nodebots

javascript and robotics in the real world

By Wilson Mendes

Nodebots - Javascript and robotics in the real world

Wilson Mendes

This book is for sale at http://leanpub.com/nodebots-javascript-and-robotic-in-the-real-world

This version was published on 2020-05-12

Contents

About the author

GDE (Google Developer Expert) AngularJS, GDG Salvador organiser, passionate about technology and active in communities focused on web development, including AngularJS, JavaScript, HTML5, CSS3, workflow, web performance, security and *Internet of things*. Participates in the organisation of events, lectures at conferences in Brazil and in other countries and contributes to few open source projects.

Introduction

This book is for anyone who wants to take the first steps on Nodebots or has an interest in deliving into some concepts that are poorly demonstrated on the subject.

It will be shown contents with simple and low-cost sensors, however relating the sensors with real integrations of a nodebots application, such as integration between external APIs from events reading some sensor data, among others.

In addition, the book is covering important topics in the test questions, showing good practices for testing coverage and how to test with quality and performance, with integrations in external services for automation of code quality and *build* validation tasks.

Another point is the evolution of the application with a focus on the architecture itself. How to use software architecture standards in any type of project and how to take advantage of each of them applying to your system.

Acknowledgements

I would like to say thank initially my mother (who has played the role of mother and father for many times), this wonderful person who teaches me many of the values that I take with me until today, being my example to this day and my brother Leonardo AraÃºjo who was always with me.

I would like to thank my wife, Nilana Rocha, for all her support. It was one of the crucial factors and without a doubt, this book would not be a reality. Thank you so much for everything.

A great time to remember all my friends from LinguÃ�gil Group and #horaExtraSSA from Salvador-BA. You were people who motivated me to be the professional that I am today, encouraging and motivating me to seek and share knowledge. Thank you all. For some Recife's folks who are amazing people and presented me with cheese curd, macaxeira (or aipim?) and much love.

Finally, I would like to thank Nelson Glauber[1] and Fagner Brack[2] for their support in reviewing the contents of this book and for everyone who Have contributed directly or indirectly to the composition of this book, such as Rafael Gomes[3], Juliano Bersano[4], Sarah Atkinson[5] and several others who supported me in this process and suggested improvements at this stage.

Thank you all.

Wilson Mendes

[1] https://twitter.com/nglauber
[2] https://twitter.com/fagnerbrack
[3] https://twitter.com/gomex
[4] https://twitter.com/julianobersano
[5] https://twitter.com/SarahvAko

Nodebots and microcontrollers

What are nodebots?

NodeBots is a term used to define the concept of control over open hardware, an electronic hardware designed and offered in the same manner and with the same licenses as free code software, sensors and other electronic components using NodeJS. And you can use several elements: from sensors, servo motors[6], wheels, motion detectors, cameras, LED displays, audio players and more.

In some moments the concept of Nodebots is directly connected to the concept of IoT - Internet of Things. Internet of Things * is a technological revolution in order to connect everyday electronic devices such as home appliances and even industrial machines and devices with internet access and other technical innovations in important fields such as home automation from Sensors.

The whole idea of NodeBots evolved according to increasing capabilities in NodeJS and the effort of some developers like Nikolai Onken, J□□rn Zaefferer, Chris Williams, Julian Gautier and Rick Waldron who worked to develop the various modules we use in NodeBots applications nowadays. The node-serialport[7] module, created by Chris Williams, was the kick-start because it allows access to devices using low-level serial ports.

Julian Gautier then implemented Firmata[8], a protocol used to access microcontrollers, such as Arduinos, via code using JavaScript for communication between physical components.

Rick Waldron went a little further. Using the Firmata library as a base, he created a framework to assist in building Nodebots and Internet stuff called Johnny-Five.

The Johnny-Five framework makes it easy to control various components, from LEDs to various other types of sensors in a simple and practical way. This is what many NodeBots now use to achieve some awesome feats!

Microcontrollers

When we talk about nodebots, we are indirectly mentioning microcontrollers. A microcontroller is a smaller and simpler computer that has a simple programmable physical circuit board (we'll call it pins, inputs, etc.) that can detect multiple inputs and outputs.

An Arduino is one of the several types of microcontrollers, being one of the most common for experiments and validations between software and hardware integration. There are other types of microcontrollers too, which can be powered by Node including:

[6] https://pt.wikipedia.org/wiki/Servomotor
[7] https://www.npmjs.com/package/serialport
[8] https://www.npmjs.com/package/firmata

- Raspberry Pi[9];
- Tessel[10];
- Espruino[11];
- BeagleBone[12];

In this book, I will use Arduino UNO[13] in the examples, but feel free to use the microcontroller of your choice.

NodeJS

NodeJS is a JavaScript execution runtime built on the Javascript engine Javascript V8, enabling the use of Javascript in other environments beyond the web and with an important aspect of non-blocking input model usage and Event-driven data output. It aims to help programmers to create high-scalability applications such as web servers with concurrent connections, asynchronous scripts and even integration with electronic components, that is our case.

It was created by Ryan Dahl in 2009, and its development is maintained by the community and the Node Foundation, of which companies such as IBM, Google, Red Hat, Joyent, among others.

Installing on Windows

Installing NodeJS on Windows is quite simple. One way is to visit the official project website[14] and download the installer in the format, click on the installation options and finish the installation. When finished open your command prompt by accessing the Windows command prompt from the "Run> cmd" command and, after starting the program, enter the following command:

```
1 $ node -v
```

It should display at the prompt the current version of NodeJS on your terminal. This completes the installation in the Windows operational system environment.

Installing on Linux and Mac OS X

For Linux and Mac OS X systems you can use various formats such as downloading the node on the site (as we did on Windows installation), via package manager by operational system, but a way to unify the installation format for UNIX platforms is to use the NVM - Node Version Manager[15] which is a version manager of NodeJS based on bash script.

It's something very simple to install as well. You can install locally via Curl or Wget, respectively:

[9] https://www.raspberrypi.org/
[10] https://tessel.io/
[11] http://www.espruino.com/
[12] http://beagleboard.org/bone
[13] https://www.arduino.cc/en/Main/ArduinoBoardUno
[14] https://nodejs.org/en/download/
[15] https://github.com/creationix/nvm

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.35.3/install.sh | bash
```

Wget:

```
$ wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.35.3/install.sh | ba\
sh
```

> Packages like CURL and WGET may not be installed on your operating system by default. If necessary, check the best method for your operating system or access the NVM repository on Github to check the installation steps or possible problem solutions.

Next, you should open your file that stores the default configuration of your terminal, which can be located in `~/.bash_profile`, `~/.zshrc`, `~/.profile` or `~/.bashrc`, and add these lines at the end of this configuration file so that you will load NVM next time you access the command line.

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh" # This loads nvm
```

With this, as soon as you recharge your terminal the NVM will be available. Now just install the version of NodeJS of your choice. In this book, we will use version 12.16.2.

```
$ nvm install 12.16.2
$ nvm use 12.16.2
$ nvm alias default 12.16.2
```

After these commands NVM will download the specific version of NodeJS, making it accessible via the terminal. To verify that the command completed successfully, enter the command:

```
$ node -v
```

The result should be `v12.16.2`. If this was the return of your command, you're all set for the next steps. If you have any problems make sure that the NVM loading code has been inserted into the configuration file of your terminal and start another instance of your terminal.

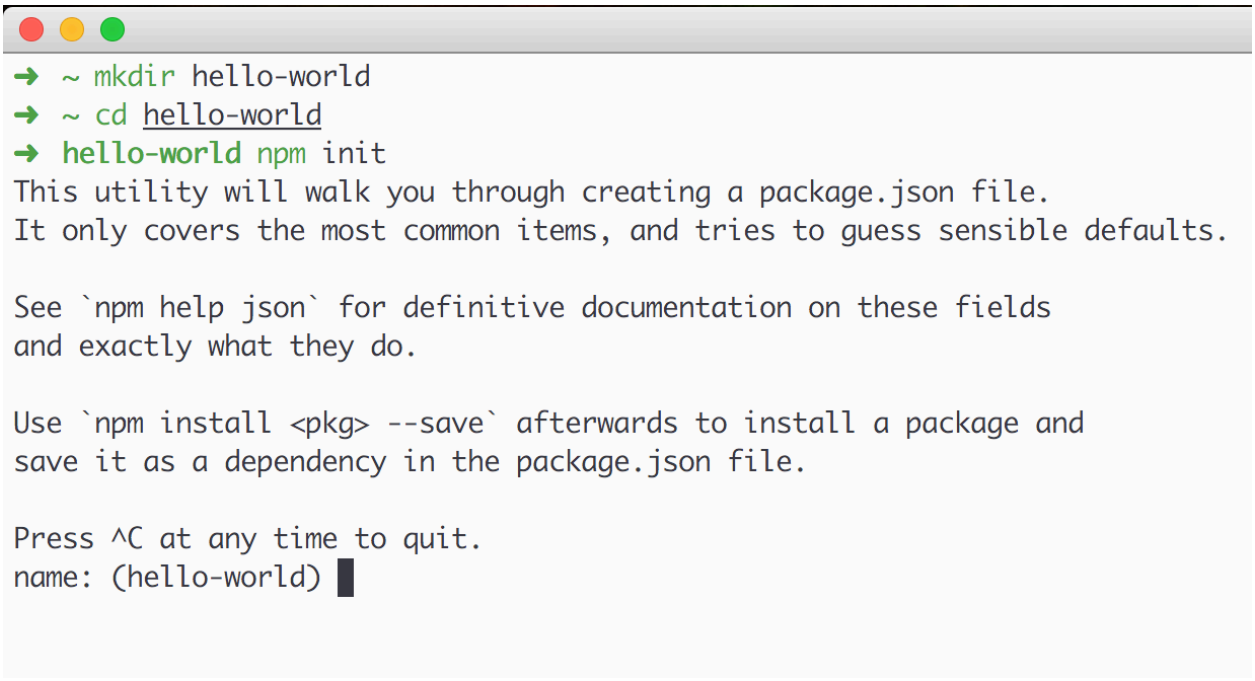
A file with the commands in this topic was created for the installation of NVM and Node with the version used in this book. If you want to use it, please download nvm-install.sh[16] file in this gist link.

[16] https://gist.github.com/willmendesneto/4c951413bacbb8850837a53bcdada30d

Managing dependencies with NPM

Starting your project and knowing the package.json file

As a first step, let's create the `"hello-world"` folder and add some initial information in the project. For this we will use the command npm init[17].



Npm init command

To continue you will need to answer some basic questions about the project, such as: - Package name; - Project version; - Description of the project; - Name of the project's main file. This will be produced at the end of your project, after all minification, obfuscation and other optimisation procedures of your javascript code;

You can rest assured that none of them is mandatory. If you do not know or do not want to respond now, just hit the "Enter" key until the end or run this command with `-y` flag, that means you are answering `yes` for all those questions. It will then create a `package.json` file with this information from your repository.

[17] https://docs.npmjs.com/cli/init

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this ok? (yes)
```

Command output

Adding Packages

Now that we have our package.json[18] with all the basic configurations of our repository, we will install our first package to integrate with our project!

NPM functions are the official package manager for NodeJS applications, being the largest ecosystem of libraries and *open source* modules in the world.

With it you can add packages in your application from the command npm install[19], informing the name of the package published in the official site of npm and the format that we want to save the Package in the application. We have some options for adding packages, which are:

- locally as a development dependency: the package will be installed locally and accessible in the `node_modules` folder and the package information will be saved in the `devDependencies` key of your JSON file. To use this option add the flag `--save-dev`;
- locally as a dependency of your project: the package will be installed locally and accessible in the node_modules folder and the package information will be saved in the dependencies key of your JSON file. To use this option add the flag `--save`;
- globally: the package will be installed with global scope and accessible in any other project. To use this option add the flag `--global`;

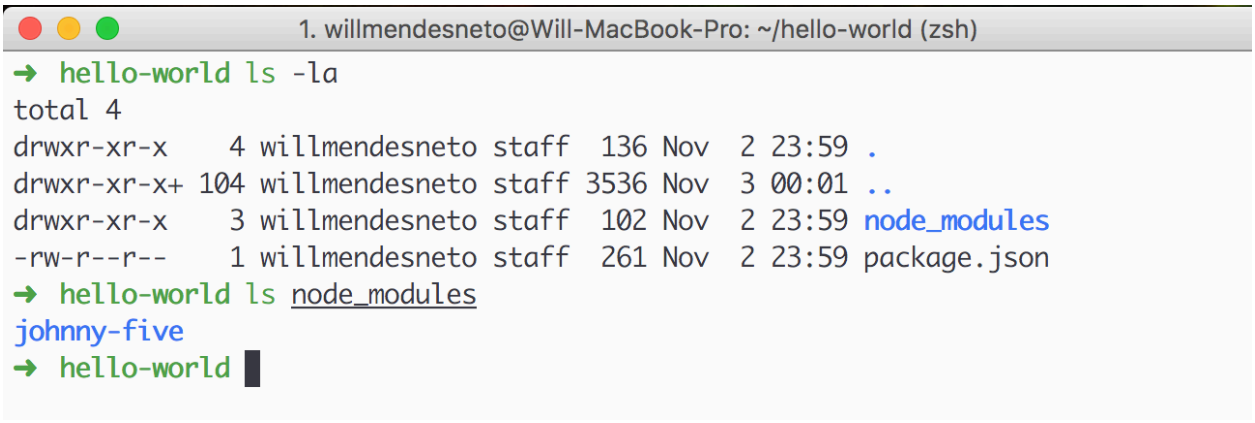
[18] https://docs.npmjs.com/files/package.json
[19] https://docs.npmjs.com/cli/install

> Packages saved as a development dependency and globally won't be used when you publish your project, so use these options carefully.

For our initial project we will install the Johnny-Five framework as one of the development dependencies by running the command:

```
$ npm install --save johnny-five
```

You may notice that we now have some new files in our project folder. Firstly, the folder `node_modules` was created and inside it, we have our package installed successfully.



Listing the folder `node_modules`

Another new content information were added in the package in the `dependencies` block of our `package.json` file.

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "johnny-five": "^1.4.0"
  }
}
```

> NPM has other standard commands and we can use them in our application. If you would like to know more about these commands, go to the page about these commands in the official NPM documentation[20].

Adding NPM Commands

NPM is a very interesting and flexible tool, with the possibility of creating specific commands executed from `npm run your-command`.

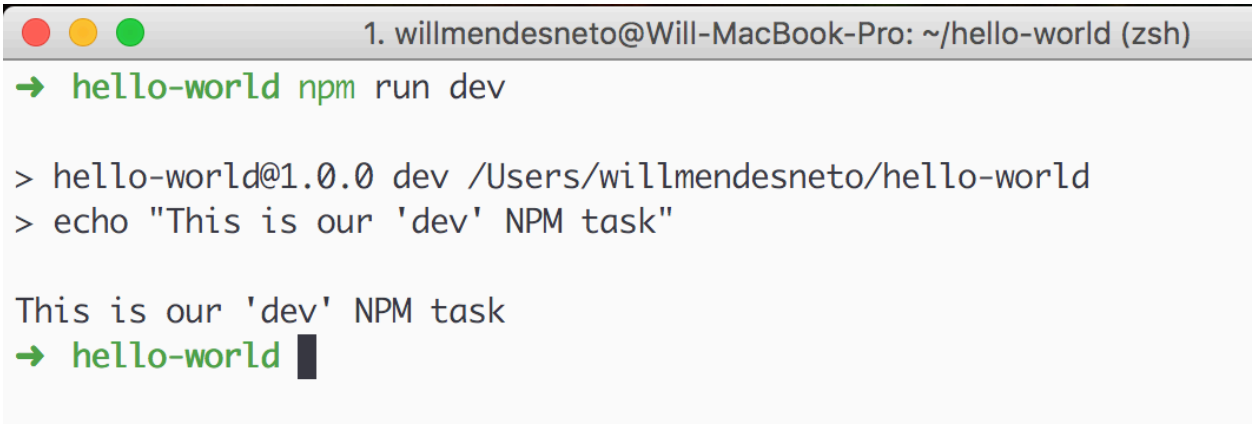
In our example, we will create a sample command to run our code. Let's open our `package.json` in our editor/IDE ideally and add the following code:

```
1  {
2    "name": "hello-world",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "dev": "echo \"This is our 'dev' NPM task\"",
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "johnny-five": "^1.4.0"
14   }
15 }
```

Notice that in line 7 we add our new NPM command through the JSON scripts field. Notice that we already had the "test" command which is a standard npm command and it's in the same area as our new command, as this is the default area for adding commands to be executed by NPM.

To run the command, just access the terminal or command prompt and type `npm run dev`. The result will return as follows:

[20] https://docs.npmjs.com/misc/scripts



Npm run dev output

Arduino

Arduino ... ardu-WHAT?

Arduino is an open-source platform based on easy-to-use and plug-and-play hardware and integration with sensors from the software. Being a fully malleable and open platform anyone can use it in projects of the most diverse as simple data checks received by light sensors, temperature, humidity, home automation and more.

Some advantages are:

- Cost: the value of an Arduino is very low. Currently, the cost of an Arduino UNO[21] is something around $ 50.00 and Arduino Nano[22] costs between $ 15.00 and $ 20.00, being more or less according to the model of your choice;
- Cross-platform: Arduino is compatible with all operating systems and platforms;
- Simple: It does not require of those who will manipulate it a vast knowledge in electronics. Just have a basic notion of development and you can already do things pretty cool;

About Open source hardware

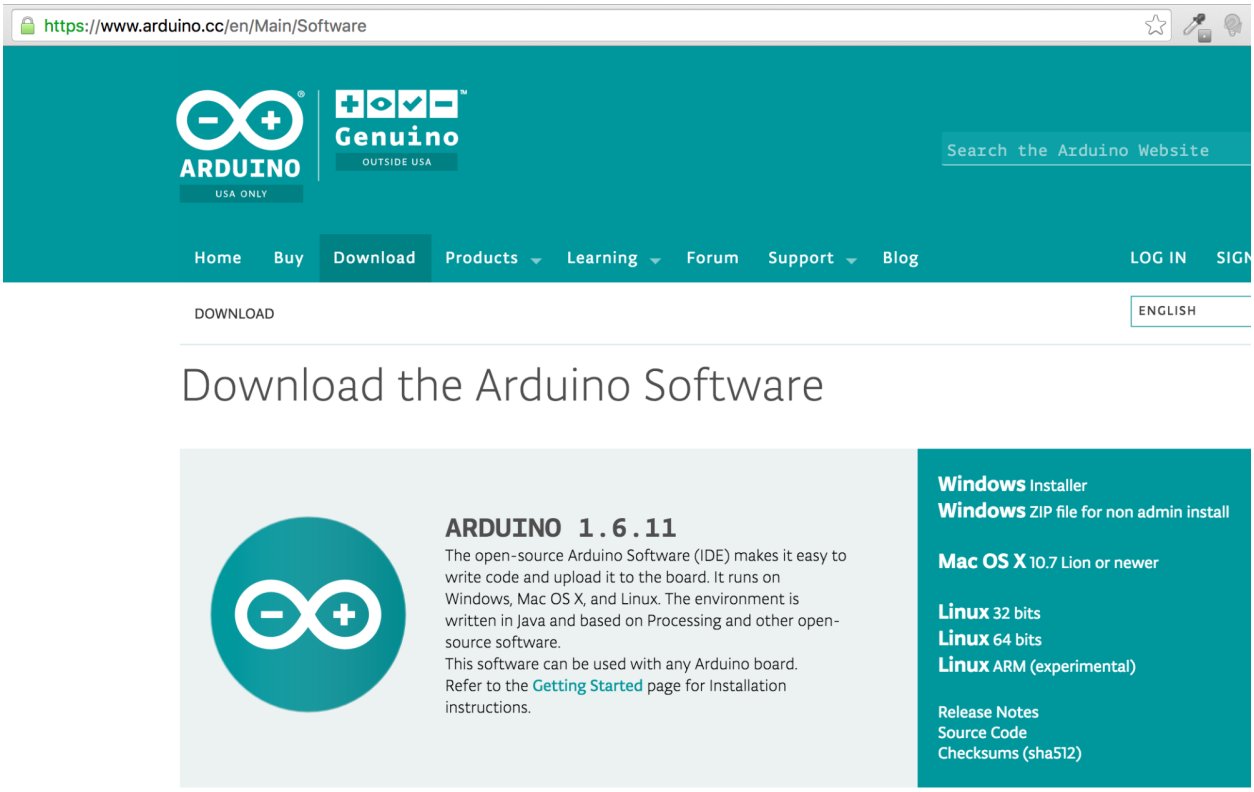
`Open source hardware` is an electronic hardware with the same culture as free code software. This term is used for the first time to reflect the idea of open and public information about hardware, such as diagrams, product structures and layout data of a printed circuit board.

With the growth of programmable logic devices, the sharing of open logic schemes has also spread. In this case, the hardware specifications are available for everyone, which means you can create or evolve your hardware from that content without any problem.

[21] https://www.arduino.cc/en/Main/ArduinoBoardUno
[22] https://www.arduino.Cc/en/Main/ArduinoBoardNano

Installing Arduino IDE

Installing the Arduino IDE is quite simple. Just go to the official project website[23] and on the main page, we can find all the download options per operating system. Check your operating system and download the installer.



Official website of the Arduino project

There is a page on the Arduino Project Wiki with solutions to the most common problems[24], if you have any kind of inconvenience with the installation and first setup Of the Arduino IDE.

Initial Arduino Setup

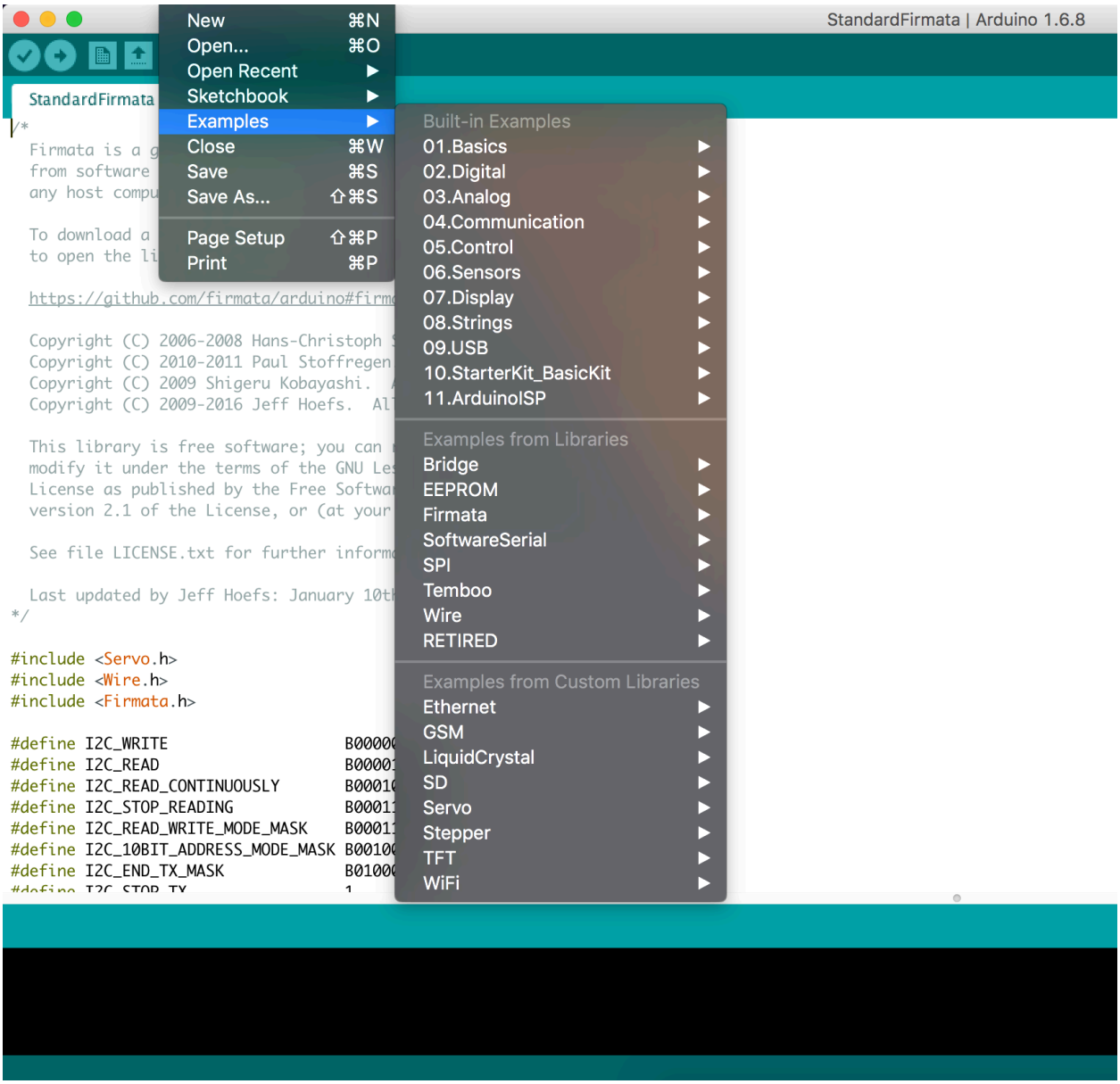
> You can code using your preferred editor or IDE and start this step using the interchange[25] package. The purpose of the following content is to facilitate the Arduino setup for developers who are having the first contact with the Arduino platform.

After the installation of the Arduino IDE, we will now access the program and verify its operation. Firstly we realize that the Arduino IDE has some examples integrated as a mediator and facilitator

[23] https://www.arduino.cc/en/Main/Software
[24] https://github.com/arduino/Arduino/wiki/Building-Arduino
[25] https://github.com/johnny-five-io/nodebots-interchange

for those who have never had contact with the platform. To check the complete list of examples, just access `"File› Examples"`.

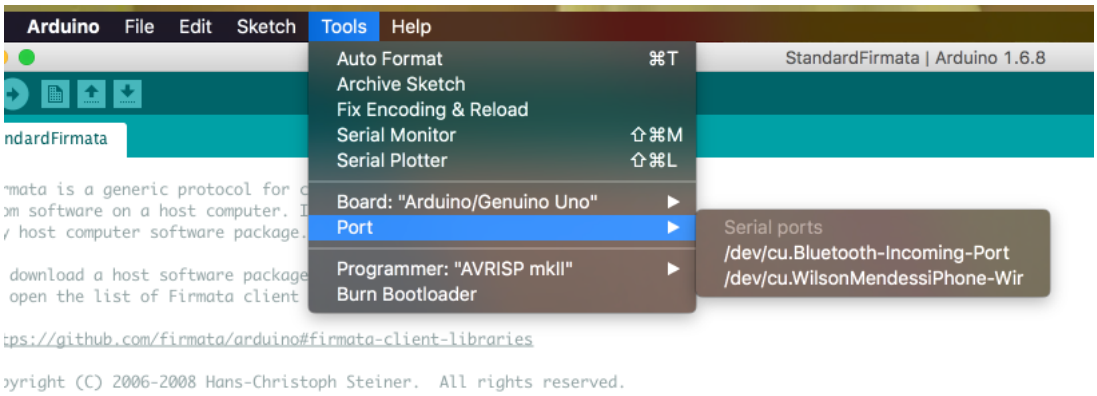
Note that in the example codes are written in [C language](https://pt.wikipedia.org/wiki/C_(program language% C3% A7% C3% A3o) and not in Javascript, but nothing prevents you from running the example code based in other programming languages.



Accessing the Arduino IDE Sample List

Let's now connect our Arduino board into our operating system. Arduino IDE has already stored the configuration of my Arduino, which appears in the item "Board: Arduino Genuino/UNO", but in the first time will appear in the "Port" option, which has the complete listing of serial ports based

in your operational system.



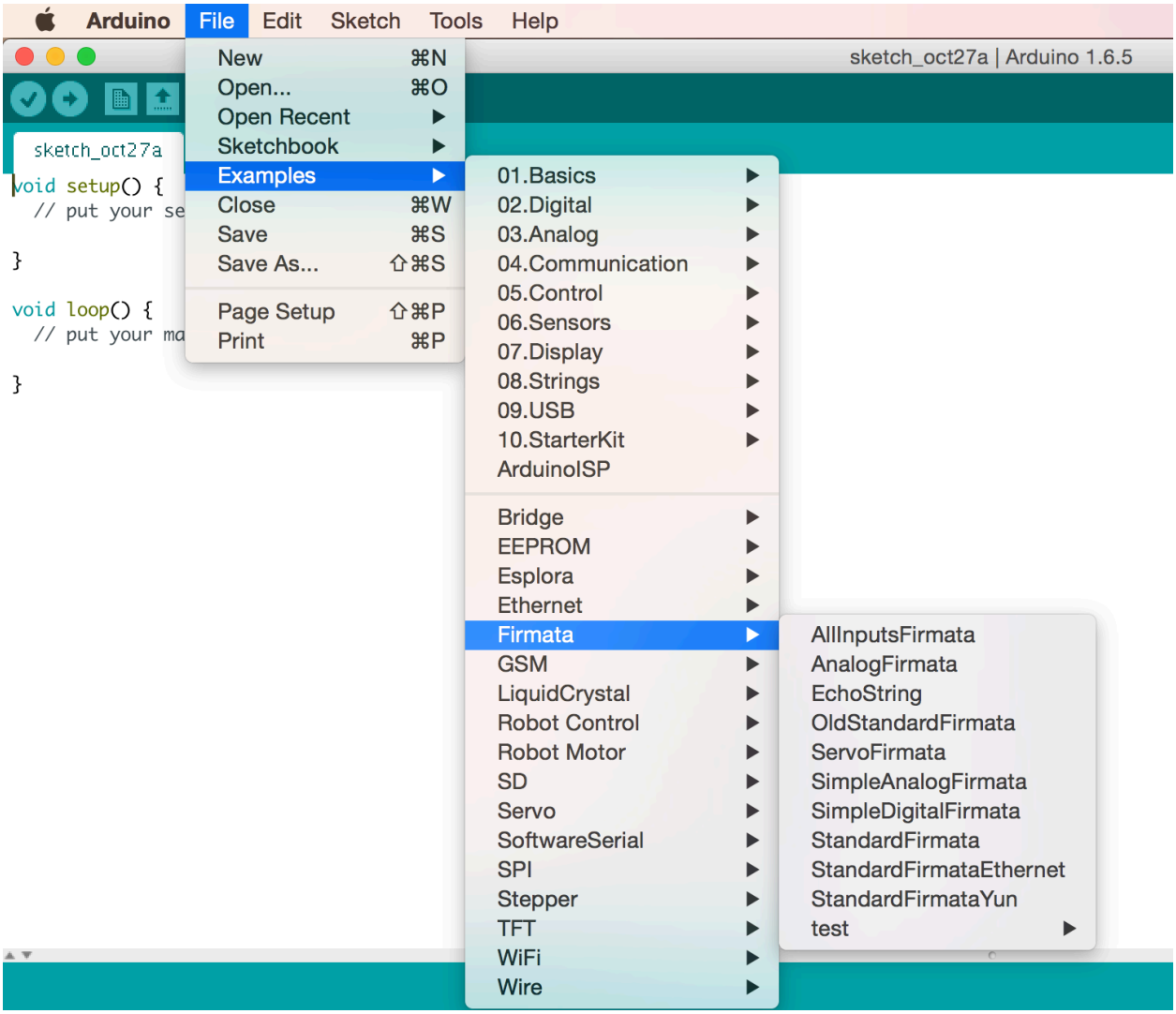
Integrating the Arduino board to the operating system

The name will appear with the prefix "/dev/cu." and will have the name of the Arduino, facilitating the integration. Choose the port which your Arduino is connected and that's all: the connection was successful.

Firmata

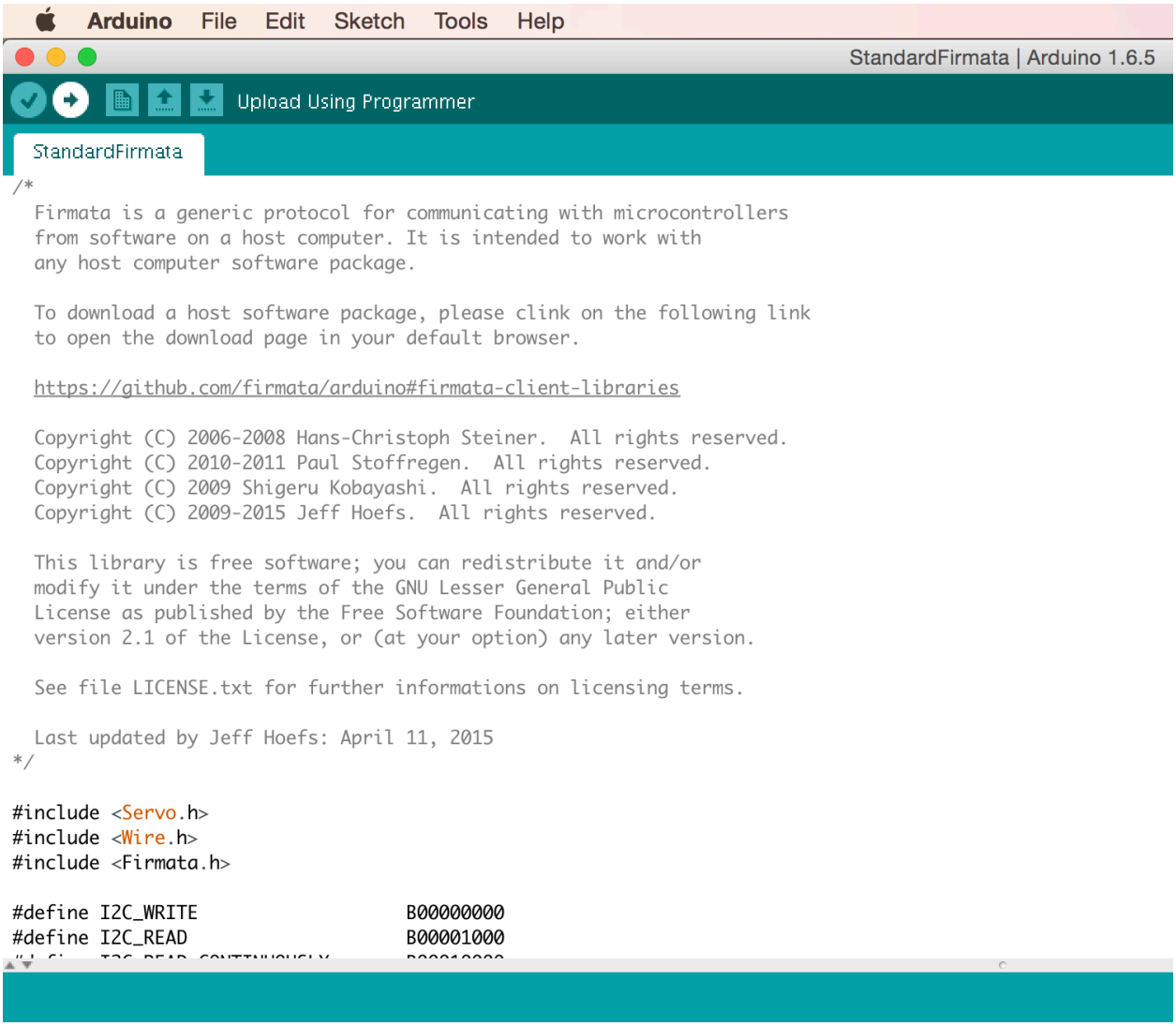
Firmata is a protocol for communicating with microcontrollers software on a computer (or smartphone/tablet, etc). The protocol can be implemented in the firmware of any microcontroller architecture, such as any computer software package.

Our next step after installing the Arduino IDE is to add the Firmata protocol in our Arduino. Let's open our Arduino IDE and access the `"Files› Examples› Firmware› StandardFirmata"` option.



Accessing the Firmware Sample Options in Arduino IDE

With the Arduino plugged into your computer you can run the following code and wait for the IDE shows the message that everything happened successfully.



Everything is OK. Firmata running

Johnny Five

Johnny-Five is an open source framework that allows you to control micro-controllers and components using very similar functions that would be used if you were programming only for the Arduino platform itself, but using JavaScript and implementing Firmata protocol for communication with software and computer host.

This allows you to write a custom `firmware` without having to create your own protocol and objects for the programming environment you are using. In summary, Johnny-Five is a node package that allows you to program micro controllers using JavaScript!

Adding johnny Five to the project

Like any good NodeJS package, adding Johnny-five to the project is a pretty simple task. For this we will use the command that we have already seen, `npm install`, and install johnny five locally, saving as a dependency of project development.

```
$ npm install --save johnny-five
```

After this command, NPM will create the folder node_modules and within it, we will have our johnny-five package installed and accessible in the context of our project.

We can also check that our `package.json` has changed. In it were added the information of the name of our NodeJS package and the installed version, as we can see in the code below.

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "johnny-five": "^1.4.0"
  }
}
```

Creating a Hello World

Now that all the setup of our project has been done, let's create our example code, and nothing better than the good old "Hello World", welcoming you to the Nodebots world.

Let's create a simple code. First, we will create the `index.js` file at the root of our project and import the Johnny-five package using the require command.

```
...
const five = require('johnny-five');
...
```

This command makes the request of the package, making it accessible to our project. Now we'll meet our first class Jonny Five: the Board.

The Board class returns to the application an object that represents the physical board itself in the electronics. All device objects depend on this object to be initialized.

```
...
const board = new five.Board();
...
```

The board object has a `.on()` method, which is commonly used in Javascript applications for creating event handlers. This method accepts 2 parameters:

- Name of the event;
- function to be executed when the event is triggered;

In this example, we will call this method with the `ready` option, which verifies when the code is already accessing the physical card used.

```
...
board.on('ready', () => {
  console.log('Hello World!');
});
...
```

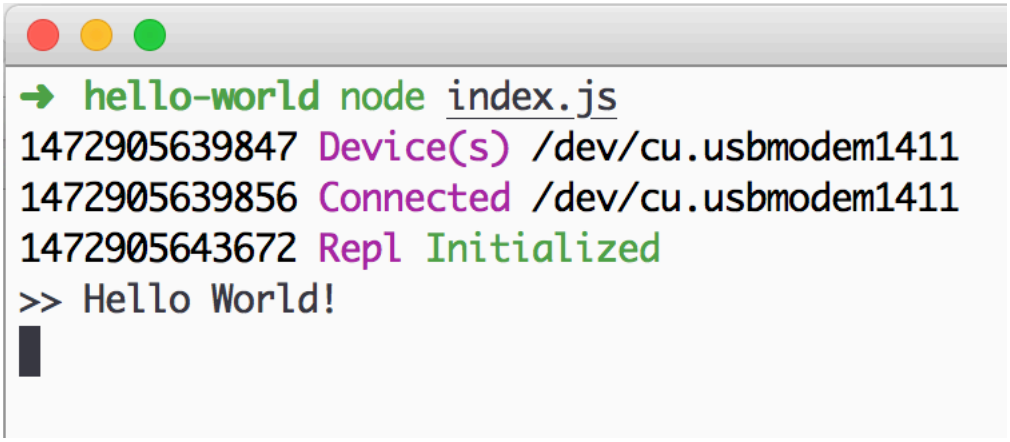
And the final content of our `index.js` has been pretty tight, as you can see below.

```
const five = require('johnny-five');
const board = new five.Board();

board.on('ready', () => {
  console.log('Hello World!');
});
```

Now we can run our code via the command line by typing the command:

```
$ node index.js
```

And the result returned will be the message *"Hello World"*, as you can see in the figure below.



Running the command `node index.js`

If you want to make it easier, we can use npm start, one of the standard NPM commands we can create in our `package.json`, making it accessible via the command line.

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "echo \"This is our 'dev' NPM task\"",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "johnny-five": "^1.4.0"
  }
}
```

After adding the command, just type in the `npm start` command line and the result will be the same message as the previous command.

In this topic, you have seen the integration between our Javascript code and the hardware. In the next chapters, we will see more examples showing in a simple and fun way how to integrate Nodebots into our daily life.

First project: Build Checker

What is a pipeline build

In some presentations, I realised that this term is not known for many people, even those who are more familiar with approaches such as continuous integration and continuous delivery.

Build pipeline is a concept that was built in the middle of 2005 and it's based on the idea of task parallelization, separating each step into small acceptance criteria for the application. It is worth remembering that these steps can be automatic or manual.

Creating a Build Checker

Whenever we get in touch with the Arduino, for example, we make the example of flashing the LED's, commonly known as blink.

In this example, I will show you a more attractive way of approaching this example for our everyday life, based on models such as Hubot[26] and Retaliation[27] to check our build pipeline and find out the health of our application using Arduino + NodeJS + Johnny-Five in an introduction to NodeBots.

Anatomy of a build checker

The project was based on CCmenu[28], a project created by ThoughtWorker Erik Doernenburg to check and show the status of a particular project on a seamless integration server.

In our case we get the idea for something physical, using open hardware and NodeJS. Our application will consume an XML with the information returned by Travis-CI. From this data we check the current state of the application and return it using some artifices like Arduino and LEDs to warn our team that something wrong happened with our build and we must correct as soon as possible.

Materials needed

For this project we will use:

[26] https://hubot.github.com/
[27] https://github.com/codedance/Retaliation
[28] http://ccmenu.org/

- 1 Protoboard: A protoboard is nothing more than a plate with holes and conductive connections for mounting experimental electrical circuits, without the need for welding. A simple protoboard costs between $ 5.00 to $ 10.00 and can be found in any electric store;
- 2 LEDs (*light-emitting diode*): 1 red to signal the broken build and 1 green to signal that the build was successfully completed. A LED costs less than $ 0.50 and can be found at any electrical store;
- Arduino with 2 GND (ground) ports;

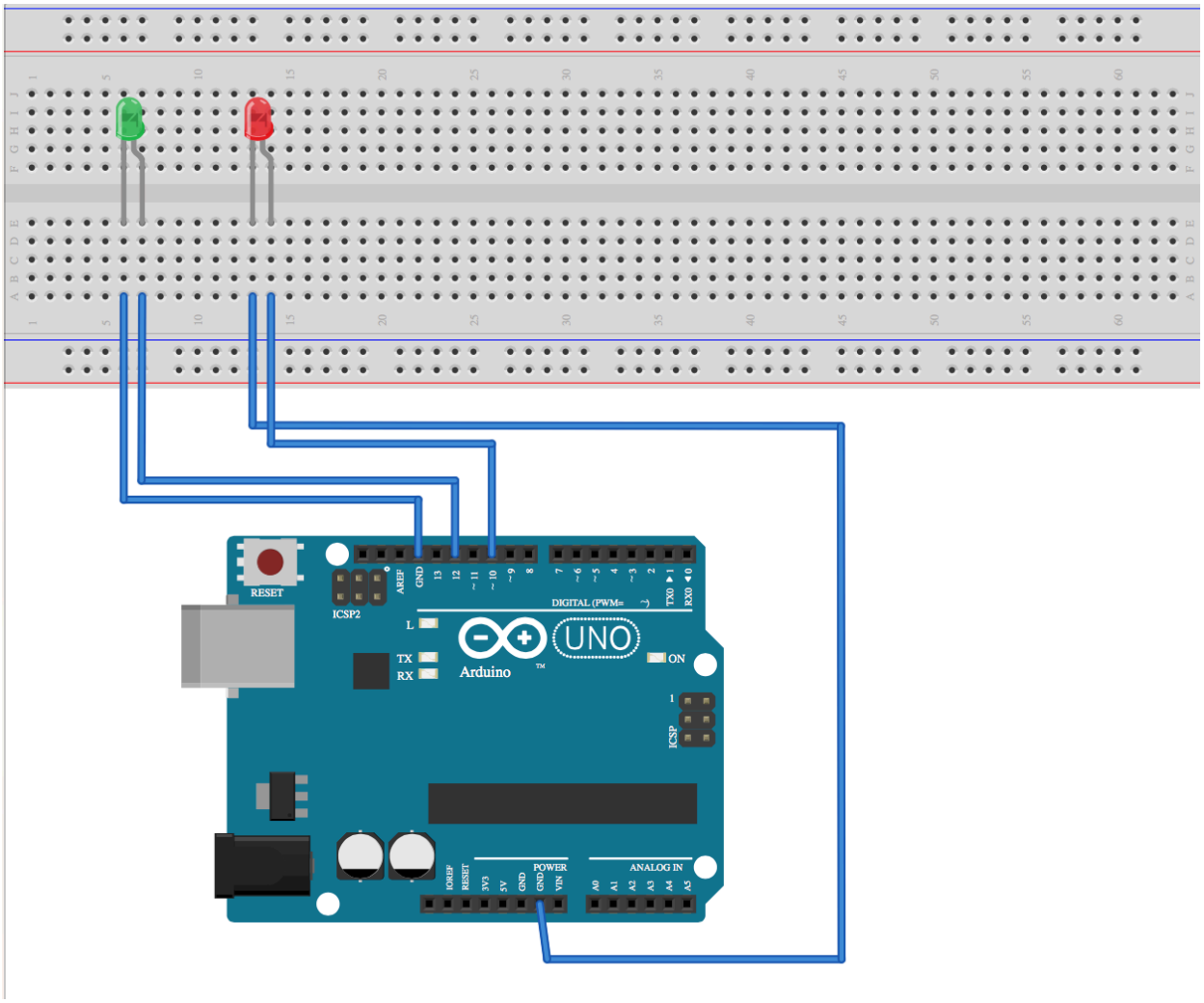
Material needed for Build Checker

GND ports are called “ground” ports. They are electrical conductors that connect to the Earth - that is, to the Electric Earth. Since it is always neutral and (theoretically) present in every electric circuit, it is always taken as a reference point for the measurement of potentials, containing zero volts.

Now just plug the 2 LEDS, linking as follows:

- Successful build on the number 12 door + a GND port of our Arduino;
- Error build on port number 10 + a GND port of our Arduino;

The following image illustrates the assembly of the components with the Arduino.



Connecting the Arduino: ports and LEDs

Controlling the LED

Knowing the components that we will use, we now go to our initial code that will control our LED. First, we will create the folder `src`, where will be the code of our application.

Let's create the folder for our `build-checker` project and navigate to the created folder.

```
$ mkdir build-checker
$ cd build-checker
```

We will start our application by typing the `npm init` command with the `-y` flag which means that all the answers that were previously made will be answered and registered with the default response. After this, we will install the johnny-five package locally as a dependency in the folder of our project.

```
$ npm init -y
$ npm install --save johnny-five
```

Inside the folder of our project we will create the folder `src` and inside it, our file `index.js` where will be our content.

```
$ mkdir src
$ touch src/index.js
```

In the `index.js` file we will import the Johnny-five package using the `require` command and create our protoboard instance to add the LED.

```
...
const five = require('johnny-five');
const board = new five.Board();
...
```

For the first activity with the component, we will use a new Johnny Five class called `LED`. To use this class we need to pass the value of the pin that the `LED` is connected to the Arduino.

```
...
const led = new five.Led(12);
...
```

Our first functional code will look something like this.

```
const five = require('johnny-five');
const board = new five.Board();

board.on('ready', () => {
  const ledSuccess = new five.Led(12);
  const ledError = new five.Led(10);

  ledSuccess.blink();
  ledError.blink();
});
```

Now let's put our code to work on our Arduino. Inside the folder of our project, we will enter in our command prompt/terminal the following command:

```
$ node src/index.js
```

After this command, our prompt/command line is showing that the command has run successfully and the result will be that our two LEDs will be flashing.

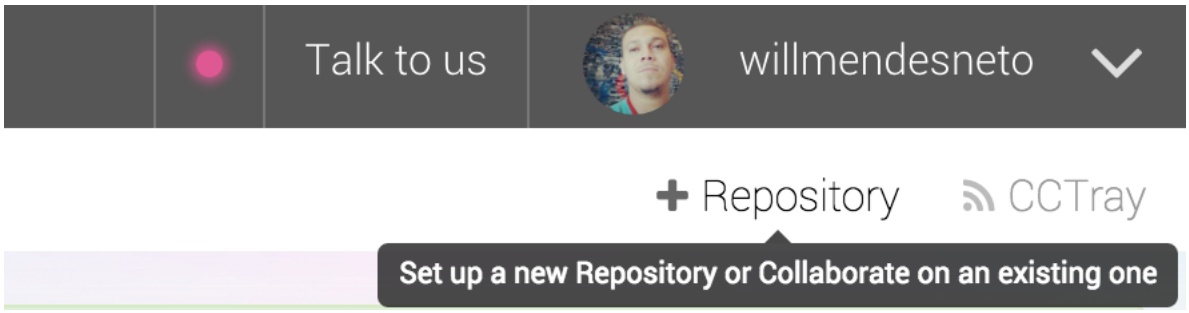
Quite simple, is not it? In the next topic, we will think a little more about our architecture.

Creating the CI/CD build information request

We already have our build checker abstraction, let's now add the last functionality that is to read the build info on our continuous delivery server and/or continuous integration server.

From there we created a request to read the content via HTTP GET in SNAP-CI[29], our continuous integration service. SNAP-CI uses a build pipeline concept that is very interesting and one of its pros is the faster feedback, giving the possibility of parallelism or not, and definition of steps for the total build. For more information on Build Pipeline, I recommend reading Martin Fowler's Continuous Integration article[30].

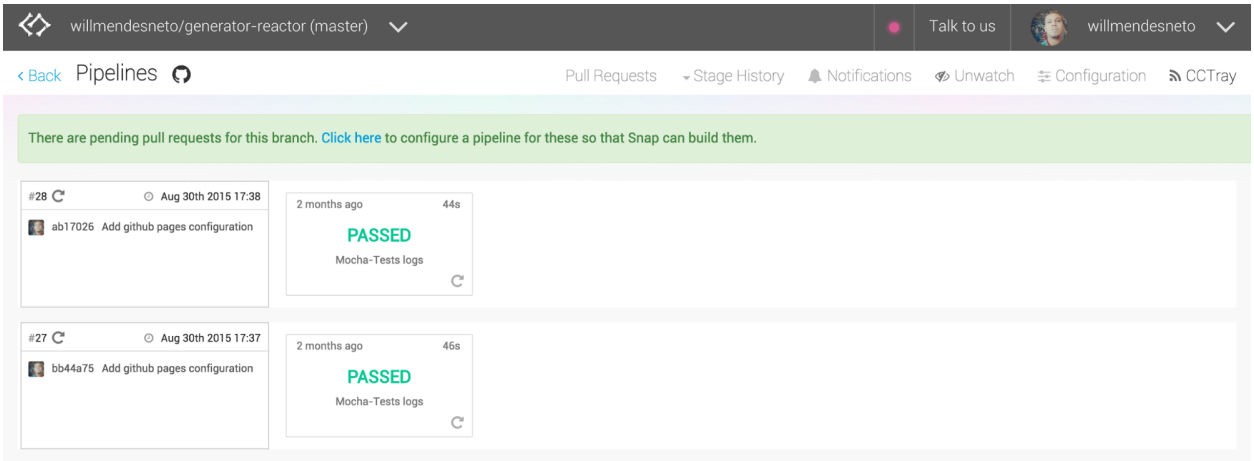
We go to the SNAP-CI website, we log in and register a project. If you do not have a registration you will have to create one, but it is very fast.



Adding repositories in Snap-CI

When registering the project it will appear in the upper part, on the right, a field with the name "CCTray" that, when we click, it directs to the XML file with the information of the build.

[29] https://snap-ci.com
[30] http://www.martinfowler.com/articles/continuousIntegration.html



Viewing pipelines created in Snap-CI

And these are the information that we will consult with our build-checker.

```
<Projects>
  <Project
name="brasil-de-fato/news-service (master) :: CONTRACT-TEST"
activity="Sleeping"
lastBuildLabel="77"
lastBuildStatus="Success"
lastBuildTime="2016-01-21T18:46:48Z"
webUrl="https://snap-ci.com/brasil-de-fato/news-service/branch/master/logs/defaultPi\
peline/77/CONTRACT-TEST"/>
</Projects>
```

Analysing the information we realise that the only information we should validate for our build is the data from the lastBuildStatus field. It is returned if the build was successfully completed, successfully completed, or is happening at the time of validation.

The `lastBuildStatus` field can contain:

- `Success`: the last job on the server has been successfully completed;
- `Failure`: the last job on the server was terminated with error;
- `Pending`: the task is happening right now on the server;
- `Exception` and `Unknown`: Something unexpected occurred with the current task on the server. The reasons are the most diverse, as the server had a swing in the middle of the task or he never ran a certain task yet, so he does not have the information;

Let's now add our URL containing the CCTray information from our project and create it with the request for this information. For this, we will add the package NodeJS request[31], an HTTP client that was developed in order to facilitate the creation of HTTP or HTTPS requests. To add the new packages we will enter the following command.

[31] https://github.com/request/request

```
$ npm install --save request
```

> The use of the request is very simple, accepting 2 parameters being the first the URL and the second the function that will manipulate the requested information. For more details, see full documentation of the request package in Github[32].

Now let's add the package in our project and create the request using the RequestJS module.

```
...
const request = require('request');
const five = require('johnny-five');
const board = new five.Board();
...
```

And we'll add the CCTray address that we've copied from our continuous integration server into our code and create the HTTP request to read the information.

```
// For educational purposes this code is using this format
// For computational resource issues use the URL as a string
// instead of manipulating the Array in the final project
const CI_CCTRACKER_URL = [
  'https://snap-ci.com',
  'willmendesneto',
  'generator-reactor',
  'branch',
  'master',
  'cctray.xml'
].join('/');
```

Let's explain a bit about RequestJS *callback*. It returns 3 parameters:

- `error`: an object with the error information that happened. If the request does not return any error it has the default value `null`;
- `response`: object with the request response information;
- `body`: String with the information of the body of the return of the requisition;

In our code then we will analyse the return of the requisition and create the appropriate treatments. The first treatment will be the verification of the error and in case we have an error we will treat the error.

[32] https://github.com/request/request#table-of-contents

```
...
request(CI_CCTRACKER_URL, function(error, response, body) {
  if (error) {
    console.log('Something is wrong in our CI/CD =(');
    return;
  }
  ...
});
```

If the response does not return any error, we will treat the message to turn the LEDs on and off for visual feedback.

```
...
if(body.indexOf('Failure') !== -1) {
  console.log('Your CI/CD is broken! Fix it!!!!');
  ledSuccess.off();
  ledError.on();
} else {
  console.log('Your CI/CD is ok!');
  ledSuccess.on();
  ledError.off();
}
...
```

With the request created, we will only create a gap between each request, using a simple `setInterval`. We will use a time of 500 milliseconds for code validation, but this value can be changed to whatever is ideal for you.

```
setInterval(() => {
  request(CI_CCTRACKER_URL, function(error, response, body) {
    ..
  });
}, 500);
```

And the final content of our `src/index.js` was as follows:

```
const request = require('request');
const five = require('johnny-five');
const board = new five.Board();

// For educational purposes this code is using this format
// For computational resource issues use the URL as a string
// instead of manipulating the Array in the final project
const CI_CCTRACKER_URL = [
  'https://snap-ci.com',
  'willmendesneto',
  'generator-reactor',
  'branch',
  'master',
  'cctray.xml'
].join('/');

board.on('ready', () => {

  const ledSuccess = new five.Led(12);
  const ledError = new five.Led(10);

  setInterval(() => {
    request(CI_CCTRACKER_URL, function(error, response, body) {
      if (error) {
        console.log('Something is wrong in our CI/CD =(');
        return;
      }

      if(body.indexOf('Failure') !== -1) {
        console.log('Your CI/CD is broken! Fix it!!!!');
        ledSuccess.off();
        ledError.on();
      } else {
        console.log('Your CI/CD is ok!');
        ledSuccess.on();
        ledError.off();
      }

    });
  }, 500);
});
```

It will query the data at a preconfigured interval and check the current state of the build, based on

information from all pipelines. If there is no word *"Failure"* in the response to the request, something wrong has happened and our *build checker* will turn on the red light, otherwise, the green light will remain on, signalling that everything is ok.

Tuning the architecture of our application

Now that we've finished the first step and have seen our working with our components, let's think about improving our architecture.

We can see that we have some rather loose values, which do not say much if we do not access the Johnny-Five framework documentation, such as numbers 12 and 10. For these and other configurations we will create a `configuration.js` file with Project information.

The initial contents of this file will be:

```
// src/configuration.js
module.exports = {
  LED: {
    SUCCESS: 12,
    ERROR: 10
  },
  CI_CCTRACKER_URL: 'https://snap-ci.com/willmendesneto/generator-reactor/branch/mas\
ter/cctray.xml',
  INTERVAL: 1000
};
```

Now let's change our `src/index.js` to use our file with the default settings of our application.

```
const five = require('johnny-five');
const CONFIG = require('./configuration');
const board = new five.Board();

board.on('ready', () => {

  const ledSuccess = new five.Led(CONFIG.LED.SUCCESS);
  const ledError = new five.Led(CONFIG.LED.ERROR);

  setInterval(() => {
    request(CONFIG.CI_CCTRACKER_URL, function(error, response, body)
    {
      ...
    });
  }, CONFIG.INTERVAL);
});
```

Our code is starting to get a little more expressive, do you agree? But is there still something we can improve on? Of course yes!

We are talking about *build checker*, but we have no abstraction for this operation. The idea is that our final code is just an initialization of our app, with all relevant information within this abstraction.

Let's then create our file with the LED information abstractions and the HTTP request by accessing the settings.

```
1  const CONFIG = require('./configuration');
2  const request = require('request');
3  const five = require('johnny-five');
4
5  let intervalId = null;
6  function BuildChecker() {
7    this.ledSuccess = new five.Led(CONFIG.LED.SUCCESS);
8    this.ledError = new five.Led(CONFIG.LED.ERROR);
9  };
10
11 BuildChecker.prototype.stopPolling = () => {
12   clearInterval(intervalId);
13 };
14
15 BuildChecker.prototype.startPolling = () => {
16   const self = this;
17
18   intervalId = setInterval(() => {
19     request.get(CONFIG.CI_CCTRACKER_URL, function(error, response, body) {
20       if (error) {
21         console.log('Somethink is wrong with your CI =(');
22         return;
23       }
24
25       if(body.indexOf('Success') !== -1) {
26         console.log('Your CI is ok!');
27         self.ledSuccess.on();
28         self.ledError.off();
29       } else {
30         console.log('Somethink is wrong with your CI =(. Fix it!!!!');
31         self.ledSuccess.off();
32         self.ledError.on();
33       }
34
35     });
```

```
36    }, CONFIG.INTERVAL);
37  };
38
39  module.exports = BuildChecker;
```

And our `src/index.js` file will only invoke and start our code so that the LEDs start blinking.

```
1  const BuildChecker = require('./build-checker');
2  const five = require('johnny-five');
3  const board = new five.Board();
4
5  board.on('ready', () => {
6    buildChecker = new BuildChecker();
7    buildChecker.startPolling();
8  });
```

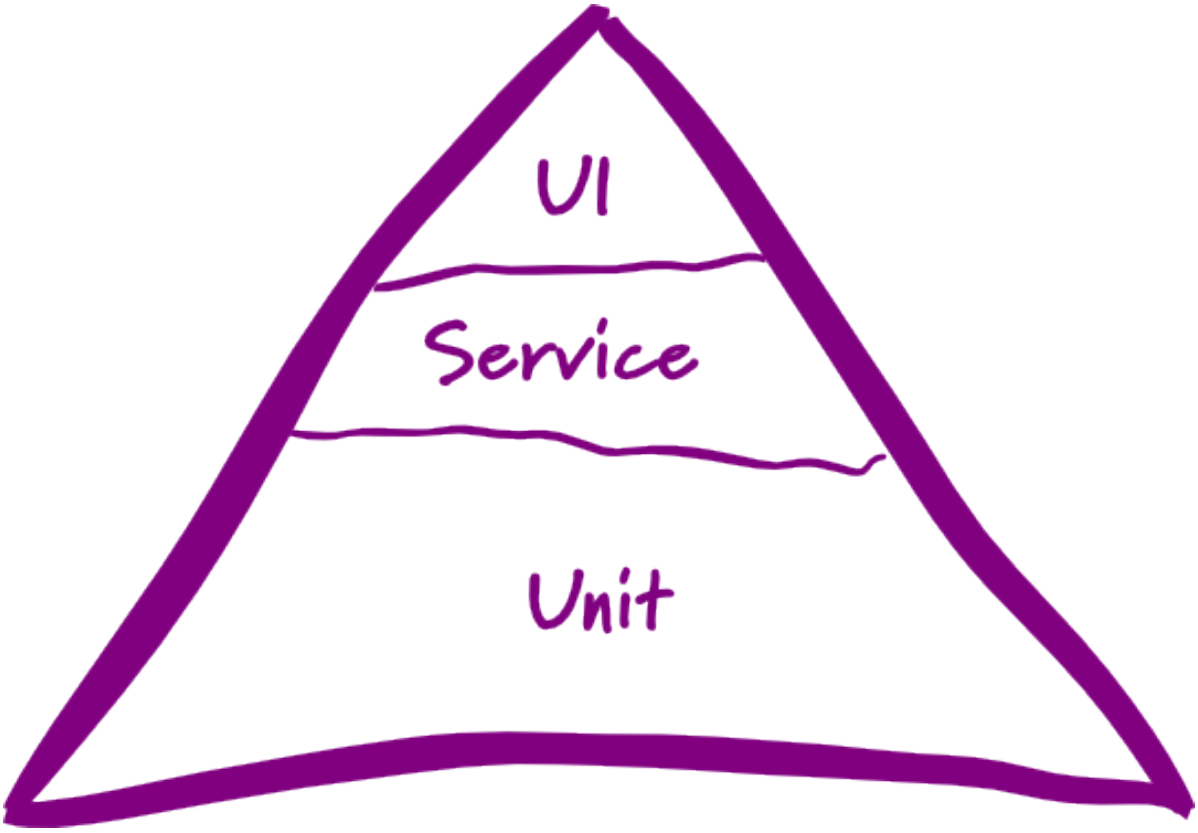
Finishing this separation of concepts, we improve the readability, maintainability and several other variants of our application. It is worth mentioning that this is a good practice and that, in the course of the book, we will always be thinking about improvements to our final code.

Creating unit tests for build checker

This time something simple, but without good information about it is like adding drive tests on Nodebots applications. Unit testing is not something new, but you do not find content on this topic in Arduino, robots and hardware applications open easily, so we'll cover a bit on this topic in this book.

Unit testing is just one of several ways to test your software and have a reliability in the end product. Based on the Test Pyramid[33], this is the way you should organise the testing of your application.

[33] http://martinfowler.com/bliki/TestPyramid.html



Pyramid of tests

We'll talk only about unit tests, if you'd like to know more about all layers, read the post "TestPyramid"[34] by Martin Fowler.

The idea of the unit test is to validate and certify that your code is doing what it intends to do, giving feedback on the errors quickly before we deploy our project to production.

One aspect that nobody explains very well is about the tests in Nodebots, which has as the main objective, in this case, to create the electrical simulations and with mocks and stubs thus simulating the communication between components.

Let's now create a folder for our unit tests with the name `test`.

```
1 $ mkdir test
```

The unit tests will use the test framework MochaJS[35], SinonJS[36] for *spies*, *stubs* and *mocks* and ShouldJS[37] for *assertions*. Let's then install these packages as a dependency of project development.

[34] http://martinfowler.com/bliki/TestPyramid.html
[35] https://mochajs.org
[36] http://sinonjs.org
[37] https://shouldjs.github.io

```
$ npm install --save-dev mocha sinon should
```

A fundamental NodeJS package in this step is mock-Firmata[38], created by Rick Waldron to make testing setup in Johnny-Five applications easier. The integration is very simple, you just need to load and create your fake component of the board in the test, as we can see in our setup file of `test/spec-helper.js` tests.

```
require('should');
const mockFirmata = require('mock-firmata');
const five = require('johnny-five');

const board = new five.Board({
  io: new mockFirmata.Firmata(),
  debug: false,
  repl: false
});
```

Let's then add a simple test to check the integration of our tests. First, we will create a file with some MochaJS settings inside the `test` folder. This will be the initial content of our `mocha.opts`.

```
--reporter spec
--recursive
--require test/spec-helper.js
--slow 1000
--timeout 5000
```

A quick explanation of the configuration information used:

`--reporter spec`: A type of reporter* used to display messages of test information; `--recursive`: flag to identify that the tests should run recursively inside the folder; `--require` test/spec-helper.js: *setup* file to be loaded before running the unit tests; `--low 1000`: Maximum time in milliseconds of tolerance between tests. If this time exceeds this time will be shown the total time of that test with a differentiated colour so that we can make the necessary changes; `--timeout 5000`: Maximum time in milliseconds of tolerance for the completion of each assertion. If this time exceeds this time our tests will return with an error message;

We will create a file with the name `test/index.js` with a fairly simple assertion.

[38] https://github.com/rwaldron/mock-firmata

```
describe('Test validation', () => {
  it('1 + 1 = 2', () => {
    (1 + 1).should.be.equal(2);
  });
});
```

Our tests use the `describe` and `it` scope methods. `Describe` is used to group scenarios, in our case `Test validation`. `It` is the identification of one of the test points. Note also that we have the `should.be.equal` method that will compare if the first value is equal to the second.

Configuring the test suite in the repository

Let's go to our prompt/terminal and enter the following command.

```
$ ./node_modules/bin/mocha
```

We will then see this information at our prompt/terminal and our initial setup was a success!

Now, let's create the scenarios for our tests. Let us then define the scenarios that we must cover in our tests:

- Initial information when creating the `BuildChecker` instance;
- When we start *polling* and the server sends information from a successfully completed build;
- When we start *polling* and the server sends information from a finished build with failures;
- When we stop our *polling* and we will not do more requests of data for our server;

One way to validate when the build checker should flash the LED is to create a stub for the request using the node-request to validate the response by expected types (`success` and `error`) and use some spies for the LEDs.

For this simulation, we will create some *fixtures* with the server responses model for success and error. Let's then create our folder with the data inside our folder containing our tests.

```
$ mkdir test/fixtures
$ touch test/fixtures/success.xml test/fixtures/error.xml
```

And i will add the information for each file.

```
<!-- test/fixtures/error.xml -->
<Projects>
  <Project
    name="Error-project"
    activity="Sleeping"
    lastBuildLabel="22"
    lastBuildStatus="Failure"
    lastBuildTime="2016-01-04T02:20:25Z"
    webUrl="https://google.com"/>
</Projects>
```

```
<!-- test/fixtures/success.xml -->
<Projects>
  <Project
    name="Success-project"
    activity="Sleeping"
    lastBuildLabel="36"
    lastBuildStatus="Success"
    lastBuildTime="2016-03-22T21:09:02Z"
    webUrl="https://google.com"/>
</Projects>
```

Let's then create the scenario to validate our code. Some things we should keep in mind about our unit testing framework is that its `beforeEach` method, which happens before every `it` method. We will use as documentation of each step that must occur to reproduce the specific scenario.

Let's then explain more about the contents of this file and why of each test. We create the tests of the instance of our `BuildChecker` and its initial attributes.

```
const BuildChecker = require('../src/build-checker');
const five = require('johnny-five');
const request = require('request');
const sinon = require('sinon');

describe('BuildChecker', () => {

  beforeEach(() => {
    buildChecker = new BuildChecker();
  });

  it('should have the led success port configured', () => {
    (buildChecker.ledSuccess instanceof five.Led).should.be.equal(true);
  });

  it('should have the led error port configured', () => {
    (buildChecker.ledError instanceof five.Led).should.be.equal(true);
  });

});
```

Now we will validate when we stop our polling. Let's now use the spy method of the sinon to check if the code used the clearInterval method to end with the requests. For this, we will check if the `global.clearInterval` was used once, by accessing the boolean `calledOnce`, which is an internal counter added by the `sinon.spy` method for the tests.

```
...
describe('#stopPolling', () => {
  beforeEach(() => {
    sinon.spy(global, 'clearInterval');
    buildChecker.stopPolling();
  });

  it('should remove interval', () => {
    global.clearInterval.calledOnce.should.be.true;
  });
});
...
```

And now the server scenarios responding successfully and failed. For this, we will assign our data from the *fixtures* folder to variables.

```
...
const fs = require('fs');
const successResponseCI = fs.readFileSync(__dirname + '/fixtures/success.xml', 'utf8\
');
const errorResponseCI = fs.readFileSync(__dirname + '/fixtures/error.xml', 'utf8');
...
```

Note that in each of the success and failure cases we are using the `sinon.useFakeTimers` method, which is a way to simulate events linked to timer objects in Javascript.

When we call the `clock.tick` the method with the information contained in the configuration file, we simulate time and time changes at the time of testing, which helps us force the call to the polling event, which uses `setInterval`.

```
...
clock = sinon.useFakeTimers();
clock.tick(CONFIG.INTERVAL);
...
```

We now use the syntax stub method for the `node-request` package. With this, we can change the return when the `request.get` method is called. In this case, we can simulate the response of each request, based on the information of our fixtures.

```
...
sinon.stub(request, 'get').yields(null, null, successResponseCI);
...
```

An important point in our tests is to remember to restore all the `stub` and `fakeTimers` information. We will use the `afterEach` method that is always called after each `it` method, and we will add our objects by calling the restore method added by `sinon`.

```
...
afterEach(() => {
  request.get.restore();
  clock.restore();
});
...
```

Based on our test scenarios, this is the test content of our build checker file:

```
// test/build-checker.js

const BuildChecker = require('../src/build-checker');
const CONFIG = require('../src/configuration');
const five = require('johnny-five');
const request = require('request');
const sinon = require('sinon');
const fs = require('fs');
const successResponseCI = fs.readFileSync(__dirname + '/fixtures/success.xml', 'utf8\
');
const errorResponseCI = fs.readFileSync(__dirname + '/fixtures/error.xml', 'utf8');
let clock = null;

describe('BuildChecker', () => {

  beforeEach(() => {
    buildChecker = new BuildChecker();
  });

  it('should have the led success port configured', () => {
    (buildChecker.ledSuccess instanceof five.Led).should.be.equal(true);
  });

  it('should have the led error port configured', () => {
    (buildChecker.ledError instanceof five.Led).should.be.equal(true);
  });

  describe('#stopPolling', () => {
    beforeEach(() => {
      sinon.spy(global, 'clearInterval');
      buildChecker.stopPolling();
    });

    it('should remove interval', () => {
      global.clearInterval.calledOnce.should.be.true;
    });
  });

  describe('#startPolling', () => {
    beforeEach(() => {
      sinon.spy(global, 'setInterval');
      buildChecker.startPolling();
    });
```

```
44
45      afterEach(() => {
46        global.setInterval.restore();
47      });
48
49      it('should creates polling', () => {
50        global.setInterval.calledOnce.should.be.true;
51      });
52
53      describe('When the CI server send success response', () => {
54        beforeEach(() => {
55          clock = sinon.useFakeTimers();
56          sinon.stub(request, 'get').yields(null, null, successResponseCI);
57          sinon.spy(buildChecker.ledSuccess, 'on');
58          sinon.spy(buildChecker.ledError, 'off');
59          buildChecker.startPolling();
60          clock.tick(CONFIG.INTERVAL);
61        });
62
63        afterEach(() => {
64          request.get.restore();
65          clock.restore();
66        });
67
68        it('should turn on the success led', () => {
69          buildChecker.ledSuccess.on.calledOnce.should.be.true;
70        });
71
72        it('should turn off the error led', () => {
73          buildChecker.ledError.off.calledOnce.should.be.true;
74        });
75
76      });
77
78      describe('When the CI server send error response', () => {
79        beforeEach(() => {
80          clock = sinon.useFakeTimers();
81          sinon.stub(request, 'get').yields(null, null, errorResponseCI);
82          sinon.spy(buildChecker.ledError, 'on');
83          sinon.spy(buildChecker.ledSuccess, 'off');
84          buildChecker.startPolling();
85          clock.tick(CONFIG.INTERVAL);
86        });
```

```
87
88        afterEach(() => {
89          request.get.restore();
90          clock.restore();
91        });
92
93        it('should turn off the success led', () => {
94          buildChecker.ledSuccess.off.calledOnce.should.be.true;
95        });
96
97        it('should turn on the error led', () => {
98          buildChecker.ledError.on.calledOnce.should.be.true;
99        });
100
101     });
102
103   });
104
105 });
```

This is just one of several unit testing formats for your application. With this, we finish our first project with unit tests based on our possible scenarios, but if you want to download or fork the final code, access the build checker project repository in Github[39]. Let's go to our next project with Nodebot and Johnny-five?

[39] https://github.com/willmendesneto/build-checker

Second project: Fire alarm

Our second example project will be that of an intelligent fire alarm. Our fire alarm will check for the temperature and, in the case of a fire, it will activate the audible alarm and send an SMS to the registered cell phone.

A simple example, but that shows some interesting integration points, such as integration with API's, data reading of a temperature sensor and integration with the Piezo sound sensor.

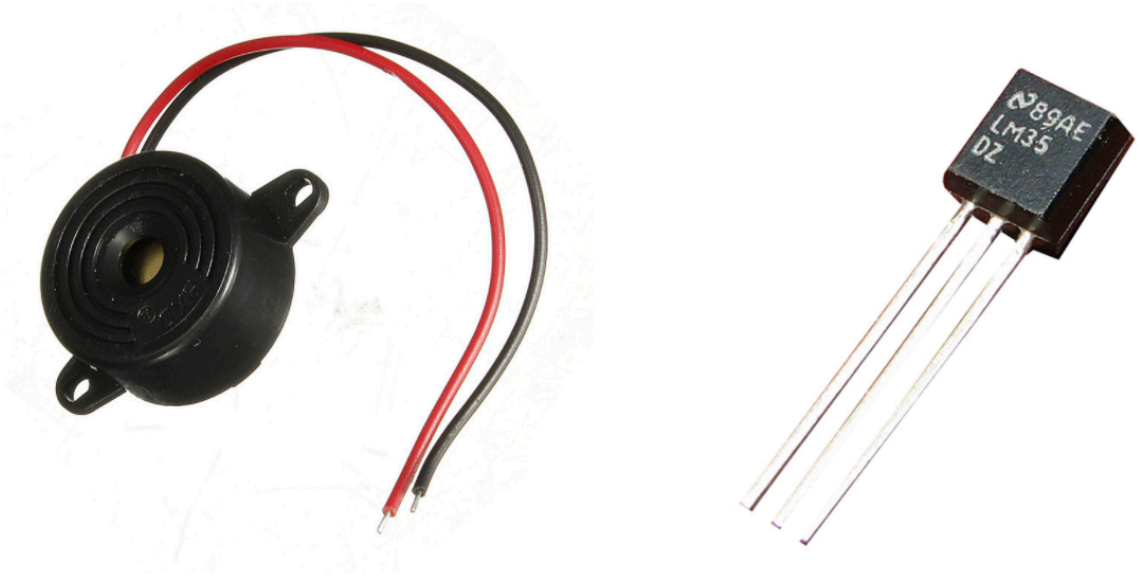
Anatomy of a fire alarm

The project was based on a simple fire alarm home system. By default systems of this type do a check from time to time and activate the alarm some different temperature pattern is encountered.

Materials needed

For this project we will use:

- 1 Protoboard: A protoboard is nothing more than a plate with holes and conductive connections for mounting of experimental electrical circuits, without the need of welding;
- 1 Piezo alarm sensor: It will be used for sound feedback to the end user, in our case to the tenant (s) of the property. This sensor is very simple and costs less than $ 2.00 can be found in any electric store;
- 1 temperature sensor: Johnny-Five works with a wide range of temperature sensors. In this case, we will use the LM35 temperature sensor. This sensor is very simple and costs less than $ 1.50 can be found in any electric store;



Piezo and Temperature Sensors

- For a complete list of supported sensors, go to Johnny's Temperature Sensor Wiki[40] page and check the list of sensor reference codes.

Some sensors require a specific voltage port for their correct operation (some sensors call VND). In our case we will use a 5 volt port for the temperature sensor.

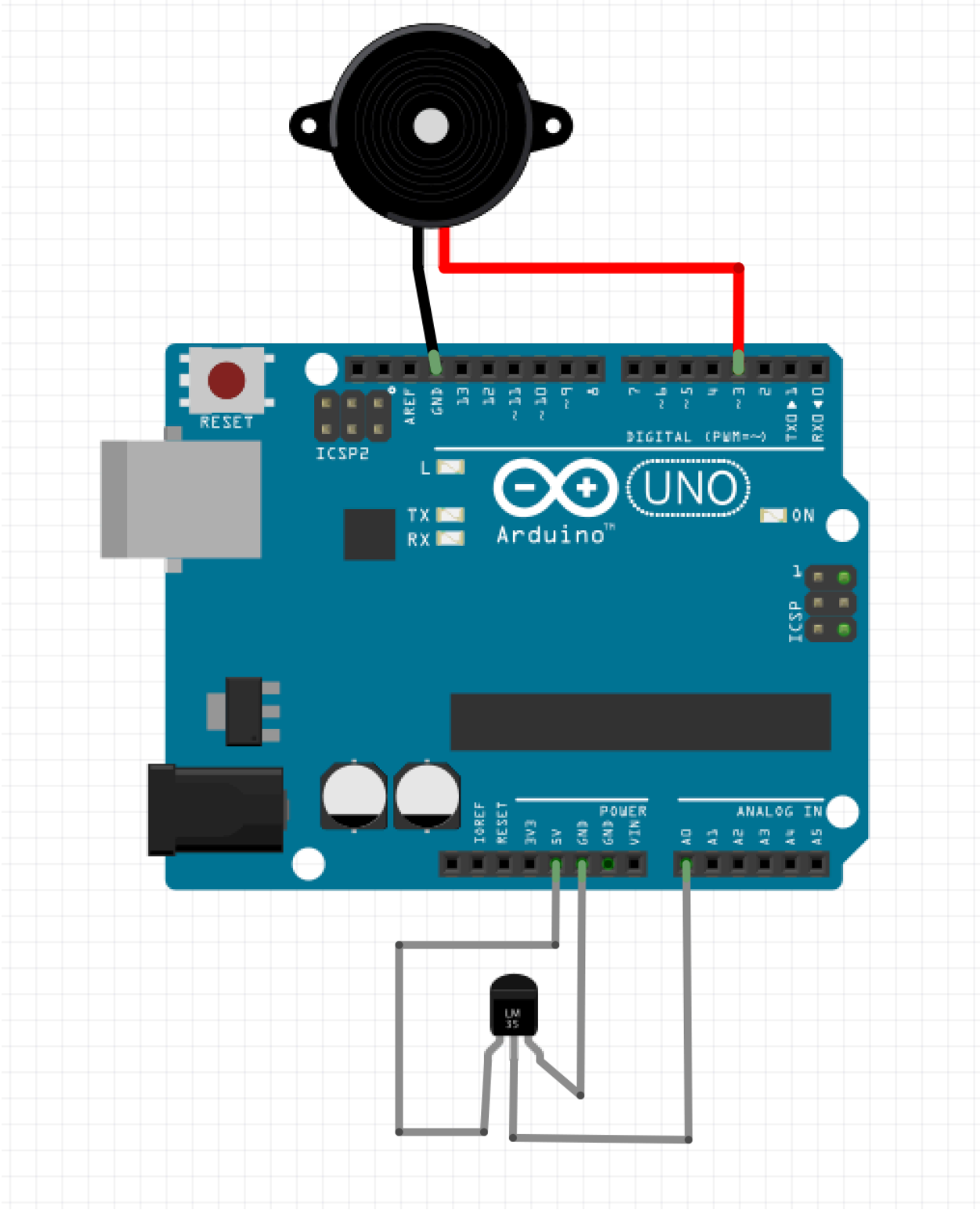
Other sensors may need an analog port. Analog ports are used for the sensor to send voltage data to the Arduino so we can read and interpret your information.

For this project we will mount the sensors in the Arduino as follows:

- Piezo alarm sensor: Attach the black piezo wire to the GND port and the red wire to our arduino's number 3 port;
- LM35 temperature sensor: attach the grounding pin to the GND port, the voltage pin on the 5-volt port and the data output pin on the analog port "A0";

The following image illustrates the assembly of the integrated components with the Arduino.

[40] https://github.com/rwaldron/johnny-five/wiki/thermometer



Integration of components used in Fire Alarm

Controlling the Flame Sensor

With the LM35 sensor connected to the Arduino board, let's now read the ambient temperature information. For this we will use the Thermometer class of Johnny five. When we create a new object Thermometer, we must pay attention to some parameters:

- `controller`: Name of the sensor used. You can refer to the complete list of sensors supported by Johnny Five in the Thermometer class documentation[41];
- `pin`: The pin information used for the analog connection on the Arduino. It is used in analog sensors for reading the temperature information;
- `toCelsius`: An optional method that we can rewrite to handle the analog data and transform it into the temperature format of your preference. In our case, we will use the Celsius format;

Based on this information, the main file of our fire alarm will be:

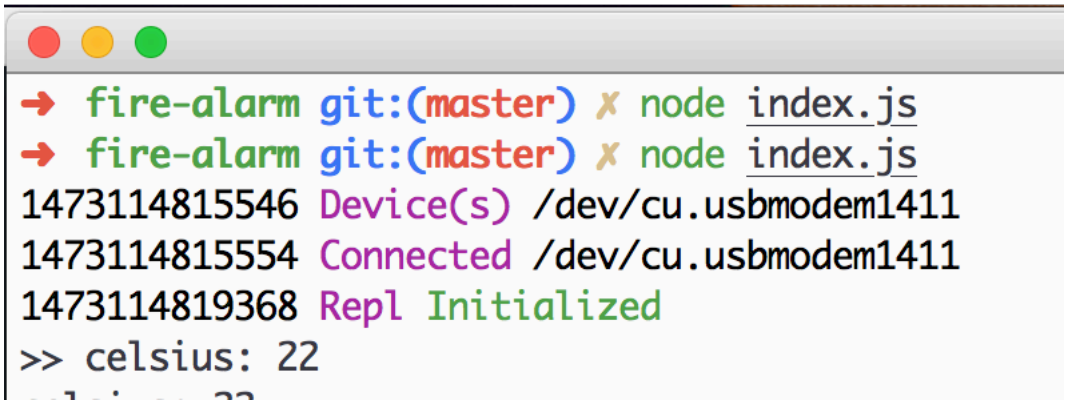
```
1  const five = require('johnny-five');
2  const board = new five.Board();
3
4  function FireAlarm() {
5    return new five.Thermometer({
6      controller: 'LM35',
7      pin: 'A0'
8    });
9  };
10
11 board.on('ready', function() {
12   const temperatureSensor = new FireAlarm();
13
14   setInterval(function() {
15     console.log('celsius: %d', temperatureSensor.celsius);
16   });
17 });
```

Let's then validate the functionality of our code with the Arduino platform. Inside the folder of our project, we will enter in our command line/prompt/terminal the following command:

```
1  $ node src/index.js
```

And this will be the result of our code.

[41] https://github.com/rwaldron/johnny-five/wiki/thermometer



Reading sensor temperature information using javascript

The code is quite simple, as you can see. In the next topic, we will think a little more about our architecture and how to evolve this code for something easier to maintain.

Evolving our initial code

Our initial code is functional, but evolving this code to the next steps is somewhat complex. To facilitate the next steps of our project, we will make some adjustments in our initial code.

You may notice that we have several configurations, such as the `controller` name and `pin` of the sensor. To improve the handling of these settings that are constant, that is, they do not change throughout the life of our application, we will add them in a file with our specific settings.

This will be added to our `src/configuration.js` file.

```
1  // src/configuration.js
2  module.exports = {
3    FIRE_ALARM: {
4      // https://github.com/rwaldron/johnny-five/wiki/thermometer
5      CONTROLLER: 'LM35',
6      PIN: 'A0'
7    },
8    INTERVAL: 1000
9  };
```

This is the contents of our `src/fire-alarm.js` file. Notice that we are now invoking the external configuration code and adding the values in the `CONFIG` variable. This step is interesting because we unlink the basic settings of our class project, which now has the responsibility of dealing with the sensors of the `fire alarm` project.

```
// src/fire-alarm.js
const CONFIG = require('./configuration');
const five = require('johnny-five');
let intervalId = null;

function FireAlarm() {
  this.temperatureSensor = new five.Thermometer({
    controller: CONFIG.FIRE_ALARM.CONTROLLER,
    pin: CONFIG.FIRE_ALARM.PIN
  });
};

FireAlarm.prototype.stopPolling = () => {
  clearInterval(intervalId);
};

FireAlarm.prototype.startPolling = () => {
  const self = this;
  intervalId = setInterval(function() {
    console.log('celsius: %d', self.temperatureSensor.celsius);
  }, CONFIG.INTERVAL);
};

module.exports = FireAlarm;
```

And our main `src/index.js` file will have a simpler content, having the responsibility of initiating the project and polling the `FireAlarm` class instance.

```
// src/index.js
const FireAlarm = require('./fire-alarm');
const five = require('johnny-five');
const board = new five.Board();

board.on('ready', () => {
  fireAlarm = new FireAlarm();
  fireAlarm.startPolling();
});
```

When we run our code from the command, we will see the same result at our command line/prompt.

```
$ npm start
```

With these changes, we have a code of easy maintenance and much more readability to be used in our application. Of course, this is one of the several approaches that can be used in your project, but the focus of this topic is to pass on the idea of always thinking about how to improve our project.

Integrating with Piezo for audible warning

Twilio is a service that allows developers to embed voice, *VoIP* and SMS messages into applications from a RESTful API that provides the voice and SMS capabilities for applications.

The alarm sensor we are going to use is Piezo because it is a simple component to handle and quite cheap, having its value around $ 1.00.

The integration of Piezo into our project is somewhat trivial since Johnny Five already has the `five.Piezo` class.

```
...
this.piezo = new five.Piezo(3);
...
```

This class accepts the pin number that is bound to the sensor and when activating we have the `play` method, which accepts an object with the information:

- `time`: interval between each note;
- `song`: array that accepts arrays of 2 positions with the information of musical note and the time;

You can see an example of the play method being used in the code below.

```
self.piezo.play({
  song: [
    ['G5', 1/4]
  ],
  tempo: 200
});
```

In our project, we are going to make the integration divided into 2 parts. First, we will add the piezo instance in the constructor of our class so that the piezo is accessible in the other methods.

```
...
function FireAlarm() {
  this.piezo = new five.Piezo(3);
  ...
}
...
```

With our instance added and accessible, let's use it in our `startPolling` method. Let's add another validation by accessing the `piezo.isPlaying` boolean that contains the piezo initialization information in our project and, if the temperature is over the threshold and the piezo is accessible, we will trigger our audible alarm. With this, the method will remain as the following code.

```
...
FireAlarm.prototype.startPolling = () => {
  self = this;
  intervalId = setInterval(function() {
    if (self.temperatureSensor.celsius >= CONFIG.FIRE_ALARM.LIMIT && !self.piezo.isP\
laying) {
      self.piezo.play({
        song: [
          ['G5', 1/4],
          [null, 7/4]
        ],
        tempo: 200
      });
      console.log(`Up to the limit: ${self.temperatureSensor.celsius}`);
    } else {
      console.log(`That's ok: ${self.temperatureSensor.celsius}`);
    }
  }, CONFIG.INTERVAL);
};
...
```

Our final `FireAlarm` code will contain the following content:

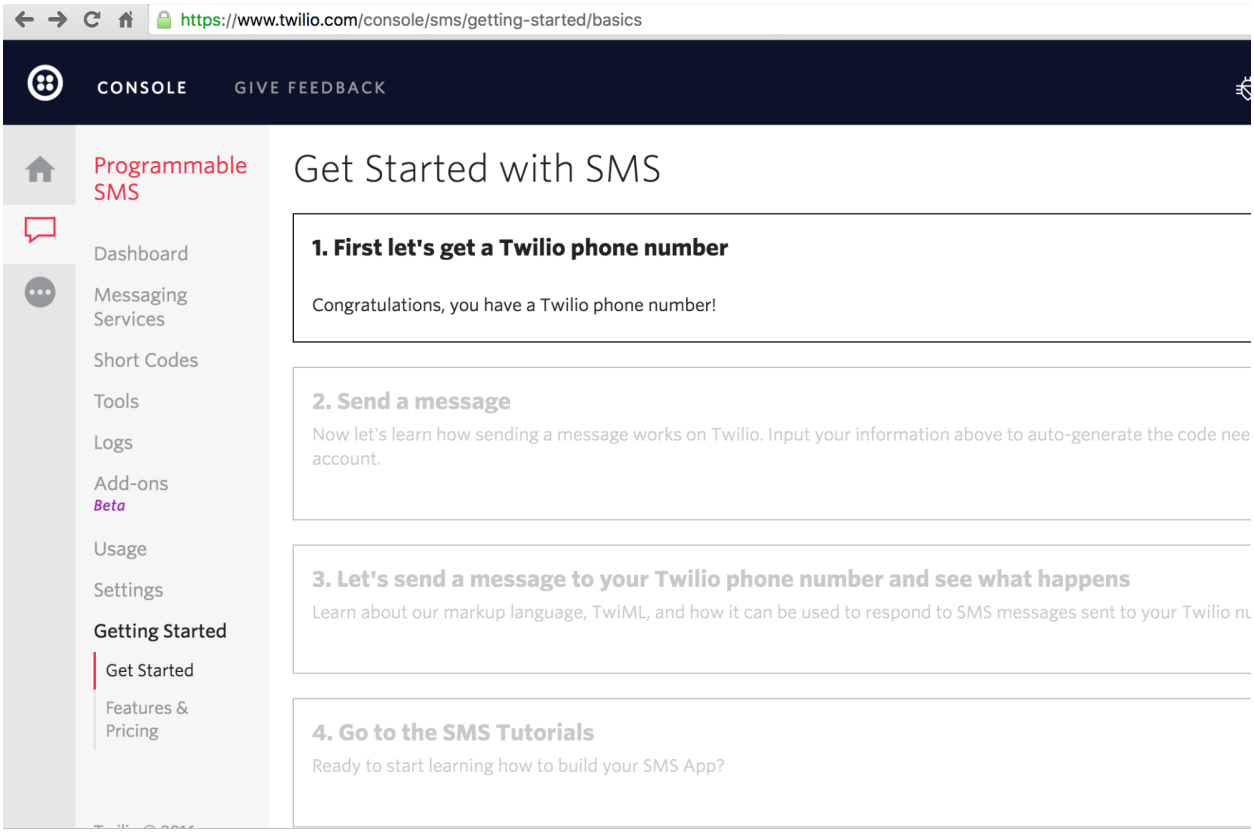
```
const CONFIG = require('./configuration');
const request = require('request');
const five = require('johnny-five');
let intervalId = null;

function FireAlarm() {
  this.piezo = new five.Piezo(3);
  this.temperatureSensor = new five.Thermometer({
    pin: CONFIG.FIRE_ALARM.PIN
  });
};


FireAlarm.prototype.stopPolling = () => {
  clearInterval(intervalId);
};

FireAlarm.prototype.startPolling = () => {
  self = this;
  intervalId = setInterval(function() {
    if (self.temperatureSensor.celsius >= CONFIG.FIRE_ALARM.LIMIT && !self.piezo.isP\
laying) {

      self.piezo.play({
        song: [
          ['G5', 1/4],
          [null, 7/4]
        ],
        tempo: 200
      });

      console.log(`Up to the limit: ${self.temperatureSensor.celsius}`);
    } else {
      console.log(`That's ok: ${self.temperatureSensor.celsius}`);
    }

  }, CONFIG.INTERVAL);
};

module.exports = FireAlarm;
```

With this, we have the first feedback for the users of our application. The next step will be to add the SMS sending functionality using the Twilio API.

Sending SMS to your phone using Twilio

Twilio is a service that allows developers to incorporate voice, VoIP and SMS messaging into applications from a RESTful API that provides voice and SMS capabilities to applications. Client libraries are available in multiple languages â€‹â€‹and, of course, have a client for NodeJS called twilio-node[42].

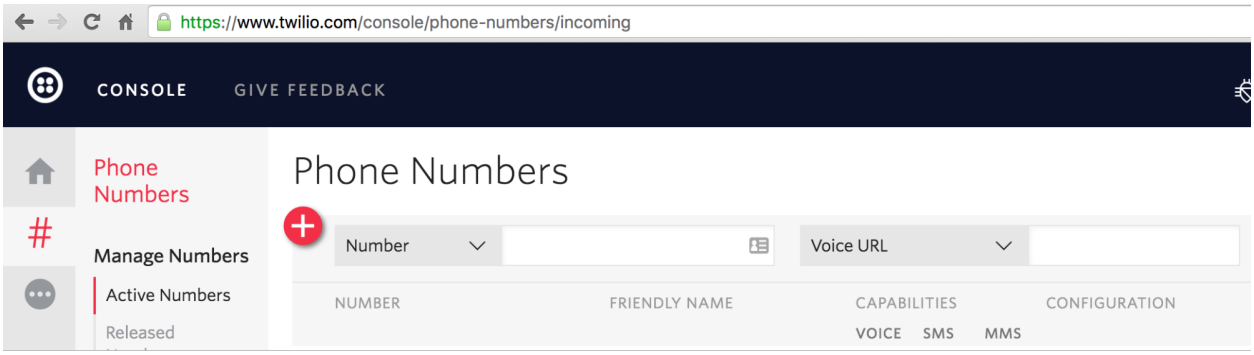
Integrating SMS into our application will be very simple. We go to the site to Twilio and we will enable the SMS service.



SMS setup page in Twilio

It has some paid numbers if you really want to use some product. In our case, we will use the number generated by the service itself and create a trial account on the platform.

[42] https://twilio.github.io/twilio-node



Adding a phone number on Twilio

Now, with Twilio set up, let's start integrating with our fire alarm. First let's add the phone information, SSID of your account, and Twilio's authentication token. Let's add this information in our `src/configuration.js`.

```
module.exports = {
  FIRE_ALARM: {
     LIMIT: 30,
     PIN: 'A0',
     PHONE_NUMBER: ''
  },
  TWILIO: {
    PHONE_NUMBER: '',
    ACCOUNT_SSID: '',
    AUTH_TOKEN: ''
  },
  INTERVAL: 1000
};
```

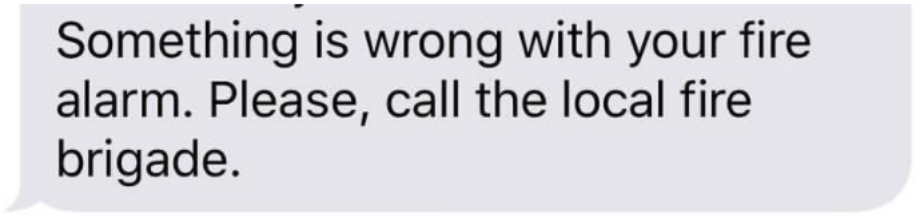
And let's continue with the integration of Twilio into our code by accessing and reading the information added in the `src/configuration.js` file with the Twilio node package.

```
// fire-alarm.js

const CONFIG = require('./configuration');
const request = require('request');
const five = require('johnny-five');
const twilio = require('twilio');
let intervalId = null;

const client = new twilio.RestClient(CONFIG.TWILIO.ACCOUNT_SSID, CONFIG.TWILIO.AUTH_\
TOKEN);

```

```
function FireAlarm() {
  this.piezo = new five.Piezo(3);
  this.temperatureSensor = new five.Thermometer({
    pin: CONFIG.FIRE_ALARM.PIN
  });
};

FireAlarm.prototype.stopPolling = () => {
  clearInterval(intervalId);
};

FireAlarm.prototype.startPolling = () => {
  self = this;
  intervalId = setInterval(function() {
    if (self.temperatureSensor.celsius >= CONFIG.FIRE_ALARM.LIMIT && !self.piezo.isP\
laying) {

      self.piezo.play({
        song: [
          ['G5', 1/4],
          [null, 7/4]
        ],
        tempo: 200
      });

      client.messages.create({
        body: 'Something is wrong with your fire alarm. Please, call the local fire \
brigade.',
        to: CONFIG.FIRE_ALARM.PHONE_NUMBER,
        from: CONFIG.TWILIO.PHONE_NUMBER
      });

      console.log(`Up to the limit: ${self.temperatureSensor.celsius}`);
    } else {
      console.log(`That's ok: ${self.temperatureSensor.celsius}`);
    }
  }, CONFIG.INTERVAL);
};

module.exports = FireAlarm;
```

Let's now run our code by typing the command npm start and this will be the information printed at our prompt/command line.

When we execute our code and the temperature exceeds the configured limit, in addition to the Piezo sensor that will trigger the alarm our customer of Twilio will be activated and will send us an SMS using the previously added configurations. Then you will receive the following message on your cell phone.



SMS sent by Twilio API

With this we have seen the complete integration of our Fire Alarm. Of course, this is the first step, you can evolve the code and add new features. As we already know: the sky is the limit!

But what about the unit tests? We know it's working, but we have to make sure the code has an acceptable level of quality even to evolve our project and add new features. Let's now create our unit tests.

Creating unit tests for Fire Alarm

As commented on in the "Building unit tests for build checker" content, unit testing is just one of several ways to test your software and have reliability in the final product with some automatic validations before deploying our project to production.

Let's now create a folder for our unit tests with the name test. By default, we will instantiate the [MochaJS]test framework (https://mochajs.org), SinonJS[43] for *spies*, *stubs* and *mocks* and ShouldJS[44] for *assertions*. Let's then install these packages as a dependency of project development.

```
1 $ mkdir test
2 $ npm install --save-dev mocha sinon should
```

For our tests, we will reuse the contents of `test/spec-helper.js` and `test/mocha.opts` that we created for our build checker project. It uses the mock-Firmata package[45] to setup the tests in our Johnny-Five application.

[43] http://sinonjs.org
[44] https://shouldjs.github.io
[45] https://github.com/rwaldron/mock-firmata

```
//test/spec-helper.js
require('should');
const mockFirmata = require('mock-firmata');
const five = require('johnny-five');

const board = new five.Board({
  io: new mockFirmata.Firmata(),
  debug: false,
  repl: false
});
```

And this will be the initial content of our `mocha.opts`.

```
--reporter spec
--recursive
--require test/spec-helper.js
--slow 1000
--timeout 5000
```

A quick explanation of the configuration information used:

`--reporter spec`: The type of reporter used to display the test information messages; `--recursive`: flag to identify that the tests should run recursively within the folder; `--require test/spec-helper.js`: setup file to be loaded before running the unit tests; `--low 1000`: Maximum time in milliseconds of tolerance between tests. If this time exceeds this time will be shown the total time of that test with a differentiated colour so that we can make the necessary changes; `--timeout 5000`: Maximum time in milliseconds of tolerance for the completion of each assertion. If this time exceeds this time our tests will return with an error message;

Now, let's create the scenarios for our tests. Let us then define the scenarios that we must cover in our tests:

- Initial application settings;
- Initial information when creating the instance of `FireAlarm`;
- When we start the polling and the sensor signals that the temperature is within the acceptable limit;
- When we start the polling and the sensor signals that the temperature is above the acceptable limit;
- When the piezo sensor is activated;
- When the Twilio SMS API is triggered;

One way to validate when the fire alarm should trigger the audible alarm is to create a *stub* for trigger and use some *spies* for the Twilio API.

As a first step, we will create the tests of our configuration file, which we will divide between the temperature sensor information and the Twilio API. We will make the request for our `src/configuration.js` and we will check the information of the pin to be connected to the sensor, the value of the interval to be used to check the sensor information and the acceptable temperature limit of the environment in which `FireAlarm` will be working and your phone will be set to receive SMS.

```
const CONFIG = require('../src/configuration');

describe('Configuration', () => {

  describe('Temperature information', () => {

    it('should have the sensor port configured', () => {
      CONFIG.FIRE_ALARM.should.have.property('PIN').which.is.a.String()
    });

    it('should have the sensor alarm limit configured', () => {
      CONFIG.FIRE_ALARM.should.have.property('LIMIT').which.is.a.Number()
    });

    it('should have the user phone configured', () => {
      CONFIG.FIRE_ALARM.should.have.property('PHONE_NUMBER').which.is.a.String()
    });
  });
});
```

Now we evaluate the information that will be passed to Twilio, which in our case will be the SSID key, authentication token and the telephone number provided by Twilio.

```
  ...
  describe('SMS information', () => {

    it('should have account ssid configured', () => {
      CONFIG.TWILIO.should.have.property('ACCOUNT_SSID').which.is.a.String()
    });

    it('should have auth token configured', () => {
    CONFIG.TWILIO.should.have.property('AUTH_TOKEN').which.is.a.String()
    });

    it('should have the user phone configured', () => {
      CONFIG.TWILIO.should.have.property('PHONE_NUMBER').which.is.a.String()
```

```
    });
  });
  ...
```

Our `src/configuration.js` test file will look like this.

```
const CONFIG = require('../src/configuration');

describe('Configuration', () => {

  describe('Temperature information', () => {

    it('should have the sensor port configured', () => {
      CONFIG.FIRE_ALARM.should.have.property('PIN').which.is.a.String()
    });

    it('should have the sensor alarm limit configured', () => {
      CONFIG.FIRE_ALARM.should.have.property('LIMIT').which.is.a.Number()
    });

    it('should have the user phone configured', () => {
      CONFIG.FIRE_ALARM.should.have.property('PHONE_NUMBER').which.is.a.String()
    });
  });

  describe('SMS information', () => {

    it('should have account ssid configured', () => {
      CONFIG.TWILIO.should.have.property('ACCOUNT_SSID').which.is.a.String()
    });

    it('should have auth token configured', () => {
      CONFIG.TWILIO.should.have.property('AUTH_TOKEN').which.is.a.String()
    });

    it('should have the user phone configured', () => {
      CONFIG.TWILIO.should.have.property('PHONE_NUMBER').which.is.a.String()
    });
  });

  it('should have the interval polling information', () => {
    CONFIG.should.have.property('INTERVAL').which.is.a.Number()
  });
```

```
38
39 });
```

Let's then create the scenario to validate the code of our `FireAlarm`. One of the things we should keep in mind about our unit testing framework is that its `beforeEach` method, which happens before each `it` method, is going to be used as documentation for playback of each specific scenario.

Let's then explain more about the contents of this file and why of each test. We create the tests of the instance of our `FireAlarm` and its initial attributes.

```
1  const proxyquire = require('proxyquire');
2  const CONFIG = require('../src/configuration');
3  const five = require('johnny-five');
4  const request = require('request');
5  const sinon = require('sinon');
6
7  describe('FireAlarm', () => {
8
9    beforeEach(() => {
10     this.sandbox = sinon.sandbox.create();
11     this.createMessagesSpy = this.sandbox.spy();
12     const restClientMessage = {
13       messages: {
14         create: this.createMessagesSpy
15       }
16     };
17
18     twilioMock = {
19       RestClient: function() {
20         return restClientMessage
21       }
22     };
23
24     const FireAlarm = proxyquire('../src/fire-alarm', {
25       twilio: twilioMock
26     });
27     fireAlarm = new FireAlarm();
28   });
29
30   afterEach(() => {
31     this.sandbox.restore();
32   });
33
```

```
34    it('should have the termometer sensor configured', () => {
35      (fireAlarm.temperatureSensor instanceof five.Thermometer).should.be.equal(true);
36    });
37
38  });
```

Let's validate when we stop our *polling*. Let's now use the `spy` method of the sinon to check if the code used the `clearInterval` method to end with the requests. For this, we will check if the `global.clearInterval` was used once, by accessing the boolean `calledOnce`, which is an internal counter added by the `sinon.spy` method for the tests.

```
1   ...
2   describe('#stopPolling', () => {
3     beforeEach(() => {
4       this.sandbox.spy(global, 'clearInterval');
5       fireAlarm.stopPolling();
6     });
7
8     it('should remove interval', () => {
9       global.clearInterval.calledOnce.should.be.true;
10    });
11  });
12  ...
```

And now the scenarios that happen when the sensor is started. For this, we will assign some spies for the setInterval functions and for the piezo sensor. Our first validation will be to certify that the range is being started when we call the `fireAlarm.startPolling()` method.

```
1   ...
2   describe('#startPolling', () => {
3     beforeEach(() => {
4       this.piezoPlaySpy = this.sandbox.spy();
5
6       const piezoStub = {
7         isPlaying: false,
8         play: this.piezoPlaySpy
9       };
10      this.sandbox.stub(fireAlarm, 'piezo', piezoStub);
11
12      this.sandbox.spy(global, 'setInterval');
13      fireAlarm.startPolling();
14    });
```

```
15
16      afterEach(() => {
17        global.setInterval.restore();
18      });
19
20      it('should creates polling', () => {
21        global.setInterval.calledOnce.should.be.true;
22      });
23    });
24    ...
```

To validate the temperature case above the limit we will use the `clock.tick` method with the information contained in the configuration file, simulating the time and time changes at the time of the test, which helps us to force the event Polling, which uses `setInterval`.

In this scenario we use the `stub` to simulate the return of the sensor with a value above the acceptable and we make sure that the Piezo sensor and the Apollo of Twilio were triggered.

```
1     ...
2     describe('When the temperature is up to the limit', () => {
3
4       beforeEach(() => {
5         clock = this.sandbox.useFakeTimers();
6
7         this.sandbox.stub(fireAlarm, 'temperatureSensor', {
8           celsius: CONFIG.FIRE_ALARM.LIMIT + 1
9         });
10        fireAlarm.startPolling();
11        clock.tick(CONFIG.INTERVAL);
12      });
13
14      afterEach(() => {
15        clock.restore();
16      });
17
18      it('should trigger piezo sensor alarm', () => {
19        this.piezoPlaySpy.calledOnce.should.be.true;
20      });
21
22      it('should send the SMS to user', () => {
23        this.createMessagesSpy.calledOnce.should.be.true;
24      });
25
```

```
26      });
27      ...
```

Now we will validate the ambient temperature scenario in acceptable levels. The checks use the same approaches as the previous one, but in this case, we make sure that the piezo sensor and the Twilio API **were not triggered**.

```
1       ...
2       describe('When the temperature is NOT up to the limit', () => {
3
4         beforeEach(() => {
5           clock = this.sandbox.useFakeTimers();
6
7           this.sandbox.stub(fireAlarm, 'temperatureSensor', {
8             celsius: CONFIG.FIRE_ALARM.LIMIT - 1
9           });
10
11          fireAlarm.startPolling();
12          clock.tick(CONFIG.INTERVAL);
13        });
14
15        afterEach(() => {
16          clock.restore();
17        });
18
19        it('should NOT trigger piezo sensor alarm', () => {
20          this.piezoPlaySpy.calledOnce.should.be.false;
21        });
22
23        it('should NOT send the SMS to user', () => {
24          this.createMessagesSpy.calledOnce.should.be.false;
25        });
26
27      });
28      ...
```

Based on our test scenarios, this is the test content of our fire alarm file:

```
const proxyquire = require('proxyquire');
const CONFIG = require('../src/configuration');
const five = require('johnny-five');
const request = require('request');
const sinon = require('sinon');

describe('FireAlarm', () => {

  beforeEach(() => {
    this.sandbox = sinon.sandbox.create();
    this.createMessagesSpy = this.sandbox.spy();
    const restClientMessage = {
      messages: {
        create: this.createMessagesSpy
      }
    };

    twilioMock = {
      RestClient: function() {
        return restClientMessage
      }
    };

    const FireAlarm = proxyquire('../src/fire-alarm', {
      twilio: twilioMock
    });
    fireAlarm = new FireAlarm();
  });

  afterEach(() => {
    this.sandbox.restore();
  });

  it('should have the termometer sensor configured', () => {
    (fireAlarm.temperatureSensor instanceof five.Thermometer).should.be.equal(true);
  });

  describe('#stopPolling', () => {
    beforeEach(() => {
      this.sandbox.spy(global, 'clearInterval');
      fireAlarm.stopPolling();
    });

```

```
44      it('should remove interval', () => {
45        global.clearInterval.calledOnce.should.be.true;
46      });
47    });
48
49    describe('#startPolling', () => {
50      beforeEach(() => {
51        this.piezoPlaySpy = this.sandbox.spy();
52
53        const piezoStub = {
54          isPlaying: false,
55          play: this.piezoPlaySpy
56        };
57        this.sandbox.stub(fireAlarm, 'piezo', piezoStub);
58
59        this.sandbox.spy(global, 'setInterval');
60        fireAlarm.startPolling();
61      });
62
63      afterEach(() => {
64        global.setInterval.restore();
65      });
66
67      it('should creates polling', () => {
68        global.setInterval.calledOnce.should.be.true;
69      });
70
71      describe('When the temperature is up to the limit', () => {
72
73        beforeEach(() =>  {
74          clock = this.sandbox.useFakeTimers();
75
76          this.sandbox.stub(fireAlarm, 'temperatureSensor', {
77            celsius: CONFIG.FIRE_ALARM.LIMIT + 1
78          });
79          fireAlarm.startPolling();
80          clock.tick(CONFIG.INTERVAL);
81        });
82
83        afterEach(() => {
84          clock.restore();
85        });
86
```

```
87        it('should trigger piezo sensor alarm', () => {
88          this.piezoPlaySpy.calledOnce.should.be.true;
89        });
90
91        it('should send the SMS to user', () => {
92          this.createMessagesSpy.calledOnce.should.be.true;
93        });
94
95      });
96
97      describe('When the temperature is NOT up to the limit', () => {
98
99        beforeEach(() => {
100         clock = this.sandbox.useFakeTimers();
101
102         this.sandbox.stub(fireAlarm, 'temperatureSensor', {
103           celsius: CONFIG.FIRE_ALARM.LIMIT - 1
104         });
105
106         fireAlarm.startPolling();
107         clock.tick(CONFIG.INTERVAL);
108       });
109
110       afterEach(() => {
111         clock.restore();
112       });
113
114       it('should NOT trigger piezo sensor alarm', () => {
115         this.piezoPlaySpy.calledOnce.should.be.false;
116       });
117
118       it('should NOT send the SMS to user', () => {
119         this.createMessagesSpy.calledOnce.should.be.false;
120       });
121     });
122   });
123 });
```

With that we finished our first project with unit tests, covering all our possible scenarios. In the next chapter, we'll look at other services that make life easier for us by sending the code to production, such as seamless integration servers, code coverage, and complexity in an automated way.

Supporting Your Code on Multiple Operating Systems

In this stage of the book, we will then validate and verify the coverage of tests in our project on different operating systems, as well as enable different web services for workflow improvements, such as continuous integration tools and code coverage.

This step is very important because these tools help us in the security process of our code, checking different criteria of acceptance of our application in an automated way.

Adding Continuous Integration Servers to Your Project

Like all quality projects, our Nodebots project will be concerned with some other aspects, such as automating the test suite, build and other tasks relevant to our project.

For this, we will rely on the help of a continuous integration server. There are several in the market, being free or paid, and in this stage of the book, we will know a little more about the operation and configuration of two of them: Travis-CI and Appveyor.

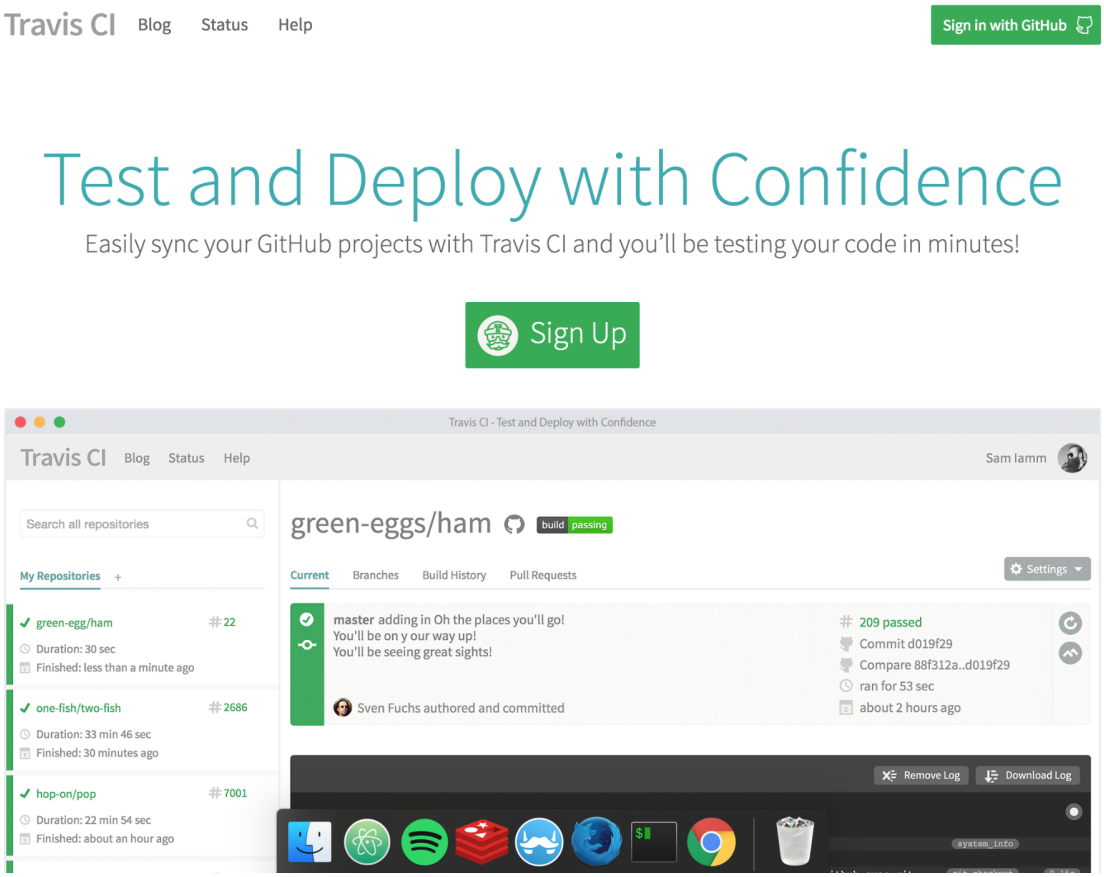
Travis-CI: Checking Your Code on Linux and OSX

Knowing that currently, the most used operating systems are Unix/Linux, Windows and OSX we will create checks for each of them and for this the Travis-CI comes into play.

It is one of the most famous services of continuous integration[46] and assists in the process of integrating the new features or bug fixes of the code of the current project in several Environments, and can even deploy for production if all the validation steps are correct.

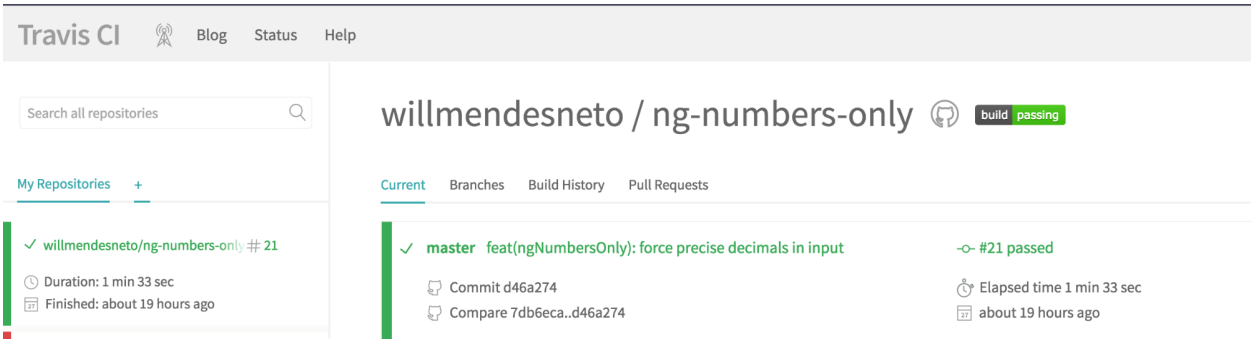
Let's go to the official project site travis-ci[47] and enable access using our Github account. Click on the *"Sign up"* button and enable access to your repositories.

[46] http://blog.caelum.com.br/integracao-continua/
[47] https://travis-ci.org/



Travis-CI Service Site

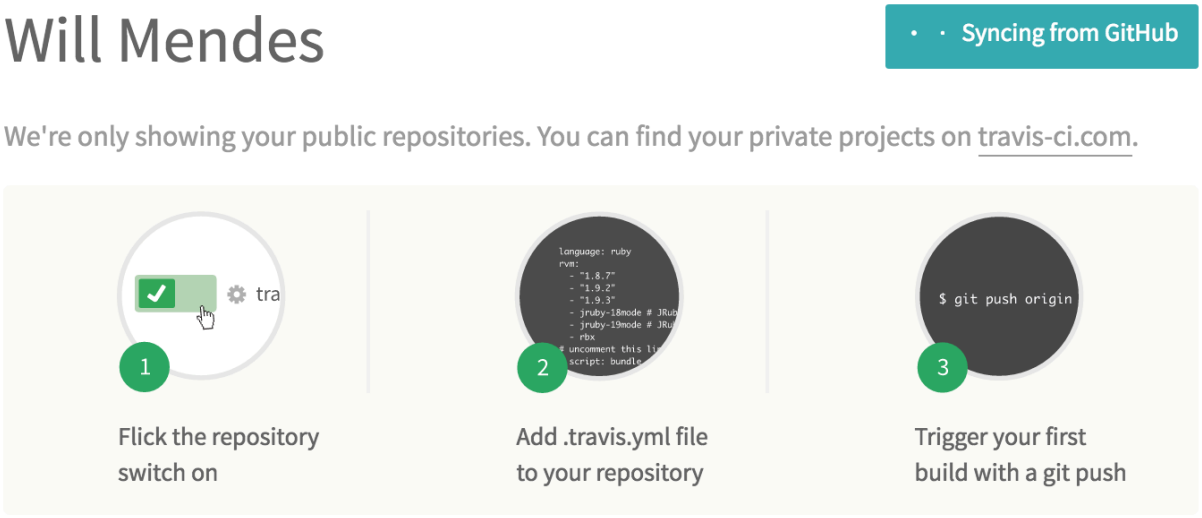
After this step, you will be redirected to a new page with all your repositories. To add a new one just click on the "+" icon next to the text *"My Repositories"*.



Page of a repository configured in Travis-CI

Now, you will be redirected to a new page with all your repositories. To add a new one just click on the "+" icon next to the text "My Repositories".

This next step is very simple since the page has a tutorial showing each of the steps to enable the integration of Travis-CI with its repository in Github, as we can see in the image below.



Synchronizing repositories with the service

On the same page, all your repositories will be listed so you can choose and enable Travis-CI integration with your project. To enable it, just click the grey button with an "X" and when it changes colour to green it means that everything went as expected and its repository is synchronised with Travis-CI.

Travis-CI is fully configurable and you can add information from a wide range of commands, from commands to be invoked before, during or after the build, and even configure the types of operating systems that the tasks should take place.

These settings will be in the `.travis.yml` file that will be in the root folder of our project. Let's explain a bit more about configuring these tasks in Travis-CI.

Activating travis webhook

First, in the `.travis.yml` file, we will add the `os` field, with the appropriate information of the operating systems used for our tests.

```
...
os:
  - linux
  - osx
...
```

We will also add the `"node_js"` field, which will be our information about the NodeJS versions that

the tasks should be used in our tasks. In our case, we will only add one version, but we could add several others based on our support needs, for example.

```
...
node_js:
  - '12.16.2'
...
```

Our continuous integration server is nothing more than a container with a complete operating system. So we can also configure environment variables in it. In this case, we will add the variable `NO_SERIALPORT_INSTALL`, specifying that we should not install the 'serialport' package in this case because it is a test that uses a `mock` of a physical board.

NOTE: The idea of this book is to focus on the concepts directly related to Nodebots and integrations with the javascript repository created, so I will not explain the concept of `containers`. If you want to know more about this concept used by Travis-CI, visit the official Docker project website[48].

```
...
env:
  - NO_SERIALPORT_INSTALL=1
...
```

We can also define the set of tasks that will be used before and after our Travis script. In this case, we will use `before` for the commands that must occur before our main script and `after` for the commands that must occur after the Travis commands, as you can see in the following code snippet:

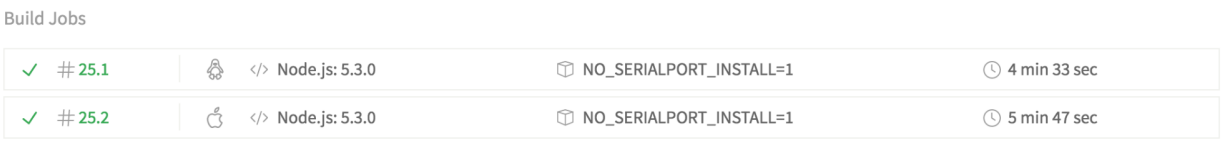
```
...
before_script:
  - 'npm install'


after_script:
  - 'make test'
...
```

In this case, we are installing our dependencies and running our tests. All this in a very simple and well-defined way. The contents of our `.travis.yml` file with all the changes will be as follows:

[48] https://www.docker.com

```
language: node_js
os:
  - linux
  - osx
node_js:
  - '12.16.2'
before_script:
  - 'npm install'


after_script:
  - 'make test'
env:
  - NO_SERIALPORT_INSTALL=1
```

We can see that the Travis-CI build is a bit different now since we are running the same setup on Linux and OSX operating systems, identified by the icons of each operating system.



List of used operating systems

With the integration tested, let's then put the Travis-ci badge in our `README.md` file in the repository. With this, you will see an image with the status of the build.

```
[![Build Status](https://travis-ci.org/willmendesneto/build-checker.png?branch=maste\
r)](https://travis-ci.org/willmendesneto/build-checker)
```

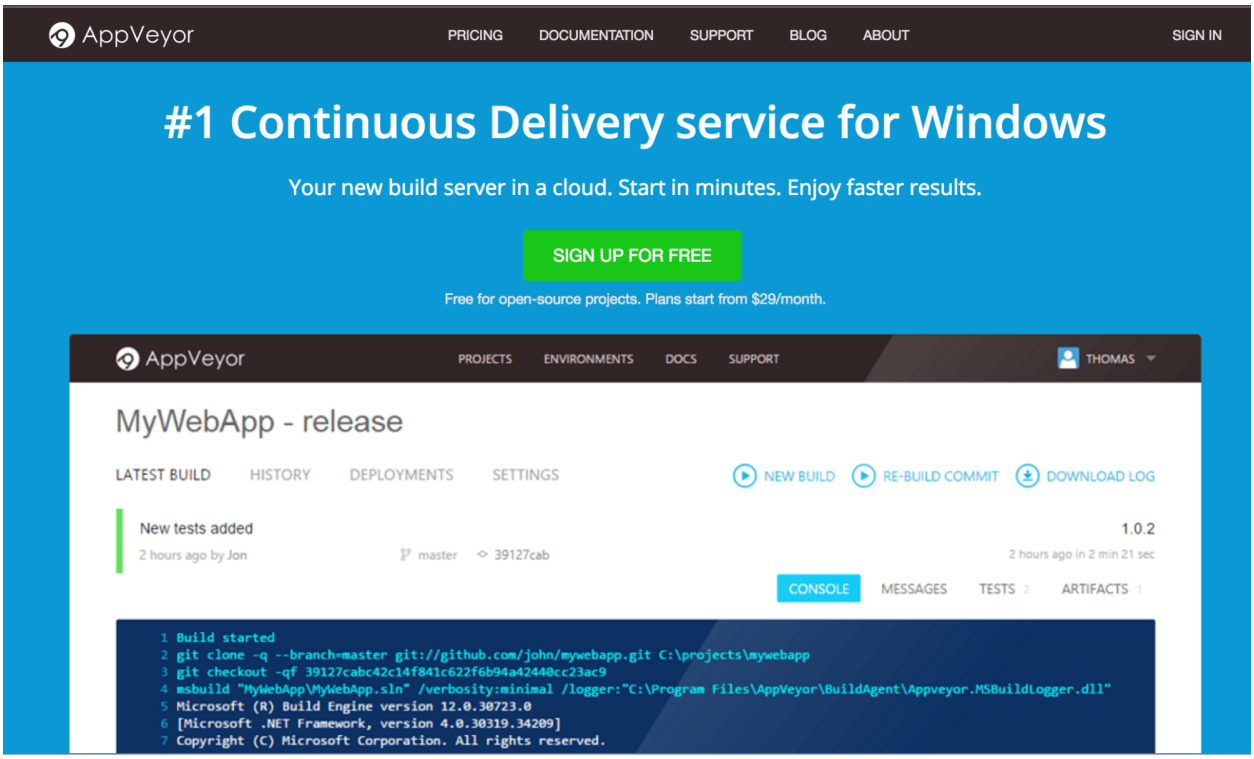
With this, we have finished our integration with Travis-CI continuous integration server and we have our entire suite of tests running on Linux and OSX systems. In this next step we will configure the same tasks, but to be verified by the Windows operating system, using another continuous integration server called Appveyor.

Appveyor: Checking Your Code on Windows

Many projects are developed on Unix-based operating systems by default and adding support for Windows was considered a big challenge for some, since building a Windows test environment was not trivial, requiring the purchase of software licenses.

The continuous integration service Appveyor[49] is one of the solutions used for testing projects hosted on GitHub in Windows environments, facilitating this process and ensuring that our code is cross-Platform[50], running on major operating systems.

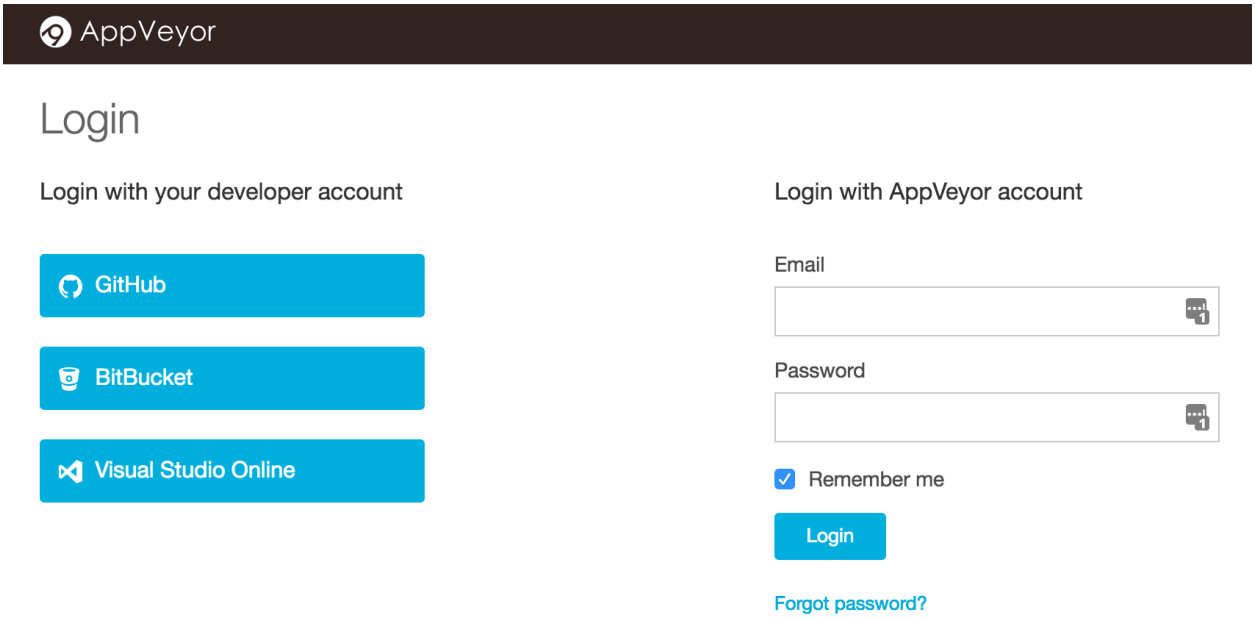
Adding Appveyor support to our project is a fairly simple task. We will then visit the official website of the project and create a login with our information.



Appveyor Site

On the login page, we have some options listed with support for some of the major code repositories on the internet. In this option we will use Github, to facilitate the next steps, but it is worth remembering that you can use any of the options supported for *login*.

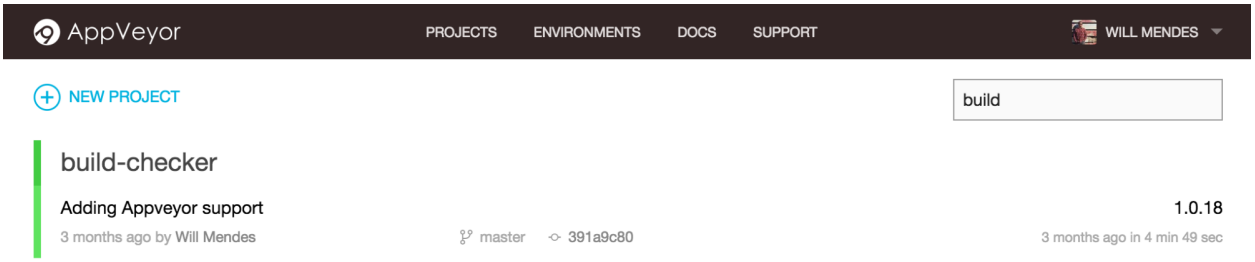
[49] https://www.appveyor.com/
[50] https://en.wikipedia.org/wiki/Cross-platform



Logging in to Appveyor

With your user created and your access working, the main page after login is a page listing all of your repositories based on a category located on the left side of the site. In our case, we will choose the `build-checker` project and click on the 'Add' button to add the support to our project.

We will then see the page of our project with the information specific to it, currently.



Now that we have our project configured we will create our `appveyor.yml` file, where our test settings will be. This file is very similar to Travis-CI in some ways.

The contents of our `appveyor.yml` file with all the changes will be as follows:

We will add the version of NodeJS used in the environment field of our configuration file.

```
...
environment:
  matrix:
    - nodejs_version: "12"
...
```

The platform field will be used to describe the platforms used. Note that in this case instead of having the differentiation between operating systems, we have between operating system platforms (x86 and 64x).

This is also interesting if there is a need to have a notion of the difference in performance, performance and other aspects of the same application on different platforms.

```
...
platform:
  - x86
  - x64
...
```

The install field will list our initial commands. Note that ps is a command to install NodeJS with the one specified in the file.

After this step we cleared the cache using the npm cache clean command for security measure in order to avoid possible false positives in our tests and, after completing this command, we will then install our dependencies using the npm install command.

```
...
install:
  - ps: Install-Product node $env:nodejs_version
  - npm cache clean --force
  - npm install
...
```

The test_script field will have the list of our commands to execute at the time of running our tests. We are directly accessing the node_modules folder and invoking the tests from them with the `node_modules/.bin/Istanbul cover node_modules/mocha/bin/_mocha -- -R dot` command, because we use the make test command in our `npm test`.

```
...
test_script:
   Run the test
  - cmd: node_modules/.bin/istanbul cover node_modules/mocha/bin/_mocha -- -R dot
```

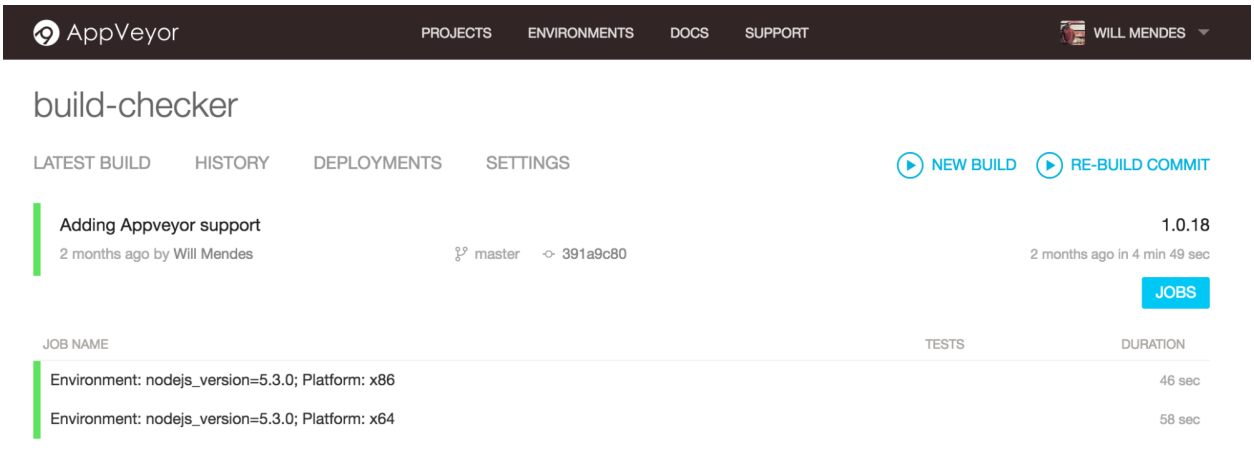
As our case does not require the creation of a build, we will add the information in our file with the `off` value and we will configure our build to be finalised as soon as possible by adding the `fast_finish` field with the value `true`.

```
build: off
matrix:
  fast_finish: true
...
```

The final content of our `appveyor.yml` file with all changes will be as follows:

```
 Fix line endings on Windows
init:
  - git config --global core.autocrlf true
environment:
  matrix:
    - nodejs_version: "12"
platform:
  - x86
  - x64
install:
  - ps: Install-Product node $env:nodejs_version
  - npm cache clean --force
  - npm install
test_script:
   Run the test
  - cmd: node_modules/.bin/istanbul cover node_modules/mocha/bin/_mocha -- -R dot
build: off
matrix:
  fast_finish: true
```

Notice that in this case, we have the list of our differentiated build by platforms on the listing page.



List of platforms used by the service

With the new integration tested, we will then update the README.md file from the repository with the Appveyor badge. With this, you will see an image with the status of the build, as well as what we have inserted previously.

```
[![Build Windows Status](https://ci.appveyor.com/api/projects/status/github/<nome-do\
-seu-usuario-ou-organização>/<nome-do-seu-repositório>?svg=true)](https://ci.appveyo\
r.com/project/<nome-do-seu-usuario-ou-organização>/<nome-do-seu-repositório>/branch/\
master)
```

Notice that we have two tags in this code snippet. Replace this information as follows:

- `<your-user-or-organisation-name>`: name of your user or organisation;
- `<your-repository-name>`: name of your repository;

For example, based on the example repository, our badge will have the following content.

```
[![Build Windows Status](https://ci.appveyor.com/api/projects/status/github/willmend\
esneto/build-checker?svg=true)](https://ci.appveyor.com/project/willmendesneto/build\
-checker/branch/master)
```

As you might realise adding support for multiple operating systems and platforms is quite a simple task with Appveyor. The next steps in the book will be more focused on improving the automation of checking our code coverage.

Code coverage for your code

Once our repository is communicated with continuous integration services, we will now add new tools. This time the focus is on the coverage of our code, checking if everything is being properly validated in an automated way even before we continue with the other development stages of our Nodebots.

Checking the code coverage of our project: Getting to know Istanbul

Istanbul[51] is a NodeJS package for verifying code coverage in our repository using various parameters such as code-line coverage, functions, declarations, and reverse engineering[52].

Let's then add this package to our project using the following command.

[51] http://gotwarlost.github.io/istanbul
[52] Https://en.wikipedia.org/wiki/Reverseengineering

```
$ npm install --save-dev istanbul
```

To verify that it is integrated into our repository, simply type in our prompt/command line.

```
$ ./node_modules/.bin/istanbul help
```

```
1. willmendesneto@Will-Ma
➜ build-checker git:(master) ./node_modules/.bin/istanbul help

Usage: istanbul help config | <command>

`config` provides help with istanbul configuration

Available commands are:

      check-coverage
              checks overall/per-file coverage against thresholds from coverage
              JSON files. Exits 1 if thresholds are not met, 0 otherwise


      cover   transparently adds coverage information to a node command. Saves
              coverage.json and reports at the end of execution


      help    shows help


      instrument
              instruments a file or a directory tree and writes the
              instrumented code to the desired output location


      report  writes reports for coverage JSON objects produced in a previous
              run


      test    cover a node command only when npm_config_coverage is set. Use in
              an `npm test` script for conditional coverage


Command names can be abbreviated as long as the abbreviation is unambiguous
istanbul version:0.4.4


➜ build-checker git:(master)
```

Output from command `istanbul help`

As we previously integrated MochaJS[53] into our repository when we created the project tests, we can simply type the following command at our command prompt.

[53] https://mochajs.org/

```
$ ./node_modules/.bin/istanbul cover ./node_modules/.bin/_mocha
```

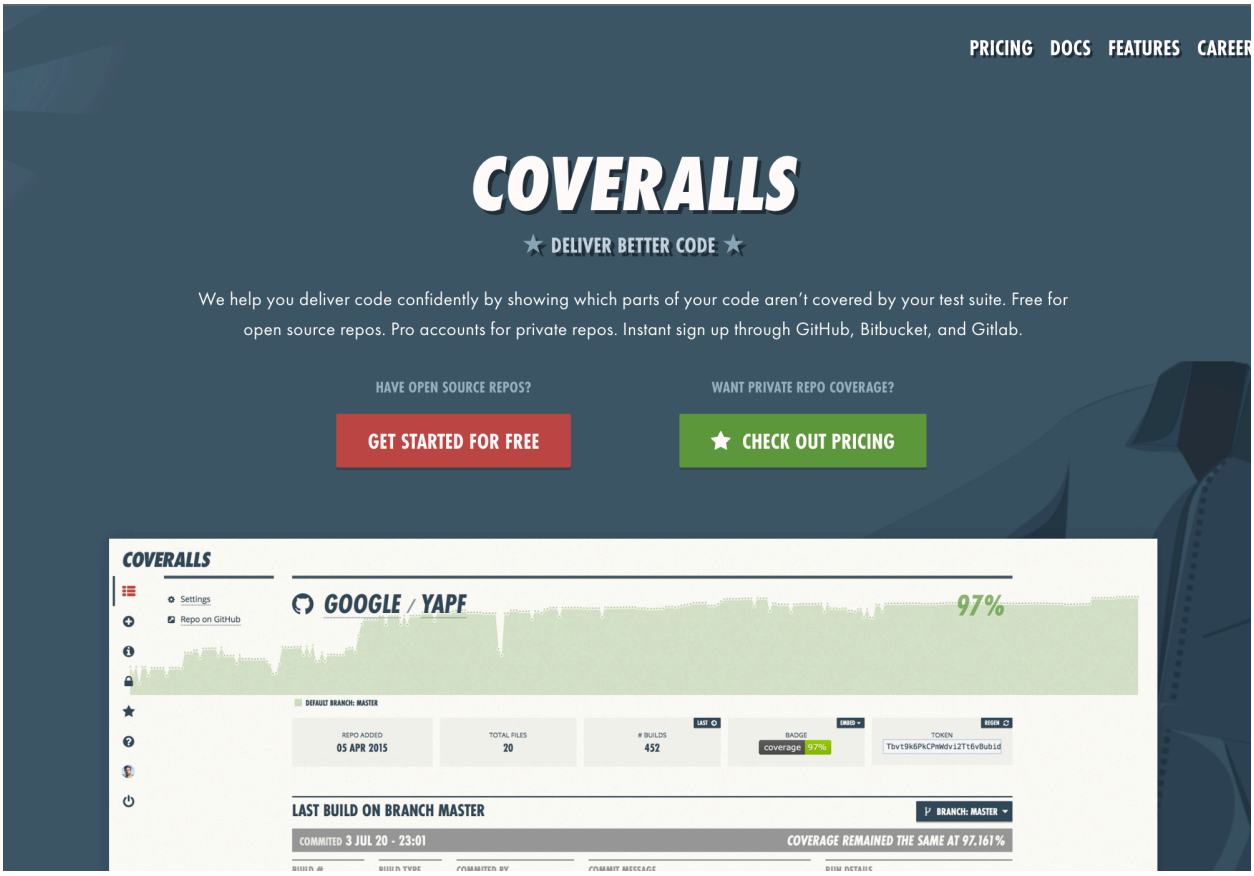
The return will be the same as the image below. You may notice that we now have some new information in the footer of test messages, such as percentages of rows, functions, branches, and declarations of methods, classes, or objects.

Output of the `mocha` command

Notice that we now have a new folder called coverage with some files and all this information listed in our command line. We will use them in the next steps for integration with the Coveralls service.

Integrating Continuous Integration Server with Coveralls

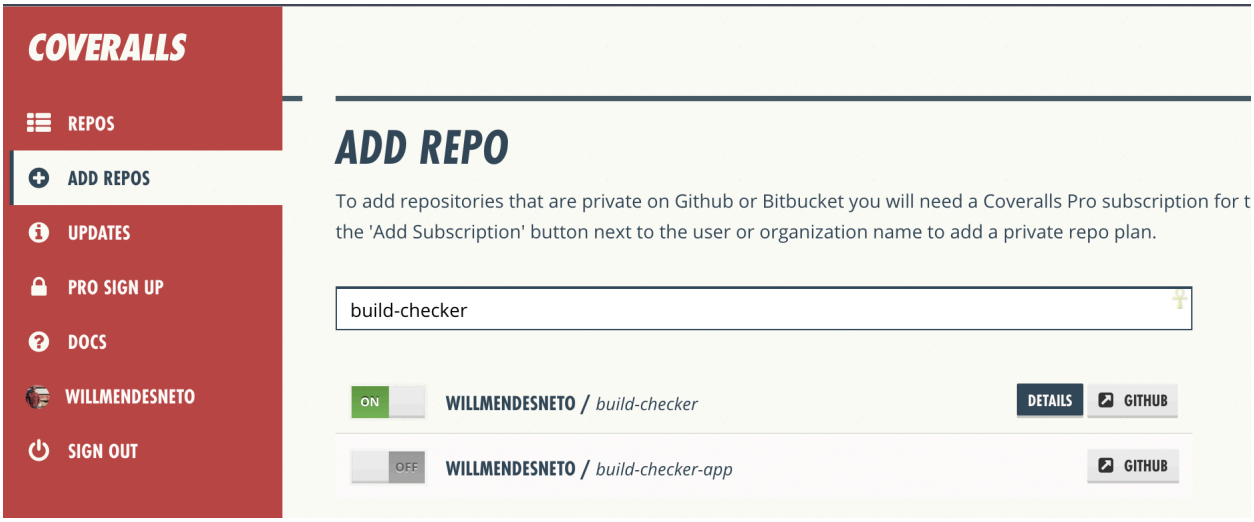
With the code coverage information collected, we will then integrate a new service called coveralls[54]. It will be used to integrate code coverage data and make it visible by adding a badge in our `README.md`.



Coveralls service website

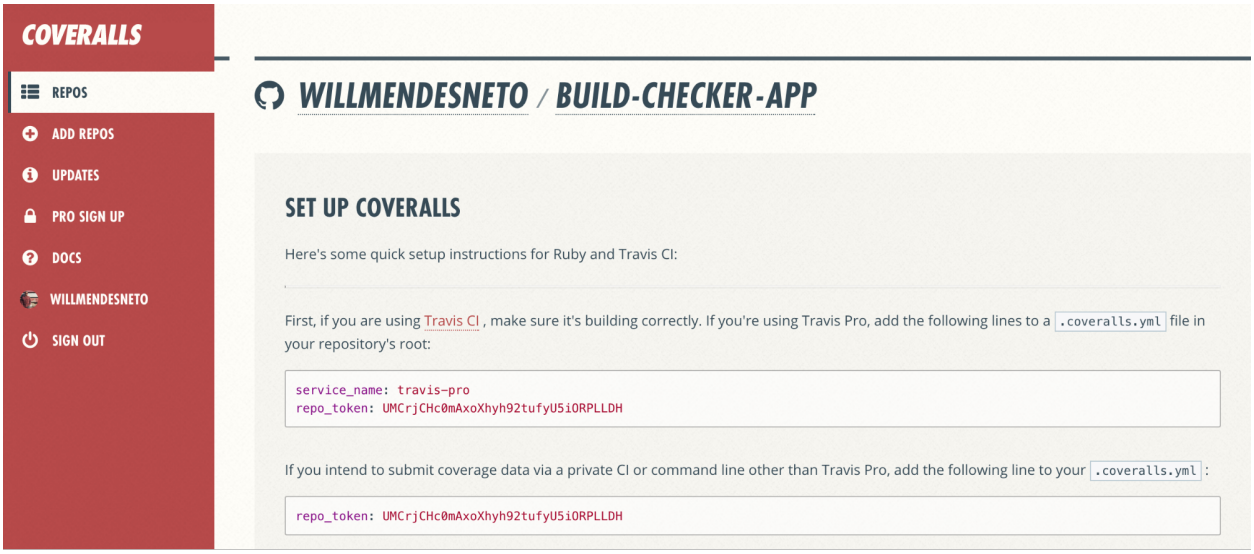
The login is very simple and you will have to enable integration with your Github. After this step, you will see a list with all your repositories registered in Github. Click the button to the left of your listed repository and wait for the message `"Off"` to become `"On"`.

[54] https://coveralls.io/



Adding repositories

Note that with the repository enabled, we now have a link to the details page. By clicking this link we will be directed to a page with all the initial information for the project setup in coveralls. For our solution, we will use the option to add coveralls information to the `.coveralls.yml` file.



We will then copy this content from the file option on the setup page and create the new file in our project. Within our local repository, we will type the following command via prompt/command line.

```
$ touch .coveralls.yml
```

We will open this file in our editor and we will add the content to this file. After this step, we will add the NodeJS package to our list of development dependencies to integrate the coveralls infrastructure into our project by typing the following command.

```
$ npm install --save-dev coveralls
```

Once we submit a new code, we can see that we have the percentage of code coverage information visible on the coveralls website in the area of our repository. With this we can follow all variations of code coverage, we create validations and more.

After that, we can add a new badge with the code coverage information for our project in the `README.md` the file contained in the project repository. The badge pattern is quite simple:

```
[![Coverage Status](https://coveralls.io/repos/<nome-do-seu-usuario-ou-organização>/\
<nome-do-seu-repositório>/badge.svg?branch=master)](https://coveralls.io/r/<nome-do-\
seu-usuario-ou-organização>/<nome-do-seu-repositório>?branch=master)
```

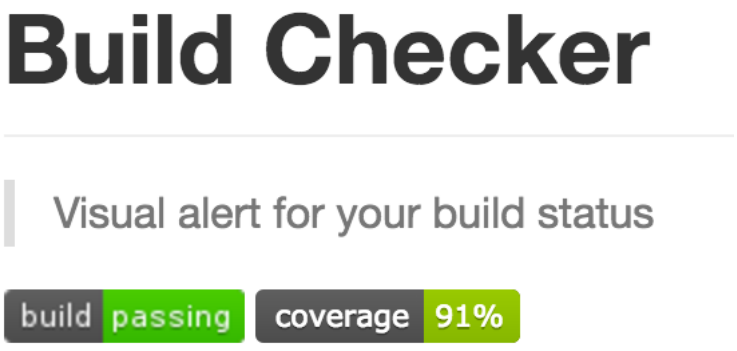
Notice that we have two tags in this code snippet. Replace this information as follows:

- `<your-user-or-organisation-name>`: name of your user or organisation;
- `<your-repository-name>`: name of your repository;

For example, based on the example repository, our badge will have the following content.

```
[![Coverage Status](https://coveralls.io/repos/willmendesneto/build-checker/badge.sv\
g?branch=master)](https://coveralls.io/r/willmendesneto/build-checker?branch=master)
```

After adding and saving this code, the final result to be rendered will be something similar to the following image.



And with this, we conclude our integration with the coveralls service. This is just a simple example of one of the many features of this service and I strongly recommend that you read the coveralls documentation[55] so that you have a greater This service.

Checking code complexity with PlatoJS

PlatoJS is a NodeJS package that will help us in some validations of our nodebots code. It creates a report using some data generated through static analysis of the code of our project that shows us some information such as code complexity, maintenance difficulty, lines of code, possible implementation errors, among other relevant data.

> If you would like to know more about PlatoJS, please visit the repository in the Github of the project[56]

Its installation is very easy. Just type the command:

```
$ npm install --save-dev plato
```

And after this step, the plato was installed locally as a development dependency in our `node_modules` folder of our project. Our next step is to add a new NPM command. Now we will have the `code-analysis` command that will trigger the plate to our project.

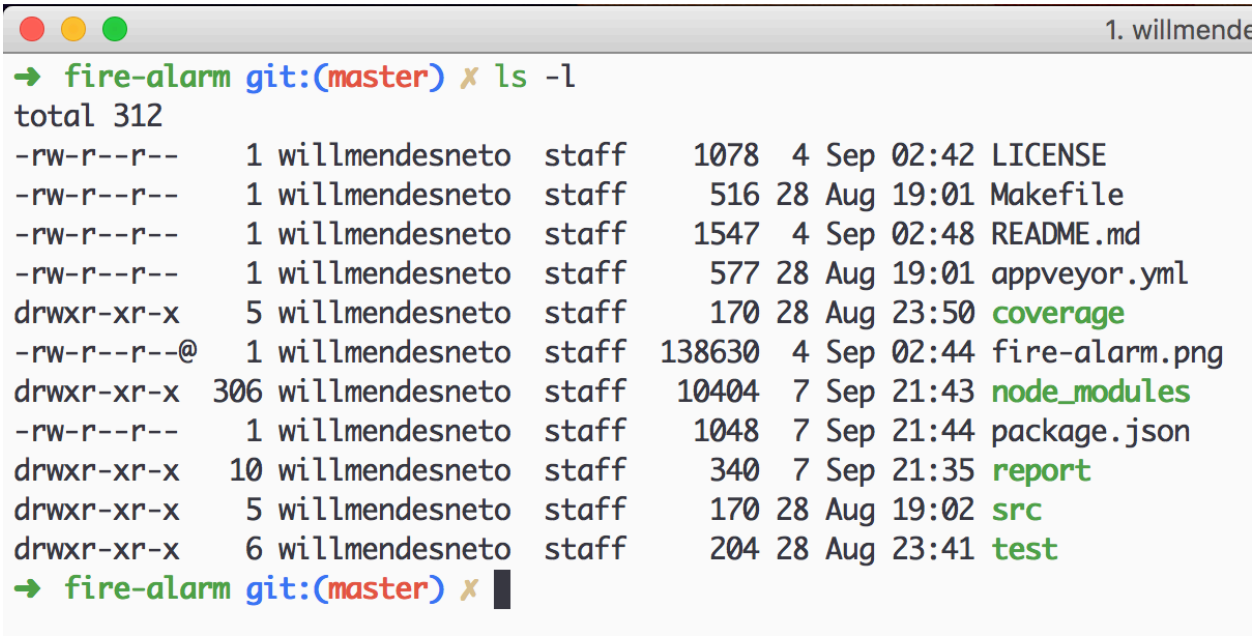
```
{
  ...
  "scripts": {
    "start": "nodemon ./src/index.js -e js,json --watch ./src",
    "test": "make test",
    "code-analysis": "plato -r -d report src test"
  },
  ...
}
```

And to trigger the PlatoJS, just type:

```
$ npm run code-analysis
```

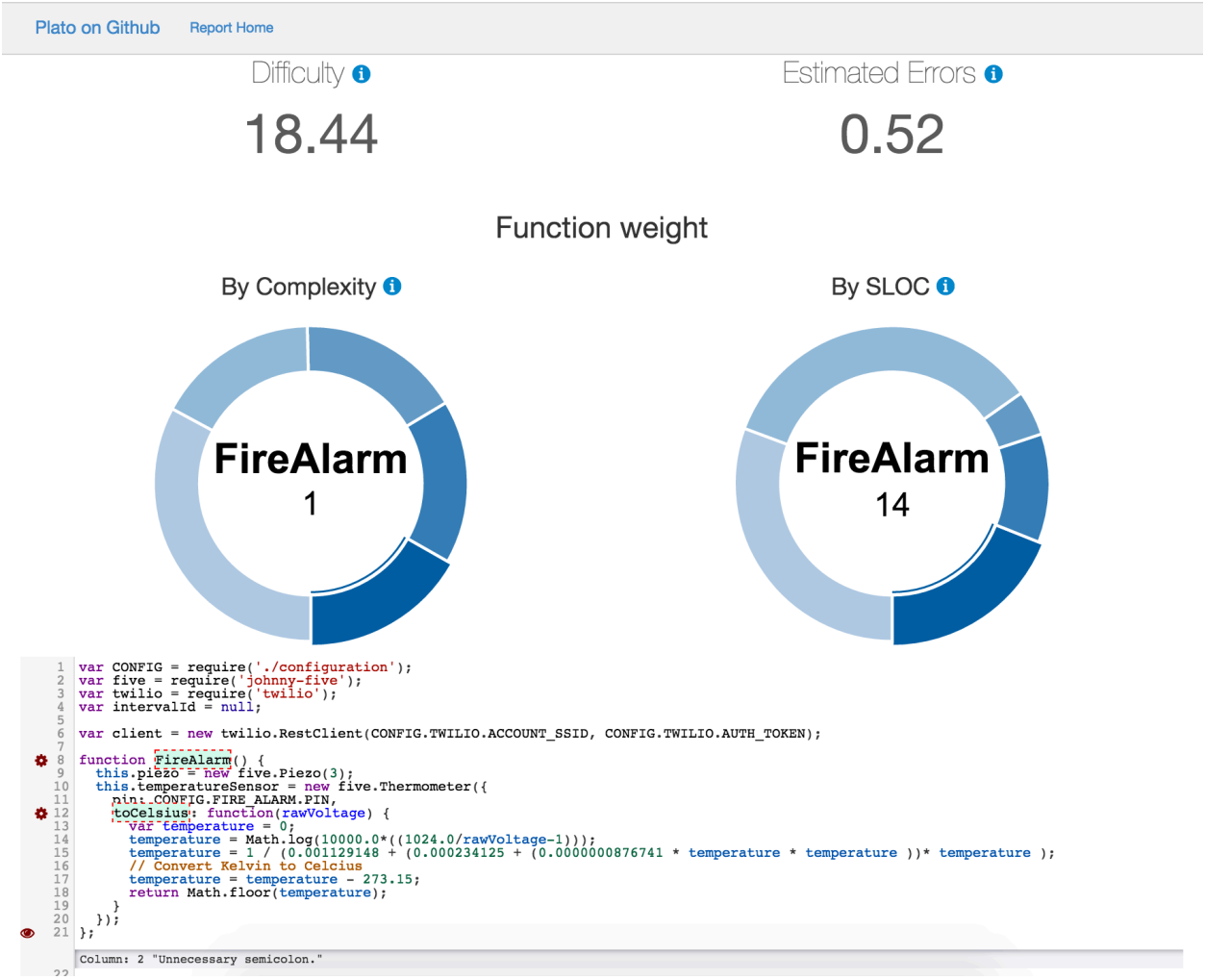
And after this command will be created a folder of name `report` with the information of the analysis of our repository.

[55] https://coveralls.zendesk.com/hc/en-us
[56] https://github.com/es-analysis/plato



Within this folder, we will have several files with the information returned from the PlatoJS analysis that we can see more details by accessing the `index.html` file in our browser.

This page will have information on each file and graphs showing data such as level of complexity and lines of code, as we can see in the figure below.



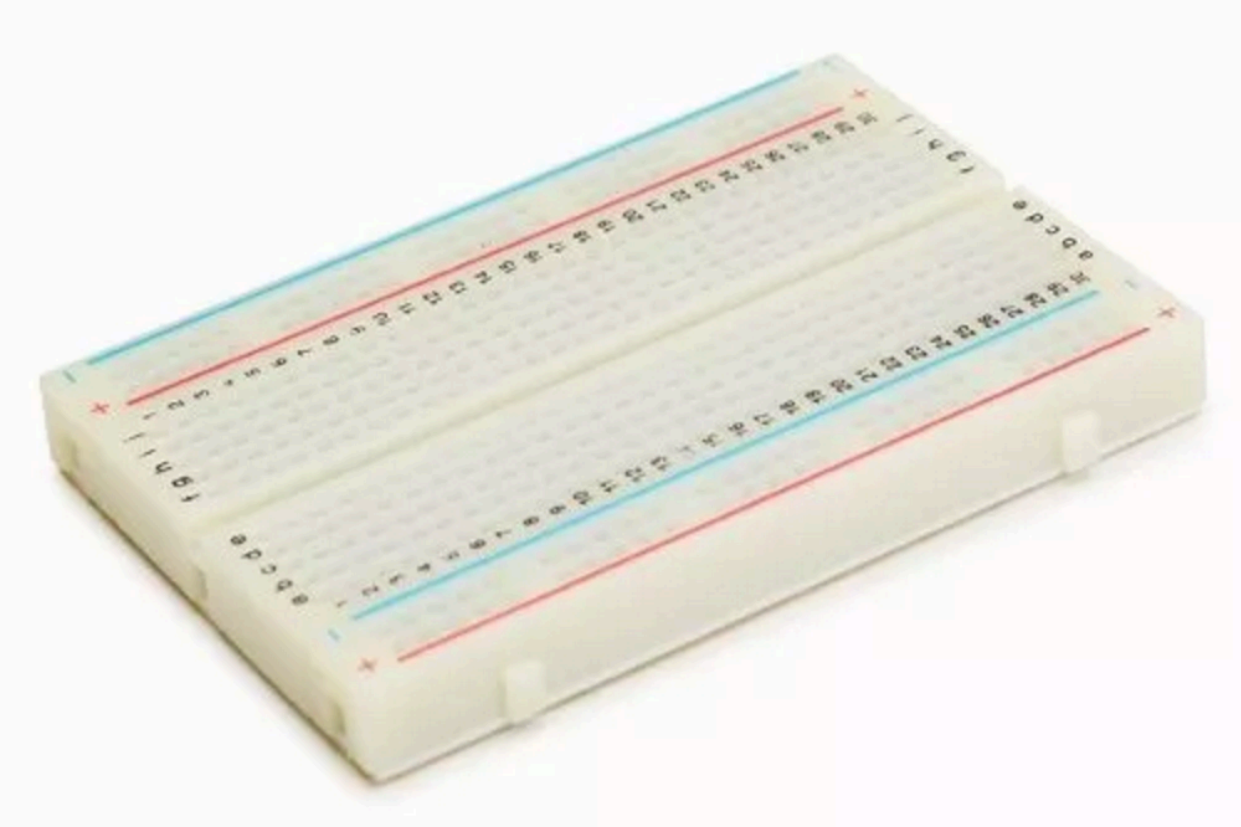
Report page with information extracted by PlatoJS

Appendix

Breadboard

These are great units for making temporary circuits and prototyping when you need prototype a circuit quickly and temporarily.

Breadboard are boards that can simulate electronic component connections and since they are not permanent, it is easy to remove a component if you make a mistake, or just start over and do a new project.



Breadboard

Piezo

Piezo is a sensor that emits a beep, such as a horn. This signal can be created in an application Nodebots from a numeric value or an abstraction of sound/music notes, which makes their

manipulation simpler.

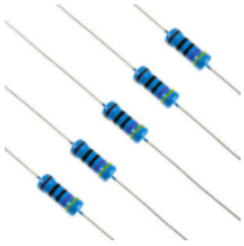
Typical uses of buzzers include alarm devices, timers, and confirmation of user input such as a mouse click or keystroke.

Piezo

Resistors

The resistors are widely used in electronics as one of the first electronic components that users have the first contact and one of the most used. They are small enough in pill form with stripes in most cases. Because resistance is an essential element of nearly every electronic circuit, you'll use them in just about every circuit that you build.

A resistor is an electronic component that limits the flow of electrons dissipating energy in the form of heat, since the electricity has to struggle to flow through something with a high resistance. It uses a large amount of energy and converting it into heat.



Resistors

LED (Light-emitting diode)

LED is an abbreviation for LED (Light-emitting diode), a semiconductor device that converts electricity into light.

Light-emitting diode

Sensors

A device that converts real-world (analog) data into data that a computer can understand using ADC and converting data from Analog to Digital format. We will use sensors to detect events or environment changes and we will send it for you to read in our application.

Protective Conductor (Ground or Ground Wire)

It is the input wire of an electric conductor that has the function of "grounding" all the devices that need to use its potential as a reference or its electrical properties.

In power systems, the grount connector has the functions of electrical reference for voltage, protection systems, control for overload/power and equipment protection.

Jumper Wires

It is a short insulated wire with bare (stripped of insulation) ends. You use them to connect two points in a breadboard circuit.

Jumpers

Push Button

It is a simple switch mechanism for controlling some aspect of a machine or a process.

Push button

Next steps

Finishing now the contents of this book with some rather didactic examples we already see the power of Javascript allied to robotics in our applications. These are examples that can be incorporated into our daily lives and teachings to carry out various other ideas that will arise.

I hope you have enjoyed the contents of this book and the examples as much as I have had the pleasure of sharing this content. If you have questions, questions or even a conversation, please contact us at `willmendesneto@gmail.com`.

Thank you very much!