

ecø, a Rainforest Product

Pablo Acereda García David Emanuel Craciunescu
Laura Pérez Medeiro

September 2020

1 Server

Set it up

Not much is needed, except for following the guide provided in the ArchLinux Wiki. Do not worry if you have to repeat the entire installation more than three times, because you either forgot something, something went wrong in the process or spend days trying to figure out why you can't connect to your Wi-Fi.

It took us an entire week ***just*** to achieve connecting the server to the network - without any other packages, just the basic shell, one user and the Wi-Fi connection: no UI, no text editor, no nothing else.

If you get stuck up, refer to this manual, but at least try once to install it yourself.

What is inside?

Application

The application itself is running as a daemon in the server. That means that any user can access from anywhere to the webpage and the API.

But that does not mean that is not such an easy path. First, maven and nodejs must be installed in order to even be able to run the application - front and back.

Afterwards, the daemon must be added to initialize after start up ¹. Take as an example:

```
1 [Unit]
2 Description=Spring Boot Application for Eco Project. Launches the
   API and the analytical threads.
3 After=auditd.service systemd-user-sessions.service time-sync.target
4
5 [Service]
6 User=root
7 TimeoutStartSec=0
```

¹the daemons from the database should also be implemented, otherwise the services will not have data to access to.

```

8 | Type=simple
9 | KillMode=process
10 | WorkingDirectory=/UBICUA/webpage/eco-webpage
11 | ExecStart=mvn spring-boot:run
12 | Restart=always
13 | RestartSec=2
14 | LimitNOFILE=5555
15 |
16 | [Install]
17 | WantedBy=multi-user.target

```

DDNS

Having a server is great, but having a server that gives service to all the internet (knowing its location) can be even better. Specially if it is an application what we are talking.

But in order to do that, it is not enough to *just* create the server. It necessary to make it available to the internet. To achieve that, first, some ports need to redirect request to the server. Because the server is not the main node of the LAN, that is the router, this last one must take the requests receive by external endpoints and redirect them.

Normally, it just redirects the responses to requests made by an internal the end-point, the server. That is because the router is the only one that nodes the internal directions of its *children*. But, it can also be configured to redirect the requests that get to a specific port - router port, namely P_e , to a port a specific machine and port, internal port or server port, namely P_i .

Those two ports, P_e and P_i do not need to be the same. In fact, it is advisable for them to be different, for security reasons.

To redirect the ports, go to your router IP, and your telcom provider should have a page where you can create those redirections [1].

Editor

Nombre	Protocolo	Puerto/Rango Externo	Puerto/Rango Interno	Direccion IP	Activar
ssh-arch	TCP	22	22	192.168.1.1	ON
mongo-repl-1	TCP	27018	27018	192.168.1.1	ON
mongo-repl-2	TCP	27019	27019	192.168.1.1	ON
mongo-repl-3	TCP	27020	27020	192.168.1.1	ON

Figure 1: Port Redirection Example

Now, the server is *known* to the network, but the network still does not *know* the *name full name* of the server. To give a server a *name* a DDNS service can be used. It is not on the scope of this documentation explaining DNS protocol. So cutting to the chase, DuckDNS does the trick.

What it does it, it requests the router its IP address from a device in the LAN. Then, it sends the IP to AWS and assigns that IP to a domain. It usually uses *myDomainName.duckdns.org*. That way, the server, is know by the net².

```
1 echo url="https://www.duckdns.org/update?domains=<domain>&token=<token>&ip=" | curl -k -o /duckdns/duck.log -K -
```

It comes in specially handy taking that now the application itself does not work with dynamic IPs that change every now and then (and therefore nothing works). Now, the application/database/user *shout* a requests to *domain*, and that domain solves the requests.

To install it, follow these steps. Instead of launching it manually, use the *CRON* service. For Arch users, we recommend using *CRONIE*.

We recommend keeping the crontab file generated with the periods of execution, in case you want to change them. Also, to include an execution after reboot would be advisable, better safe than sorry³.

```
1 # Moment Who Command
2 @reboot /duckdns/duck.sh
3
4 # Minute Hour Day of Month Month Day of Week
5 # (0-59) (0-23) (1-31) (1-12 or Jan-Dec) (0-6 or Sun-Sat)
6 */5 * * * * /duckns/duck.sh >/dev/null 2>&1
```

MongoDB

All to know about MongoDB and how it works is in this same documentation, under the section *database*.

Regarding its installation, ArchLinux got it removed from its official repositories, due to some issues with its License. But still, MongoDB can be downloaded from the official Ubuntu repositories.

It is also highly advisable to create a service for MongoDB so it automatically get launched after reboot.

Mosquitto

All the transcendent information regarding this broker, has already been explained in the *device* section. That being said, it is obviously necessary to have somewhere to run it into.

Can download it using **pacman**: `mosquitto`.

²To be more precise, the router is known, and it is the router the one that redirects the requests

³Depending on the terminal being used, the behavior might get affected. Each terminal (*bash*, *zsh*, etc.) has different redirection mechanisms.

It is recommended to launch the process when the server is initiated. To do that, do as before and create a `systemd service`.

Vim

According to archlinux wiki, Vim is a “*terminal text editor. It is an extended version of vi with additional features, including syntax highlighting, a comprehensive help system, native scripting (vimscript), a visual mode for text selection, comparison of files (vimdiff), and tools with restricted capabilities such as rview and rvim.*”

What for some is a nightmare, impossible to understand; for us is the perfect tool to develop/create/maintain content/code directly on the server.

It obviously needs to be supported by some other tools, like maven or nodejs⁴

Zsh

zsh is a shell that operates as interactive shell and as a scripting language interpreter. It is compatible with POSIX (although, not by default) it offers advantages such as improved tab completion and globbing.

It is a substitute for the default ArchLinux shell, bash. The avid reader has probably noticed the aforementioned advantages that *Zsh* has. Those reasons are more than enough to pivot to his shell.

Also, some plugins like OhMyZsh⁵ can be included to improve the user experience (if you are ever considered to have any with *just* a shell, UI - we do).

2 Device

As a first option, ESP32 was chosen as the hardware device to develop in this project but due to some technical problem this has change in the last moment. That is the reason why it can be found information about ESP32 or Arduino UNO.

Getting started with ESP32

In order to start using an ESP32 with Arduino IDE, it is necessary to have the Arduino IDE and the Git GUI for getting the library ESPRESSIF.

Expressif is used to have support on Windows 10 machines for the ESP devices, to install it must be followed steps:

1. Start Git GUI and click on clone existing repository.
2. Select `https://github.com/espressif/arduino-esp32.git` as source location and as a target directory `[ARDUINO_DIR]/hardware/espressif/esp32` where

⁴If it gives any error while trying to install with pacman, try to download the binaries directly - remember to change the version.

⁵On GitHub: `https://github.com/ohmyzsh/ohmyzsh/`.

ARDUINO_DIR is where Arduino IDE has been installed. Usually it is located inside /Documents/Arduino. Also select the option which said recursively clone submodules.

3. When the installations has finished, open a GitBash session pointing to the target directory and execute git sub module update -init -recursive
4. Open the target directory and double-click the get.exe

Then, plug the device and wait until the drivers are installed, start the Arduino IDE and select Tools > Board > ESP32 Arduino > ESP32 Dev Module. And check if the port is attached to.

Another option is with Arduino 1.8 or higher:

1. Click inside File > Preference menu
2. In the setting tab there is a placeholder that said 'Additional Boards Manager URLs', paste there the following link
3. Click OK button and save the setting
4. Click on Tools > Boards > Boards Manager and search esp32 to install the module "esp32 by Espressif Systems"

Take into account that before uploading the code, it must be chosen the corresponding board.

Troubleshooting - Booting up ESP32

It is a typical mistake that happens when it fail to connect the board. This error said something like '***A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header***'. That means that the device is not in flashing/updating mode and it can be fixed:

1. Make sure that the proper COM port is selected, upload speed to 115200 or lower and the correct board is chosen
2. Hold on the BOOT button
3. Press upload in the IDE
4. When the message

```
1 Connecting...
2 Writing at 0x00001000 (100 %)
```

release the finger from the BOOT button

5. Then, the message ***Done uploading*** must be shown

Libraries

The libraries finally used in this project were:

- **DHT** : This library was developed by Adafruit in order to control their DHT11 or DHT22 humidity and temperature sensor easily. That library is compatible with all architectures, so it can be used in all Arduino and ESP boards. Inside the code, it must be define the pin where the sensor is located and the DHTTYPE, also it is necessary to used a constructor with the structure `DHT <nameSensor> (pin, DHTTYPE)` and then, inside the setup function used the `.begin()`. With this library the temperature can be taken in Celsius or in Farenheins (using true as a paramenter inside the `readTemperature` function).
- **SimpleTimer** : It is a simple library to launch time actions. It is based on the idea of cexecuting a particular piece of code every milliseconds indicates without using interrupts. It must be taken into account that it uses polling so it can not be guarantee when a callback is fired and also, that the the callbacks functions used inside the functions must be declares as void type. The functions available inside this library are:
 - `int setInterval (long d, timer_callback f)`. Call function f every d milliseconds.
 - `int setTimeout(long d, timer_callback f)`. After f has been called, the interval is deleted, therefore the value `timerId` is no longer valid.
 - `boolean isEnabled(int timerId)`. Return if the timer is enabled
 - `void enable(int timerId) / void disable (int timerId)`: enable or disable the specified timer
 - `void toggle(int timerId)`. Enables the specified timer if it's currently disabled, and vice-versa.
 - `void restartTimer(int timerId)`.
 - `void deleteTimer(int timerId)`.
 - `void getNumTimers()`. Returns the number of used slots in a timer object.
- **ArduinoJson** : In order to send data with JSON format it was selected this library as it has an extended documentation with useful examples. Disclaimer, the version 5 and version 6 of this library has huge differences so be cautious choosing the version.This project use version 6.

But when the ESP was used, there were also WiFi and MQTT library as we had WiFi in that device. The libraries for that purpose where:

- **WiFi**. It has a similar procedure that the ESP8266 but with a more generic name. This library allows to use a begin method that given the ssid and password initialize the network connection. After that, using

a loop it can be controlled that WiFi status and wait until the status is WL_CONNECTED so that is the moment where the device is successfully connect. This library is included in the espressif repository downloaded previously. But it can also be downloaded inside the library manager or here

- PubSubClient. It is library to manage with MQTT protocol with which publish and subscribe messaging are supported. Disclaimer, this library is not fully compatible with the ESP32 and can present problems from time to time. It can subscribe at QoS 0 or QoS 1, but default, the maximum message size is 256 bytes, 15 seconds for the keepalive interval and MQTT 3.1.1. All this parameter can be modified inside PubSubClient.h file.

Mosquitto MQTT as a broker

For storing the data inside a database, it was decided to used a broker which activate an script that insert the data. Mosquitto was the first decision as it is open source and is written in Python and can run on practically any system.

In order to install mosquitto in our server, which has Arch as an operative system the following steps:

1. Download the most recent archive. `wget http://mosquitto.org/files/source/mosquitto-1.6.9.tar.gz`
2. Unpack the archive, enter inside the new directory, build and install the package.

```
1 tar xzf mosquitto-1.6.9.tar.gz
2 cd mosquitto-1.6.9.tar.gz
3 make
4 sudo make install
5
```

3. Then remove the leftover installation files and the package archive.

```
1 sudo rm -Rd mosquitto-1.6.9
2 sudo rm mosquitto-1.6.9.tar.gz
3
```

4. The following steps consist on configure mosquitto, in order to do that it is necessary to create a system account and group, then copy configuration file from the example directory and add `/var/run/mosquitto.pid` to the pid_file line.

```

1  sudo useradd -r -s /bin/false mosquitto
2  sudo cp /etc/mosquitto/mosquitto.conf.example
3  /etc/mosquitto/mosquitto.conf
4  sudo vim /etc/mosquitto/mosquitto.conf
5

```

5. Then create a systemd script with the following information:

```

1  sudo vim /etc/systemd/system/mosquitto.service
2
3  [Unit]
4  Description=Mosquitto MQTT Broker daemon
5  ConditionPathExists=/etc/mosquitto/mosquitto.conf
6  Requires=network.target
7
8  [Service]
9  User=mosquitto
10 Group=mosquitto
11 Type=simple
12 ExecStartPre=/usr/bin/rm -f /var/run/mosquitto.pid
13 ExecStart=/usr/local/sbin/mosquitto -c
14 /etc/mosquitto/mosquitto.conf -d
15 ExecReload=/bin/kill -HUP $MAINPID
16 PIDFile=/run/mosquitto.pid
17 Restart=on-failure
18
19 [Install]
20 WantedBy=multi-user.target
21
22

```

6. A common error that can be found using mosquitto is *is mosquitto_pub: error while loading shared libraries: libmosquitto.so.1: cannot open shared object file: No such file or directory*. In order to solve it, the following steps must be done:

```

1  vim /etc/ld.so.conf
2
3  Add inside the .conf file:
4  include ld.so.conf.d/*.conf
5  include /usr/local/lib
6  /usr/lib
7  /usr/local/lib
8

```

Then execute `/sbin/ldconfig` and create a soft link `ln -s /usr/local/lib/libmosquitto.so.1 /usr/lib/libmosquitto.so.1`

7. Finally run mosquitto, start the systemd service and enable the systemd service to run on boot `sudo systemctl enable mosquitto`

Using this way of communication was when the ESP start to overheat up to a point where the ESP return

```
1      a fatal error occurred: Failed to connect to ESP32: Timed
2      out...}
```

Doing a reset, changing the COM port and the Serial frequency was some of the attempts done to recover the utility of the device. But then a new error appears:

```
1      C:/msys32/mingw32/lib/python2.7/site-packages/serial/
2      serialwin32.py", line 62, in open
3      raise SerialException("could not open port {!r}: {!r}".
4      format(self.portstr, ctypes.WinError()))
      serial.serialutil.SerialException: could not open port
      '/??/COM4': Windows Error(3, 'The system cannot find the
      path specified.')
```

At this point it was tried to fix it, as it is mentioned here but without any result.

Also, it was tried to use another ESP32 that it was bought previously and which was used to take measurements from other plants and the actual problem was that the COM port was no longer available. As it is mentioned in this tutorial, the CP2012 driver was downloaded without any results and it was checked with different cables which has data wires but anything worked.

Smartnest

Smartnest is a web services used in this project to connect Alexa to the ESP using MQTT. This services provides a web page where the ESP can be controlled using a button or with voice command if it connect to assistant such as Alexa. But it also gives the opportunity to use other assistant such as Siri or other services such as IFTT or Espurna firmware.

Alexa integration

First, open Alexa application and download the skill. Once it is downloaded, it must be enable and before diving your username and password the app is completely downloaded. After the first link, Alexa will start discovering devices and she will get all the devices you have created in your account. Alexa will show you a confirmation of how many devices she found and then you can configure them, add them to your rooms or routines.

More information about this project can be found in his official documentation

ESP32 code

In the code, it can be found the setup and loop functions and also `WiFiConnection()` for connecting the device to the WiFi which details has been indicated as a global values (SSID and password), `initMQTT()` which starts the MQTT protocol in order to connect with smartnest given the broker, port, username, password and id client (also define as global variables), `checkMqtt()` check if MQTT service has been initialized and if not it started, `splitTopic` (`char* topic`, `char* tokens[]`, `int tokensNumber`) given a topic returns an array with the element inside that topic, `callback` (`char* topic`, `byte* payload`, `unsigned int length`) where the information is notified to the server and `measurements()` which returns the values of the different sensors.

```
1  /* ----- */
2  /*      L I B R A R I E S      */
3  /* ----- */
4  #include <WiFi.h>
5  #include <PubSubClient.h>
6  #include <WiFiClient.h>
7
8  #include "DHT.h"
9
10 #ifdef __cplusplus
11 extern "C" {
12 #endif
13 uint8_t temprature_sens_read();
14 #ifdef __cplusplus
15 }
16 #endif
17 uint8_t temprature_sens_read();
18
19 /* ----- */
20 /*      W I F I      D A T A      */
21 /* ----- */
22 #define SSID_NAME "*****" // Wifi Network name
23 #define SSID_PASSWORD "*****" // Wifi network password
24
25 /* ----- */
26 /*      S M A R T N E S T
27 *      M Q T T B R O K E R
28 *      D A T A      */
29 /* ----- */
30 #define MQTT_BROKER "smartnest.cz" //
31 #define MQTT_PORT 1883 //
32 #define MQTT_USERNAME "l.perezm" //
33 #define MQTT_PASSWORD "Ubicua2019" //
34 #define MQTT_CLIENT "5e0f5b6f4ff3c4137d21ef73" // Info given by
35     smartnest.cz/mynestV2
36
37 /* ----- */
```

```

38  /*      V A R I A B L E S      */
39  /* ----- */
40  WiFiClient    espClient;
41  PubSubClient  client(espClient);
42  const int     waterPump = 2;
43  const int     YL69Pin   = 34;
44  const int     DHTPin    = 23;
45  const int     LDR       = 35;
46
47  #define DHTTYPE DHT22
48
49  //create an instance of DHT sensor
50  DHT dht(DHTPin, DHTTYPE);
51
52
53  /*----- */
54  /*  F U N C T I O N
55   *  D E C L A R A T I O N  */
56  /*----- */
57  void WiFiConnection();
58  void initMQTT      ();
59  void checkMqtt      ();
60  int  splitTopic    (char* topic, char* tokens[] ,int tokensNumber);
61  void callback      (char* topic, byte* payload,  unsigned int
62                      length);
63  void measurements();
64
65  /* ----- */
66  void setup()
67  {
68      // Initializes the water pump.
69      pinMode(waterPump, OUTPUT);
70      Serial.begin(115200);
71      // Connects to the WiFi Network.
72      WiFiConnection();
73      // Starts the MQTT Service.
74      initMQTT();
75  }
76
77  void loop()
78  {
79      // Must be kept alive to process incoming messages and maintain
80      // connection to
81      // the server.
82      client.loop();
83      // In case MQTT is closed, it starts the service.
84      checkMqtt();
85  }
86
87  /* ----- */
88
89  /*
90   * Executes when a new message is received.
91   *
92   * CALLBACK FUNCTION IMPORTANT INFO:
93   * MQTT HAS 3 BASIC CONCEPTS:

```

```

93  *   1. TOPICS:
94  *       Similar to the address of a message.
95  *       In this case, it should look like Home/bedroom/
        waterpumpAlexa/Turn.
96  *       Where the msg is ON or OFF.
97  *   2. PUBLISH/SUBSCRIBE SYSTEM:
98  *       Devices can subscribe to any topic and, of course, receive
        the
99  *       messages from those topics they are subscribed to.
100  *       In order to publish a message, a topic must be provided!
101  *   3. BROKER:
102  *       Central device. It allows and restricts the connections.
        Its job is
103  *       also to receive, filter, redirect and publish messages to
        all the
104  *       devices connected.
105  *       In this case, the broker is Smartnest Server.
106  */
107
108 void callback(char* topic, byte* payload, unsigned int length)
109 {
110     // Representation of the connected topic.
111     Serial.print ("Topic:");
112     Serial.println(topic);
113
114     // Obtains the topic and prints the message.
115     int tokensNumber = 10;
116     char *tokens [tokensNumber];
117     char message[length + 1];
118     splitTopic(topic, tokens, tokensNumber);
119
120     sprintf(message, "%c", (char) payload[0]);
121     for (int i = 1; i < length; i++)
122     {
123         sprintf(message, "%s%c", message, (char) payload[i]);
124     }
125
126     // Prints message.
127     Serial.print ("Message:");
128     Serial.println(message);
129
130     // Notifies the experienced state alteration due to the received
        message.
131     char reportChange[100];
132     sprintf(reportChange, "%s/report/powerState", MQTT_CLIENT);
133
134     // Sends the new status to the Server.
135     // If this were to be placed in the loop, the device would get
        locked.
136     if(strcmp(tokens[1], "directive") == 0 &&
        strcmp(tokens[2], "powerState") == 0)
137     {
138         // Starts the water pump.
139         if(strcmp(message, "ON") == 0)
140         {
141             digitalWrite(waterPump, LOW);
142             client.publish(reportChange, "ON");
143

```

```

144
145         // ** YL-69 moisture ***
146         int const readYL69value = analogRead(YL69Pin);
147         // map inversely to 0..10%
148         int const convertedPercentage = map(readYL69value, 4095,
1200, 0, 100);
149         Serial.println("HUMIDITY: ");
150         Serial.print(convertedPercentage);
151         if (convertedPercentage < 80){
152             digitalWrite(waterPump, LOW);
153             client.publish(reportChange, "ON");
154         }
155         Serial.println("NO SE HA PODIDO REGAR");
156
157     }
158     // Stops the water pump.
159     else if(strcmp(message, "OFF") == 0)
160     {
161         digitalWrite(waterPump, HIGH);
162         client.publish(reportChange, "OFF");
163     }
164 }
165 }
166
167 /*
168  * Connection to the WiFi Network.
169  */
170 void WiFiConnection()
171 {
172     // Connect in case it has not been connected already.
173     if(WiFi.status() != WL_CONNECTED)
174     {
175         WiFi.mode(WIFI_STA);
176         WiFi.begin(SSID_NAME, SSID_PASSWORD);
177         // Tries to connect
178         Serial.println("Connecting ...");
179         while (WiFi.status() != WL_CONNECTED)
180         {
181             delay(500);
182             Serial.print(".");
183         }
184     }
185
186     // Connected Successfully.
187     Serial.println('\n');
188     Serial.print ("Connected to ");
189     Serial.println(WiFi.SSID());
190     Serial.print ("IP address:\t");
191     Serial.println(WiFi.localIP());
192     delay(500);
193 }
194
195 /*
196  * Initialization of MQTT service.
197  */
198 void initMQTT()
199 {

```

```

200 client.setServer (MQTT_BROKER, MQTT_PORT);
201 client.setCallback(callback);
202
203 while (!client.connected())
204 {
205     Serial.println("Connecting to MQTT...");
206     // Connected
207     if (client.connect(MQTT_CLIENT, MQTT_USERNAME, MQTT_PASSWORD))
208     {
209         Serial.println("Connected");
210     }
211     // Failed Connection
212     else
213     {
214         Serial.print("Failed with state ");
215         Serial.print(client.state());
216         delay(0x7530);
217     }
218 }
219
220 // Initialization of the Topic and Subscriber.
221 // Allows to show in the Serial any request to the service.
222 char reportTopic [100];
223 char publishTopic[100];
224
225 sprintf(reportTopic, "%s/report/online", MQTT_CLIENT);
226 sprintf(publishTopic, "%s/#", MQTT_CLIENT);
227 client.subscribe(publishTopic);
228 client.publish (reportTopic, "true");
229 }
230
231 /*
232 * Splits a string by the character "/".
233 * Can be specifically used to split topics.
234 * Eg.:
235 * input:  Home/bedroom/waterpumpAlexa/Turn
236 * output: [Home, bedroom, waterpumpAlexa, Turn]
237 */
238 int splitTopic(char* topic, char* tokens[], int tokensNumber)
239 {
240     const char s[2] = "/";
241     int pos = 0;
242
243     // Breaks the input into tokens
244     tokens[0] = strtok(topic, s);
245
246     while(pos < tokensNumber - 1 &&
247           tokens[pos] != NULL)
248     {
249         pos++;
250         tokens[pos] = strtok(NULL, s);
251     }
252
253     return pos;
254 }
255
256 /*

```

```

257  * Starts the MQTT service if it has not been initialized yet.
258  */
259  void checkMqtt ()
260  {
261      if(!client.connected())
262      {
263          initMQTT();
264      }
265  }
266
267  void measurements(){
268      // *** DHT22 measurement ***
269      //use the functions which are supplied by library.
270      double const humidity = dht.readHumidity();
271      // Read temperature as Celsius (the default)
272      double const temperature = dht.readTemperature();
273      // Check if any reads failed and exit early (to try again).
274      if (isnan(humidity) || isnan(temperature)) {
275          Serial.println("Failed to read from DHT sensor!");
276          delay(1000); // wait a bit
277          return;
278      }
279
280      // print the result to Terminal
281      Serial.print("Humidity (DHT22): ");
282      Serial.print(humidity);
283      Serial.print(" %\t");
284      Serial.print("Temperature (DHT22): ");
285      Serial.print(temperature);
286      Serial.println(" C ");
287
288      // *** internal temperature ***
289      //convert raw temperature in F to Celsius degrees
290      Serial.print("Temperature (internal): ");
291      Serial.print((temprature_sens_read() - 32) / 1.8);
292      Serial.println(" C");
293
294      // ** YL-69 moisture ***
295      int const readYL69value = analogRead(YL69Pin);
296      // map inversely to 0..10%
297      int const convertedPercentage = map(readYL69value, 4095, 1200, 0,
298          100);
299      Serial.print("Moisture (YL-69): ");
300      Serial.print(convertedPercentage);
301      Serial.print("%\n");
302
303      // ** LDR measurement ** //
304      int const readLDRvalue = analogRead(LDR);
305      Serial.println("Amoung of lighth: ");
306      Serial.print(readLDRvalue);
307      Serial.print("% ");
308  }

```

Arduino UNO code

As with this device does not count with WiFi or Bluetooth module, it was decided to code in Arduino the part relative to the sensor and data collections inside a JSON. This JSON will be send by the Serial port where the data can be read using a python script that collect that data and sends to the mongodatabase. The libraries needs to do this, has been mentioned before.

The code can be summarized as take function's, getter functions, createJson and the irrigate action.

Take function's are functions called by the timer that call the getter's functions which returns values from the different sensor. The reason why it is code in that way is due to the fact that the timer library only is able to call void functions.

The create json function provides a structure which can be summarized as identifier: object with the time when the measure has been taken, which gives a default value that is change when the script read the data and the sensor's value.

```
1  #include "DHT.h"
2  #include "SimpleTimer.h"
3  #include "ArduinoJson.h"
4
5  #define YL69 A0          // Analog pin for moisture sensor
6
7  #define DHTPIN 2         // Digital pin for external temperature and
    humidity
8  #define DHTTYPE DHT22
9
10 #define LDR A2           // Analog pin for LDR
11
12 #define waterPump A5     // Analog pin for waterpum
13
14 // Init DHT sensor
15 DHT dht(DHTPIN, DHTTYPE);
16
17 // Create timers
18 SimpleTimer extTimer;
19 SimpleTimer sendJSON;
20 SimpleTimer humidity;
21 SimpleTimer lighth;
22
23 // Variables for measurements
24 double extTem;
25 double extHumidity;
26 double intHumidity;
27 double luminosity;
28 double* arrayHumidity;
29
30 bool irrigate;
31
32 /*----- */
33 /*  F U N C T I O N
34  *  D E C L A R A T I O N  */
35 /*----- */
```



```

36 void takeExternalData();
37 void takeHumidity();
38 void takeLigth();
39
40 void createJSON();
41 void merge(JsonObject dest, JsonObjectConst src);
42
43 double getExtHumidity();
44 double getExtTemp();
45 double getHumidity();
46 double getLigth();
47
48 bool isIrrigating();
49
50 /*          *
51  * MAIN CODE *
52  *          */
53
54 void setup() {
55
56     Serial.begin(9600);
57     dht.begin();
58
59     extTimer.setInterval(10000, takeExternalData);
60     humidity.setInterval(10000, takeHumidity);
61     sendJSON.setInterval(20000, createJSON);
62 }
63
64 void loop() {
65     extTimer.run();
66     sendJSON.run();
67     humidity.run();
68 }
69
70
71 /*****  A U X I L I A R      F U N C T I O N S
       *****/
72
73 // Function that given two JSON return another with the combination
   of both
74 void merge(JsonObject dest, JsonObjectConst src) {
75     for (auto kvp : src) {
76         dest[kvp.key()] = kvp.value();
77     }
78 }
79
80 // Returns the JSON file which contains device and measurements
   data
81 void createJSON(){
82     // Init JSON files as Static elements which 480 of size
83     StaticJsonDocument<480> doc1, doc2;
84
85     // First document that only contains id device info
86     JsonObject dev=doc1.to<JsonObject>();
87     doc1["device"] = "5f4d3798d0df9a7447ca25e2";
88
89     // Second document which contains all the measurements

```

```

90 // Minute value is replaced before in the Python script
91 JSONArray datos = doc2.createNestedArray("humidityInt");
92 JsonObject root = datos.createNestedObject();
93 root["minute"]="46";
94 JsonObject measure = root.createNestedObject("measure");
95 measure["watered"]= irrigate;
96 JSONArray value = measure.createNestedArray("value");
97 value.add(intHumidity);
98
99
100 // In case DHT returns an allowed value, the information
101 // is add to the document, otherwise it is omitted
102 if (extHumidity != -99999){
103     JSONArray datosExt = doc2.createNestedArray("humidityExt");
104     JsonObject infoExt = datosExt.createNestedObject();
105     infoExt["minute"]="112";
106     infoExt["measure"]=extHumidity;
107 }
108
109 JSONArray lumExt = doc2.createNestedArray("luminosityExt");
110 JsonObject infoLum = lumExt.createNestedObject();
111 infoLum["minute"]="213";
112 infoLum["measure"]=luminosity;
113
114 if (extTem != -9898){
115     JSONArray tempExt = doc2.createNestedArray("temperatureExt");
116     JsonObject infoTemExt = tempExt.createNestedObject();
117     infoTemExt["minute"]="335";
118     infoTemExt["measure"]=extTem;
119 }
120
121 // Both documents are merged
122 merge(doc1.as<JsonObject>(), doc2.as<JsonObject>());
123
124 // Send JSON to Serial port
125 serializeJson(doc1, Serial);
126 Serial.println("");
127 delay(500);
128 }
129
130 // Take Functions are only functions that call other functions
131 // that returns the data
132
133 void takeExternalData(){
134     extTem = getExtTemp();
135     extHumidity = getExtHumidity();
136 }
137
138 void takeHumidity(){
139     intHumidity = getHumidity();
140     isIrrigating();
141 }
142
143 void takeLigth(){
144     luminosity = getLigth();
145 }
146

```

```

147
148 // Get measurements values
149 double getExtTemp()
150 {
151     // By default the temperature is given in Celsius
152     double const temperature = dht.readTemperature();
153     if (isnan(temperature) ) {
154         return -9898;
155     }
156     return temperature;
157 }
158
159 double getExtHumidity()
160 {
161     double const humidity = dht.readHumidity();
162
163     // Check if the read failed and exit early (to try again).
164     if (isnan(humidity) ) {
165         return -99999;
166     }
167     return humidity;
168 }
169
170 double getHumidity(){
171     double const readValueYL69 = analogRead(YL69);
172     double const convertedPercentage = map(readValueYL69, 0, 1023,
173         100, 0);
174
175     return convertedPercentage;
176 }
177
178 double getLigth(){
179     int const readLDRvalue = analogRead(LDR);
180     int ligth = map(readLDRvalue, 0, 370, 0, 100);
181
182     return ligth;
183 }
184
185 // Function that activate waterpump in case the plant need water
186 bool isIrrigating(){
187     if (intHumidity <= 35){
188         digitalWrite(waterPump, LOW);
189         humidity.setInterval(5000, takeHumidity);
190         irrigate = true;
191     }
192     else{
193         digitalWrite(waterPump, HIGH);
194         humidity.setInterval(10000, takeHumidity);
195         irrigate = false;
196     }
197
198     return irrigate;
199 }

```

Database script

This script is written in python due to the fact that is a language that provides a lot of libraries to manage easily json structures.

It is a python script which can read the data send by the Arduino in the serial port and sends this data to the database using the uri given by mongodb. Inside that uri it can be found the username, password, host (where the database is storage) and the accessible port.

Also, it is necessary to defined the database and the collection where the data is going to be stored. Then it is defined where is the serial port (in Linux, it can be found inside /dev/ttyACM0 but that is the case of the Arduino UNO, for other devices this can change), then the frequency that it will be used is 9600 Hz but can be used until 11520 Hz and the timeout.

For knowing more about the queries used inside the code, you can checked the database information.

```
1 import time
2 import datetime
3 import serial
4 import json
5 from pymongo import MongoClient
6 from bson.objectid import ObjectId
7
8 # CONNECTION TO MONGODB
9 uri = "mongodb://gardener:eco-app-plant@chir0n.duckdns.org:10072,
10      chir0n.duckdns.org:20072,chir0n.duckdns.org:30072/admin"
11 client = MongoClient(uri)
12
13 mongodb_db = "eco" # Database
14 db = client[mongodb_db] # Use
15 collection = db['measurements'] # Select
16
17 # DEFINE SERIAL PORT
18 serial_port = "/dev/ttyACM0" # If ESP32
19 # is used, it might be /dev/ttyUSB0
20 ser = serial.Serial(serial_port, 9600, timeout=0) # Using an
21 # ESP32 the frequency must be 115200
22
23 id = ObjectId() # Document
24 identification
25 generationHour = datetime.datetime.today().hour # Time
26 # when a new document is created
27
28 while True:
29     # Data for define the match filter date and hour
30     dt = datetime.datetime.today()
31     year = dt.year
32     month = dt.month
33     day = dt.day
34     hour = dt.hour
```

```

32
33     date      = datetime.datetime(year, month, day, hour) # Python
                    uses datetime as equivalent to Mongo Date.
34
35
36
37     # Every hour is generated a new id
38     if generationHour != hour:
39         id = ObjectId()
40         generationHour = hour
41
42     match_filter = {
43         "_id": id,
44         "plant": "",
45         "device": "",
46         "date": date,
47         "hour": hour
48     }
49
50     try:
51         serial_info = ser.readline() # Read from
52         serial port
53
54         if serial_info:
55
56             info = json.loads(serial_info) # Convert
57             to json
58
59             # Setting id for device and plant
60             match_filter["device"] = ObjectId(info["device"])
61             info.pop("device")
62
63             dev = client["eco"]["device"].find_one(
64                 {"_id": match_filter["device"]},
65                 {"_id": 0, "plant": 1}
66             )
67             match_filter["plant"] = ObjectId(dev["plant"]) #
68             Adding plant inside the match filter
69
70             # Measurements to add
71             values = { "$push": info}
72
73             minute = (int) (5* round(dt.minute/5)) #
74             Measurements are taken every 5 minutes,
75
76             # for
77             that it is used a 5-based time system
78
79             info["humidityInt"][0]["minute"] = minute
80             info["humidityExt"][0]["minute"] = minute
81             info["luminosityExt"][0]["minute"] = minute
82             info["temperatureExt"][0]["minute"] = minute
83
84             # Update operation
85             doc = collection.update(match_filter, values, upsert=True
86 ) # The upsert value is used to generate a new
87
88             # document when it doesn't exist a file that match
89
90

```

```

80         # with match_filter value's
81         print(info)
82
83     else:
84         print("Waiting\n")
85
86 except serial.SerialTimeoutException:
87     print("Error! Could not read data from serial port")
88
89 finally:
90     time.sleep(30)

```

Provide wifi to the Arduino

A way to provide the Arduino with WiFi is using the ESP8266 WiFi module which must be powered by 3.3 V. It used AT commands to get WiFi point access via serial port which can be configured with different velocities. Once this is configured, it can send information using an IP address and a port.

In case it is wanted to receive data, it cleans TCP/IP and resends the information via serial port with the clean data.

The code necessary to dump the console to serial port and viceversa, consists on adding the library `SoftwareSerial.h`, initialize `SoftwareSerial` with the RX and TX port, include the `begin` option for software serial inside the `setup` function and, finally, adding the `Serial` and `SoftwareSerial` available method inside the `loop` to read and write data. So the code must look similar to:

```

1  #include <SoftwareSerial.h>
2  SoftwareSerial BT1(3, 2); // RX | TX
3
4  void setup(){
5      Serial.begin(9600);
6      BT1.begin(9600);
7  }
8
9  void loop(){
10     String B= "." ;
11     if (BT1.available()){
12         char c = BT1.read() ;
13         Serial.print(c);
14     }
15     if (Serial.available()){
16         char c = Serial.read();
17         BT1.print(c);
18     }
19 }

```

Information relative to the AT operations can be found [here](#). The necessary commands would be `AT+MODE=(1,2 or 3)`, `AT+CWJAP= [SSID][PASSWORD]`

for connecting to the WiFi and AT+CIPMUX=1 for enable multiple simultaneous connections.

This module brings the option to keep sending data to the serial port where the script reads the data to send it to the database and also, receiving orders such as active the waterpump in case the plant need it.

3 Database

This project uses MongoDB as database engine, to store and manipulate data at will.

Why MongoDB

One of the reasons to use MongoDB, instead of other database is because it is open-source. The developers of this project deeply believe in the open-source community, having the content being publicly available allows developers to collaborate with us and bring free, secure and private applications: anybody can suggest improvements, modify, or fix bugs in the application.

Other reason would be that, compared to other SQL solutions, is the advantages for these type of projects, IoT:

- SQL solutions are designed to vertically. This is not suitable for IoT applications, where replication and sharding are, in most cases, mandatory - and therefore, horizontal scaling is best.
- The flexibility brought by a NoSQL solution allows for a faster development:
 - Model does not need to be as detailed, it admits fast adaptation of data and model through the different development phases of the project.
 - Allows different data types even for same field, so data types do not need to be specified in advanced.
 - Those data types can be redefined, without the need of changing the whole schema or the information already stored in the database.
- Also, having so much data coming from so many different devices all at ones means that, probably, data consistency is not one of the main pillars of the application - it might be more important to be able to use that information (despite outdated) to use analytic queries, predict behaviors, etc. Once again, NoSQL is more suited for this task.

Servers Setup

To store data it must be used a MongoDB server. With that objective, it can be used a server such as the ones provided by the Atlas Cluster (more info here) or create a private MongoDB server in a personal computer/server.

In case of using a private server, please refer to server creation documentation of this project. If more information is needed, it is recommended to use the MongoDB official documentation - some links have already been included in the guide done by the developers of this project.

Disclaimer: Please be advised, if you want to connect from outside your LAN and you use Port Forwarding, consider changing the default ports for every type of connection you need when accessing from a remote source, otherwise some security risks might appear in your application.

How to connect

Once the servers are running, it is used the mongo interactive shell as a command interface to those servers.

If the server is local, just specifying the port is sufficient to connect:

```
1 mongo --port <port>
```

But, in case it is not a local instance, or you want to connect from outside the LAN, the host address must be specified. In which case, it might be necessary to activate Port Forwarding in your router - **in case of doubt, please refer to your Telecom provider to ensure you do not break anything.**

```
1 mongo --host <ip-address>:<port>
```

This also allows connections in replica sets, it is specially useful as in this cases it does not connect to the Primary Node, but to the node that has been specified.

To always connect to the primary node, a step forward must be taken. Besides the host IP Address and Port on where MongoDB server is running, it is also needed the name of the replica set:

```
1 mongo --host <repl-set-name>/<ip-address-1>:<port-1>,<ip-address  
-2>:<port-2>,...
```

As shown before, it admits a list of nodes (pair of IP Address and Port), so in case the first node is not available, it tries the next one.

Finally, the authentication process can be done inside the Mongo Shell:

```
1 use <database>  
2 db.auth(<username>, <password>)
```

But it can also be done before connecting to the server. In order to do that, it must be specified the username and password of the user that is going to connect to the database, and also, the database on which to authenticate it (authorization must be enabled, and some user created).


```
1 mongo --host <host> -u <username> -p <password> --
authenticationDatabase <database>
```

The authentication database must be the database where the user was created.

Schema

It follows the NoSQL design and data modeling directives. There have been used patterns and other model relationships to implement the schema. Also, the collections have been optimized to reduce the workload on write operations, as an IoT application tends to suffer more from that type of operations.

The schema has been designed following the requirements of data insertion, and read operations, trying to minimize the impact of certain frequent operations, such as creating new documents.

No free or open-source data modeler has been found that would fit the design of the database schema. Instead, the Moon Modeler free trial version can be a substitute.

The resulting schema is [2]:

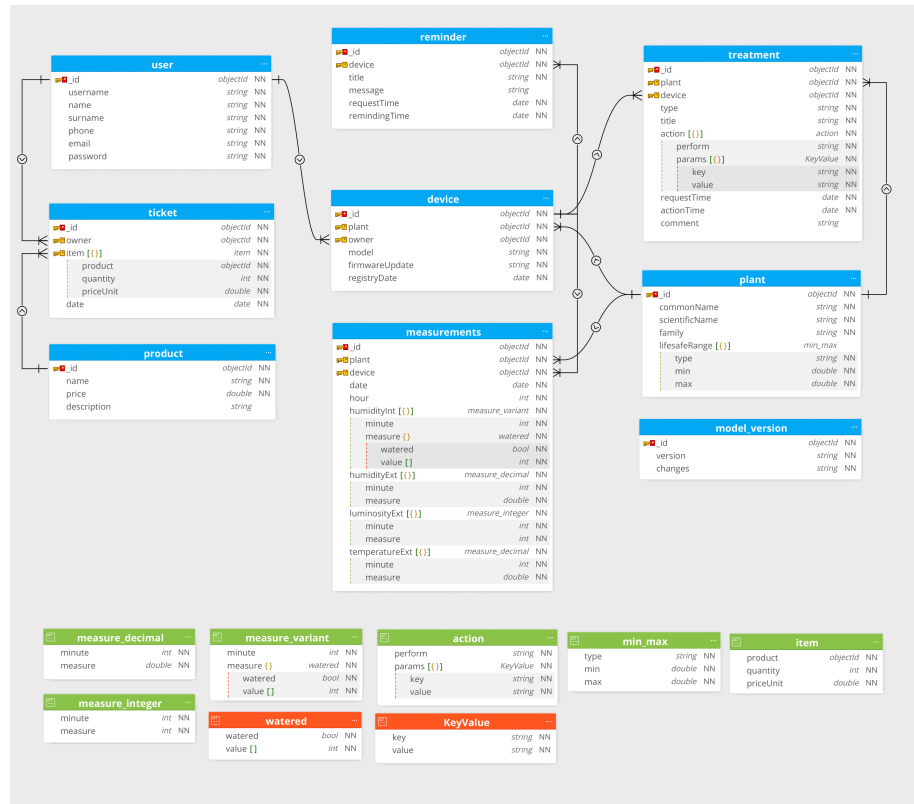


Figure 2: Database NoSQL Schema.

Collections

Before even starting, it should be taken into account that a collection is the equivalent to a table in a relational SQL database. That being said, the collections and their designs are going to be discussed in the following sections.

Plant The main objective of the device is to take care of plants, That is why, all the convenient information must be stored. The behavior of the device will be affected by parameters like the acceptable values of temperature, humidity, etc.

The collection used goes as follows:

```
1 {  
2   "_id": "<objectId>",  
3   "commonName": "<string>",  
4   "scientificName": "<string>",  
5   "family": "<string>",  
6   "lifesafeRange": [ {  
7     "type": "<string>",  
8     "min": "<double>",  
9     "max": "<double>"  
10  } ]  
11 }
```

javascript

The range of values that a plant can resist without endangering its safety are represented as subdocuments containing the edge values (minimum and maximum values) of that range.

Device The device stores each Ecø device information. That means knowing which plant it is tracking or the latest software update being installed. This last piece of information allows to notify the owner of the device to receive and install the latest firmware update:

```
12 {  
13   "_id": "<objectId>",  
14   "plant": "<objectId>",  
15   "owner": "<objectId>",  
16   "model": "<string>",  
17   "firmwareUpdate": "<string>",  
18   "registryDate": "<date>"  
19 }
```

javascript

Measurements One of the most complex structures to understand in this context. It uses patterns to simplify and reduce the development and maintenance complexity (for more information go to patterns documentation from this

project). It uses the bucket pattern by grouping the data into chunks of information of one hour in total. That way, by specifying the current day (`date`) and then the `hour`, it is certain that all the information that is going to be loaded into the database has a certain order and structure.

The documents of this collection would have the following shape:

```
20 {
21   "_id": "<objectId>",
22   "plant": "<objectId>",
23   "device": "<objectId>",
24   "date": "<date>",
25   "hour": "<int>",
26   "humidityInt": [{
27     "minute": "<int>",
28     "measure": {
29       "watered": "<bool>",
30       "value": ["<int>"]
31     }
32   }],
33   "humidityExt": [{
34     "minute": "<int>",
35     "measure": "<double>"
36   }],
37   "luminosityExt": [{
38     "minute": "<int>",
39     "measure": "<int>"
40   }],
41   "temperatureExt": [{
42     "minute": "<int>",
43     "measure": "<double>"
44   }]
45 }
```

javascript

Also, the developers have decided that all measures are to be taken at time (`minute`) multiples of 5, as it is the least common multiple of all parameters - that helps of reading and organizing the information. This prevents exceeding the database workload and losing data, as the total amount of write and read operations is greatly reduced.

Treatment In order to take care of the plant, some treatments are necessary. For the moment, the only available treatment in this project is watering the plant, although some others might be included in the near future.

Information such as when the request was made, or the type of action carried out is what this collection stores. For the action, it is used an standardize JSON file through all the interactions of the project. That being said, the structure used by those documents just store the function that is going to be performed and the parameters that function takes.

The overall document look like this:

```
46 {
47   "_id": "<objectId>",
```

```

48   "plant": "<objectId>",
49   "device": "<objectId>",
50   "type": "<string>",
51   "title": "<string>",
52   "action": {
53     "perform": "<string>",
54     "params": [
55       {
56         "key": "<string>",
57         "value": "<string>"
58       }
59     ]
60   },
61   "requestTime": "<date>",
62   "actionTime": "<date>",
63   "comment": "<string>"
64 }

```

javascript

Each parameter from the function may have different data type.

Reminder Reminders are linked to each device, so a reminder has to be set manually for all devices. The reminders can repeat the notification, or not. In order to ease having to take into account the dates, it also takes a certain number of repetitions.

Not all the fields are necessary. For example, if the reminder does not repeat, the date or the number of repetitions do not need to be set. It can be more easily understood by taking into a look at the . The last column of the collection (NN) indicates if that key needs to be set.

```

65 {
66   "_id": "<objectId>",
67   "device": "<objectId>",
68   "title": "<string>",
69   "message": "<string>",
70   "requestTime": "<date>",
71   "remindingTime": "<date>"
72 }

```

javascript

User The Ecø service users have an account linked to the webpage and the device (in case they have one).

In this collection there is a One-To-Many relationship from **user** to the **device** collection. Out of the several available options (*embed* or *reference* in the *one* or in the *many* side), an array of devices has been referenced in the *one side*. This is usually not the preferred behavior, although it is the most suitable for this occasion. For more information regarding this design decision refer to the **relationships** section in the patterns documentation.

```

73 {

```

```

74     "_id": "<objectId>",
75     "username": "<string>",
76     "name": "<string>",
77     "surname": "<string>",
78     "phone": "<string>",
79     "email": "<string>",
80     "password": "<string>"
81 }

```

javascript

Product In the *rainforest* store there are several products available (and more will be in the future). To store their information, it is kept a very simple structure of the most basic information.

```

82 {
83     "_id": "<objectId>",
84     "name": "<string>",
85     "price": "<double>",
86     "description": "<string>"
87 }

```

javascript

Ticket It is a way to simplify a *many-to-many* relationship between **product** and **user**. It does not use the MongoDB standard, but there is a reason behind this behavior. This information could make the size of both the **product** and **user** collections exponentially. To avoid that, it is created a different collection where only the data from the *transactions* is kept.

```

88 {
89     "_id": "<objectId>",
90     "owner": "<objectId>",
91     "product": [
92         {
93             "item": "<objectId>",
94             "quantity": "<int>",
95             "priceUnit": "<double>"
96         }
97     ],
98     "date": "<date>"
99 }

```

javascript

Model Version It simply helps tracking the changes through versions. It is self-explanatory. It also should help implementing the schema versioning pattern.

```

100 {
101     "_id": "<objectId>",
102     "version": "<string>",

```

```

103 |     "changes": "<string>"
104 | }

```

javascript

Operations

Once the servers are up and running, several operations can be performed on data stored. These operations are given the name of CRUD:

- Create Operations
- Read Operations
- Update Operations
- Delete Operations
- Bulk Write

Each of those types of operations are explained in a different documentation.

There are several forms to operate with information in MongoDB. One of them is to use MongoDB Compass or by the aforementioned Mongo Shell. The first of them does not belong to the scope of this manual, so only the Mongo Shell method will be explained.

Disclaimer: Should be taken into account that the information here displayed does not refer to any language driver, but for the mongo shell method.

Read Operations Retrieve information documents (information) from a given collection. Can use two methods: `findOne()` or simply `find()`. The first method retrieves only one document (the first encountered) that matches a certain condition; the second operation finds any document matching a condition.

db.collection.findOne()

The query uses a JSON document as filter. It can contain the parameters:

- **query (document):** *Optional.* Specifies query selection criteria using query operators.
- **projection (document):** *Optional.* Specifies the fields to return using projection operators. Omit this parameter to return all fields in the matching document.

It returns a single document that satisfies the criteria specified as the first argument to this method.

Although similar to the `find()` method, the `findOne()` method returns a document rather than a cursor.

db.collection.find()

Just as before, the query uses a JSON document. It can be used with the following parameters:

- **query (document):** *Optional.* Specifies query selection criteria using query operators. To return all documents in a collection, omit this parameter or pass an empty document `{}`.

- **projection** (*document*): *Optional*. Specifies the fields to return using projection operators. Omit this parameter to return all fields in the matching document.

It returns a cursor to the documents that match the query criteria. When the `find()` method returns a *document*, the method is actually returning a cursor to the documents.

Create Operations After being connected, two different operations can be used: `insertOne` and `insertMany`. Their names are self-explanatory, not much detail is needed regarding that point.

In both cases, if the `_id` is not specified for a document, MongoDB will automatically assign one. Meaning, writing an `_id` manually for each document inserted is not mandatory.

Also, if the *collection* where the document is inserted does not exist, MongoDB will automatically create one, just as with the `_id`. Therefore, a collection does not need to be created before a document insertion.

db.collection.insertOne()

Takes 2 parameters:

- **document** (*document*): A JSON document to insert into the specified collection.
- **writeConcern** (*document*): *Optional*. A document expressing the write concern. Omit to use the default write concern.

It returns:

- **acknowledged** (*boolean*): **true** if the operation ran with write concern or **false** if write concern was disabled.
- **insertedId** (*ObjectId*): the `_id` value of the inserted document.

db.collection.insertMany()

Takes 3 parameters:

- **document** (*document*): An array of JSON documents to insert into the specified collection.
- **writeConcern** (*document*): *Optional*. A document expressing the write concern. Omit to use the default write concern.
- **ordered** (*boolean*): *Optional*. A boolean specifying whether the mongod instance should perform an ordered or unordered insert. Defaults to **true**. Executing an ordered list of operations on a sharded collection will generally be slower than executing an unordered list since with an ordered list, each operation must wait for the previous operation to finish. In **ordered** operations, if an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list. On the other hand, in **unordered** if an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

It can also be used a bulk write operation to insert documents into the DB. Please refer to the section **Bulk Operations** in this same page.

It returns:

- **acknowledged** (*boolean*): **true** if the operation ran with write concern or **false** if write concern was disabled.
- **insertedId** (*ObjectId*): the `_id` value of the inserted document.

Update Operations Written information can also be altered. In order to do that, MongoDB counts with 3 operations: `updateOne`, `updateMany`, `replaceOne`.

db.collection.updateOne()

Takes the following parameters:

- **filter** (*document*): The selection criteria for the update. The same query selectors as in the `find()` method are available. Specify an empty document `{ }` to update the first document returned in the collection.
- **update** (*document/pipeline*): The modification to apply. Can be one of the following:
 - **Update document**: Contains only update operator expressions.
 - **Aggregation pipeline**: Contains only the following aggregation stages:
 - * `$addField` and its alias `$set`.
 - * `$project` and its alias `$unset`.
 - * `$replaceRoot` and its alias `$replaceWith`.
- **upsert** (*boolean*): *Optional*. When **true**, `updateOne()` either:
 - Creates a new document if no documents match the **filter**. For more details see upsert behavior.
 - Updates a single document that matches the **filter**.
- **writeConcern** (*document*): *Optional*. A document expressing the write concern. Omit to use the default write concern.
- **collation** (*document*): *Optional*. allows users to specify language-specific rules for string comparison, such as rules for letter-case and accent marks.
- **arrayFilters** (*array*): *Optional*. An array of filter documents that determine which array elements to modify for an update operation on an array field.
- **hint** (*document/string*): *Optional*. A document or string that specifies the index to use to support the query predicate.

It returns a document with the following information:

- **acknowledged** (*boolean*): **true** if the operation ran with write concern or **false** if write concern was disabled.
- **matchedCount** (*int*): number of matched documents.
- **modifiedCount** (*int*): number of modified documents.

- `upsertedId (ObjectId)`: `_id` for upserted document.
- db.collection.updateMany()**
- `updateMany()` updates all documents matching the `filter` parameter.
- `filter (document)`: The selection criteria for the update. The same query selectors as in the `find()` method are available. Specify an empty document `{ }` to update all documents in the collection.
 - `update (document/pipeline)`: The modification to apply. Can be one of the following:
 - **Update document**: Contains only update operator expressions.
 - **Aggregation pipeline**: Contains only the following aggregation stages:
 - * `$addField` and its alias `$set`.
 - * `$project` and its alias `$unset`.
 - * `$replaceRoot` and its alias `$replaceWith`.
 - `upsert (boolean)`: *Optional*. When **true**, `updateMany()` either:
 - Creates a new document if no documents match the `filter`. For more details see `upsert` behavior.
 - Updates a single document that matches the `filter`.
 - `writeConcern (document)`: *Optional*. A document expressing the write concern. Omit to use the default write concern.
 - `collation (document)`: *Optional*. Collation allows users to specify language-specific rules for string comparison, such as rules for letter-case and accent marks.
 - `arrayFilters (array)`: *Optional*. An array of filter documents that determine which array elements to modify for an update operation on an array field.
 - `hint (document/string)`: *Optional*. A document or string that specifies the index to use to support the query predicate.

It returns a document with the following information:

- `acknowledged (boolean)`: **true** if the operation ran with write concern or **false** if write concern was disabled.
- `matchedCount (int)`: number of matched documents.
- `modifiedCount (int)`: number of modified documents.
- `upsertedId (ObjectId)`: `_id` for upserted document.

db.collection.replaceOne()

The function `replaceOne()` it replaces the first matching document in the collection that matches the `filter`:

- `filter (document)`: The selection criteria for the update. The same query selectors as in the `find()` method are available. Specify an empty document `{ }` to update all documents in the collection.

- **replacement** (*document*): The replacement document. Cannot contain update operators.
- **upsert** (*boolean*): *Optional*. When **true**, `replaceOne()` either:
 - Inserts the document from the **replacement** parameter if no document matches the **filter**.
 - Replaces the document that matches the **filter** with the **replacement** document.
- **writeConcern** (*document*): *Optional*. A document expressing the write concern. Omit to use the default write concern.
- **collation** (*document*): *Optional*. Collation allows users to specify language-specific rules for string comparison, such as rules for letter-case and accent marks.
- **hint** (*document/string*): **Optional**. A document or string that specifies the index to use to support the query predicate.

It returns a document with the following information:

- **acknowledged** (*boolean*): **true** if the operation ran with write concern or **false** if write concern was disabled.
- **matchedCount** (*int*): number of matched documents.
- **modifiedCount** (*int*): number of modified documents.
- **upsertedId** (*ObjectId*): `_id` for upserted document.

Delete Operations The documents can also be deleted (careful, you will need a backup to recover the deleted data). There are two possible operations: `deleteOne` and `deleteMany`.

db.collection.deleteOne()

`deleteOne` removes a single document from a collection. It has the following parameters: - **filter** (*document*): Specifies deletion criteria using [query operators](<https://docs.mongodb.com/manual/reference/operator/>). Specify an empty document `{}` to delete the first document returned in the collection. - **writeConcern** (*document*): *Optional*. A document expressing the write concern. Omit to use the default write concern. - **collation** (*document*): *Optional*. Collation allows users to specify language-specific rules for string comparison, such as rules for letter-case and accent marks.

It returns a document containing the same parameters:

- **acknowledged** (*boolean*): **true** if the operation ran with [write concern or **false** if write concern was disabled.
- **deletedCount** (*int*): number of deleted documents.

db.collection.deleteMany()

`deleteMany` removes all documents that match the **filter** from a collection. Counts with the following parameters:

- **filter** (*document*): Specifies deletion criteria using query operators. To delete all documents in a collection, pass in an empty document { }.
- **writeConcern** (*document*): *Optional*. A document expressing the write concern. Omit to use the default write concern.
- **collation** (*document*): *Optional*. Collation allows users to specify language-specific rules for string comparison, such as rules for letter-case and accent marks.

It returns a document containing the same parameters:

- **acknowledged** (*boolean*): **true** if the operation ran with [write concern or **false** if write concern was disabled.
- **deletedCount** (*int*): number of deleted documents.

Bulk Operations Allows to perform write operations in bulk. This, of course, also includes the delete operations. It highly increases performance, as making a bulk operations means less round trips between the client and the database (refer to the external Bulk Operations in MongoDB article for more information).

The operations inside the bulk can be performed ordered or unordered. It means that ordered operations are executed serially. Therefore, if an error occurs during the execution of the bulk the remaining operations will not be executed and MongoDB will return an error.

On the other hand, executing an unordered list of operations means that they will be executed in parallel. If an error occurs, MongoDB will continue executing the remaining operations.

In addition, it supports the following operations: - insertOne - updateOne - updateMany - replaceOne - deleteOne - deleteMany

Implementation

The operations available through the REST API or the analytical thread are some examples. But those are not the only places where CRUD operations are used.

There is a script that connects the database with the reading made by the device. Which, at the same time, takes measurements and metrics from the monitored plant. In this script, there is a query to create the corresponding document that holds the aforementioned values.

There is a JavaScript file that contains the original query, which served as inspiration for its later translation to Python. It has the following content:

```

1 db.measurements.update(
2   // Match Filter
3   { "_id": "<objectId>", "plant": "<objectId>", "device": "<
   objectId>", "date": "<Date>", "hour": "<int>" },
4   // Insert new element into
5   { $push: { // Adds (appends) the arrays the values specified
6     "humidityInt": { $each: [
7       {
8         "minute": "<int>",

```

```

9       "measure": {
10         "watered": "<bool>",
11         "value": [
12           "<int>"
13         ]
14       }
15     },
16   ],
17   "humidityExt": { $each: [ // Appends the documents inside the
                           array to the
18     {                               // specified field
19       "minute": "<int>",
20       "measure": "<double>"
21     }
22   ] },
23   "luminosityExt": { $each: [
24     {
25       "minute": "<int>",
26       "measure": "<int>"
27     }
28   ] },
29   "temperatureExt": { $each: [
30     {
31       "minute": "<int>",
32       "measure": "<double>"
33     }
34   ] },
35   } },
36   // If the document does not exist to update, it creates it
37   // Also includes the fields at the filter
38   { upsert: true }
39 )

```

This script prevents having to create different queries for updating and creating new documents. This is achieved by using the `upsert` parameter, which creates a new document if there is other document that matches the `filter`

Patterns

To optimize database operations and storage of data, it is needed to take into account:

- **DB Objective**
- **Relationships**
- **Patterns**

As every IoT application, the data read from the devices is vastly greater than the data the users request of the information that is obtained from it.

That being said, it is clear that the main objective is to reduce the load of the write operations as result of the parameters obtained from the Ecø devices. That should be one of the main aspects to take into account in order to create a good model for the database.

Logically, space occupied in disk should also be another aspect to look at. As well as the development and maintenance difficulty of the code.

In the next sections it will be discussed the design decisions carried out by the developers of this project.

Relationships

Despite MongoDB being a document based database, it still has relationships among collections. To create a relationship between two documents, it is crucial to decide whether to embed or link - it may affect performance.

One-to-One It is commonly represented as a single table in a tabular database (for more information go to one-to-one relationships documentation).

Embed Prefer embedding over referencing for simplicity. If the fields are at the same level, it is very similar to a tabular database. It can be used subdocuments to organize fields, preserve simplicity and make overall documents clearer.

Reference It is used for optimization purposes (like in the extended reference pattern or the subset pattern, that are going to be explained on the patterns section). It uses the same identifier in both documents.

Examples It would be the case of the `user` and its private information (such as the `phone` or the `email` and `password`). In this case it could be separated the information into separate collections if it would be considered as a high security risk, but it would be as a future enhancement. For now the private and common information from the `user` are going to be kept together for simplicity reasons.

Another example would be the case of `plant` and the *safety measures* (that are inside the `lifesafe_range` field - should be `temperature`, `humidity` and `light_exposition_time`).

One-to-Many An object of a given type is associated with **N** objects of a second type; while in the other direction, an object of the second type can only be associated with **one** object of the *one* side.

In IoT it can also be identified the *one-to-zillions* subcase. It is the case of a relationship where the *many* side is identified as 10000 or more. It is suggested to be mindful of this relationship in every place that the relationship is used in the code. The last thing that you want to do is to retrieve a document and its zillions associated documents and then process the complete result set.

This gives two options:

Embed Using the same collection. Usually embedding in the entity most queried. Preferred over referencing for simplicity, or when there is a small number of referenced documents, as all related information is kept together. For more information go to one-to-many relationships documentation for embedded documents.

Reference Use two separate collections to keep the documents. Usually referencing in the *many* side. Preferred when the associated documents are not always needed within the most often queried documents. For more information go to one-to-many relationships documentation for referenced documents.

In the subcase of the *one-to-zillion*, there is only one option left: **reference in the *many/zillions* side** - sole representation for a *one-to-zillions* relationship, reference the document on the *one* side of the relationship from the *zillion* side - quantify the relationship to understand the maximum N value.

Examples The cases of **1-to-N** relationships using **reference** are easily observed in the schema, as they have a line joining the two collections, indicating the *one* as a strip, and the *many* side as an inverted arrow:

- user and device - device and reminder - device and measurements (mind *one-to-zillions*) - plant and device - plant and measurements (mind *one-to-zillions*)

In all the previous cases, embedding mechanism could have been used, but it would mean that the size of the documents would have grown exponentially as new sub documents were added (specially with the *device-reminder* and the *device/plant-measurements* cases).

In the cases of the *one-to-zillions* relationships, a pattern has been applied to reduce the number of entries, although it is still probable for it to reach the *zillions* state.

On the other hand, the **embedding** mechanism has been used in:

- treatment.action - measurements.humidity_int - measurements.humidity_ext - measurements.luminosity_ext - measurements.temperature_ext

In a tabular database (such as MySQL), it would be necessary referencing tables to achieve a similar structure.

In all of the **embedding** cases, it has been done in the most queried collections, in both cases they would represent the *one* side of the relationship (*treatment* and *measurement*).

Many-to-many In a normalized model, it can not be linked two tables as a *many-to-many* relationship. Under the hood, an additional relationship table needs to be created to define this relationship - *jump* table. It splits the relationship into two *one-to-many* relationships.

Embed Usually, only the most queried side is considered. Should prefer embedding on the most queried side, specially over the information that is primarily static over time and may profit from duplication.

Reference Prefer referencing over embedding to avoid managing.

Examples The most remarkable case would be `ticket` collection, as a middle collection for `user` and `product`. In this case, it is ****referenced**** the `_id` of the *many* sides into the middle collection, to avoid overcharging the *many* collections with non-entirely collection-related data.

There are more no-implicit cases where an N-to-N relationship appears. The cases where two collections have a *many-to-many* relationship thanks to a common intermediate table can not be considered as such. Some of those cases would be:

- `user` and `reminder` - `user` and `plant` - `user` and `treatment` - `user` and `measurements`
- `device` and `treatment`

Patterns

Patterns are the most powerful tool for designing database schema (and any code, and web content, etc. - anything that is done as it should, let's be honest). Despite their usefulness, patterns are not full solutions to problems. Patterns are smaller sections of full solutions. They are reusable unit of knowledge. They bring solutions that can scale and perform under stress.

It should also be taken into account that applying patterns may lead to:

- *Duplication*: Duplicating data across documents.
- *Data Staleness*: Accepting staleness in some pieces of data.
- *Data integrity issues*: Writing extra application side logic to ensure referential integrity.

		Use Case Categories						
		Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
Patterns	Approximation	✓		✓	✓		✓	
	Attribute	✓	✓					✓
	Bucket			✓			✓	
	Computed	✓		✓	✓	✓	✓	✓
	Document Versioning	✓	✓			✓		✓
	Extended Reference	✓			✓		✓	
	Outlier			✓	✓	✓		
	Preallocated			✓			✓	
	Polymorphic	✓	✓		✓			✓
	Schema Versioning	✓	✓	✓	✓	✓	✓	✓
	Subset	✓	✓		✓	✓		
	Tree and Graph	✓	✓					

Figure 3: Patterns list and their use cases.

The patterns from the previous table are just recommendations, but some of the patterns that are not ticked in the table could also be considered for the project at hand, but it is not usually the case (according to MongoDB documentation on pattern. The patterns used for this project and that should be considered in the future of the project are:

Attribute Pattern The attribute pattern is suitable for big documents, that have many similar fields but there is a subset of fields that share common characteristics and it is sorted or queried on that subset of fields; or the fields needed to sort are only found in a small subset of documents; or both of the previous conditions are met within the documents.

Pros:

- Fewer indexes are needed.
- Queries become simpler to write and are generally faster.

Study Case This is the case of `treatment.action` field. The original idea was to use separated fields for each type of treatment, or even a separated collection to reference each treatment. But that would mean either unnecessary fields or joining collections (inefficient in most cases). So it was decided to join all the actions performed by the device into a single well-structured and standardized field. All the fields or reference were substituted by a simulating a *key-value pair* field `action`.

For this occasion, instead of using the standard two-field structure: `k: "<field>", v: "<value>"`, it has been expanded to use to use more intuitive names such as:

```
105 "action": [  
106   {  
107     "perform": "<string>",  
108     "params": [ {  
109       "k": "<string>",  
110       "v": "<string>"  
111     } ]  
112   },  
113   { ... }  
114 ]
```

javascript

This structure also allows concatenating actions to create more elaborated treatments (right now the available treatments are not many, but it would necessary in a future expansion).

Bucket Pattern The bucket pattern is useful with data coming in as a stream over a period of time (time series data). This might suggest, at first hand, each measurement in its own document. This approach would be fit for a tabular database way of handling data.

This is going to lead to application scalability and index size issues. Instead, data can be bucketed by time, so documents hold measurements from a particular time span.

Pros:

- Reduces the overall number of documents in a collection.
- Improves index performance.
- Can simplify data access by leveraging pre-aggregation.

Study Case In `measurements` collection, it is stored the measures from the humidity inside the smart pot, and the humidity, light and temperature outside the pot.

The aforementioned structure typical of a tabular database was discarded and substituted by a single collection to store all four measures, with a time span of one hour.

As the minimum common period to read the is 5 minutes, the document is created at the beginning of each hour, and then updated every five minutes.

There is a special case for the interior humidity of the pot, which happens when the pot is being watered - the measurement period is drastically reduced from 5 minutes to measures every 5 to 15 seconds.

Polymorphic Pattern The polymorphic pattern is used when all documents in a collection are of similar, but not identical, structure. It is useful when wanted to access documents from a single collection. Grouping documents together based on the queries helps improve performance.

It is the solution when there are a variety of documents that have more than similarities than differences and the documents need to be kept in a single collection.

Pros:

- Easy to implement.
- Queries can run across a single collection.

Study Case It has been implemented in the `plant` collection, in the `lifesafe_range`. Before implementing this pattern, there were 6 different fields to specify the ranges between which the plant was safe to live in (minimum and maximum humidity, temperature and light exposition - last one indicating time in hours).

This was then changed for three fields containing documents (called `min_max`) that had a `min` and `max` field, in order to indicate the span of the three parameters.

Finally, it was implemented a single field (`plant.lifesafe_range`), that simply specifies the parameters and its span. This allows not to standardize the different spans into one single field. This also allows the option of not setting any range in case they are unknown to a certain plant. It also allows to set a new one in case something else needs to be considered for a certain plant.

Schema Versioning Pattern The schema versioning pattern is used to support changes between versions in the data schema.

An application starts with an original schema which eventually needs to be altered. When that occurs it can be created and save the new schema to the database with a `schema_version` or simply `version`.

This field will allow to know how to handle a particular document. Alternatively, the application can deduce the version based on the presence or absence of some fields, but the former method is preferred.

It can be assumed that documents that *do not* have this field are version 1. Each new schema version would then increment the `schema_version` field value and could be handled accordingly in the application.

Pros:

- No downtime needed to update version.
- Control of schema migration.
- Reduced future technical debt.

Cons:

- Might need two indexes for the same field during migration.

Study Case It has been created a collection called `model_version`. This helps on creating a model that documents the changes suffered in collection when the version changes.

Therefore, the `version` field on each document will be related to the `model_version` entrance.

Greatly reduces the possible technical debt of migrating the whole data stored so far - should take into account that Ecø is an IoT application, and therefore the amount of data stored could easily reach the *zillions* relationship. Therefore, changing the structure, fields content, adding new information, etc. to all the documents could take days, months or even years depending on the change required.

Subset Pattern The subset pattern addresses the issues associated with a working set that exceeds RAM, resulting in information being removed from memory. This is frequently caused by large documents which have a lot of data that isn't actually used by the application.

Pros:

- Reduction in the overall size of the .
- Shorter disk access time for the most frequently used data.

Cons:

- Must manage the subset.
- Pulling in additional data requires additional trips to the database.

Study Case The `measurements` collection itself could easily be part of the `device` documents. But that would mean that the documents would grow exponentially in no time, even reaching the 16MB limit per document.

To avoid that, what should be the `measurements` field was transformed into a new collection entirely.

It was also taken into account to decide the size (actually, the time period span) in the bucket pattern.

Setting Up the Cluster

First thing is first, in order to have a cluster, it is necessary to create a single server.

Server Configuration

Each server has its own configuration file, which are simple text files that allow to encode the different options of each server, does not matter if it is and `mongod` instance or a `mongos` instance. Each configuration file must end with the extension

.conf They use **YAML format** - *YAML does not support tab characters for indentation: use spaces instead.*

The usage of this kind of files simplifies **large-scale deployments**. For more information refer to the MongoDB Configuration file documentation.

Using them is simple, just use the one of the following commands:

```
1 mongod --config <path-to-file>
2 mongod -f <path-to-file>
```

bash

Configuration File Parameters

There are more parameters than the ones used in these files, but the ones used for this project are:

systemLog

- **path** (*string*): The path of the log file to which `mongod` or `mongos` should send all diagnostic logging information, rather than the standard output or the host's syslog. MongoDB creates the log file at the specified path.
- **logAppend** (*boolean*): When true, `mongos` or `mongod` appends new entries to the end of the existing log file when the `mongos` or `mongod` instance restarts. Without this option, `mongod` will back up the existing log and create a new file.
- **destination** (file — syslog): If you specify *file*, you must also specify `systemLog.path`. If you do not specify `systemLog.destination`, MongoDB sends all log output to standard output.

processManagement

- **fork** (*boolean*): Enable a daemon mode that runs the `mongos` or `mongod` process in the background. By default `mongos` or `mongod` does not run as a daemon.

net

- **port** (*integer*): The TCP port on which the MongoDB instance listens for client connections. Default:
 - **27017** for `mongod` (if not a shard member or a configuration server member) or `mongos` instance.
 - **27018** if `mongod` is a shard member.
 - **27019** if `mongod` is a configuration server member.
- **bindIp** (*string*): The hostnames and/or IP addresses and/or full Unix domain socket paths on which `mongos` or `mongod` should listen for client connections. You may attach `mongos` or `mongod` to any interface. To bind to multiple addresses, enter a list of comma-separated values.

security

- **authorization** (*string*): Enable or disable Role-Based Access Control (RBAC) to govern each user's access to database resources and operations. Set this option to one of the following:
 - **enabled**: A user can access only the database resources and actions for which they have been granted privileges.
 - **disabled**: A user can access any database and perform any action.
- **keyFile** (*string*): The path to a key file that stores the shared secret that MongoDB instances use to authenticate to each other in a sharded cluster or replica set. **keyFile** implies **security.authorization**. See Internal/Membership Authentication for more information.
- a single key string (same as in earlier versions),
- multiple key strings (each string must be enclosed in quotes), or
- sequence of key strings.

storage

- **dbPath** (*string*): The directory where the **mongod** instance stores its data.
- **directoryPerDB** (*boolean*): When true, MongoDB uses a separate directory to store data for each database. The directories are under the **storage.dbPath** directory, and each subdirectory name corresponds to the database name.

replication

- **replSetName** (*string*): The name of the replica set that the **mongod** is part of. All hosts in the replica set must have the same set name. If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

Replication

Database replication is a methodology that allows to have multiple servers at the same time, one of them being the main server, with several secondary servers connected.

This brings the possibility of having the same data separated in different servers (that can be even in different locations). This means that if the main server fails, the application service will not be interrupted, it will be redirected to other of the secondary servers (which will become primary by a process called election). There will always be service as long as a primary can be elected.

There are different type of nodes: - Primary node: Receives **all write operations**. Can only be one primary per replica set. - Secondary node: Maintains a copy of the primary's data. Data is replicated via an asynchronous process that copies the operations from the primary oplog. - Hidden Node: It acts just like a secondary, but it is hidden from the application. Used for reporting and backup tasks. Needs to have its priority set to 0, it cannot become primary. - Delayed Node: Contain a copy of the data, but that data can be an earlier of delayed copy of the dataset. They are mainly used for recoverability purposes,

specially for unsuccessful application upgrades and operator errors. They have a priority of 0 and it is advisable for them to be hidden. They can vote in elections (if their vote is set to 1). - Arbiter Node: They cannot become primary, they do not have a copy of the data, but they can vote. It is a solution when another secondary node cannot be added to the replica set. They have exactly 1 election vote.

Reading operations can be done against any primary or secondary node of the cluster, which is handy for, for example, analytical queries, where it does not matter if data is a little outdated.

On the other hand, write operations can only be done against the primary node, for consistency reasons.

Launch the Server

All the previous information comes in handy on understanding how everything is built. But it does not say how to setup the server.

1. Go to MongoDB Enterprise and get it installed, just follow the steps.
2. Make sure MongoDB has been included to your PATH, if it has not, it is highly advisable to do so.
3. To avoid using any system folder, create a new one wherever pleases you.

In example:

```
1 mkdir /var/mongodb
2 sudo chown pace:pace -R /var/mongodb
```

4. Run mongod by using the standalone-server.conf file.

```
1 systemLog:
2   path: "/var/mongodb/db/mongod1.log"
3   logAppend: true
4   destination: file
5 processManagement:
6   fork: true
7 net:
8   port: 27001
9   bindIp: "localhost,192.168.1.69"
10 security:
11   authorization: "enabled"
12   keyFile: "/var/mongodb/pki/eco-keyfile"
13 storage:
14   dbPath: "/var/mongodb/db/1"
15   directoryPerDB: true
```

- (a) May need to configure/create the paths if the suggested here are not used:

```
1 mkdir -pv /var/mongodb/db/1
```

```
2 | chmod -Rv 700 /var/mongodb/db/1
```

5. Now, connect to the server by using `mongo` and the necessary parameters

(a) May need the `--host` or `--port` parameters.

6. Run the initial configurations script `init_config.js` to create new users using the command:

```
1 | load("init_config.js")
```

(a) Might need to specify the path where that file is stored in your PC.

(b) Contains the code:

```
1 | db = db.getSiblingDB('admin')
2 |
3 | db.createUser({
4 |   user: "root",
5 |   pwd: "root",
6 |   roles : [ { db: "admin", role: "root" } ]
7 | })
8 |
9 | db.auth({
10 |   user: "root",
11 |   pwd: "root"
12 | })
13 |
14 | db.createUser({
15 |   user: "data-analyst",
16 |   pwd: "analyst",
17 |   roles : [
18 |     { db: "admin", role: "clusterAdmin" },
19 |     { db: "admin", role: "backup" },
20 |     { db: "admin", role: "restore" },
21 |     { db: "admin", role: "readAnyDatabase" }
22 |   ]
23 | })
24 |
25 | db.createUser({
26 |   user: "eco-manager",
27 |   pwd: "manager",
28 |   roles : [ { db:"eco", role:"dbOwner" } ]
29 | })
30 |
31 | db.createUser({
32 |   user: "gardener",
33 |   pwd: "eco-app-plant",
34 |   roles : [ { db:"eco", role:"readWrite" } ]
35 | })
36 |
37 | db = db.getSiblingDB('eco')
38 |
39 | db.model_version.insert(
```

```

40 [
41   { version: "0", changes: "Creation of eco database and
42     model_version and a way to track changes between versions." },
43   { version: "0.1", date: new Date(), changes: "Created users."
44     },
45   { version: "0.2", date: new Date(), changes: "Schema design:
46     Add 6 collections (owner, device, reminder, measurements, plant
47     , treatment)" },
48   { version: "0.2.1", date: new Date(), changes: "Apply MongoDB
49     patterns to collections." }
50 ]
51 )

```

(c) Now that the users and *eco* database have been created, it is time to create the replication cluster.

7. Connect to the standalone server and shut it down, this will allow to set a new configuration (change the previously set configuration) and maintain the databases and users created.
8. Being connected to the Mongo Shell, execute the following commands:

```

1 use admin
2 db.shutdownServer()

```

9. To enforce the security of the new replica set is convenient to use a keyFile. To set up this file it is necessary to:

- (a) Create the file and give the necessary permissions (specifying the path to the file).
- (b) Give the appropriate permissions to the file.

```

1 sudo mkdir -pv /var/mongodb/pki
2 openssl rand -base64 741 > /var/mongodb/pki/eco-keyfile
3 chmod 600 /var/mongodb/pki/eco-keyfile
4 mkdir -pv /var/mongodb/db/{2,3}

```

10. It is necessary to create different files for every node in the server (need to change the port, the dbPath, logPath). Refer to table [1] and to snippet [1].

- (a) One again, some new folder will be necessary before launching the servers.

```

1 mkdir -pv /var/mongodb/db/{2,3}
2 chmod -Rv 700 /var/mongodb/db/{2,3}

```


11. After connecting to the server that is going to act as the primary of the cluster (in this case the `mongo-repl-1.conf`), it must be initiated the replica set and added the rest of the members (if the replica set is not local, change the host ip address). Can use either of the two following methods.

```
1 rs.initiate(  
2   {  
3     _id: "eco-repl",  
4     version: 1,  
5     members: [  
6       { _id: 0, host : "localhost:27001" },  
7       { _id: 1, host : "localhost:27002" },  
8       { _id: 2, host : "localhost:27003" }  
9     ]  
10  }  
11 )
```

or

```
1 rs.initiate()  
2 rs.add( { host: "localhost:27002" } )  
3 rs.add( { host: "localhost:27003" } )
```

12. To check that everything is running as it should run the command in the MongoShell:

```
1 rs.status()
```

The users and databases previously created in the standalone node should now be available in the rest of the nodes.

13. Finally, execute the following commands to include some data into the database. Containing users for the application, devices, etc. [reflst:data]:

```
1 load(data.js)
```

Also, some other information that has the part of the data regarding the products and some purchases [3].

```
1 load(product_integration.js)
```

Disclaimer: The command has been executed when connecting from the main folder of the repository. To know where you are when being connected, execute `pwd()` in the Mongo Shell. Also, remember to clone this repository, it will make this guide easier to follow.

Reconfigure a Node

As it might have been suggested earlier, to reconfigure a running node it is needed to first stop the node, and then launch the new configuration using the same paths and ports. The rest of the values can usually be changed.

In order to avoid problems while shutting down a node, use one of the following options, in order of preference:

14. Use **shutdownserver**:

```
1 use admin
2 db.shutdownServer()
```

15. Use **--shutdown**:

It is not mandatory, but it is recommended to specify the `--dbpath`.

```
1 mongod --shutdown --dbpath <path-to-running-instance>
```

16. Use **Ctrl+C**:

Needs to have the process running on a terminal, without the `--fork` option.

17. Use **kill**:

Needs to know the PID (in Linux):

```
1 ps -ef | grep mongo
2 kill <mongod-PID>
```

After that, just run the new configuration (use one of the next options):

```
1 mongod --config <path-to-new-config>
2 mongod -f <path-to-new-config>
```

Reconfigure a Running Replica Set

Being connected to the mongo shell, to the Primary node, there is a command that returns the current configuration of the replica set.

```
1 rs.config()
```

The thing is, the JSON that represents the configuration returned in that function call, can be stored in a variable as a simple text. This allows to edit the content of the configuration (i.e: change the IP:Port pair of a given host). All the elements of the JSON can be accessed using dot notation.

```

1  cfg = rs.config()
2  cfg.members[1].host = "localhost:27017"

```

But it does not mean that by changing the variable with the configuration actually affects the configuration of the replica set. In order to do that, a reconfiguration must be performed.

```

1  rs.reconfig(cfg)

```

Code and tables

```

1  systemLog:
2    path: "/var/mongodb/db/mongod1.log"
3    logAppend: true
4    destination: file
5  processManagement:
6    fork: true
7  net:
8    port: 27001
9    bindIp: "localhost,192.168.1.69"
10 security:
11   authorization: "enabled"
12   keyFile: "/var/mongodb/pki/eco-keyfile"
13 storage:
14   dbPath: "/var/mongodb/db/1"
15   directoryPerDB: true
16 replication:
17   replSetName: "eco-repl"

```

Listing 1: Replica Set Configuration File. There are other two, that change the path, the port and the dbPath to their respective values.

```

1  db = db.getSiblingDB('admin')
2
3  db.auth({
4    user: "gardener",
5    pwd: "eco-app-plant"
6  })
7
8  db = db.getSiblingDB('eco')
9
10 plant_id = ObjectId()
11 owner_id = ObjectId()
12 device_id = ObjectId()
13
14 db.plant.insert(
15   [
16     {
17       "_id": plant_id,
18       "commonName": "Kalanchoe",

```

```

19         "scientificName": "Kalanchoe Blossfeldiana",
20         "family": "Asteraceae",
21         "lifesafeRange": [
22             {
23                 "type": "temperature",
24                 "min": 10.0,
25                 "max": 28.0
26             },
27             {
28                 "type": "humidity",
29                 "min": 40.0,
30                 "max": 80.0
31             },
32         ]
33     },
34 ],
35 { "ordered": false }
36 )
37
38 db.owner.insert(
39     [
40         {
41             "username": "pablo.acereda",
42             "name": "Pablo",
43             "surname": "Acereda Garcia",
44             "phone": "000000001",
45             "email": "pablo.acereda@eco.com",
46             "password": "3410
a597eaaa5894e92b2b51bc3934aa80d58c30f3efcc802826ad9ea1506992"
// acereda
47         },
48         {
49             "username": "javier.albert",
50             "name": "Javier",
51             "surname": "Albert Segui",
52             "phone": "000000002",
53             "email": "javier.albert@eco.com",
54             "password": "72
d0166b5707d129dc321e56692fe454c034552ee9e2b38f5a7f1c1306a632ea"
// albert
55         },
56         {
57             "username": "ana.castillo",
58             "name": "Ana",
59             "surname": "Castillo Martinez",
60             "phone": "000000003",
61             "email": "ana.castillo@eco.com",
62             "password": "
b6e81b71ac210e45b216fccb3302f1ebc798947ba9ddca6b83ed1ddc63b2ff70
" // castillo
63         },
64         {
65             "username": "dave.craciunescu",
66             "name": "Dave",
67             "surname": "Craciunescu",
68             "phone": "000000004",
69             "email": "dave.craciunescu@eco.com",

```

```

70         "password": "0653076
bc4dd143b243b5c7d896bd31c53450ec06dfe4053a2d1710056f011fa" //
craciunescu
71     },
72     {
73         "_id": owner_id,
74         "username": "laura.perez",
75         "name": "Laura",
76         "surname": "Perez Medeiro",
77         "phone": "000000005",
78         "email": "laura.perez@eco.com",
79         "password": "
a5ed602ee512bda8b2b18d6d4b06d6f176e7e3fb15a0cf5b23028b9849bd0d62
" // perez
80     }
81 ],
82 { "ordered": false }
83 )
84
85 db.device.insert(
86 [
87     {
88         "_id": device_id,
89         "plant": plant_id, // Kalanchoe
90         "owner": owner_id, // laura.perez
91         "model": "eco-pot-2020",
92         "firmwareUpdate": "1.0",
93         "registryDate": new Timestamp()
94     }
95 ],
96 { "ordered": false }
97 )
98
99 db.treatment.insert(
100 [
101     {
102         "plant": plant_id,
103         "device": device_id,
104         "type": "scheduled",
105         "title": "irrigation",
106         "action": [
107             {
108                 "perform": "watering",
109                 "params": [
110                     {
111                         "k": "time",
112                         "v": "5 s"
113                     },
114                     {
115                         "k": "flow",
116                         "v": "min" // can be: min, max,
percentage or a number
117                     }
118                 ]
119             }
120         ],
121         "requestTime": new Date("<2020-08-06T12:00:00Z>"), // Z

```

```

122         means UTC time
123         "actionTime": new Date("<2020-08-07T12:00:00Z>"),
124         "comment": "Water the plant due to low pot humidity."
125     },
126     { "ordered": false }
127 )
128
129 db.model_version.insertOne(
130     { version: "0.3", date: new Date(), changes: "Initial data
131       insertion (users, plants and first device registry)."}
132 )

```

Listing 2: Users, Plants and other Data

```

1  db = db.getSiblingDB('admin')
2
3  db.auth({
4      user: "gardener",
5      pwd: "eco-app-plant"
6  })
7
8  db = db.getSiblingDB('eco')
9
10 db.owner.renameCollection('user')
11
12 db.model_version.insert(
13     [
14         { version: "0.3.1", date: new Date(), changes: "OWNER is
15           now USER collection."},
16         { "ordered": true }
17     ]
18 )
19 var aprobado_id = ObjectId()
20 var eco_id = ObjectId()
21
22 db.product.insert(
23     [
24         {
25             "_id": eco_id,
26             "name": "eco smart pot",
27             "price": 60.0,
28             "description": "Rainforest best product. A little
29               something so you don't have to worry about your plants"
30         },
31         {
32             "name": "red pot",
33             "price": 10.0,
34             "description": "Phones have covers, we have colorful
35               pots"
36         },
37         {
38             "name": "blue pot",
39             "price": 10.0,
40             "description": "Phones have covers, we have colorful
41               pots"
42         }
43     ]
44 )

```

```

39     },
40     {
41         "name": "green pot",
42         "price": 10.0,
43         "description": "Phones have covers, we have colorful
pots"
44     },
45     {
46         "name": "pink pot",
47         "price": 10.0,
48         "description": "Phones have covers, we have colorful
pots"
49     },
50     {
51         "_id": aprobado_id,
52         "name": "aprobado",
53         "price": 0,
54         "description": "Profes, mirad que pedazo de proyecto os
estamos haciendo. Esto se merece un 10."
55     }
56 ],
57 { "ordered": false }
58 )
59
60 pablo_id = ObjectId("5f3184c0ede1e7f42f37c62b")
61 david_id = ObjectId("5f3184c0ede1e7f42f37c62e")
62 laura_id = ObjectId("5f3184bfede1e7f42f37c629")
63
64 db.ticket.insert(
65     [
66         {
67             "owner": pablo_id,
68             "item": [
69                 {
70                     "product": aprobado_id,
71                     "quantity": 2,
72                     "priceUnit": 0
73                 }
74             ],
75             "date": new Date()
76         },
77         {
78             "owner": david_id,
79             "item": [
80                 {
81                     "product": aprobado_id,
82                     "quantity": 1,
83                     "priceUnit": 0
84                 }
85             ],
86             "date": new Date()
87         },
88         {
89             "owner": laura_id,
90             "item": [
91                 {
92                     "product": aprobado_id,

```

```

93         "quantity": 5,
94         "priceUnit": 0
95     },
96     {
97         "product": eco_id,
98         "quantity": 1,
99         "priceUnit": 60.0
100     }
101 ],
102     "date": new Date()
103 }
104 ],
105 { "ordered": false }
106 )
107
108 db.model_version.insert(
109     [
110         { version: "0.4", date: new Date(), changes: "Include
111         PRODUCT and TICKET collections."},
112         { version: "0.4.1", date: new Date(), changes: "TICKET.DATE
113         field added."},
114         { version: "0.4.2", date: new Date(), changes: "Bug fixes.
115         Incorrect data type for PRODUCT.DESCRPTION"},
116         { version: "0.4.3", date: new Date(), changes: "Sub-
117         document for items in TICKET.{ITEM, QUANTITY, PRICEUNIT}.",
118         { version: "0.4.4", date: new Date(), changes: "Other Bug
119         fixes. REMINDER now has the same time fields as TREATMENT"}
120     ],
121     { "ordered": true }
122 )

```

Listing 3: Products Integration

Table 1: Replica Set Configuration Files.

type	PRIMARY	SECONDARY	SECONDARY
config filename	mongo-repl-1.conf	mongo-repl-2.conf	mongo-repl-3.conf
port	27001	27002	27003
dbPath	/var/mongodb/db/1	/var/mongodb/db/2	/var/mongodb/db/3
logPath	/var/mongodb/db/mongod1.log	/var/mongodb/db/mongod2.log	/var/mongodb/db/mongod3.log
replSet	eco-repl	eco-repl	eco-repl
keyFile	/var/mongodb/pki/eco-keyfile	/var/mongodb/pki/eco-keyfile	/var/mongodb/pki/eco-keyfile

4 Sub Projects Details

This project has used state of the art technologies in order to ease the development and scalability future issues. Also, it also accomplishes a familiar and

simple UI for the users to interact with Rainforest products.

The project has been splitted up into two sub-projects. On the one hand, there is the front-end project - developed in Angular.js. On the other hand, there is the back-end project - developed in Java.

Therefore, they need to be connected somehow.

Sub-Projects

Having separated the front-end and the back-end allows to have an independent implementation of each part of the development stack. This means that the same back-end could be used by different applications. For example, different web application or phone application - iOS, Android, Windows Phone, etc. but use the same back-end.

As expected, the **front-end** will contain the UI, the part of the project where the user can interact, make requests, look at the information displayed, etc. It also contains a small part of logic. This logic is just the necessary to give access to certain pages, or to manipulate the data obtained from a back-end call - the front part of the RESTful API.

The **back-end** includes the main logic of the program, and the database where data is stored. To connect both, the back-end part of the RESTful API handles the requests from the front, retrieves the data and manipulates it. It also counts with threads for analytical purposes.

These two projects are independent of each other. In order for them to interact, the `.jar` created out of the front-end project must be inserted into the back-end project. In order to do so, Maven comes in handy.

Implementation

1. Should have previously created the front-end and back-end projects separately.
2. Duplicate the `pom.xml` file in the back-end project to the parent folder. The parent folder is the one containing both services.
3. The parent `pom.xml` must contain the following:
 - (a) Should specify that the `<artifactId>` is the parent module.
 - (b) Does not need the `<dependencies>`.
 - (c) Keeps the `<parent>` and the `<properties>` attributes.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
  //www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://
  maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
6   <name>eco-smart-pot</name>
7   <description>Eco smart pot. A Tamagotchi in real life.</
    description>
```

```

8
9     <groupId>org.rainforest.eco</groupId>
10    <artifactId>parent</artifactId>
11    <version>0.0.1-SNAPSHOT</version>
12    <packaging>pom</packaging>
13
14    <parent>
15        <groupId>org.springframework.boot</groupId>
16        <artifactId>spring-boot-starter-parent</artifactId>
17        <version>2.3.2.RELEASE</version>
18        <relativePath/> <!-- lookup parent from repository -->
19    </parent>
20
21    <properties>
22        <java.version>1.8</java.version>
23    </properties>
24
25 </project>

```

4. Back to the back-end folder. In its *pom.xml*:

- (a) The parent attributes are the `<groupId>`, `<artifactId>` and `<version>` created in the parent folder *pom.xml*.
- (b) Keeps its originals `<artifactId>` and its `<dependencies>`.
- (c) To avoid future errors, also include the `maven-surefire-plugin` to the `<build>` ; `<plugins>` attribute.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
  //www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://
  maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
6   <artifactId>eco-webpage</artifactId>
7   <name>eco-webpage</name>
8   <description>Eco smart pot. A Tamagotchi in real life.</
  description>
9
10  <parent>
11      <groupId>org.rainforest.eco</groupId>
12      <artifactId>parent</artifactId>
13      <version>0.0.1-SNAPSHOT</version>
14  </parent>
15
16  <dependencies>
17      <dependency>
18          <groupId>org.springframework.boot</groupId>
19          <artifactId>spring-boot-starter-data-mongodb</artifactId>
20      </dependency>
21      <dependency>
22          <groupId>org.springframework.boot</groupId>
23          <artifactId>spring-boot-starter-web</artifactId>
24      </dependency>

```

```

25
26     <dependency>
27         <groupId>org.springframework.boot</groupId>
28         <artifactId>spring-boot-devtools</artifactId>
29         <scope>runtime</scope>
30         <optional>true</optional>
31     </dependency>
32     <dependency>
33         <groupId>org.projectlombok</groupId>
34         <artifactId>lombok</artifactId>
35         <optional>true</optional>
36     </dependency>
37     <dependency>
38         <groupId>org.springframework.boot</groupId>
39         <artifactId>spring-boot-starter-tomcat</artifactId>
40         <scope>provided</scope>
41     </dependency>
42     <dependency>
43         <groupId>org.springframework.boot</groupId>
44         <artifactId>spring-boot-starter-test</artifactId>
45         <scope>test</scope>
46         <exclusions>
47             <exclusion>
48                 <groupId>org.junit.vintage</groupId>
49                 <artifactId>junit-vintage-engine</artifactId>
50             </exclusion>
51         </exclusions>
52     </dependency>
53 </dependencies>
54
55 <build>
56     <plugins>
57         <plugin>
58             <groupId>org.springframework.boot</groupId>
59             <artifactId>spring-boot-maven-plugin</artifactId>
60         </plugin>
61         <plugin>
62             <groupId>org.apache.maven.plugins</groupId>
63             <artifactId>maven-surefire-plugin</artifactId>
64             <version>2.19.1</version>
65         </plugin>
66     </plugins>
67 </build>
68
69 </project>

```

5. Copy the *pom.xml* file from the back-end to the front-end folder:

- (a) Remember to change the `<artifactId>` and `<name>` for this project.
- (b) The `<parent>` attribute should stay as it is.
- (c) Need to specify where the compiled project should be kept in the back-end folder.
- (d) Also, need to include a couple of plugins and tell Maven how to

compile the project.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http:
3   //www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://
5     maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7
8   <artifactId>eco-front</artifactId>
9   <name>eco-front</name>
10  <description>Eco smart pot. A Tamagotchi in real life.</
11    description>
12
13  <parent>
14    <groupId>org.rainforest.eco</groupId>
15    <artifactId>parent</artifactId>
16    <version>0.0.1-SNAPSHOT</version>
17  </parent>
18
19  <!-- Install npm and dependencies -->
20  <build>
21    <resources>
22      <resource>
23        <directory>target/frontend</directory>
24        <targetPath>static</targetPath>
25      </resource>
26    </resources>
27
28    <plugins>
29      <plugin>
30        <groupId>com.github.eirslett</groupId>
31        <artifactId>frontend-maven-plugin</artifactId>
32        <version>1.3</version>
33
34        <configuration>
35          <nodeVersion>v14.2.0</nodeVersion>
36          <npmVersion>6.14.4</npmVersion>
37          <workingDirectory>src/app</workingDirectory>
38        </configuration>
39
40        <executions>
41          <execution>
42            <id>install node and npm</id>
43            <goals>
44              <goal>install-node-and-npm</goal>
45            </goals>
46          </execution>
47
48          <execution>
49            <id>npm install</id>
50            <goals>
51              <goal>npm</goal>
52            </goals>
53          </execution>
54
55          <execution>
56            <id>npm run build</id>
```

```

54         <goals>
55             <goal>npm</goal>
56         </goals>
57
58         <configuration>
59             <arguments>run build</arguments>
60         </configuration>
61     </execution>
62 </executions>
63 </plugin>
64 </plugins>
65 </build>
66
67 </project>

```

- Now that the front-end has an `<artifactId>`, include the following snippet to the `<dependencies>` section. This connects the front-end and the back-end.

```

1 <dependency>
2   <groupId>org.rainforest.eco</groupId>
3   <artifactId>eco-front</artifactId>
4   <version>${project.version}</version>
5   <scope>runtime</scope>
6 </dependency>

```

- Finally, and now that both sub-projects `<artifactId>` has been set, include this snippet to the parent *pom.xml*:

```

1 <modules>
2   <module>eco-front</module>
3   <module>eco-webpage</module>
4 </modules>

```

- Get to the parent folder and run:

```

1 mvn clean install
2 cd eco-webpage
3 mvn spring-boot:run

```

4.1 Projects Interaction

The way the two independent projects/services interact can be summarized by the following image [4]:

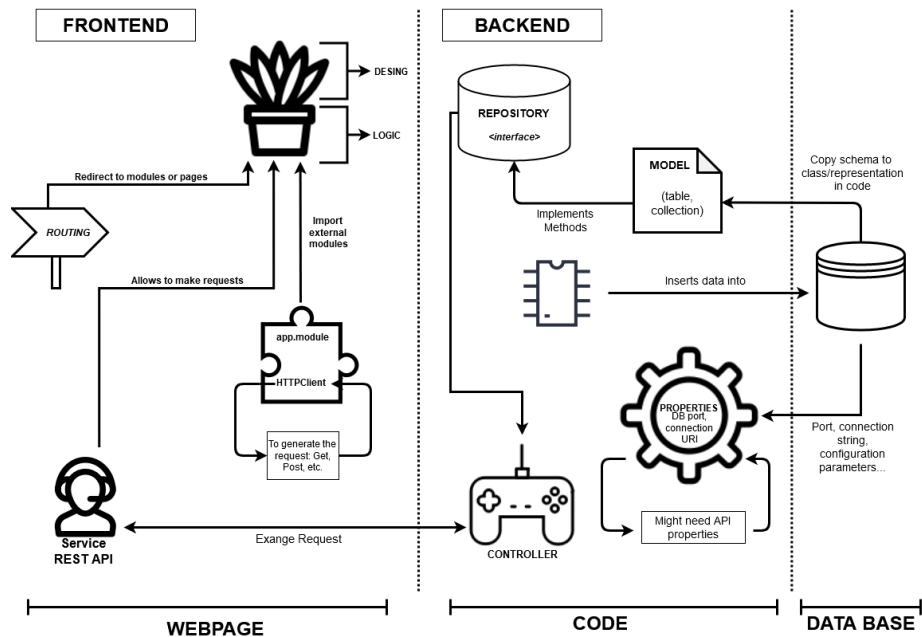


Figure 4: How modules interact with each other.

This image has been used before, but it is the best one to explain how everything works.

The **database** stores the data provided by the **device**. The collections in the database are mimicked by the the models in the back-end. This **models** help implementing the **repositories** that provide the operations for the **controllers**.

The actual configuration for the **database** is done in the project back-end **properties**.

The **controllers** are called by the **service REST API** in the front-end side of the application. The last one, returns the response to the **web pages** and the **logic**. Every component is accessed by the redirections provided by the **router**. The external modules to be used in the UI of the application are import through the **app.module**.

5 Back-end

To develop this part of the project, and due to the lack of other technology that would be of common knowledge between the development team, Spring Boot was selected as the framework to create the micro service.

According to the official documentation, Spring Boot:

- Create stand-alone Spring applications.
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files).

- Provide opinionated ‘starter’ dependencies to simplify the build configuration.
- Automatically configure Spring and 3rd party libraries whenever possible.
- Provide production-ready features such as metrics, health checks, and externalized configuration.
- Absolutely no code generation and no requirement for XML configuration.

Which, as a backbone, uses Java JDK. A well known technology by all the developers in this project. Specially, taking into account that part of the service provided by this part of the project is a Thread to monitor the state of the plant.

Also, as Java started being even more private than in the past. As proud members of the open-source community, the team members decided to use the Open JDK.

This part of the project is the one that handles the requests made from the webpage and performs the necessary operations to complete the task at hand. The answer given to the requester is called **Response**.

What it does is basically to create a JPA:

- **Model**: representation of each collection in MongoDB - sub-documents are represented as referenced objects.
- **Repository**: it is what connects the code with the database. Uses models to know the *shape* of the collection. Provides default operations to perform against the database in exchange.
- **Controller**: the logic of the requests is hidden in this layer. Its task is to execute the necessary functions to retrieve the data, manipulate it and return the desired result.

How to Deploy it

Remember, this project has been separated into two different modules. So either follow the steps to create those modules or launch the web application and the service application.

Following these steps is how the Spring Boot project has been created:

1. Direct to Spring Initializr and create a project with the following characteristics:
 - (a) **Project**: Maven Project
 - (b) **Language**: Java
 - (c) **Spring Boot**: 2.3.3
 - (d) **Project Metadata**:
 - i. **Group**: con.rainforest
 - ii. **Artifact**: eco
 - iii. **Name**: eco
 - iv. **Description**: Eco smart pot. A Tamagotchi in real life.

v. **Package name:** com.rainforest.eco

vi. **Packaging:** Jar

vii. **Java:** 8

(e) **Dependencies:** Search for:

- i. **Spring Web:** Build web, using RESTful, applications using Spring MVC.
- ii. **Spring Data MongoDB:** Store data using MongoDB.
- iii. **Spring Boot DevTools:** Allows to configure application to auto re-run itself when a file is changed.

2. Include the dependencies that need to be added manually: `log4j2` and their respective configuration (can find it on this same document, on the previous sections).

3. Create the REST API. For more information of the code, please refer to the Code section of this document. In order to do that:

Now there is an empty project ready to start a new project.

Other Dependencies

During the development of an application it is quite common to be in a situation where part of the code has been changed/adapted, and that means that the application needs to be re-executed in order for the changes to appear.

To do that, the dependency Spring Boot DevTools can be included. Adding the following syntax to the *pom.xml* file allows to auto-update the application and reflect the last changes made to it.

```
1 <dependency> <!-- Get file changes during runtime -->
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-devtools</artifactId>
4   <scope>runtime</scope>
5   <optional>true</optional>
6 </dependency>
```

In order to track the execution of the different components of the back-end module, a log has been included into the project.

In Spring Boot it is usually used the *log4j2* logging system. It is supposed to be an improved *log4j* - mainly because it fixes some inherent problems in the LogBack's architecture.

In order to work with it, a dependency must be included into the *pom.xml* file:

```
1 <dependency> <!-- Logging -->
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-log4j2</artifactId>
4 </dependency>
```


Besides that, a configuration file, by the name of `log4j2.xml` must be created. It is recommended to place the file under the folder `src/main/resources`. Following the instructions from the official documentation the configuration file in this project would be like this⁶:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Configuration status="WARN" monitorInterval="30">
3   <Properties>
4     <Property name="LOG_PATTERN">
5       %d{yyyy-MM-dd HH:mm:ss.SSS} %5p ${hostName} --- [%15.15
6       t] %-40.40c{1.} : %m%n%ex
7     </Property>
8   </Properties>
9   <Appenders>
10    <Console name="ConsoleAppender" target="SYSTEM_OUT" follow=
11    "true">
12      <PatternLayout pattern="${LOG_PATTERN}"/>
13    </Console>
14  </Appenders>
15  <Loggers>
16    <Logger name="com.rainforest.eco" level="debug" additivity=
17    "false">
18      <AppenderRef ref="ConsoleAppender" />
19    </Logger>
20
21    <Root level="info">
22      <AppenderRef ref="ConsoleAppender" />
23    </Root>
24  </Loggers>
25
26  <!-- Rolling File Appender -->
27  <RollingFile name="FileAppender" fileName="logs/eco.log"
28    filePattern="logs/eco-%d{yyyy-MM-dd}-%i.log">
29    <PatternLayout>
30      <Pattern>${LOG_PATTERN}</Pattern>
31    </PatternLayout>
32    <Policies>
33      <SizeBasedTriggeringPolicy size="10MB" />
34    </Policies>
35    <DefaultRolloverStrategy max="10"/>
36  </RollingFile>
37</Configuration>
```

REST API

It is not a standard but a set of recommendations and constraints for RESTful web services. A REST API suggests to create an object with the data from the requests from the user and send it as response. Some other recommendations include:

⁶It might occur that no logging file is created. The messages might appear on the terminal after running the Java Application.

- **Client-Server:** A makes a request to B - being both systems - through an URL hosted in B. B returns a response for that request.
- **Stateless:** REST is stateless: the client making the request should provide all the information the server needs to respond to that request. Meaning that, no matter the circumstances the response will be received if the input is all the same.
- **Cacheable:** A response should be defined as cacheable or not.
- **Layered:** The requesting client need not know whether it is communicating with the actual server, a proxy, or any other intermediary.

Create a REST API

In order to create a REST API that fits a certain application it is always necessary to follow the next steps 5:

Model Generate one for every table/collection in the database. It represents the shape of a certain document/entry. In the case of MongoDB, to represent nested objects⁷, it is usually implemented a new object that is embedded as an attribute.

It is linked to a certain document in the database by inserting the annotation `@Document(collection="<collectionName>")`. Some common error is to forget this annotation, which might lead to an unexpected error.

The `_id` needs the annotation `@Id` to avoid some more unexpected errors.

Repository is what could be called as the real connection between the *model* and the *database*. It provides simple operations such as creating new documents⁸, deleting them, finding on fields. *Repositories* are interfaces, not objects. They usually extend from `CrudRepository<T, S>` or `MongoRepository<T, S>`, the *T* is the model.

For this project it has been used *CrudRepository*. Both of them supply the basic operations, but the one we chose provided some more operations that the *CrudRepository* and was a little bit easier to use.

It also needs the annotation `@Repository` to be recognized as such and be able to use it in the controller without having to add packages analyzers to the main Application class.

Controller is where the *Business Logic* of the service is implemented. It uses the models to parse the incoming data and to generate the response output for a client request. It links a URL pattern⁹ and a type of request (GET, POST,

⁷In Java code.

⁸Specific MongoDB case.

⁹Like for example `"/api/users"`.

PUT, DELETE) to a specific function. That means that two functions cannot share the same time of URL and type of requests¹⁰.

All the controllers have the following annotations:

```
1 @Controller
2 @EnableAutoConfiguration
3 @RequestMapping("/api")
```

These annotations indicate that the class is a *Controller*, and therefore allowing *classpath* scanning to get the rest of the necessary components. Next, the *EnableAutoConfiguration* auto-configures beans that the class is likely to need. Finally, the *RequestMapping* on top of the class includes the indicated string at the end of the URL used to access the API. By convention, it is simply added */api*. Others also include the version of the API, for compatibility reasons.

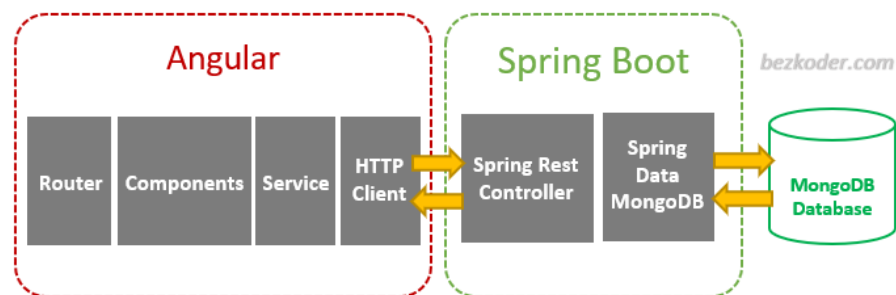
In order for the *controller* to recognize the *repositories*, the annotation *@Autowired* is set on top of their declaration.

Finally, each function has the following annotation headers, in example:

```
1 @RequestMapping(value="/devices", method=RequestMethod.POST)
2 @ResponseBody
```

The first annotation adds the *value* string at the end of the URL when calling the API, it is a way to differentiate requests to the same *methods* in the same API. The method value indicates what type of method is called when the function is executed. In this project four have been used: POST, GET, PUT and DELETE.

The other annotation is VERY important. Do not make the same mistakes that we did. Spending 4 days debugging the API and trying every tutorial available is not a very pleasant thing to do. The only solution that we found was starting from a simplistic HelloWorld Application and then include the project's requirements one by one. The *@RequestBody* is used to bind the HTTP response body with a domain object in the return type.



¹⁰Been there, takes a while to notice why the function that previously worked does not work now.

Figure 5: How does the REST API work

Each collection (except for the model version that should not be affected by any outer actor) has a *controller* with the following functions:

Create It creates a new document in the corresponding collection. Most of the collections do not even need to check certain conditions before creating a new document.

This is not the case of, for example, *user* collection. That is because no *user* can be created if the username or the email of the new user are currently being used.

The structure of each function goes as follows:

```
1 @RequestMapping(value="/<api_direction>", method=RequestMethod.POST
2 )
3 @ResponseBody
4 public ResponseEntity<Model> createDocument(@RequestBody Model
5     document)
6 {
7     String LogHeader = "[<api_direction>: createDocument] ";
8
9     try {
10         Log.logger.info(LogHeader + "Requested");
11
12         Model _document = modelRepository.save(
13             new Model(
14                 document.getSomeField(),
15                 document.getSomeOtherField(),
16                 document.getSomeThisField(),
17                 document.getDescriptionField()
18             )
19         );
20
21         Log.logger.info(LogHeader + "Successful");
22         return new ResponseEntity<>(_document, HttpStatus.OK);
23     } catch (Exception e) {
24         Log.logger.error(LogHeader + "some error occurred: " + e);
25         return new ResponseEntity<>(null, HttpStatus.
26             INTERNAL_SERVER_ERROR);
27     }
28 }
```

Get All Retrieves all the documents in a collection. No filters are included in this function.

```
1 @RequestMapping(value="/<api_direction>", method=RequestMethod.GET)
2 @ResponseBody
3 public ResponseEntity<List<Model>> getAllDocuments()
4 {
5     String LogHeader = "[<api_direction>: getAllDocuments] ";
6 }
```

```

6
7
8     try {
9         Log.logger.info(LogHeader + "Requested");
10        List<Model> documents = new ArrayList<Model>();
11
12        modelRepository.findAll().forEach(documents::add);
13
14        if (documents.isEmpty()) {
15            Log.logger.info(LogHeader + "No documents found");
16            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
17        }
18
19        Log.logger.info(LogHeader + "Successful");
20        return new ResponseEntity<>(documents, HttpStatus.OK);
21    } catch (Exception e) {
22        Log.logger.error(LogHeader + "some error occurred: " + e);
23        return new ResponseEntity<>(null, HttpStatus.
24            INTERNAL_SERVER_ERROR);
25    }
26 }

```

Get By Id In this case, documents are filtered on their *_id*. That means that only one document is going to be returned, as the *_id* cannot be repeated.

```

1  @RequestMapping(value="/<api_direction>/{id}", method=RequestMethod
2     .GET)
3  @ResponseBody
4  public ResponseEntity<Model> getDocumentById(@PathVariable("id")
5     String id)
6  {
7     String LogHeader = "[<api_direction>/id: getDocumentById] ";
8
9     try {
10        Log.logger.info(LogHeader + "Requested");
11        Optional<Model> documentData = modelRepository.findById(id);
12
13        if (documentData.isPresent()) {
14            Log.logger.info(LogHeader + "Successful");
15            return new ResponseEntity<>(documentData.get(), HttpStatus.OK
16        );
17        } else {
18            Log.logger.info(LogHeader + "No document found with id: " +
19            id);
20            return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
21        }
22    } catch (Exception e) {
23        Log.logger.error(LogHeader + "some error occurred: " + e);
24        return new ResponseEntity<>(null, HttpStatus.
25            INTERNAL_SERVER_ERROR);
26    }
27 }

```

Update Documents might need to be updated. Some data might be inserted incorrectly. A user might want to change its email, etc.

```
1 @RequestMapping(value="/<api_direction>", method=RequestMethod.PUT)
2 @ResponseBody
3 public ResponseEntity<Model> updateDocument(@RequestBody Model
4     document)
5 {
6     String LogHeader = "[<api_direction>: updateDocument] ";
7
8     try {
9         Log.logger.info(LogHeader + "Requested");
10        Optional<Model> documentData = modelRepository.findById(
11            document.getId());
12
13        if (documentData.isPresent())
14        {
15            Model _document = documentData.get();
16            _document.setId      (document.getId());
17            _document.setField   (document.getField());
18            _document.setOtherModel (document.getOtherField());
19
20            Log.logger.info(LogHeader + "Successful");
21            return new ResponseEntity<>(modelRepository.save(_document),
22                HttpStatus.OK);
23        } else {
24            Log.logger.info(LogHeader + "The document to update has not
25                been found");
26            return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
27        }
28    } catch (Exception e) {
29        Log.logger.error(LogHeader + "some error occurred: " + e);
30        return new ResponseEntity<>(null, HttpStatus.
31            INTERNAL_SERVER_ERROR);
32    }
33 }
```

Delete By Id To delete a single document, it is as simple as using its *_id*. At the beginning we thought of using other fields, like *username* for the *user* collection - as it is a unique value. But depending on the username, passing it through the URL request might lead to errors.

Given a certain document to delete its *_id* is always going to be known. The reason behind this statement is that no element is loaded statically in the webpage. That means, that any information appearing at any point of the webpage, other than the components, has to be loaded from database. It also makes the system a little bit more secure, as finding an *_id* is harder than finding a short string representing, for example, a *username*. Therefore, it is almost impossible to know data outside the website.

```
1 @RequestMapping(value="/<api_direction>/{id}", method=RequestMethod
2     .DELETE)
```

```

2  @ResponseBody
3  public ResponseEntity<HttpStatus> deleteDocument(@PathVariable("id"
4  ) String id)
5  {
6      String LogHeader = "[</api_direction>/id: deleteDocument] ";
7      try {
8          Log.logger.info(LogHeader + "Requested");
9          Optional<Model> document = modelRepository.findById(id);
10
11         if (document.isPresent()) {
12             Log.logger.info(LogHeader + "The document \"" + document.get
13             ().getId() + "\" is going to be deleted");
14             modelRepository.deleteById(id);
15         }
16
17         Log.logger.info(LogHeader + "Successful");
18         return new ResponseEntity<>(HttpStatus.NO_CONTENT);
19     } catch (Exception e) {
20         Log.logger.error(LogHeader + "some error occurred: " + e);
21         return new ResponseEntity<>(null, HttpStatus.
22         INTERNAL_SERVER_ERROR);
23     }
24 }

```

Delete All It deletes all documents from a collection. Although it might sound like a great way to clean everything up, it does not always work. Careful, the REST API might return the error *405 method not allowed*.

```

1  @RequestMapping(value="/devices/", method=RequestMethod.DELETE)
2  @ResponseBody
3  public ResponseEntity<HttpStatus> deleteAllDevices()
4  {
5      String LogHeader = "[/devices: deleteAllDevices] ";
6
7      try {
8          Log.logger.info(LogHeader + "Requested");
9          deviceRepository.deleteAll();
10
11         Log.logger.info(LogHeader + "Successful");
12         return new ResponseEntity<>(HttpStatus.NO_CONTENT);
13     } catch (Exception e) {
14         Log.logger.error(LogHeader + "some error occurred: " + e);
15         return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
16     }
17 }

```

Custom The functions introduced in the previous sub-sections do not cover all the behavior and information that this application should provide. Other functions have been implemented to increase the API functionality.

Some functions can use *CrudRepository* to implement new functions. Other need the *MongoTemplate* class. It allows to perform queries/aggregation pipelines on a pre-established connection to the database (do not try to configure the connection on your own, the right variables in the *application.properties* file are more than enough to connect to the database - it is safer and simpler).

Get By Other Field As it has been mentioned before, a *repository* extends from *CrudRepository*. That means that some function calls are already implemented, like the ones that have been explained before.

But, it also allows to implement custom functions to search by a specific field. In order to do that, the *repository* must specify the definition of a function as follows:

```
1 TypeToReturn<Model> findByField(FieldType field);
```

Remember that the *repository* is an interface. No further implementation is needed. This code (adapting it to the specific *model* and query - might return just one element or more) works in the same way as the *findById* function.

Some examples are the ones implemented in:

- Reminder:

```
1 List<Reminder> findByTitle(String title);
```

- Ticket

```
1 List<Ticket> findByOwner(ObjectId owner);
```

- Treatment

```
1 List<Treatment> findByDevice(ObjectId device);
```

The actual implementations are not going to be shown, as the changes between functions are minimal, and quite similar to the *findById* functions in all *controllers*.

After Values The aforementioned *MongoTemplate* comes in handy for this type of queries. Actually it does for every query... but let's specify for those that do not get into any of the previously explained cases.

One example would be getting all the documents where a specific field is greater than a certain value.

This is the case of the *MeasurementsController*, function *getMeasurementsAfterDate*. In this case, the documents where the measures are newer than a certain date are returned.


```

1 public ResponseEntity<List<Measurements>> getMeasurementsAfterDate(
    @RequestBody DayRequest date)
2 {
3     String LogHeader = "[/measurements/from: getMeasurementsAfterDate
        ] ";
4
5     try {
6         Log.logger.info(LogHeader + "Requested");
7
8         Query query = new Query();
9         query.addCriteria(Criteria.where("device").is(new ObjectId(date
            .getDevice())));
10        query.addCriteria(Criteria.where("date").gte(date.getToday()));
11        List<Measurements> measurements = mongoTemplate.find(query,
            Measurements.class);
12
13        if (!measurements.isEmpty()) {
14            Log.logger.info(LogHeader + "Successful");
15            return new ResponseEntity<>(measurements, HttpStatus.OK);
16        } else {
17            Log.logger.info(LogHeader + "No measures found after date: "
                + date.getToday());
18            return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
19        }
20
21    } catch (Exception e) {
22        Log.logger.error(LogHeader + "some error occurred: " + e);
23        return new ResponseEntity<>(null, HttpStatus.
            INTERNAL_SERVER_ERROR);
24    }
25 }

```

Between Values Just as the situation before, but in this occasion adding the condition that the field value must also have an upper limit. This is the situation in *Product* in the function *getProductBetweenPrices*; and *measurements* in the function *getMeasurementsBetweenDates*. They are both very similar (except) one uses decimals and the other one uses dates.

```

1 @RequestMapping(value="/measurements/between", method=RequestMethod
    .GET)
2 @ResponseBody
3 public ResponseEntity<List<Measurements>>
    getMeasurementsBetweenDates(@RequestBody BetweenDatesRequest
        date)
4 {
5     String LogHeader = "[/measurements/between:
        getMeasurementsAfterDate] ";
6
7     try {
8         Log.logger.info(LogHeader + "Requested");
9
10        Query query = new Query();

```

```

11 query.addCriteria(Criteria.where("device").is(new ObjectId(date
12 .getDevice())));
13 query.addCriteria(Criteria.where("date").gte(date.getMinDate())
14 .lte(date.getMaxDate()));
15 List<Measurements> measurements = mongoTemplate.find(query,
16 Measurements.class);
17
18 if (!measurements.isEmpty()) {
19     Log.logger.info(LogHeader + "Successful");
20     return new ResponseEntity<>(measurements, HttpStatus.OK);
21 } else {
22     Log.logger.info(LogHeader + "No measures found between the
23     dates: " + date.getMinDate() + " and " + date.getMaxDate());
24     return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
25 }
26
27 } catch (Exception e) {
28     Log.logger.error(LogHeader + "some error occurred: " + e);
29     return new ResponseEntity<>(null, HttpStatus.
30     INTERNAL_SERVER_ERROR);
31 }
32 }

```

There is another similar implementation, where only the initial date is given and where the upper value is the pre-computed *tomorrow* date (in *Reminder* and *Treatment*).

```

1 @RequestMapping(value="/treatments/programmed", method=
2 RequestMethod.GET)
3 @ResponseBody
4 public ResponseEntity<List<Treatment>>
5     getTreatmentsProgrammedNextDay(@RequestBody DayRequest today)
6 {
7     String LogHeader = "[/treatments/programmed:
8     getTreatmentsProgrammedNextDay] ";
9
10     try {
11         Log.logger.info(LogHeader + "Requested");
12
13         Query query = new Query();
14         query.addCriteria(Criteria.where("device").is(new ObjectId(
15         today.getDevice())));
16         query.addCriteria(Criteria.where("actionTime").gte(today.
17         getToday()).lte(today.getTomorrow()));
18         List<Treatment> treatments = mongoTemplate.find(query,
19         Treatment.class);
20
21         if (!treatments.isEmpty()) {
22             Log.logger.info(LogHeader + "Successful");
23             return new ResponseEntity<>(treatments, HttpStatus.OK);
24         } else {
25             Log.logger.info(LogHeader + "No treatments found for date: "
26             + today.getToday());
27             return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
28         }
29     }
30 }

```

```

22 }
23 } catch (Exception e) {
24     Log.logger.error(LogHeader + "some error occurred: " + e);
25     return new ResponseEntity<>(null, HttpStatus.
        INTERNAL_SERVER_ERROR);
26 }
27 }

```

Thread

Having an API to get all the measurements, create new tickets, etc. is a nice API just by itself. But that is not enough. The developers of this project decided that wanted to schedule new treatments for the plant automatically depending on the state of the abiotic factors.

In order to achieve that, it has been used the Java `ScheduledExecutorService` object. This thread allows to *schedule* tasks. Which is exactly what we are trying to achieve.

The data is read every approximately 5 minutes, but according to the experience of the developers and some local farmers, it is quite uncommon for the abiotic factor to change too drastically in that short period of time and therefore affecting the well-being of the plant.

This process is quite time consuming with large amount of data and too many devices to take care of. That being said, a better time period should be chosen than just *getting new data*. For that precise reason, the thread is programmed every 10 minutes.

```

1  @SpringBootApplication
2  public class Application {
3
4      public static void main(String[] args) {
5          SpringApplication.run(Application.class, args);
6
7          Log.logger.info("Starting Thread: TreatmentThread");
8          Log.logger.info("Starting Thread: Analyze pot measurements to
        schedule treatments");
9          // Create pool with 10 threads
10         int numThreads = 1;
11         final ScheduledExecutorService schExService = Executors.
            newScheduledThreadPool(numThreads);
12         // Create runnable thread
13         final Runnable treatmentAnalyzer = new TreatmentThread();
14         // Thread scheduling
15         int minute = 60;
16         schExService.scheduleWithFixedDelay(treatmentAnalyzer, minute,
            10 * minute, TimeUnit.SECONDS);
17         Log.logger.info("Thread Started: Currently Running");
18     }
19 }
20 }

```

The thread is launched after the rest of application starts up.

With that amount of time, the thread is only executed a few times an hour, and it does not overload the database nor the server, allowing other requests to be attended.

On the other hand, and due to problems with *@Autowired* dependencies needed for the execution (like database connection) the logic had to be implemented as a API call. The reason behind this is that the *Application.java* class (main Spring Boot class) cannot send the database dependency to the thread. Therefore, when trying to connect to the database to execute the query the instance turns out to be null/empty: so a **NullPointerException** is raised by the compiler.

The solution was to move the logic of the function to the API, and make a request through the thread to the API. Be advised, once the connection with the API is initialized, it is necessary to request some kind of action, or it will stay idle until it is either closed or a request is made. A simple request on the response code solves this problem.

```
1 public class TreatmentThread implements Runnable
2 {
3     @Override
4     public void run()
5     {
6         try {
7             callAnalyzerApi();
8         } catch (Exception e) {
9             Log.logger.error("Some error occurred " + e);
10        }
11    }
12
13    /**
14     * Call the API to execute a function that retrieves the data and
15     * creates
16     * treatments depending on their values.
17     * @return Boolean indicating the execution state of the remote
18     * call.
19     */
20    private Boolean callAnalyzerApi() {
21        try {
22            Log.logger.info("Requesting measurements");
23
24            // Creating Request
25            URL url = new URL("http://localhost:8080/api/analyze/
26            measurements");
27            HttpURLConnection con = (HttpURLConnection) url.
28            openConnection();
29            con.setRequestMethod("GET");
30            con.setRequestProperty("Content-Type", "application/json");
31
32            int responseCode = con.getResponseCode();
33            Log.logger.info("The process finished with code: " +
34            responseCode);
35
36            con.disconnect();
37        }
38    }
39 }
```

```

33         Log.logger.info("Returning measurements");
34
35         return true;
36     } catch (Exception e) {
37         Log.logger.info("Some error occurred while retrieving
38         measurements: " + e);
39         return false;
40     }
41 }
42 }

```

Once in the API code, several dependencies are necessary. The first group of dependencies are the *repositories* for:

- **MeasurementsRepository.** Use to recover the measurements under certain conditions.
- **PlantRepository.** Each plant has data on the ranges of abiotic factors that are safe for the plant to life healthy. That data is retrieved.
- **TreatmentRepository.** Sometimes, the factors on which the plant is surrounded are harmful for its well-being. Some actions might be taken or recommended to the user to mitigate the risk factor.

The other dependency for this function is **MongoTemplate**. *MongoTemplate* is the primary implementation of **MongoOperations**. Which means that any query can be requested using the appropriate syntax (not as easy as it looks, the Java MongoDB driver does not implement query functions as easy to use as the *pymongo* driver for Python).

It is going to be used to create a *find* function that filters measurements by specifying a range of dates (current time and yesterday at the same hour) and then ordering the result by:

- **device: (-1 DESC)** - All the devices must be grouped to later iterate them orderly and be able to make sense of the measurements.
- **date: (-1 DESC)** - The measurements also need to be ordered by date, from most recently to later on the day. Doing otherwise would mean comparing unrelated measurements.
- **hour: (-1 DESC)** - Same as before but with the hour.
- **plant: (-1 DESC)** - There might happen that the device changes the plant it takes care of in the range of values that are being read.

After this point, all that is left is to make any sense of the data collection. The current analyzer has a very basic behavior. In the future more through analysis could be performed. Right now, its behavior goes as follows.

1. Iterate through all measurements documents (and therefore all devices).
2. From the current device to analyze, get the last temperature.
3. If there is any temperature, check if it is within the safe values.

- (a) If is not safe, create a treatment for the user to notice that the plant is being burned or frozen.
4. Do the same with the internal humidity, and create a treatment, if the humidity is too low, to water the plants¹¹.
5. If any or none treatment was created in the whole iteration, that means that device's plant is currency safe.

```

1 @Controller
2 @EnableAutoConfiguration
3 @RequestMapping("/api")
4 public class AnalyticalController
5 {
6     @Autowired
7     MeasurementsRepository measurementsRepository;
8     @Autowired
9     PlantRepository plantRepository;
10    @Autowired
11    TreatmentRepository treatmentRepository;
12
13    @Autowired
14    MongoTemplate mongoTemplate;
15
16    @RequestMapping(value="/analyze/measurements", method=
17        RequestMethod.GET)
18    @ResponseBody
19    public ResponseEntity<Boolean> analyzeData()
20    {
21        String LogHeader = "[/analyze/measurements: analyzeData] ";
22
23        try {
24            Log.logger.info(LogHeader + "Requested");
25
26            DayRequest day = new DayRequest(new Date(System.
27                currentTimeMillis()));
28
29            // Query to return measurements from the last 24h
30            List<Measurements> measurements = getMeasurementsToAnalyze(
31                day.getYestarday(), day.getToday());
32            Map<String, List<MinMax>> plants = getMappedPlantRanges();
33
34            // Interpret measurements
35            Measurements last = null;
36            boolean done = false; // Does not need to look at more
37            measurements
38            boolean needsNext = false; // Needs more measurements
39            for (Measurements m : measurements) {
40                if (done && last != null) {
41                    done = last.getDevice().equals(m.getDevice());
42                }
43
44                if (!done) {
45                    if (!needsNext) {
46                        List<Double> temperatures = m.getLastNTemperatureExt(1)
47
48            ;

```

¹¹Auto watering unavailable due to the accident with the ESP-32s

```

43         if (temperatures != null) {
44             MinMax temperatureRange = getRange(plants.get(m.
45 getPlant()), "temperature");
46             // Freeze
47             if (temperatures.get(0) < temperatureRange.getMin())
48 {
49                 Log.logger.info("Prescribing Treatment: avoid
freezing plant: " + m.getPlant() + "(monitored by " + m.
getDevice() + ")");
50                 // Treatment parameters
51                 String plant = m.getPlant();
52                 String device = m.getDevice();
53                 String type = "scheduled";
54                 String title = "warmup";
55                 List<Action> action = Arrays.asList(new Action("
move-hotter", null));
56                 Date requestTime = new Date(System.
currentTimeMillis());
57                 Date actionTime = incrementTime(requestTime, "
minute", 30);
58                 String comment = "The temperature is low for
the plant. Move it to a place with more temperature.";
59                 // Create new treatment
60                 Treatment treatment = new Treatment(plant, device,
type, title, action, requestTime, actionTime, comment);
61                 // Save treatment
62                 treatmentRepository.save(treatment);
63                 Log.logger.info("Treatment Prescribed");
64                 // Burn
65             } else if (temperatures.get(0) > temperatureRange.
getMax()) {
66                 Log.logger.info("Prescribing Treatment: avoid
burning plant: " + m.getPlant() + "(monitored by " + m.
getDevice() + ")");
67                 // Treatment parameters
68                 String plant = m.getPlant();
69                 String device = m.getDevice();
70                 String type = "scheduled";
71                 String title = "cooldown";
72                 List<Action> action = Arrays.asList(new Action("
move-colder", null));
73                 Date requestTime = new Date(System.
currentTimeMillis());
74                 Date actionTime = incrementTime(requestTime, "
minute", 30);
75                 String comment = "The temperature is high for
the plant. Move it to a place with less temperature.";
76                 // Create new treatment
77                 Treatment treatment = new Treatment(plant, device,
type, title, action, requestTime, actionTime, comment);
78                 // Save treatment
79                 treatmentRepository.save(treatment);
80                 Log.logger.info("Treatment Prescribed");
81             }
82         }
83     }
84     // Treatments have been prescribed. Unless there

```

```

83         // is any change in the following treatment section
84         // this device does not need any more treatments.
85         done = true;
86     } else {
87         Log.logger.info("Needs next");
88         needsNext = true;
89     }
90
91     List<Integer> humidityInt = m.getLastNHumidityInt(1);
92
93     if (humidityInt != null) {
94         MinMax humidityIntRange = getRange(plants.get(m.
getPlant()), "humidity");
95
96         if (humidityInt.get(0) < humidityIntRange.getMin()) {
97             Log.logger.info("Prescribing Treatment: water plant
: " + m.getPlant() + "(monitored by " + m.getDevice() + ")");
98             // Treatment parameters
99             String plant = m.getPlant();
100             String device = m.getDevice();
101             String type = "scheduled";
102             String title = "irrigation";
103             List<Action> action = Arrays.asList(
104                 new Action("watering", Arrays.asList(
105                     new KeyValue("time", "5 s"),
106                     new KeyValue("flow", "max"))));
107             Date requestTime = new Date(System.
currentTimeMillis());
108             Date actionTime = incrementTime(requestTime, "
minute", 10);
109             String comment = "Water the plant due to low
pot humidity.";
110             // Create new treatment
111             Treatment treatment = new Treatment(plant, device,
type, title, action, requestTime, actionTime, comment);
112             // Save treatment
113             treatmentRepository.save(treatment);
114             Log.logger.info("Treatment Prescribed");
115         }
116     } else {
117         Log.logger.info("Needs next");
118         needsNext = true;
119         done = false;
120     }
121     // Needs measures from different hours
122 } else {
123     Log.logger.info("Needs more information");
124 }
125 }
126 // Gets next measure
127 last = m;
128 }
129
130 Log.logger.info(LogHeader + "Successful");
131 return new ResponseEntity<>(true, HttpStatus.OK);
132 } catch (Exception e) {
133     Log.logger.error(LogHeader + "some error occurred: " + e);

```



```

134         return new ResponseEntity<> (false, HttpStatus.
135             INTERNAL_SERVER_ERROR);
136     }
137 }
138 private List<Measurements> getMeasurementsToAnalyze(Date min,
139     Date max) {
140     List<Measurements> measurements = new ArrayList<Measurements>()
141     ;
142     Log.logger.info("Measurements between dates and ordered
143         requested");
144     Query query = new Query();
145     query.addCriteria(Criteria.where("date").gte(min).lte(max));
146     Sort deviceSort = Sort.by(Sort.Direction.DESC, "device");
147     Sort dateSort = Sort.by(Sort.Direction.DESC, "date");
148     Sort hourSort = Sort.by(Sort.Direction.DESC, "hour");
149     Sort plantSort = Sort.by(Sort.Direction.DESC, "plant");
150     Sort groupBySort = deviceSort.and(dateSort).and(hourSort).and(
151         plantSort);
152     query.with(groupBySort);
153     // Return measurements
154     mongoTemplate.find(query, Measurements.class).forEach(
155         measurements::add);
156     return measurements;
157 }
158 private Map<String, List<MinMax>> getMappedPlantRanges() {
159     List<Plant> plantsList = new ArrayList<Plant>();
160     // Get all plants
161     PlantRepository.findAll().forEach(plantsList::add);
162     // Map Ranges
163     Map<String, List<MinMax>> plants = plantsList.stream().collect(
164         Collectors.toMap(Plant::getId, Plant::getLifesafeRange));
165     return plants;
166 }
167 private MinMax getRange(List<MinMax> lifesafeRanges, String type)
168 {
169     MinMax desired = null;
170     for (MinMax range : lifesafeRanges) {
171         if (range.getType().equals(type)) {
172             desired = range;
173             break;
174         }
175     }
176     return desired;
177 }
178 private Date incrementTime(Date toIncrement, String type, int

```

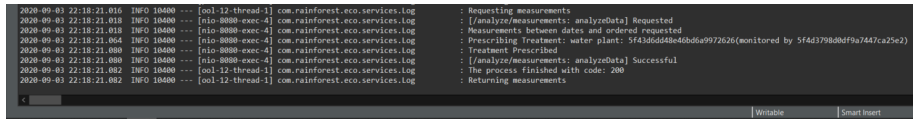
```

183     amount) {
184         Calendar cal = Calendar.getInstance();
185         cal.setTime(toIncrement);
186
187         if (type.equals("day")) {
188             cal.add(Calendar.DATE, amount);
189         } else if (type.equals("hour")) {
190             cal.add(Calendar.HOUR, amount);
191         } else if (type.equals("minutes")) {
192             cal.add(Calendar.MINUTE, amount);
193         }
194         return cal.getTime();
195     }
196 }

```

After a successful execution of the thread, a new treatment (or several) are created. Should be taken into account that, for now, the thread does not check if there are any treatments programmed so that could be one of its improvements in the near future¹².

An example of a new treatment being created would be the following [6]:



```

2020-09-03 22:18:21.816 INFO 10000 --- [ool-12-thread-1] com.rainforest.eco.services.log : Requesting measurements
2020-09-03 22:18:21.818 INFO 10000 --- [nio-8080-exec-4] com.rainforest.eco.services.log : [/analyze/measurements: analyzeData] Requested
2020-09-03 22:18:21.818 INFO 10000 --- [nio-8080-exec-4] com.rainforest.eco.services.log : Measurements between dates and ordered requested
2020-09-03 22:18:21.864 INFO 10000 --- [nio-8080-exec-4] com.rainforest.eco.services.log : Prescribing Treatment: water plant: 5f43dd4de4b8da9972626(monitored by 5f4d3798dbf9a744/ca25e2)
2020-09-03 22:18:21.880 INFO 10000 --- [nio-8080-exec-4] com.rainforest.eco.services.log : Treatment Prescribed
2020-09-03 22:18:21.880 INFO 10000 --- [nio-8080-exec-4] com.rainforest.eco.services.log : [/analyze/measurements: analyzeData] Successful
2020-09-03 22:18:21.882 INFO 10000 --- [ool-12-thread-1] com.rainforest.eco.services.log : The process finished with code: 200
2020-09-03 22:18:21.882 INFO 10000 --- [ool-12-thread-1] com.rainforest.eco.services.log : Returning measurements

```

Figure 6: Treatment created after plant with little water.

Another optimization would be to make the read (*find*) requests on secondary nodes only. If those nodes are updated oftenly (for example from 5 to 10 minutes delay from the primary), it could highly reduce the impact if the primary is receiving too many requests. Both the refreshing period and the read concern could be easily implemented. Although this beta version did not count with enough time for such enhancement.

Write operations must be done on the primary only. Secondary nodes do not allow write operations for consistency reasons.

6 Front-end

It is known that separating front-end and back-end can provides many benefits depending on the size of the project. Due to the fact that this project will be developing in the future, the decision was to separate both element.

It was decided to generated this project with Angular CLI version 10.0.4 for the front-end part as it is a common used framework which is designed to work

¹²Could be done by, when a new treatment would be created, checking if there are any soon-to-be-applied treatments.

with every other tool in an interconnected way. Also the fact that one of the team member has used before was a key to choose it.

Angular apps are built using TypeScript language (which is a superscript for JS that ensures higher security thanks to its typing system). Also it is easier to perform maintenance task.

Another benefit is the fact that Angular don't need any additional getter and setter functions because every object uses POJO, which enables object manipulation by providing all the convectional JS functionalities.

Getting start

To use the Angular framework, it is necessary to install Node.js (disclaimer, for using Angular it is required a current, active or maintenance LTS version). To check if Node has been installed properly, run `node --version` in a terminal window. Once it is installed, run the following commands to obtain Angular CLI:

```
1 npm install -g @angular/cli
```

Downloading the project and deploy the server

When it is downloaded an Angular app, the first step to be able to run the project is run `npm install --save-dev @angular-devkit/build-angular` otherwise, the app will return the following error:

```
1 Could not
2 find module "@angular-devkit/build-angular" from [PATH_PROJECT].
3 Error: Could not find module "@angular-devkit/build-angular" from
4   ... then, run
5   ng build
6 which will created a folder called node_modules, this directory is
7   only for build tools. Inside that folder it can be found
8   different modules which are defined inside package.json file.
```

When all dependencies has been install, run `ng serve --open` that will automatically launch the server and open a tab inside the default explorer with the URL . In case there is any change in the source file, the app will automatically reload.

By default, Angular launch it services in port number 4200 but this can be change in a temporal way using the command `ng serve --port <PORT>` or modifying the angular.json like this:

```
1 "serve": {
2   "builder": "@angular-devkit/build-angular:dev-server",
3   "options": {
4     "browserTarget": "eco-front:build",
5     "port": 9200
```

Starting the project and general structure

To create a new workspace use `ng new <APP_NAME>`

Inside the project, it can be found the app folder (where is the code), assets (to store images) and environment (where configuration files are stored if there are different environment where the app will be develop). Inside the app folder, there are components, services, models and helpers.

For creating a new component, run `ng g c <COMPONENT_NAME>`. Inside this new component it can be found a CSS, HTML, TypeScript (.ts) and a test file. CSS files are used for change styles, HTML is the template and .ts file is where the logic of the component is develop. That command also adds the component inside app.module.ts file automatically.

For creating a new service, run `ng g service <SERVICE_NAME>`. Services are used to share information among classes. Running that command will create an .ts file which contains an injectable decorator (this indicates that the class participate in the dependency injection system).

Models are used inside Angular to represent collections inside the database, but is more similar to the data that will be send or receive. It is just an structure to have a better organization and also makes everything easier.

Finally, the helper which is not supported at the moment but it was the part which takes care of the token storage. This part will be longer explained before in more detail.

Eco Webpage

The webpage is mainly formed by the following pages:

- Home page: it contains information about who is rainforest and summarize the main idea of this project
- Authentication page: it is where the user can login or sign up. This page is formed by a component called authentication-card where two other components are located. This page use the authentication service, in order to verify the identity of the user. It gather the data and send it to the back logic, if the data is equals to the information store in the data base it will received a token that will identify the session with the user, so the client area will at this point be accessible to the user. The other component, the sign up use that service to send the data to the back and create a new user. Further information about this page can be found in the following section.
- Product page: It is formed by three components (cart-list, filters, product-list). This page shows the products store in the database with their corresponding image, it also has a price filter for displaying some products

and a shopping bag. This page use the product service where the actions *getProducts()*, *getProductsBetweenPrices()* and *buyProducts()* are codify. For understanding better how the works it has been design:

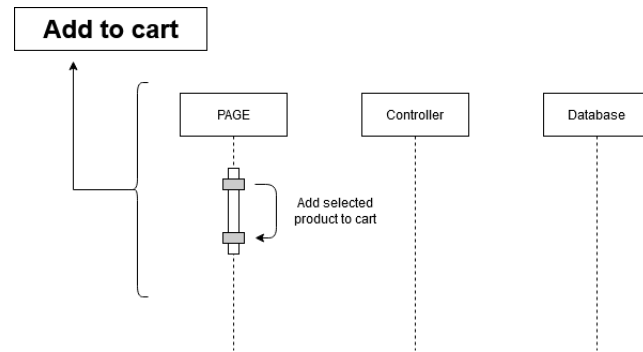


Figure 7: Add to cart user iteration

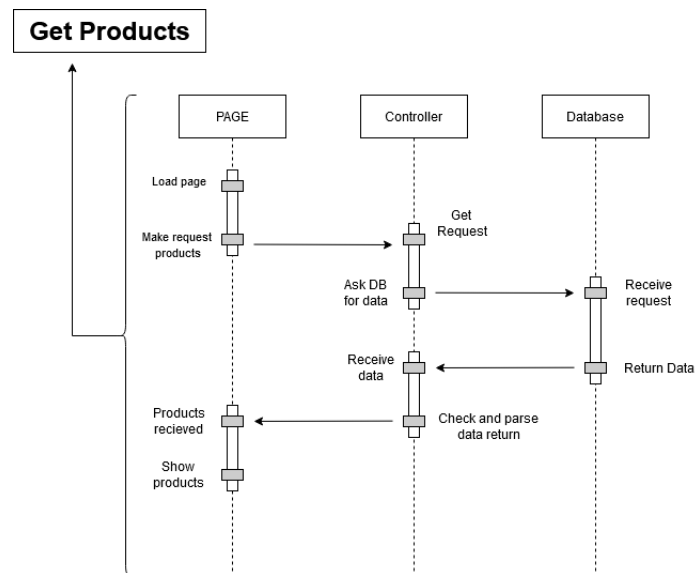


Figure 8: Getting products

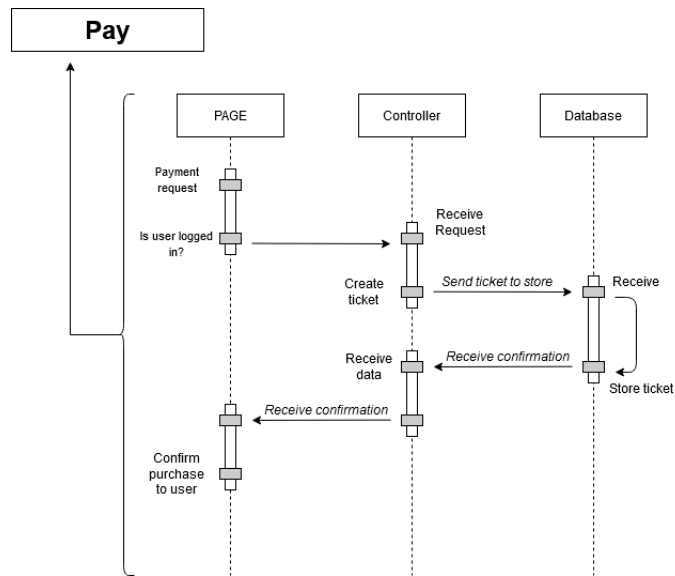


Figure 9: Paying procedure

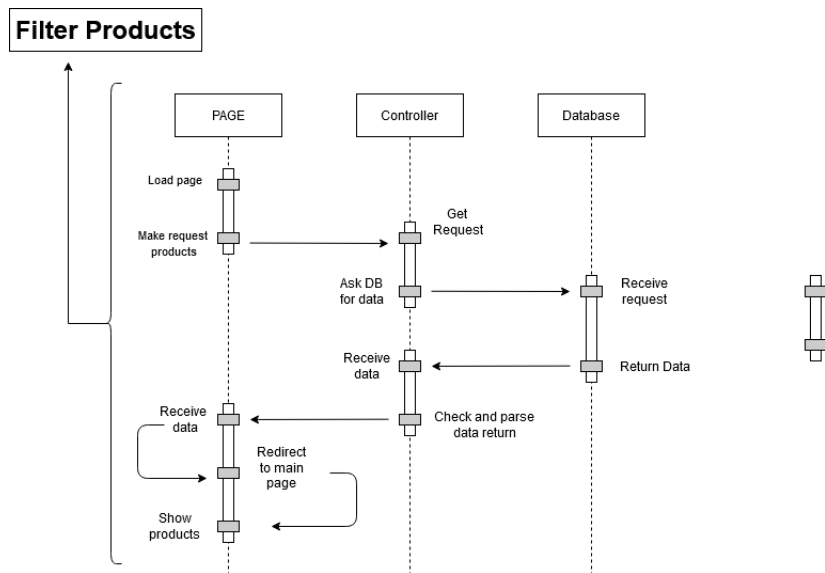


Figure 10: Displaying products by filter

- Client Area page: That is the most interesting page as it is where personal info, devices associated with the user, notifications and graphics which displays the different measurements. Inside this page it can be found five different components (charts where there are two other components, device-list, measurements, notifications and profile info). This page

use the client-area service, where the actions `getDevices()`, `getPlant()`, `getReminders()`, `getTreatments()` and `getMeasurements()`

- Error pages: In order to display the most commons error (as 202 accepted, 401 unauthorized, 403 forbidden which is not used as it has relation with guards, 404 not found) it has been created some graphical representation for them. In a nearly future it will be added some other pages, as the funny and famous error 418, or the 5xx errors.

Security

Guards

Front-end

Another issue that will be implemented in the future are the guards, which allows to store a jwt that identify the session with the user in a security way. It exist different types of guards:

- `canActivate`, which checks if a user can visit a route
- `canActivateChild`, checks if a user can visit a route child
- `CanDeactivate`, checks if a user can exit a route.
- `Resolve`, performs route data retrieval before route activation.
- `CanLoad`, checks to see if a user can route to a module that lazy loaded.

It was tried to implement a `canActivate` guard for the client area in order to force the user to log in if they want to have access to it. For doing that it is necessary to implement a service which implements the `canActivate` function. Then, this guard is associated with a route inside `app.routing-module.ts`. As guards has given many problems in the back side, this idea is implemented in other way. Inside `client-area-card.component.html` it can be found a `ngIf` that in case the user is login in, displayed the page otherwise it is hidden.

Further information about how to implement guard can be found in this tutorial or in the official documentation

Back-end

Spring boot has some dependencies for developing a JWT authentication as `io.jsonwebtoken`, which must be included inside the `pom.xml` and also it will be necessary to include the `spring-boot-starter-security`. To sum up, it will be develop two operations:

- Generating a JWT, which expose a POST API with mapping `/api/login` in this project. It will be the responsible to create the token in case the user is correctly login.
- Validating a JWT, in case the user tries to access to the protected url, the access will be denied. As it has been mentioned before, the front service ask to the back if it can gives access to the user.

The following diagram explains in more detail the how this should work (this diagram has been taken and modified for this page):

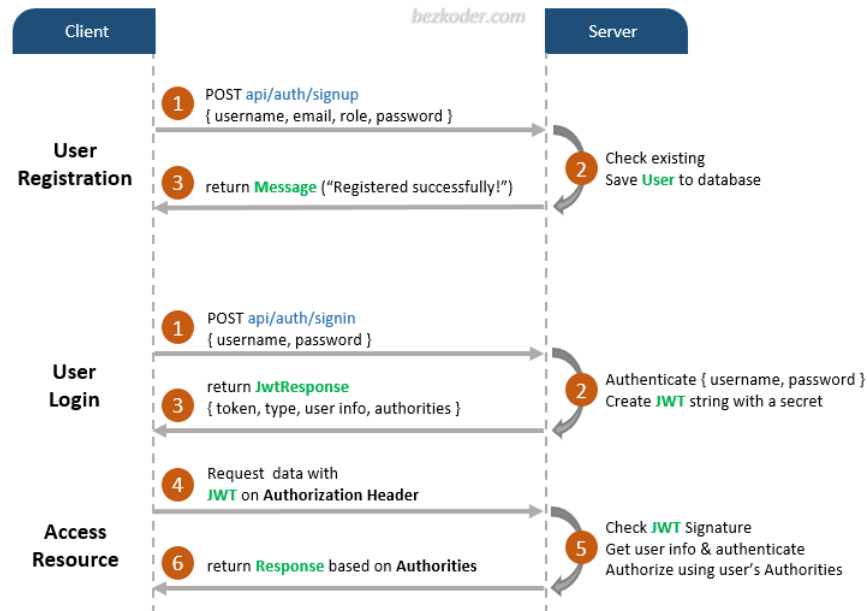


Figure 11: Login interaction

Summing up, for develop this operation it will be necessary to create:

- Controllers which handles signup/login request and authorized request
- Interfaces which has interfaces that extend Spring Data JpaRepository (the responsible to interact with database).
- Models, which as same as happend in the front part are structures to definde the request and response.

Before following different tutorial, the webpage always returns us a 403 error. This error was supposed to be resolve adding a root user inside the app.properties as it is explain here

7 Tools

Throughout the project many different tools and technologies have been used for implementing the software, testing if the code works, etc. These are the ones that have been used the most:

Draw.io

It is a free online website that allows to create custom schemas. Does not matter if it is a database (relational, non-relation schema), a sequential schema, etc. It also provides a small search engine to look for icons that fit best to you schema.

It allows export the schema to a .PNG file, making it easier to use in LaTeX or Markdown.

GitHub

Used as a software version controller for the project. It made it quite easy to track all the process, create different branches to work on separate parts of the project without conflicts, etc.

It also provides the tool Github Projects, which allows to create a Kaban board. That helped integrating the development methodology and code development all in one place. And, therefore, make things simpler, as the issues could directly be related to the code.

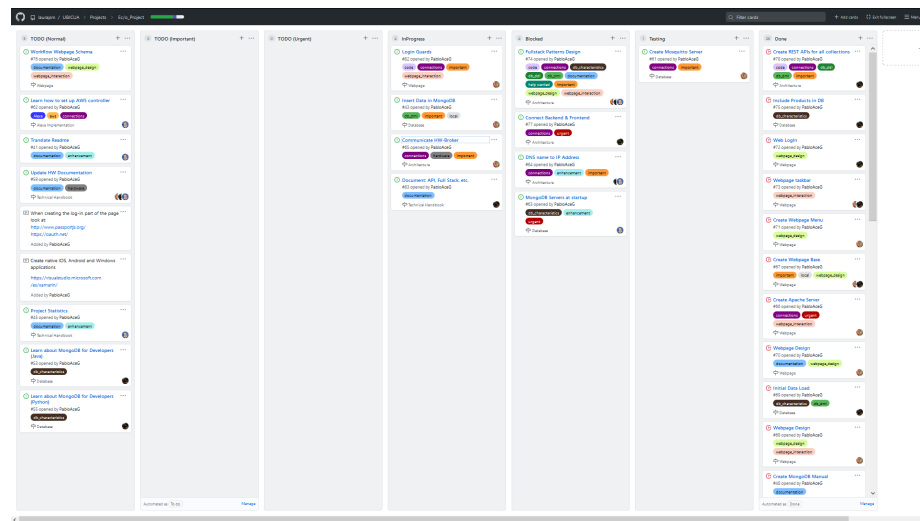


Figure 12: Screenshot of the Kanban board at some point of this project.

In order to document the contents of the repository, GitHub markdown was used. Every piece of information that could be considered as necessary to understand the project has been included into the README.md files

Overleaf

Overleaf has been used as the IDE to create and share the PDF documents of this project. As it can be guessed, these documents have been written using LaTeX.

It allows to customize every aspect of the documents without having to worry for a bad formatting error or visual mistake. Although it might also cause some unexpected problems due to packages incompatibilities, bad syntax, etc.

This platform also allows to live share and change the document live streaming, and therefore making it easier for all the group members to create new content at the same time.

Postman

Postman is a tool that allows to test API tools (download [here](#)) by generating request to the given API. It allows making the following request:

- GET
- POST
- PUT
- PATCH
- DELETE
- COPY
- HEAD
- OPTIONS
- LINK
- UNLINK
- PURGE
- LOCK
- UNLOCK
- PROPFIND
- VIEW

Despite the many types of requests it allows to send and perform, only four of them are used in this project: GET, POST, PUT and DELETE.

It has been used to test that the REST API for the `eco` MongoDB works as it should. The file with the tests used by the developers can be found by importing the file `src/main/resources/rainforest-eco-api.postman_testing.json` into Postman.

Beware that almost any of the test is static. An adjustment of some of the bodies (`body` option), or some `ids` might need to be changed in order for the test to work, as they are data dependent.

Other options can be used, although it is not recommended to do so. One of the previous options considered was the webpage `postwoman.io/`. It was discarded, as it resulted quite troublesome during the development and testing of the API.

As a last remark, notice that postman allows to send a body with the GET requests. This is not allowed in other requesters, like for example, the `HttpClient`

module in Angular. Should be taken into account, not know this incompatibility can lead to lost hours of debugging the RESTful API and searching on how to insert a body into a GET request.

Development IDE

Mainly, two different environments were used to develop this project.

The first of them was Visual Studio Code, mainly chosen for its compatibility with multiple programming languages and numerous plug-ins that made the programming experience smoother than having to use multiple IDEs.

Its integrated terminal and powerful functionalities, such as refactoring, formatting, column selection, different terminals, etc.; also had a big impact in the amount of time spent developing the project. It positively, mitigated the impact of changes born out of the agile methodology used and the need to change parts already implemented bits to meet the requirements.

It came in specially handy for the website development. It was possible to compile the front-end application while remaining inside the IDE and developing more code.

The second Environment used was Eclipse, for developing the back-end part of the application - Spring Boot. Eclipse is used in most of the tutorial available online, due to its native compatibility with Java.

That compatibility allows to execute the application, without the need for any external or internal terminal. Which means, in example, that the log messages when making a request were directly showed in the console at the bottom of Eclipse's UI.

8 Testing

Unit Test

To understand this type of testing properly, it is necessary to understand the concept of Unit: "A Unit is the smallest testable portion of system or application which can be compiled, loaded, and executed." It allows to test each module of the application separately.

That way, each part of the application is tested separately, testing if each component fulfills their functionalities.

This is the case of Postman and the REST API. The process was first creating the corresponding API function with its URL. Once the function has been completed, it is time to test it using Postman:

1. Select the type of request to be done. In this project:

- (a) GET
- (b) POST
- (c) PUT
- (d) DELETE

2. Write the URL to the API.
 3. Write the Body request, in case it is necessary.
- After making the request, it would return a result like the following 13:

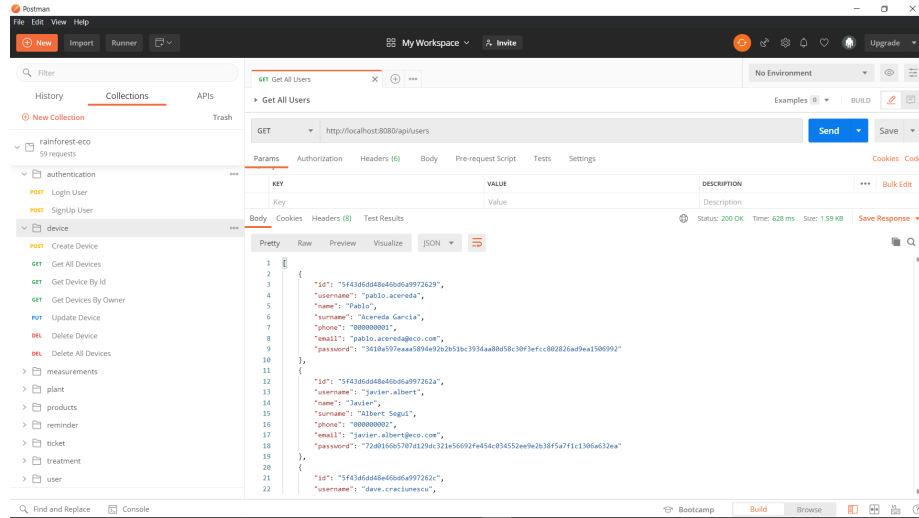


Figure 13: Testing the API

At the beginning of the development this helped finding out that a problem connecting to the API was not due to the front-end logic, but due to a bad configuration of the REST API service.

The cause actually was due to a missing annotation: `@RequestBody`. Annotation indicating a method parameter should be bound to the body of the web request. The body of the request is passed through an `HttpMessageConverter` to resolve the method argument depending on the content type of the request.

Integration Test

This type of testing involves putting together modules that independently work fine. In this case, for example, instead of testing the REST API, it would be tested the RESTful API. Using web debugging tools to know what information was retrieved and the packages sent in that process would do the trick.

Using this method, a parsing error in the client area page (`/profile`) where the `humidityInt` graph did not have any value, as the values that the page was trying to print were actually objects, not valid measures that could be printed 14.

Last 24h measurements

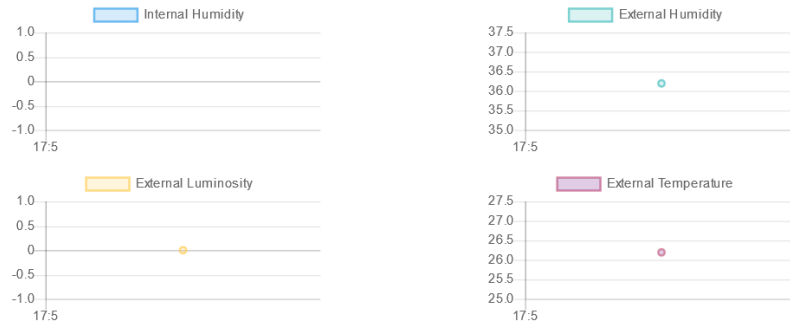


Figure 14: Testing the RESTful API

System Test

It is a test carried out by the development/testing team. All the application is put together. It allows to check the system's compliance towards the requirements; and also, overall interaction between the components:

4. Load
5. Performance
6. Reliability
7. Security Testing

When all the application was put to test, evaluating if everything was working as it should, after completing the previous test, no alarming error was obtained.

Therefore, it could be safe to assume that part of or most of this achievement was thanks to the Unitary Testing and the Integration Testing.

Although nothing failed, performance and security testing could not be done due to time issues.

Acceptance Test

An acceptance test is a meeting/presentation held to find if the requirements of the project match their objective. Therefore, it is not carried out by any development/testing team, but by the final user or customer.

In this project's case, this test is going to be carried out on *September 5th*, 2020. It is going to be done a presentation on the project, explaining the objective of the project, the business model and part of a technical interview to show the how it actually works.

9 Future Work

As a final conclusion we would like to emphasize that although the project has had ups and downs such as problems with the use of the virtual assistant due to de malfunction of the ESP or the pest control attempt that was thought but for economic reasons and the attempt to develop a forecast of adverse storms or extreme weather conditions, that was not implemented for reasons of time and also we do not consider necessary in a beta version just yet, they have caused several changes in the development of the project.

But all these problems are nothing more than easy to overcome inconveniences. We have managed to develop a fully functional device, which measures several fundamental values for the care of plants and that thanks to a simple and intuitive web page and a very complete API implementation we can say that the project is on the right track.

As future plans we intend to amend all the mistakes of the past. So we can assure that from Rainforest we will continue to improve the product with the intention of implementing what has not been possible until now. Like the use of the virtual assistant of Amazon Alexa through the re-implementation of the ESP micro-controller, the connection through MQTT by the ESP WiFi module or even in the distant future the development of our own micro-controller. Without moving away at any time the line of work that has been carried out.