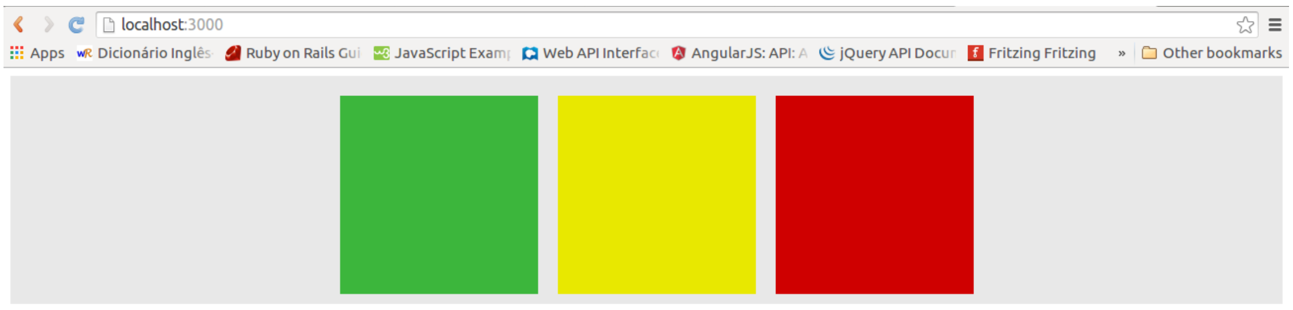
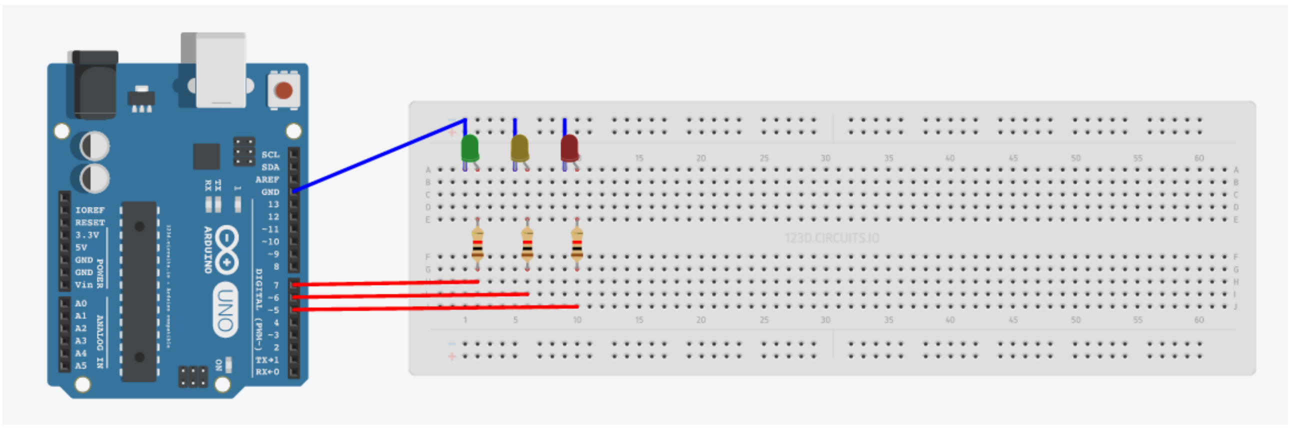
Railsduino

Learn how to build your own web app to interact with an arduino board

Throughout this tutorial it's shown how to build a simple web app which purpose it to serve as an interface to turn on and off three leds powered by an arduino board.





At this point you should already have the rails up an running so we will start by creating a rails new app by running the following command on the terminal. [1]

```
$ rails new app
```

Rails generates an hierarchy of files and folder which gives the basic suture of your rails app.

As in most of the web apps, your app should have a home page or index which needs a matching route. To generate the root route you should edit the `routes.rb` file which is located at the `/config` folder. [2]

```
root to: 'home#index'
```

With this code line you are telling rails that for a '/' url your app has a `home` controller with an `index` action. However this components are not automatically generated, thus, the next step is to generate the `home` controller with the `index` action. This can be made manually by adding an new file named `home_controller.rb` into the `/app/controller` folder or through the command line with the following command. [3]

```
$ rails generate controller Home index
```

The `rails generate` command not only generates the controller file but the related `index` view as well as other resources. If you choose to do it manually you need to create the `index.erb` file as well, placing it inside the `/app/views/home folder`.
Since rails renders the `index.erb` view automatically when entering the `index` action, it's no need to add any code to it, thus, the `home_controller.rb` should look like this:

```
class HomeController < ApplicationController
    def index
    end
end
```

Now, in order to have something displayed on the browser you have to add some html to the `index.erb` file. For the sake of simplicity lets just add some divs representing the three arduino leds – green, yellow and red.

As you can notice we add some classes to the divs. Those classes are used on the CSS and JS files.
In order to give shape and color to the divs you should edit the `application.css` file which is located inside the `/app/assets/stylesheets`.

To give behavior to your app you should edit the `application.js` which can be found inside the `/app/assets/javascripts.`
Some of the CSS classes were created as helpers of the JS file in order to change the divs' behavior as desired, namely to change the divs' color for a brighter shade when a given arduino led is on.

The main goal with the web app interface is: whenever each colored div is hovered the correspondent led should turn on. This is accomplished through jquery, which allow us to read the user interaction, and ajax requests to the serve, which allow us to send commands through serial port to the arduino board.

On the server side, for each ajax request it's needed an action on a controller. Since the serial communication is a block in its own it's advised to create a dedicated controller. To do so you just have to follow the same steps as on the `home` controller case described before, however, in this case, instead of having only one single action it's needed one action for each ajax requests which should be related to its own route as well.
Following this line of thought, the routes connected to the ajax request should be added to the `routes.rb` file.

```
$.ajax({
   type: "POST",
   url:"/green_on"
});
```

```
post '/green_on', to:
'serial#green_on'
```

Ruby has a gem called SerialPort which implements the serial communication. In order to use that gem you have to edit the

GemFile, adding the following code line: [4]

```
gem 'serialport'
```

Then you need to used `bundler` to install the gem, thus you should run `$ bundle install` on the command line.
As soon as the bundler finishes running you can then use SerialPort functions, which allows you to configure the serial port and read/write to it.

```
class SerialController < ApplicationController

  def green_on
    port = self.serial_port_config("/dev/tnt0")
    port.write "GH\n"
    port.close
  end

  private

  def serial_port_config(port)
    binding.pry
    parameters = {"baud" => 9600,
                  "data_bits" => 8,
                  "stop_bits" => 1,
                  "parity" => SerialPort::None}
    sp = SerialPort.new(port,parameters)
  end
end
```

On the other side of the communication, the serial port should be read by the arduino board. According to the message sent by the rails app, a given led should be turned on or off. [5]

```
void loop() {
    if(Serial.available()>0){
        switch(Serial.read()){

          case 'G':
              led_state(green);

           break;
        }

      }
}

void led_state(char led){

  if(Serial.read()=='L'){
    digitalWrite(led, LOW);
  }
  else{
    digitalWrite(led, HIGH);
  }

}
```

Referencies

[1] http://guides.rubyonrails.org/getting_started.html
[2] http://guides.rubyonrails.org/command_line.html
[3] http://guides.rubyonrails.org/routing.html
[4] http://www.rubydoc.info/gems/serialport/1.3.1
[5] https://123d.circuits.io/circuits/1027005-railsuino