especificação do mesmo. Caso contrário, ***err*** é **null** e o parâmetro ***c*** contém o objeto HomeController. Agora torna-se possível alterar os campos do objeto e voltar a guarda-lo na base de dados, como se pode observar no excerto de código seguinte.

```
1 HomeController.findById(req.params.controllerID, function (err, c) {
2   if(err != null){
3     return;
4   }
5   c.Name = 'New Name';
6   c.save();
7 });
```

Vistas

A componente Vista (Views) da aplicação tem como objetivo fornecer um interface gráfico pelo qual o utilizador interage com a Controlo e consequentemente com os HomeControllers. Esta é composta por duas vistas: uma dedicada à configuração dos aspetos relacionados com os HomeControllers e uma outra para a interação com os componentes que se encontram ativos nos Sockets dos HomeControllers.-

Ambas as vistas são geradas usando o motor de templates Handlebars [4] e recorrer a HTML[13], CSS[14], Javascript[15][19], Bootstrap[16], jQuery[17] e AJAX[18] (Asynchronous JavaScript And XML). As componentes jQuery e AJAX foram incluídas com intuito de comunicar com o servidor assincronamente para implementar as seguintes funcionalidades:

- **Auto Descoberta**: quando aparece um novo HomeController na rede, após o emparelhamento com o servidor, a vista de configuração automaticamente é atualizada de forma a demonstrar o HomeController.

- **Auto Update**: Quando existe alguma alteração nos campos de um HomeController, esta tem que se refletir em todas as suas instâncias. Por exemplo, quando um utilizador liga uma luz, um outro que esteja a observar o mesmo HomeController tem que também observar esta mudança de estado.

- **Interação com os HomeControllers**: Toda a interação com os

HomeControllers é não bloqueante em aspetos gráficos. Isto significa que qualquer comunicação com sentido cliente -> Web Framework é através de pedidos AJAX. Assim torna-se possível que vários utilizadores acedam e modifiquem o mesmo componente ao mesmo tempo, pois as prioridades são resolvidas e demonstradas nas vistas assincronamente.

A fig.9 apresenta um formulário de um HomeController na vista de configuração. Neste aparece todos os Sockets disponíveis assim como o seu nome. É aqui que se muda quais os componentes que estão a ser utilizados nos Sockets e os seus nomes.

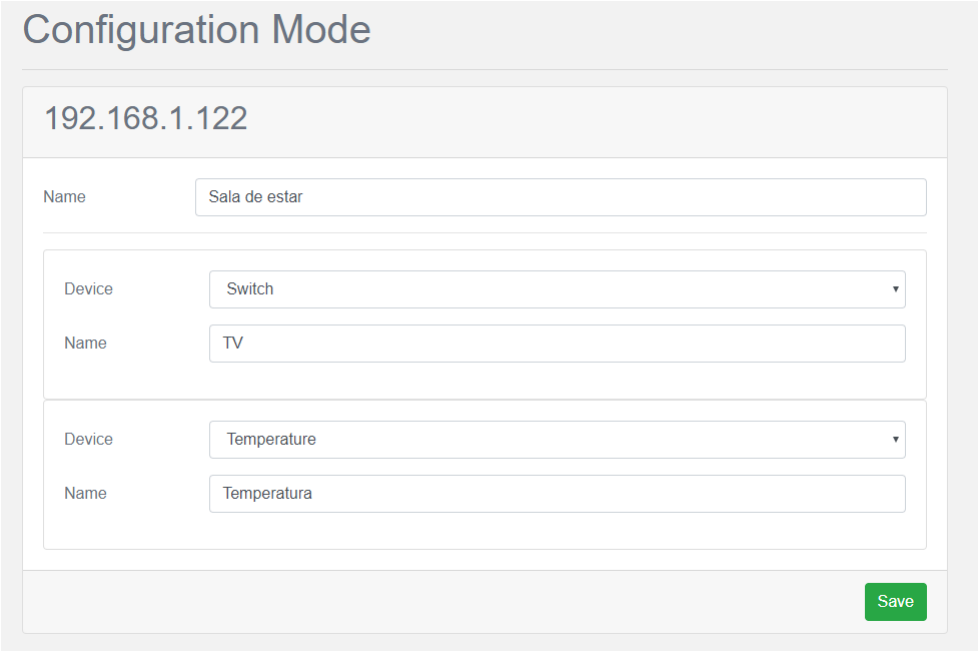


Figura 9: Formulário de configuração.

Neste caso o HomeController 192.168.1.122 tem dois Sockets: um que se encontra a suportar um interruptor (Switch) e outro suporta um sensor de temperatura. Para a futura identificação do equipamento na rede muda-se o nome para “Sala de estar”. Também é possível identificar os Sockets por nome. O primeiro Socket, responsável pelo interruptor da televisão chama-se “TV”. O segundo, responsável por adquirir a temperatura da sala, chama-se “Temperatura”.

Ao aceder agora à vista de interação, deparamo-nos com uma lista de formulários dos HomeControllers. A Fig. 10 apresenta um formulário para o HomeController previamente configurado (Fig. 9). Cada tipo de componente tem uma forma especifica de se interagir. No caso do Socket “TV”, que é um interruptor, é possível liga-lo e desliga-lo através de um botão deslisante. Já a “Temperatura” mostra apenas, em graus centigrados, a temperatura da sala de estar.

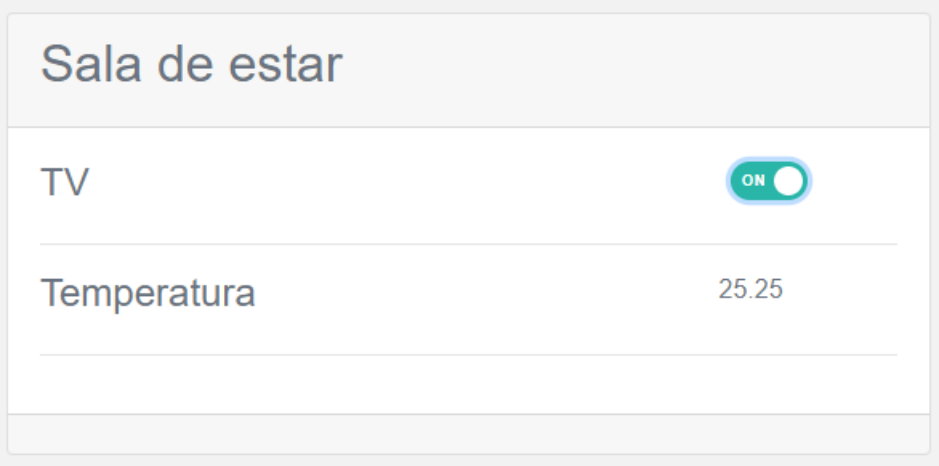


Figura 10: Formulário de utilização.

Controlo

A componente controlo tem como objetivo relacionar as vistas aos modelos e foi implementada recorrendo à Web Framework Express[2]. Encontra-se encarregue de interagir com a HomeController API, gerar as vistas usando Handlebars e responder a pedidos HTTP.

3.2 HomeController

O HomeController é uma peça de hardware baseada num microcontrolador com capacidades wifi que permite múltiplos componentes e protocolos que sejam unificados sob um único protocolo (HomeController API).

3.2.1 Software

O software do HomeController é o que vai proporcionar ao dispositivo realizar as suas tarefas. O código encontra-se em anexo com comentários detalhados. Como se pode ver na Fig. 11 encontra-se dividido em três partes: Comunication, Handlers e Drivers.

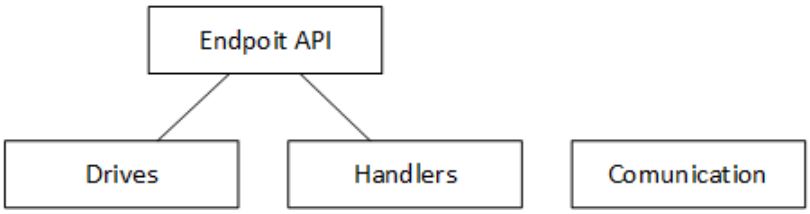


Figura 11: Diagrama de componentes do software.

Comunicação

Pode-se considerar que um HomeController tem dois estados: antes de entrar na rede e depois de entrar na rede. No primeiro estado é possível configurar o HomeController de forma a emparelhar-se com uma rede. Isto é possível pois este, quando inicia, começa com um ponto de acesso (Acess Point) com um pequeno servidor disponibilizando apenas uma página web. Esta página contém um formulário que permite indicar os dados de acesso à rede que se pretende ligar. Uma vez sucedido, o HomeController deixa de ser um ponto de acesso e passa para o segundo estado no qual é uma estação de wifi (STA). É no segundo que corre o programa principal, no qual através da HomeController API, é possível aos componentes suportados pelos Sockets. A forma como se interage com o HomeController estando este em modo estação é através de um servidor HTTP. Os pedidos e respostas da HomeController API estão implementados usando o protocolo HTTP.

O seguinte excerto de código demonstra como ativar o modo ponto de acesso no HomeController, passando a ser possível ligar-se ao mesmo como se fosse um router. Os parâmetros *SSID* e **password** representam, respetivamente, o SSID e WPA2 do ponto de acesso.

```
1 void accessPoint() {
2   WiFi.softAP(ssid, password);
3   IPAddress myIP = WiFi.softAPIP();
4 }
```

Quando em modo de ponto de acesso o servidor que corre no HomeController response ao pedido **GET http://[homecontrollerIP]/** com a página representada na Fig. 12.

Figura 12: Formulário de configuração de uma nova rede no HomeController.

Depois de preencher o formulário, que pede os dados de entrada para uma rede onde está o resto do sistema, o modo de comunicação muda para estação (STA). Neste estado o HomeController já faz parte do sistema, contudo tem primeiro que avisar o servidor que se ligou a uma nova rede. Para tal, emite de tempo em tempo um sinal de heartbeat

em broadcast usando o protocolo UDP [**?**] até que o sistema de Controlo confirme que já o identificou. O seguinte excerto de código apresenta a função encarregue de emitir o sinal de heartbeat.

```
1 void heartbeat() {
2   udp.beginPacket(IP_Remote, 3333);
3   udp.write(applicationIdentifier, 7);
4   Serial.println("Heartbeat");
5   udp.endPacket();
6 }
```

Quando recebida a confirmação pelo sistema de Controlo que o HomeController já se encontra reconhecido, desliga o sinal de heartbeat, passando a comunicar com o Controlo através de um servidor HTTP.

Drivers

O conceito de Driver foi introduzido com o intuito de permitir que uma componente de entrada/saída, independentemente do seu protocolo e interface, se possa interligar com o HomeController. O elemento encarregue de chamar a driver de uma componente é o mesmo que o suporta, ou seja, o Socket. Quando um Socket deseja interagir com um componente (por exemplo, ler temperatura de um sensor I^2C), chama a driver desse componente, que sabe lidar com o seu protocolo. No entanto, quando se muda o componente que um Socket está a suportar, é necessário preparar o pinos de GPIO que este vai utilizar. Para tal exista uma função de iniciação de Drivers. Sempre que se quer adicionar um novo

componente o seu suporte é acompanhado por três elementos: o ID do componente, a função que inicia o driver e a função de driver. O excerto de código seguinte apresenta uma função de iniciação do Driver de um relé genérico.

```
void relayDriverInit(void * slot) {
  struct Slot * caller = (struct Slot *) slot;
  int relayPin = -1;

  for (int i = 0; i < caller->pinsSize; i++) {
    if ((caller->Pins[i]->TypeFlags & PIN_TYPE_GPIO) == PIN_TYPE_GPIO) {
      relayPin = caller->Pins[i]->Number;
      break;
    }
  }
  if (relayPin == -1) {
    return;
  }
  pinMode(relayPin, OUTPUT);
}
```

A função que inicia o suporte para o componente é chamada quando o Socket muda de componente que está a suportar. Neste caso em especifico, como a interface com o relé é feita usando GPIO, a função procura um pino com capacidade GPIO no Socket (pois os Sockets são conjuntos de pinos) e coloca-o em modo OUTPUT. Desta forma torna-se possível controlar com sinais lógicos (HIGH, LOW) o relé.

Quando o Socket interage com o componente, esta interação é feita através da função de Driver. A função é responsável por interagir com o componente, pois esta conhece o seu protocolo. É também responsável por ler dados (se aplicável) de um componente e escreve-los numa área da memória, indicada no parâmetro.

```
void relayDriver(void * slot, int paramSize, uint8_t * params,
    EndpointResponse * response) {
  Serial.println("Relay Driver Call");
  struct Slot * caller = (struct Slot *) slot;
  int relayPin = -1;

  for (int i = 0; i < caller->pinsSize; i++) {
    if ((caller->Pins[i]->TypeFlags & PIN_TYPE_GPIO) == PIN_TYPE_GPIO){
      relayPin = caller->Pins[i]->Number;
      break;
    }
  }
  if (relayPin == -1) {
    return;
  }

  digitalWrite(relayPin, params[0]);
  response->length = 0;
  return;
}
```

A função em cima é responsável por mudar o estado do relé quando o Socket tenta interage com o dispositivo. Primeiro, a função vai descobrir em que GPIO o relé se encontra. Caso não encontre nenhum pino que cumpra as requisitos, a função retorna sem alterar o estado. Uma vez encontrado o pino falta afetar o estado do relé, cuja informação vem no primeiro índice do *array* de parâmetros (**params**). No caso do exemplo, se o valor for zero o relé desliga, se for um, o relé liga.

Handlers

Para comunicar com o sistema de Controlo o HomeController implementa um pequeno servidor HTTP, que contém três funções (handlers) que são chamadas dependentemente do pedido HTTP. Estas funções são: “getInterfaces”, “changeCurrentDevice” e “deviceInteract”.

getInterface

O handler getInterface tem como objetivo retornar uma String de formato JSON com a quantidade de componentes que estão nos Sockets e as suas caraterísticas. Como se pode observar no código seguinte, esta função é chamada quando o servidor de HTTP recebe um pedido com o URL “/board/config”.

```
server.on("/board/config", getInterfaces);
```

A função em baixo, demonstra como é que se cria a String JSON que é enviada como resposta. Esta String tem que conter as caraterísticas do Socket do HomeController, que são as seguintes: ID do Socket, um array com os componentes que suporta e o componente que está a ser usado no momento.

```
void getInterfaces() {
  uint8_t numberOfDevices;
  data[0] = NumberOfSlots;

  EndpointAPI_Request(0, 1, data, &rsp);
  numberOfDevices = rsp.data[0];

```

```
8  json = "[";
9
10 for (uint8_t i = 0; i < numberOfDevices; i++) {
11   json += "{\"id\":\"";
12   json += i + 1;
13
14   json += "\", \"SupportedDevices\":[";
15
16   data[0] = GetSupportedDevices;
17   data[1] = i+1;
18   EndpointAPI_Request(0, 1, data, &rsp);
19
20   Serial.println(i + 1, DEC);
21
22   for (int j = 0; j < rsp.length; j++) {
23     json += "\"";
24     json += (int)rsp.data[j];
25     json += "\"";
26     if (j < rsp.length - 1) {
27       json += ",";
28     }
29   }
30
31   json += "]";
32
33   data[0] = CurrentDevice;
34   data[1] = 1;
35   data[2] = i+1;
36
37   EndpointAPI_Request(0, 3, data, &rsp);
```

```
38
39      json += ", \"CurrentDevice\":\"";
40      json += (int)rsp.data[0];
41      json += "\"";
42      json += "}";
43
44      if (i < numberOfDevices - 1) {
45        json += ",";
46      }
47    }
48    json += "]";
49    server.send(200, "text/plain", json);
50  }
```

Depois de todos os *fors* e *ifs* que a função utiliza para criar a String, esta tem que sugir com a seguinte forma:

```
1  [
2    {
3      "id":1,
4      "SupportedDevices":[1, 2, 5],
5      "CurrentDevice":1
6    }
7  ]
```

changeCurrentDevice

O handler changeCurrentDevice tem como objetivo alterar o componente que está num determinado Socket. Como se pode observar no código seguinte, esta função é chamada quando o servidor de HTTP recebe um pedido com o URL "/board/interface/device/change".

```
server.on("/board/interface/device/change", changeCurrentDevice);
```

A função em baixo, demonstra como é que a alteração dos componentes do Sockets é efetuada. Apesar de ser o URL que chama a função, existem outros paramentos que vem dentro do pedido. Estes parâmetros indicam: o ID do Socket e o ID do componente que é para ficar.

```
void changeCurrentDevice() {
  //url: POST /interface/change  | params: interfaceNumber=1,newDevice=2
  int interfaceNumber = server.arg("interfaceNumber").toInt();

  if (interfaceNumber == 0) {
    server.send(400);
    return;
  }

  int newDevice = server.arg("newDevice").toInt();

  if (newDevice == 0) {
    server.send(400);
    return;
  }

  uint8_t data[4];
  data[0] = CurrentDevice;
  data[1] = 2; // GET = 1, POST = 2
  data[2] = interfaceNumber;
  data[3] = newDevice;

  EndpointAPI_Request(0, 4, data, &rsp);
```

```
24    if (rsp.data[0] == 0) {
25      server.send(200);
26    }
27    else {
28      server.send(500);
29    }
30  }
```

deviceInteract

O handler deviceInteract tem como objetivo interagir com Drives dos componentes. Como se pode observar no código seguinte, esta função é chamada quando o servidor de HTTP recebe um pedido com o URL "/board/interface/device/call".

```
1  server.on("/board/interface/device/call", deviceInteract);
```

A função em baixo demonstra como é que se preparar a informação que se utiliza para interagir com as drives dos componentes. Apesar de ser o URL que chama a função, existem outros paramentos que vem dentro do pedido. Estes parâmetros são usados como argumentamentos nas drives dos componentes. Eles chegam à função num String continua que é repartida nas seguintes partes: numero do Socket, quantidade de argumentos, os argumentos e um apontamento para a resposta.

```
1  void deviceInteract() {
2    if (!server.hasArg("number")) {
3      server.send(400); // bad format, number required
4      return;
5      }
6
```

```
7   int nInterface = server.arg("number").toInt();
8   int nArgs = server.args() - 1;
9
10  for (int i = 1; i < nArgs+1; i++) {
11    arguments[i-1] = (uint8_t)(server.arg(i).toInt());
12  }
13
14
15  EndpointAPI_Request(nInterface, nArgs, arguments, &rsp);
16  server.send_P(200, "application/octet-stream", (char *)rsp.data, rsp.
      length);
17 }
```

3.2.2 Hardware

Tal como mostra a Fig.13 um Home Controller precisa de ter certas caraterísticas para funcionar no sistema e estas passam por: transformador de 220 volt AC para 3.3 e 5 volt DC, um micro processador, sensores e atuadores.

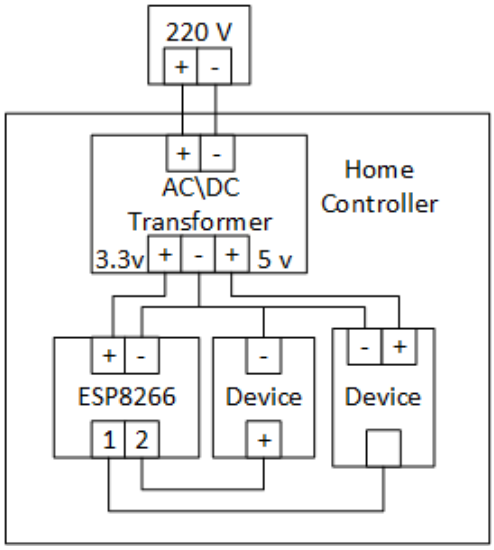


Figura 13: Esquema eléctrico de uma placa Home Controller.

Micro Controlador

O microcontrolador usado é um ESP8266 [7], pois este reúne todas as características que são precisas para criar os Home Controllers.

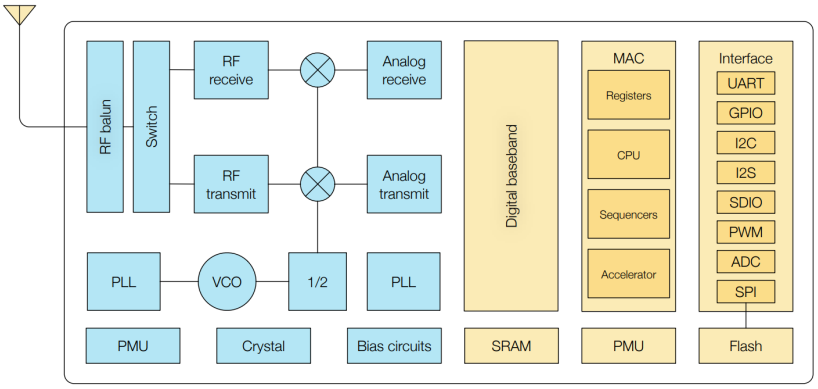


Figura 14: Diagrama de blocos funcional.

Na Fig.14 pode-se observar os constituintes da placa, mas o que merecem referencia são: a unidade central de processamento (CPU), da memoria e do radio.

CPU

O processador é um Tensilica L106 que tem uma arquitetura RISC de 32-bits. Este atinge consumos bastante baixos e tem uma capacidade máxima de clock de 160 MHz. O CPU ainda contem as seguintes interfaces: memória RAM/ROM programáveis (iBus), data RAM interface (dBus).

Memoria do CPU

A unidade de controlo de memoria (MCU) integra dois tipos: SRAM e ROM. Esta acede as memorias usando as unidades de iBus e dBus. O acesso as partes das memorias é feito a partir de pedidos e é gerenciado por um “árbitro” que arranja a sequencia de processamento de acordo

com o tempo que os pedidos chegam do processador. A memória RAM tem a capacidade de 50kB e a ROM, que é a memória programável, tem disponível 16MB.

Clock

O clock desta placa alimenta tanto o transmissor como receptor. Os cristais tem uma gama de frequencia de 24MHz a 52MHz.

Radio

O radio da placa consiste em: um transmissor e receptor de 2.4GHz, um gerador de clock de alta velocidade, um cristal oscilador e um clock de tempo real.

O receptor de 2,4 GHz converte os sinais de RF para quadratura de sinais de banda base e os converte para o domínio digital com 2 ADCs de alta resolução e alta velocidade. O ESP8266, para se adaptar às condições variáveis do canal de sinal usa filtros de RF, filtros de banda básica e os circuitos de cancelamento de compensação DC.

Já o transmissor de 2.4GHz também converte os sinais de banda de quadratura para 2,4 GHz. A função de calibração digital melhora ainda mais a linearidade do amplificador de potência, permitindo um desempenho de última geração para fornecer uma potência média de + 19,5 dBm para a transmissão 802.11b [8] e +16 dBm para a transmissão 802.11n.

Wifi

A componente Wifi da placa é capaz de implementar comunicação TCP/IP[9], o protocolo completo de 802.11 b/g/n WLAN MAC e a especificação Wi-Fi Direct. Suporta não apenas as operações do conjunto de serviços básicos (BSS) sob a função de controle distribuído (DCF), mas também a operação do grupo P2P compatível com o mais recente protocolo Wi-Fi P2P.

Sensores e Atuadores

O sensores e atuadores são os componentes que compõem os Sockets e permitem controlar e monitorizar o ambiente que se quer. Os componentes que estão prontos para serem usados no projeto são:

- Sensor de temperatura DS18B20 [20];
- LED;
- Placa genérica do relés.

Referências

[1] Mongoose Web Site,
http://mongoosejs.com/

[2] Express Web Site,
https://expressjs.com/

[3] MongoDB Web Site,
https://www.mongodb.com/what-is-mongodb

[4] Handlebars Web Site,
https://handlebarsjs.com/

[5] Amazon Smart Home Devices Web Pag.,
https://www.amazon.com/smart-home-devices/b?ie=UTF8&node=6563140011

[6] Wikipedia Domótica,
https://pt.wikipedia.org/wiki/Dom%C3%B3tica

[7] Manual do ESP8266,
https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_-
datasheet_en.pdf

[8] IEEE 802.11b/g/n,
https://www.air802.com/files/802-11-WiFi-Wireless-Standards-and-Facts.pdf

[9] TCP/IP,
https://searchnetworking.techtarget.com/definition/TCP-IP

[10] Arquitetura MVC,
https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_-
web_app_architecture/MVC_architecture

[11] HTTP Status Code,
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[12] HTTP Métodos,
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[13] HTML Informação,
https://www.w3.org/html/

[14] CSS Informação,
https://techterms.com/definition/css

[15] JavaScript,
https://www.javascript.com/

[16] Bootstrap,
https://getbootstrap.com/

[17] jquery,
https://jquery.com/

[18] AJAX,
https://developer.mozilla.org/pt-PT/docs/Web/Guide/AJAX/Como_-
come%C3%A7ar

[19] ecmascript,
https://www.ecma-international.org/publications/standards/Ecma-262.htm

[20] DS18B20,
https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

4 Conclusão

Neste projeto abordou-se o problema que o mercado gerou a volta dos dispositivos Smart Home e ainda sugeriu-se uma solução do mesmo. Os objetivos previstos foram compridos ao implementar as duas componentes que constituem o projeto. Este trabalho foi especialmente interessante pois permitiu implementar todos os conhecimentos adquiridos no curso, criando um produto que da maneira que está pensado pode evoluir em todos os aspectos. Esta evolução poderá passar por desenvolver mecanismos de comunicação mais fortes para possibilitar mais componentes e com isso mais funcionalidades na aplicação.

A Manual de utilizador

1 Passo:

O primeiro passo para utilizar o produto passa por montar o dispositivo HomeController com os componentes que quiser.

2 Passo:

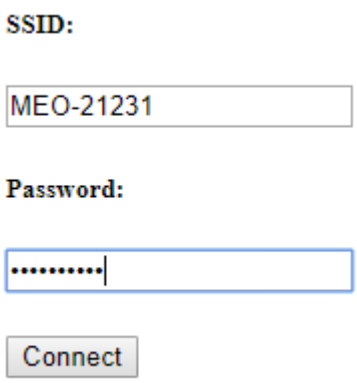
Ligar o HomeController a uma fonte de energia, de preferencisa a uma tomada com 220 Volts.

3 Passo:

Utilizar um computador ou telemóvel para se ligar à rede que o HomeController. Deverá de aparecer na lista de procura de redes como "HomeControl-001" e poderá aceder com a palavra secreta "12345qwerty".

4 Passo:

Já dentro da rede do HomeController, utilizar um navegador de internet para aceder ao URL "http://192.168.1.1/". Este irá apresentar o seguinte formulário.



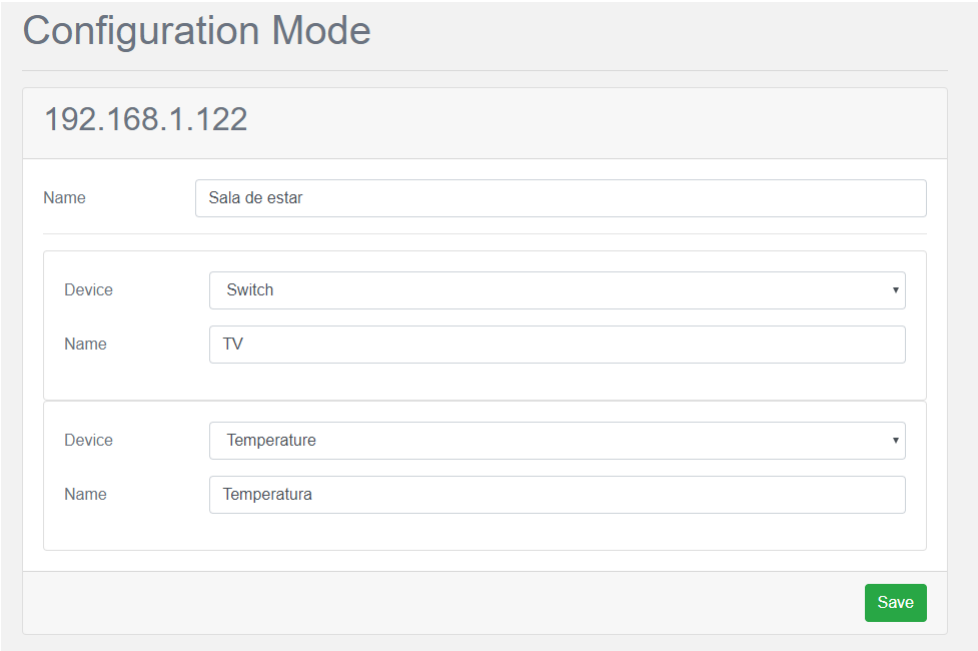
Neste formulário coloque as credenciais da sua rede de casa.

5 Passo:

Volte a ligar o seu computador ou telemóvel à rede de casa onde acabou de instalar o HomeController.

6 Passo:

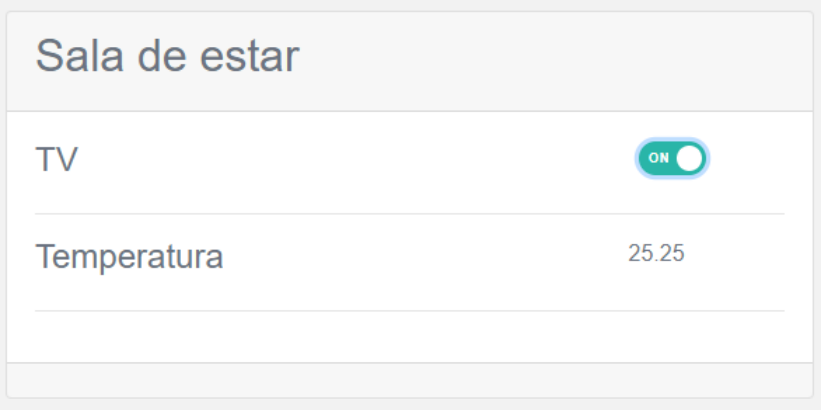
Agora entre na aplicação web e ligue-se a pagina de configuração, que deve ter ser o seguinte aspecto.



Preencha os campos de acordo com as configurações físicas que fez no 1 passo.

6 Passo:

Agora aceda ao página de utilizador, que deverá ter o seguinte aspecto.



B Codigo do HomeController

```
#include <EndpointAPI.h>
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>

// Temperature Sensor
#include <OneWire.h>
#include <DallasTemperature.h>

OneWire * oneWire;
DallasTemperature * sensors;

#define OPERATION_READ 0
#define OPERATION_WRITE 1

#define DEVICE_ID_NONE 0
#define DEVICE_ID_LIGHT_SENSOR 1
#define DEVICE_ID_TEMPERATURE 2
#define DEVICE_ID_RELAY 3

#define STATE_HEARTBEATING 0
#define STATE_SERVER_RUNNING 1

#define APPLICATION_IDENTIFIER 0xca

ESP8266WebServer server(80);
uint8_t arguments[64];
```

```
29 uint8_t data[10]; // generic buffer, use as you wish
30 EndpointResponse rsp;
31 String json;
32
33 int argSize = 0;
34
35 const char *ssid = "HomeControl-001";
36 const char *password = "12345qwerty";
37
38 WiFiUDP udp;
39 IPAddress IP_Remote(192, 168, 1, 255);
40 byte stateMachine = 0;
41
42 uint8_t applicationIdentifier[7];
43
44 //------------------------------------------------------------------------
45 //------------------------------ DRIVERS ---------------------------------
46 //------------------------------------------------------------------------
47
48 void noneInit(void * slot) {
49 }
50
51 void noneDriver(void * slot, int paramSize, uint8_t * params,
       EndpointResponse * response) {
52   response->length = 0;
53 }
54
55 void relayDriverInit(void * slot) {
56   Serial.println("Relay Driver Init Call");
57   struct Slot * caller = (struct Slot *) slot;
```

```
58   int relayPin = -1;
59
60   for (int i = 0; i < caller->pinsSize; i++) {
61     if ((caller->Pins[i]->TypeFlags & PIN_TYPE_GPIO) == PIN_TYPE_GPIO) {
62       relayPin = caller->Pins[i]->Number;
63       break;
64     }
65   }
66   if (relayPin == -1) {
67     return;
68   }
69
70   pinMode(relayPin, OUTPUT);
71 }
72
73 void relayDriver(void * slot, int paramSize, uint8_t * params,
       EndpointResponse * response) {
74   Serial.println("Relay Driver Call");
75   struct Slot * caller = (struct Slot *) slot;
76   int relayPin = -1;
77
78   for (int i = 0; i < caller->pinsSize; i++) {
79     if ((caller->Pins[i]->TypeFlags & PIN_TYPE_GPIO) == PIN_TYPE_GPIO) {
80       relayPin = caller->Pins[i]->Number;
81       break;
82     }
83   }
84   if (relayPin == -1) {
85     return;
86     }
```

```
87
88    digitalWrite(relayPin, params[0]);
89    response->length = 0;
90    return;
91  }
92  //////////////////////
93
94  void temperatureDriverInit(void * slot) {
95    struct Slot * caller = (struct Slot *) slot;
96    int pin = -1;
97
98    for (int i = 0; i < caller->pinsSize; i++) {
99      if ((caller->Pins[i]->TypeFlags & PIN_TYPE_GPIO) == PIN_TYPE_GPIO) {
100       pin = caller->Pins[i]->Number;
101       break;
102     }
103   }
104   if (pin == -1) {
105     return;
106   }
107   oneWire = new OneWire(pin);
108   sensors = new DallasTemperature(oneWire);
109   sensors->begin();
110 }
111
112 void temperatureDriver(void * slot, int paramSize, uint8_t * params,
      EndpointResponse * response) {
113   Serial.println("Temp. Driver Call");
114   struct Slot * caller = (struct Slot *) slot;
115
```

```
  sensors->requestTemperatures();
  float reading = sensors->getTempCByIndex(0);

  Serial.println(reading);
  response->length = sizeof(float);
  memcpy(response->data, &reading, sizeof(float));
}

//------------------------------------------------------------------------
//------------------------------ HANDLERS --------------------------------
//------------------------------------------------------------------------

void deviceInteract() {
  if (!server.hasArg("number")) {
    server.send(400); // bad format, number required
    return;
  }
  int nInterface = server.arg("number").toInt();

  int nArgs = server.args() - 1;

  for (int i = 1; i < nArgs+1; i++) {
    arguments[i-1] = (uint8_t)(server.arg(i).toInt());
  }

  Serial.println("Calling interface");
  EndpointAPI_Request(nInterface, nArgs, arguments, &rsp);
  Serial.println("Interface Called");
  Serial.print("Rsp data length:");
  Serial.println(rsp.length, DEC);
```

```
146   server.send_P(200, "application/octet-stream", (char *)rsp.data, rsp.
        length);
147 }
148
149 void changeCurrentDevice() {
150 //url: POST /interface/change  | params: interfaceNumber=1,newDevice=2
151   int interfaceNumber = server.arg("interfaceNumber").toInt();
152
153   if (interfaceNumber == 0) {
154     server.send(400);
155     return;
156   }
157
158   int newDevice = server.arg("newDevice").toInt();
159
160   if (newDevice == 0) {
161     server.send(400);
162     return;
163   }
164
165   uint8_t data[4];
166   data[0] = CurrentDevice;
167   data[1] = 2; // GET = 1, POST = 2
168   data[2] = interfaceNumber;
169   data[3] = newDevice;
170
171   EndpointAPI_Request(0, 4, data, &rsp);
172   if (rsp.data[0] == 0) {
173     server.send(200);
174   }
```

```
175   else {
176     server.send(500);
177   }
178 }
179 
180 void getInterfaces() {
181   uint8_t numberOfDevices;
182 
183   data[0] = NumberOfSlots;
184 
185   EndpointAPI_Request(0, 1, data, &rsp);
186   numberOfDevices = rsp.data[0];
187 
188 /*
189 * [
190 * {
191 *   "id":1,
192 *   "SupportedDevices":[1, 2, 5],
193 *       "CurrentDevice":1
194 *   }
195 * ]
196 */
197 
198   json = "[";
199 
200   for (uint8_t i = 0; i < numberOfDevices; i++) {
201     json += "{\"id\":\"";
202     json += i + 1;
203     json += "\", \"SupportedDevices\":[";
204 
```

```
205     data[0] = GetSupportedDevices;
206     data[1] = i+1;
207     EndpointAPI_Request(0, 1, data, &rsp);
208
209     Serial.println(i + 1, DEC);
210
211     for (int j = 0; j < rsp.length; j++) {
212       json += "\"";
213       json += (int)rsp.data[j];
214       json += "\"";
215       if (j < rsp.length - 1) {
216         json += ",";
217       }
218     }
219
220     json += "]";
221
222     data[0] = CurrentDevice;
223     data[1] = 1;
224     data[2] = i+1;
225
226     EndpointAPI_Request(0, 3, data, &rsp);
227
228     json += ", \"CurrentDevice\":\"";
229     json += (int)rsp.data[0];
230     json += "\"";
231     json += "}";
232
233     if (i < numberOfDevices - 1) {
234       json += ",";
```

```
235     }
236   }
237   json += "]";
238   server.send(200, "text/plain", json);
239 }
240
241 void setup() {
242   Serial.begin(9600);
243
244   EndpointAPI_Init();
245
246   EndpointAPI_ConfigPin(0, PIN_TYPE_GPIO);
247   EndpointAPI_ConfigPin(2, PIN_TYPE_GPIO);
248   EndpointAPI_ConfigPin(12, PIN_TYPE_GPIO);
249   EndpointAPI_ConfigPin(13, PIN_TYPE_GPIO);
250   EndpointAPI_ConfigPin(14, PIN_TYPE_GPIO);
251   EndpointAPI_ConfigPin(15, PIN_TYPE_GPIO);
252   EndpointAPI_ConfigPin(18, PIN_TYPE_GPIO);
253
254   EndpointAPI_NewDevice(DEVICE_ID_NONE, noneInit, noneDriver);
255   EndpointAPI_NewDevice(DEVICE_ID_RELAY, relayDriverInit, relayDriver);
256   EndpointAPI_NewDevice(DEVICE_ID_TEMPERATURE, temperatureDriverInit,
        temperatureDriver);
257
258   int temp[3] = {2, 0};
259   int tempDevices[2] = {DEVICE_ID_RELAY, DEVICE_ID_NONE};
260   EndpointAPI_NewSlot(1, temp, 2, tempDevices);
261
262   temp[0] = 0;
263   tempDevices[0] = DEVICE_ID_RELAY;
```

```
264  tempDevices[1] = DEVICE_ID_TEMPERATURE;
265  tempDevices[2] = DEVICE_ID_NONE;
266
267  EndpointAPI_NewSlot(1, temp, 3, tempDevices);
268  EndpointAPI_ChangeCurrentDevice(1, DEVICE_ID_RELAY);
269
270  WiFi.macAddress(applicationIdentifier);
271  applicationIdentifier[6] = APPLICATION_IDENTIFIER & 0xff;
272
273  WiFi.begin("MEO-341E51", "5A6CDC765F"); // ze
274  //WiFi.begin("Vodafone-B22EE0", "vung6UJq3J"); // gui
275  //WiFi.begin("MEO-179F14", "14753325"); // caio
276
277  //TODO: Only a master can call this: regulate access
278  server.on("/config/master-sync", handleAlive);
279  server.on("/board/config", getInterfaces);
280  server.on("/board/interface/device/change", changeCurrentDevice);
281  server.on("/board/interface/device/call", deviceInteract);
282
283  while (WiFi.status() != WL_CONNECTED) {
284    delay(500);
285    Serial.print(".");
286  }
287
288  server.begin();
289  Serial.println("\nConnected");
290  Serial.println(WiFi.localIP());
291
292  stateMachine = STATE_HEARTBEATING;
293  //Abrir o canal de broadcast
```

```
294   udp.begin(3333);
295 }
296
297 //--------------------------------------------------------------
298 //----------------------Communication---------------------------
299 //--------------------------------------------------------------
300
301 void heartbeat() {
302   udp.beginPacket(IP_Remote, 3333);
303   udp.write(applicationIdentifier, 7);
304   Serial.println("Heartbeat");
305   udp.endPacket();
306 }
307
308 void handleAlive() {
309 //master calls this path to notify that he knows this slave exists
310   server.send(200, "text/plain", "OK");
311
312 stateMachine = STATE_SERVER_RUNNING;
313 }
314
315 void handleRoot() {
316   if (server.method() == HTTP_GET) {
317     server.send(200, "text/html", "<form action=\"/config/change-network
      \" method = \"POST\" >\nNetwork Name:<br> <input type=\"text\" name
      =\"networkName\"><br> \n Password:<br><input type=\"text\" name=\"
      password\"><br><br> \n <input type=\"submit\" value=\"Submit\">\n</
      form> ");
318   }
319   else if (server.method() == HTTP_POST) {
```

```
320     server.send(400, "text/html", "<h1>Not valid!!</h1>");
321   }
322 }
323
324 void accessPoint() {
325 /* You can remove the password parameter if you want the AP to be open.
      */
326   WiFi.softAP(ssid, password);
327   IPAddress myIP = WiFi.softAPIP();
328
329   Serial.print("AP IP address: ");
330   Serial.println(myIP);
331 }
332
333 void APToSTA() {
334   char* newSSID = (char *)malloc(50);
335   String tempSSID = server.arg("networkName");
336   char* newPassword  = (char *)malloc(50);
337   String tempPass = server.arg("password");
338
339   tempSSID.toCharArray(newSSID, tempSSID.length());
340   tempPass.toCharArray(newPassword, tempPass.length());
341
342   Serial.println(newSSID);
343   Serial.println("\n");
344   server.send(200, "text/plain", "Trying to Connect!");
345
346   WiFi.persistent(false);                 // Turn off persistent to fix
      flash crashing issue.
347   //WiFi.mode(WIFI_OFF);
```

```
348   WiFi.mode(WIFI_STA);
349   WiFi.begin(newSSID, newPassword);
350
351   while (WiFi.status() != WL_CONNECTED) {
352     delay(500);
353     Serial.println(".");
354   }
355
356   Serial.println("\nConnected");
357   Serial.println(WiFi.localIP());
358
359   free(newSSID);
360   free(newPassword);
361 }
362
363 //---------------------------------------------------------------
364 //----------------------Test Space-------------------------------
365 //---------------------------------------------------------------
366 void loop() {
367   if (stateMachine == STATE_HEARTBEATING) {
368     heartbeat();
369     delay(1000);
370   }
371   server.handleClient();
372 }
```

C Código da Aplicação Web

C.1 Ficherio WWW

```
/**
* Module dependencies.
*/

var app = require('../app');
var debug = require('debug')('server:server');
var http = require('http');
var dgram = require('dgram');
var request = require('request');
var mongoose = require('mongoose');
const chalk = require('chalk');

var HomeController  = require('../models').HomeController;
var broadcast = dgram.createSocket("udp4");

/**
* Get port from environment and store in Express.
*/

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
* Create HTTP server.
*/

var server = http.createServer(app);

mongoose.connect("mongodb://Application:qwerty54321@ds161790.mlab.com
    :61790/guilab-project", function (error) {
```

```
30   if(error){
31     console.log(chalk.red("Error connecting with database, exiting
           application."));
32     return process.exit(1);
33   }
34   console.log(chalk.green("Connected to database, starting express
         server..."));
35   server.listen(port);
36   server.on('error', onError);
37   server.on('listening', onListening);
38   console.log(chalk.green("Express server started. Starting broadcast
         listener."));
39   broadcast.bind(3333);
40 });
41
42 broadcast.on("listening", function () {
43   broadcast.setBroadcast(true);
44   console.log("Broadcast server started.");
45 });
46
47 broadcast.on("message", function (msg, rinfo) {
48   var mac = byteArrayToMAC(msg);
49   var ip =rinfo.address;
50
51   request({method:'GET', url:'http://'+ip+'/config/master-sync'},
         function (err, rsp, body) {
52     request('http://'+ip+'/board/config', function (err, rsp, body) {
53       var slave = JSON.parse(body);
54
55       HomeController.findOne({}, function (err, hCtrl) {
```

```
56        if(err){
57          console.log(err);
58          return;
59        }
60        if(!hCtrl){
61          var temp = {};
62          temp.IP = ip;
63          temp.Name = '';
64          temp.Interfaces = [];
65
66          for(var i in slave){
67            var interface = {};
68            interface.Name = '';
69            interface.Supported = [];
70            interface.Current = slave[i].CurrentDevice;
71
72            for(var supp in slave[i].SupportedDevices){
73              interface.Supported.push(slave[i].SupportedDevices[supp]);
74            }
75
76            temp.Interfaces.push(interface);
77          }
78          var c = new HomeController(temp);
79          c.save();
80        }
81      });
82    });
83  });
84});
85
```

```
function byteArrayToMAC(array){
  var result = "";
  for(var i = 0; i < 6; i++){
    result += array[i].toString(16);
    if(i < 5){
      result+='-';
    }
  }
  return result;
}

function byteArrayToMAC(array){
  var result = "";
  for(var i = 0; i < 6; i++){
    result += array[i].toString(16);
    if(i < 5){
      result+='-';
    }
  }
  return result;
}

function normalizePort(val) {
  var port = parseInt(val, 10);

  if (isNaN(port)) {
    // named pipe
    return val;
  }
  if (port >= 0) {
```

```
116     // port number
117     return port;
118   }
119   return false;
120 }
121
122 /**
123 * Event listener for HTTP server "error" event.
124 */
125 function onError(error) {
126   if (error.syscall !== 'listen') {
127     throw error;
128   }
129
130   var bind = typeof port === 'string'
131   ? 'Pipe ' + port
132   : 'Port ' + port;
133
134   // handle specific listen errors with friendly messages
135   switch (error.code) {
136     case 'EACCES':
137       console.error(bind + ' requires elevated privileges');
138       process.exit(1);
139       break;
140     case 'EADDRINUSE':
141       console.error(bind + ' is already in use');
142       process.exit(1);
143       break;
144     default:
145       throw error;
```

```
146   }
147 }
148
149 /**
150 * Event listener for HTTP server "listening" event.
151 */
152 function onListening() {
153   var addr = server.address();
154   var bind = typeof addr === 'string'
155   ? 'pipe ' + addr
156   : 'port ' + addr.port;
157   debug('Listening on ' + bind);
158 }
```

C.2 Ficherio controller

```
1  var express = require('express');
2  var router = express.Router();
3
4  var HomeController  = require('../models').HomeController;
5  var request = require('request');
6
7  //Config route
8  router.post('/update', function(req, res, next) {
9    var controller = req.body.controller;
10
11   var temp = false;
12   HomeController.findById(controller.id, function (err, c) {
13     c.Name = controller.name;
14     for (var d in controller.devices){
15       for (var cd=0; cd < c.Interfaces.length; cd++){
```

```
16          if(controller.devices[d].id == c.Interfaces[cd]._id.toString()){
17            if(c.Interfaces[cd].Current != controller.devices[d].current){
18              c.Interfaces[cd].Current = Number(controller.devices[d].
                    current);
19              var iNumber = Number(cd)+1;
20              request({method:'GET', url:'http://'+c.IP+'/board/interface/
                    device/change?interfaceNumber='+iNumber+'&newDevice='+
                    controller.devices[cd].current}, function (err, rsp,
                    body){
21                console.log(body);
22              });
23            }
24
25            c.Interfaces[cd].Name = controller.devices[d].name;
26            temp = true;
27            break;
28          }
29        }
30      }
31      c.save();
32    });
33  });
34
35  router.get('/:controllerID/device/:deviceNumber/execute', function (req,
        res, next) {
36
37    HomeController.findById(req.params.controllerID, function (err, c) {
38      if(err){
39        return res.json({status: "Error accessing DB"});
40      }
```

```
41      if(c){
42        var iNumber = Number(req.params.deviceNumber) + 1;
43        request({method:'GET', encoding:null ,url:'http://'+c.IP+'/board/
              interface/device/call?number='+iNumber+''}, function (err, rsp
              , body){
44          var bytes = new Buffer.from(body);
45          var num = bytes.readFloatLE();
46
47          res.json({status: "ok", value: num});
48        });
49      }
50      else{
51        res.json({status: "error"});
52      }
53    })
54  });
55
56  module.exports = router;
```

C.3 Ficherio index

```
1  var express = require('express');
2  var router = express.Router();
3
4  var HomeController  = require('../models').HomeController;
5
6  //Main user route
7  router.get('/', function(req, res, next) {
8    HomeController.find({}, function (err, ctrls) {
9      res.render('index', { controllers: ctrls });
10   });
```

```
});

//Config route
router.get('/config', function(req, res, next) {
  HomeController.find({}, function (err, ctrls) {
    for(var c in ctrls){
      for(var i in ctrls[c]._doc.Interfaces){
        for(var t in ctrls[c]._doc.Interfaces[i].Supported){
          var temp =  ctrls[c]._doc.Interfaces[i].Supported[t];
          ctrls[c]._doc.Interfaces[i].Supported[t] = {};
          ctrls[c]._doc.Interfaces[i].Supported[t].Value = temp;

          var name = 'None';

          if(temp == 1){
            name = 'Light Sensor'
          }
          else if(temp == 2){
            name = 'Temperature'
          }
          else if(temp == 3){
            name = 'Switch'
          }

          ctrls[c]._doc.Interfaces[i].Supported[t].Name = name;
        }
      }
    }
    res.render('config', { controllers:ctrls});
  })
```

```
41 });
42
43 module.exports = router;
```

C.4 Ficherio users

```
1 var express = require('express');
2 var router = express.Router();
3
4 /* GET users listing. */
5 router.get('/', function(req, res, next) {
6   res.send('respond with a resource');
7 });
8
9 module.exports = router;
```

C.5 Ficherio config

```
1 <script>
2   function saveDevice(id) {
3     var c = {};
4     c.id = id;
5     c.ip = $('#'+id+' .IP').text();
6     c.name = $('#'+id+' .controller-name').val();
7     c.devices = [];
8     $.each( $('#'+id+' .interface-card'), function (i, val) {
9       var temp = {}
10      temp.name =  $(val).find(".interface-name").val()
11      temp.current = $(val).find(".option-supported").val()
12
13      temp.id =  $(val).attr("id");
```

```
14        //console.log(temp);
15        c.devices.push(temp);
16      })
17    //console.log(c);
18
19      $.ajax({
20        url: '/controllers/update',
21        dataType: 'json',
22        type: 'post',
23        contentType: 'application/json',
24        data: JSON.stringify( { "controller": c } ),
25        processData: false,
26        success: function( data, textStatus, jQxhr ){
27        },
28        error: function( jqXhr, textStatus, errorThrown ){
29          console.log( errorThrown );
30        }
31      });
32    }
33  </script>
```

C.6 Ficherio index

```
1   <script>
2     $('.temperature-reading').each(function (item, data) {
3       setInterval(function(){reqTemperature($(data).attr('data-
            controllerID'), $(data).attr('data-deviceNumber'))}, 2000);
4     });
5
6     function reqTemperature(controllerID, deviceNumber){
7       $.ajax({
```

```
8       url: '/controllers/'+controllerID+'/device/'+deviceNumber+'/
            execute',
9       dataType: 'json',
10      type: 'get',
11      contentType: 'application/json',
12      processData: false,
13      success: function( data, textStatus, jQxhr ){
14        $('#controller-'+controllerID+'   .temperature-reading').text(
              data.value);
15      },
16      error: function( jqXhr, textStatus, errorThrown ){
17        console.log( errorThrown );
18      }
19    });
20  }
21 </script>
```

C.7 Ficherio app

```
1  var express = require('express');
2  var path = require('path');
3  var favicon = require('serve-favicon');
4  var logger = require('morgan');
5  var cookieParser = require('cookie-parser');
6  var bodyParser = require('body-parser');
7  var lessMiddleware = require('less-middleware');
8
9  var app = express()
10 exphbs = require("express-handlebars");
11
12 app.set('views', path.join(__dirname, 'views'));
```

```
13
14 app.engine("hbs", exphbs({
15   defaultLayout: 'layout',
16   helpers: require('./helpers'),
17   extname: 'hbs',
18   layoutsDir: __dirname + '/views/layouts'
19 }));
20
21 app.set("view engine", "hbs");
22
23 app.use(lessMiddleware(__dirname + '/public'));
24 app.use(express.static(__dirname + '/public'));
25
26 var index = require('./routes/index');
27 var users = require('./routes/users');
28 var controllers = require('./routes/controllers');
29
30 // uncomment after placing your favicon in /public
31 //app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
32 app.use(logger('dev'));
33
34 app.use(bodyParser.json());
35 app.use(bodyParser.urlencoded({ extended: false }));
36 app.use(cookieParser());
37 app.use(express.static(path.join(__dirname, 'public')));
38
39 app.use('/', index);
40 app.use('/controllers', controllers);
41
42
```

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

C.8 Ficherio halpers

```
function ifCondition(v1, v2, options) {
  if(v1 === v2) {
    return options.fn(this);
  }
  return options.inverse(this);
}

module.exports = {
  ifCond : ifCondition
```

```
}
```

C.9 Ficherio models

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var HomeController = new Schema({
  IP : String,
  Name: String,
  Interfaces : [{
    Name: String,
    Supported : [Number],
    Current : Number
  }],
});

module.exports = {
  HomeController : mongoose.model('HomeControllers', HomeController)
};
```