Smart Bathroom App

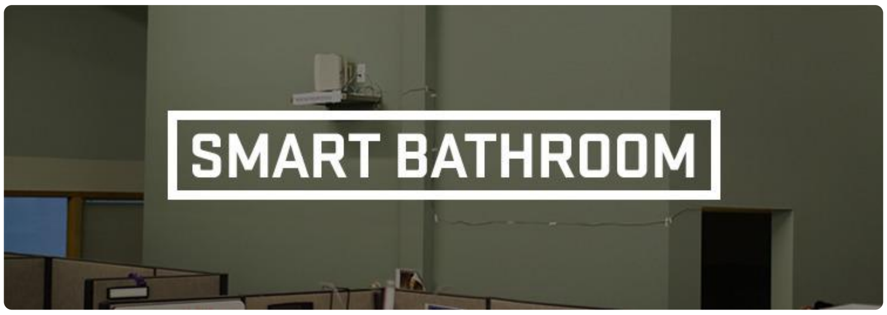
Created by Michael Szczepanski

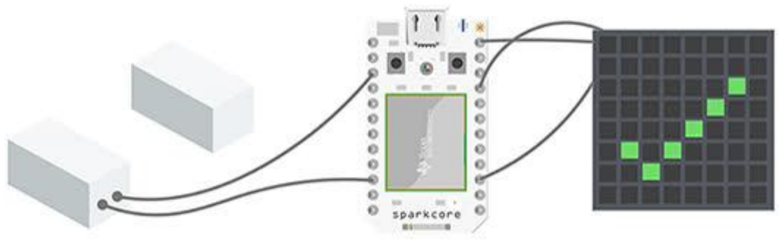
https://learn.adafruit.com/smart-bathroom-app

Last updated on 2023-08-29 02:42:50 PM EDT

Table of Contents

Overview





Looking to dip your toes into the IoT realm without hitting too many technical speed bumps? So were we, until we picked up some parts that were a little too advanced for our hobby. After some research into other solutions on Adafruit, The Spark Core stood out to pair with our other parts to get our sensors talking to the internet.

The best way to get started with any embedded project is to try to solve a small problem you have, no matter how minor, to keep you on track. Our problem was seeing people not knowing if a bathroom was in use or not and react with a quick experiment on giving people a heads up from their computers or seats.

This project will show you how to share sensor data through the Spark Core API, wire up your hardware, and dive you into JavaScript. The wiring and firmware are very simple and the web programming isn't too intense; we had this up and running within two weeks.

The Team

Michael Szczepanski

Interaction Designer ()
Michael is an interaction designer who enjoys designing user interfaces and digging into ways to illustrate them. As a designer with a background in development, Michael’s strength is bringing more iterative design into development.

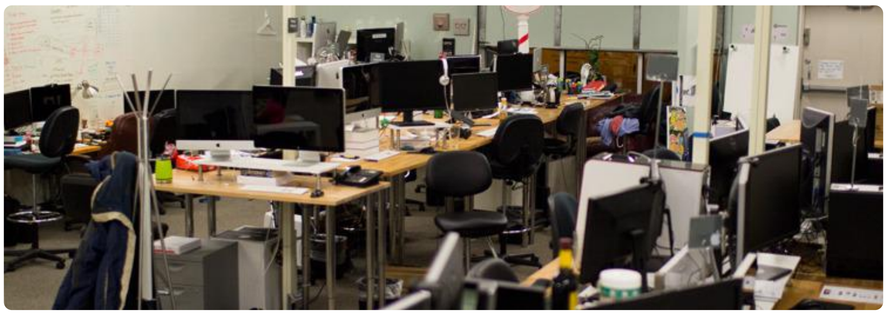
Chris Cheney

Front End Developer ()
Chris is a laid back web developer who’s passionate about well structured code, especially JavaScript, HTML, and CSS. As the executing arm of the team, Chris has developed mobile friendly markup and JavaScript that you can use on your own projects.

Chris and Michael began working together in 2007 at a web hosting company and quickly became friends when their skills aligned. Countless client projects later, they strive for creating their own tools outside of the nine to five to exert creativity

What we're building today is quite simple, but highly effective in making a change in your environment. With a few simple parts and some web development chops, you can get this up and running in a couple weeks.

The Problem

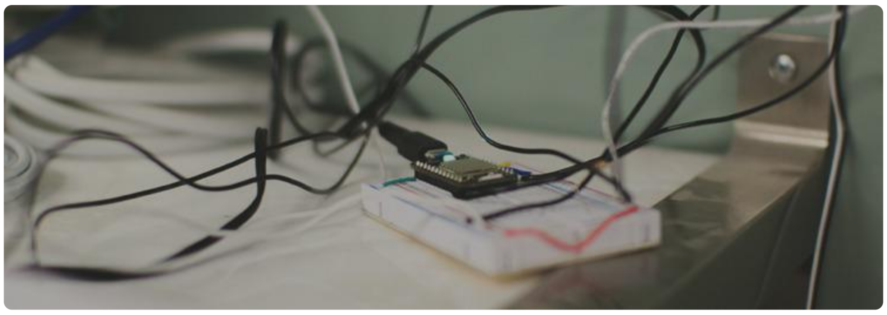


The Nerdery is a company of 500 web developers and user experience designers. With bathrooms being scattered throughout thousands of square feet, a lot of time is lost roaming through the large workplace to use a bathroom (and preferably a private one!). As a company that thrives off billable client work, both the client and the company lose when an employee has to take an extra long “lunch”.

We noticed people constantly walking up to the door, rarely getting a chance to use it due to the popularity. My friend Chris and I, who sat in view of this bathroom, cringe when seeing failed attempts of people hoping to relieve themselves. We somehow joked that an indicator could help our coworkers by cobbling together parts from recent Adafruit impulse buys. Then we found ourselves staying late to screw the reed sensor to a door frame.

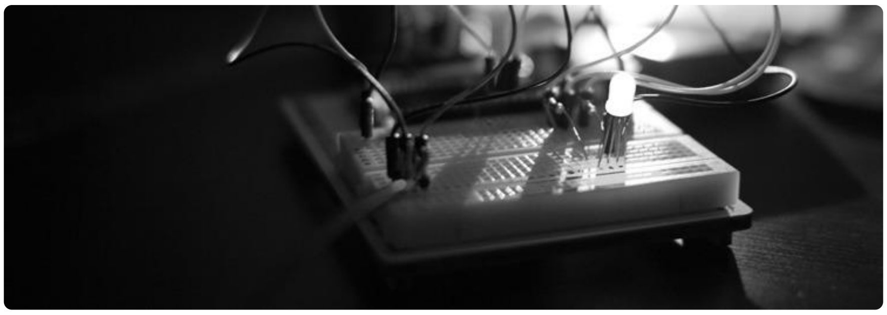
A shelf above a small cube farm looked to have some hope for getting a message across. We came to the conclusion that an airplane bathroom heads up could help save people time. First we started with an Arduino Uno to get the door to trigger the LEDs, but knew we wanted something a little more internet connected for a technology company like The Nerdery.

We looked at many different boxes that could get us on the internet fast; The Arduino Yun, Raspberry Pi, and others. None of which stood out to be as simple as the Spark Core to help provide API access from/to your sensors. We’ve still ran into some small problems, but overall it’ll help you get to spun up much quicker than configuring a web server manually.

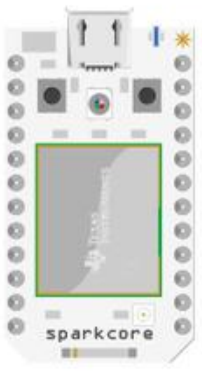


Within moments we had the Spark Core online and received values from a button using their iPhone app. After looking through the API for the web; we knew there was something to work with. Let's get this wired up and ready for the web.

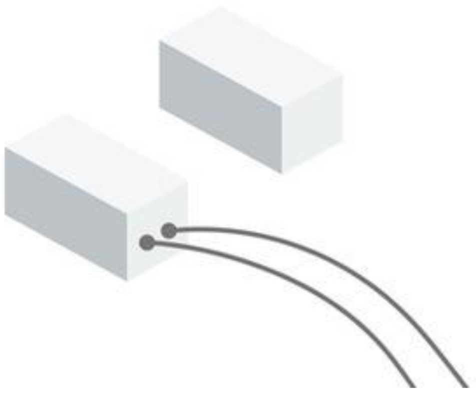
Hardware



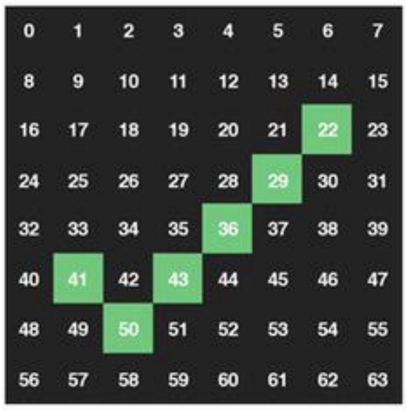
You'll only need the hardware on right side of this page.



The Spark Core team has great documentation on how to set up your Spark Core device to your wireless network. Throw in that USB cable that came with it and get the device up and running.

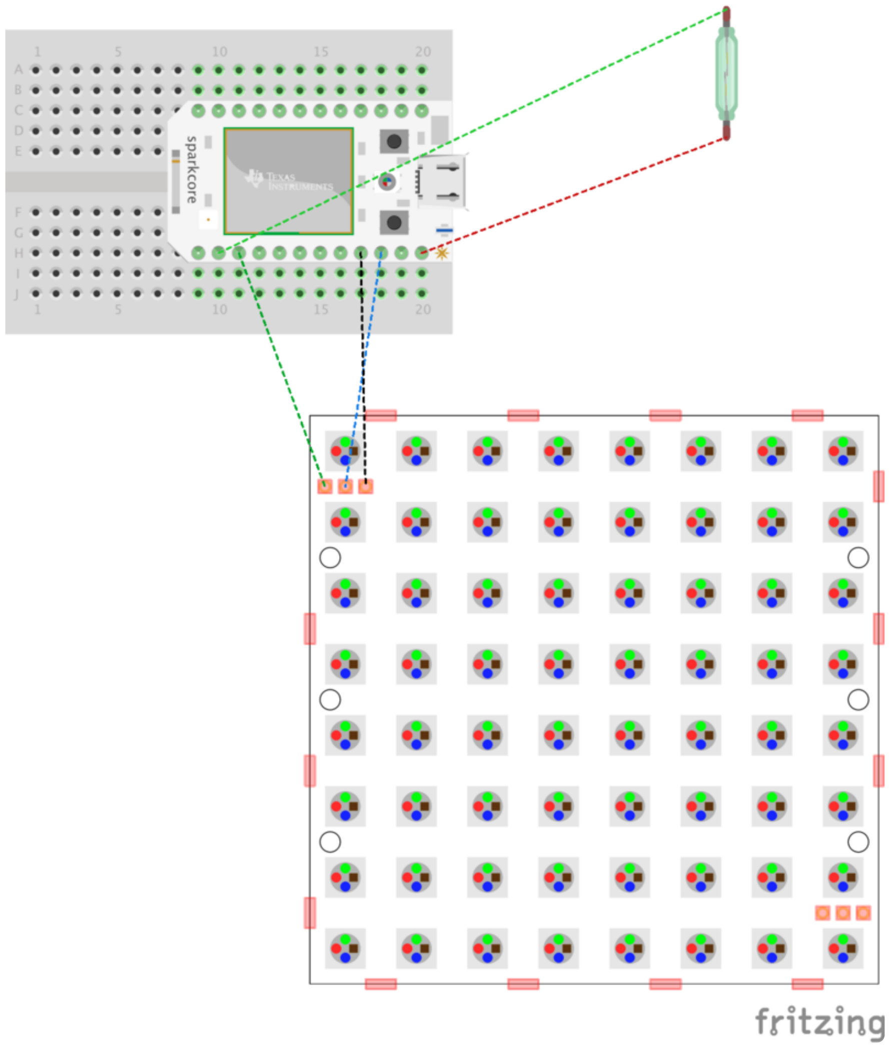


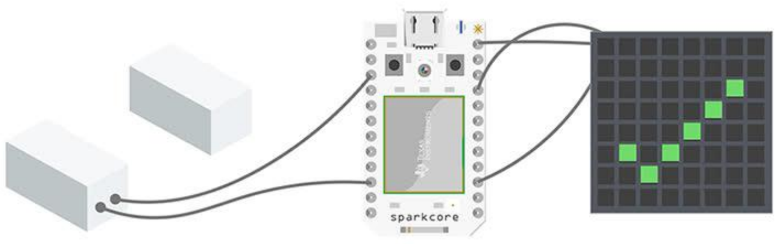
Hook up one end to 3.3V and the other to digital pin 1 on your Spark Core. The reed sensor works as a switch, one module contains the electrical components that separate and come together based on the other module which is a magnet. When the magnet is next to the electrical module, the switch gets flipped.



NeoPixels are very easy to wire; connect the VCC to 3.3V, GND to ground, and DIN to digital pin 2 on the Spark Core. As an optional step, cut out a square from a normal piece of paper and tape it over the front of the NeoPixel to help diffuse the light. If you're spendy, try using black vellum for a really solid black overlay.

Make the connections listed above and boot up the Spark Core. It'll require downloading their mobile app and pushing your sketches to the device through the web browser.


sparkcore
fritzing


sparkcore

Spark Core Software

Download the Spark sketch

The software is just as simple as the hooking up the hardware. We're going to configure the NeoPixels, set up some variables for the switch/Spark API, and run a loop to check the activity of the door.

Setting up the NeoPixels takes a little more effort in the Spark IDE than a Arduino. The library is included in the web IDE which is handy, but it requires a few variables to be defined before making the initial call to set up the NeoPixel. The code here will take care of telling it which pin we're hooked up to, how many pixels we're pushing, and the type of driver.

```
#include "neopixel/neopixel.h"

#define PIXEL_PIN D2
#define PIXEL_COUNT 64
#define PIXEL_TYPE WS2812B

Adafruit_NeoPixel strip = Adafruit_NeoPixel(PIXEL_COUNT, PIXEL_PIN, PIXEL_TYPE);
```

We'll also need to create a few basic variables along the way. Assigning A0 to "analogPin" can reference it without forgetting the number. The "sensorData" variable will hold the latest value from the sensor. "doorStatus" is where we'll access the state of the switch on the door.

```
#define sensorPin = D1;
int sensorData = 0;
int doorStatus = 0;
```

In the setup we'll start the NeoPixel and turn all the pixels to off. We'll use the Spark API calls to spit out a variable called "doorStatus", which will be 0 or 1, and share it with the web.

```
void setup() {
  strip.begin();
  strip.show(); // Initialize all pixels to 'off'
  Spark.variable(“doorStatus", &amp;doorStatus, INT);
}
```

The loop will read the sensor (reed switch) to see if the value is available (zero) or in use (1). When the status of the door changes to open, update "doorStatus" to one and share it with the Spark Core API using the Spark.variable from above, and display a red X on our NeoPixel. If that status changes, we'll indicate a green check mark for open.

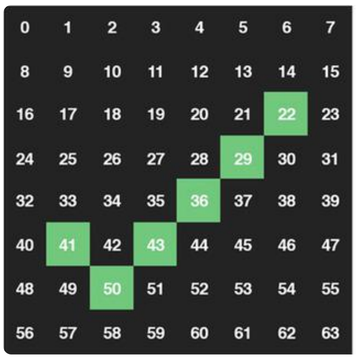
```
void loop() {

  sensorData = digitalRead(sensorPin);
  if (sensorData = 1)
  {
    doorStatus = 1;
    doorClosed();
  }
  else
  {
    doorStatus = 0;
    doorOpen();
  }
}
```

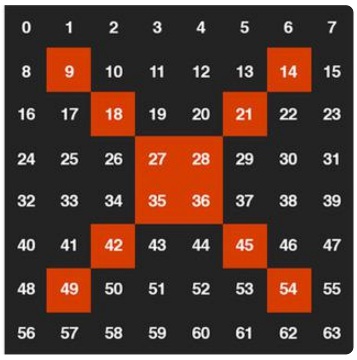
NeoPixels need to be controlled manually for a situation like this; each shape needs to be entered manually. Use the stip.setPixelColor() function to pass the number in the array (0-63) and the RGB value.

```
strip.setPixelColor(22, 27, 167, 46);
```

To make things easier in counting these, I created a quick grid in PhotoShop with numbered square guides. It’s easy canvas to create/dissect coordinates without losing count or track. Below is a template that will help you in building your own shapes in the future.

NeoPixel Guide PSD

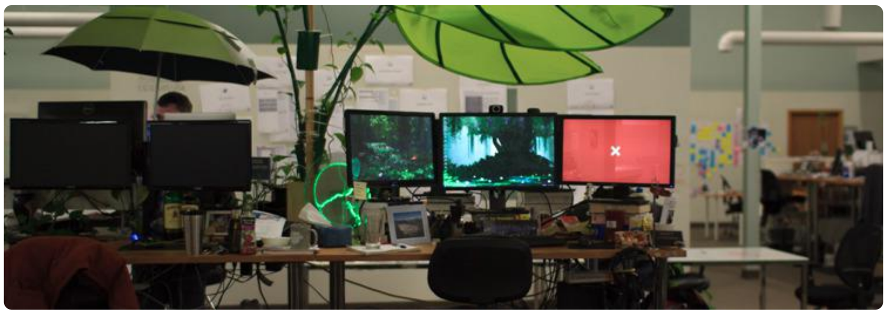




We’ll list out all the coordinates for each shape, then put them into functions that group them together. The brightness of that shape can be defined here using the strip.setBrightness() function too. (255 may induce x-ray vision) strip.show() displays what we’ve decided on and displays it.

```
void doorOpen() {
    strip.setPixelColor(0, 0, 0, 0);
    strip.setPixelColor(1, 0, 0, 0);
    strip.setPixelColor(2, 0, 0, 0);
    strip.setPixelColor(3, 0, 0, 0);
    strip.setPixelColor(4, 0, 0, 0);
    strip.setPixelColor(5, 0, 0, 0);
    strip.setPixelColor(6, 0, 0, 0);
    strip.setPixelColor(7, 0, 0, 0);
    strip.setPixelColor(8, 0, 0, 0);
    strip.setPixelColor(9, 0, 0, 0);
    strip.setPixelColor(10, 0, 0, 0);
    strip.setPixelColor(11, 0, 0, 0);
    strip.setPixelColor(12, 0, 0, 0);
    strip.setPixelColor(13, 0, 0, 0);
    strip.setPixelColor(14, 0, 0, 0);
    strip.setPixelColor(15, 0, 0, 0);
    strip.setPixelColor(16, 0, 0, 0);
    strip.setPixelColor(17, 0, 0, 0);
    strip.setPixelColor(18, 0, 0, 0);
    strip.setPixelColor(19, 0, 0, 0);
    strip.setPixelColor(20, 0, 0, 0);
    strip.setPixelColor(21, 0, 0, 0);
    strip.setPixelColor(22, 27, 167, 46);
    strip.setPixelColor(23, 0, 0, 0);
    strip.setPixelColor(24, 0, 0, 0);
    strip.setPixelColor(25, 0, 0, 0);
    strip.setPixelColor(26, 0, 0, 0);
    strip.setPixelColor(27, 0, 0, 0);
    strip.setPixelColor(28, 0, 0, 0);
    strip.setPixelColor(29, 27, 167, 46);
    strip.setPixelColor(30, 0, 0, 0);
    strip.setPixelColor(31, 0, 0, 0);
    strip.setPixelColor(32, 0, 0, 0);
    strip.setPixelColor(33, 0, 0, 0);
    strip.setPixelColor(34, 0, 0, 0);
    strip.setPixelColor(35, 0, 0, 0);
    strip.setPixelColor(36, 27, 167, 46);
    strip.setPixelColor(37, 0, 0, 0);
    strip.setPixelColor(38, 0, 0, 0);
    strip.setPixelColor(39, 0, 0, 0);
    strip.setPixelColor(40, 0, 0, 0);
    strip.setPixelColor(41, 27, 167, 46);
    strip.setPixelColor(42, 0, 0, 0);
    strip.setPixelColor(43, 27, 167, 46);
    strip.setPixelColor(44, 0, 0, 0);
    strip.setPixelColor(45, 0, 0, 0);
    strip.setPixelColor(46, 0, 0, 0);
    strip.setPixelColor(47, 0, 0, 0);
    strip.setPixelColor(48, 0, 0, 0);
    strip.setPixelColor(49, 0, 0, 0);
    strip.setPixelColor(50, 27, 167, 46);
    strip.setPixelColor(51, 0, 0, 0);
    strip.setPixelColor(52, 0, 0, 0);
    strip.setPixelColor(53, 0, 0, 0);
    strip.setPixelColor(54, 0, 0, 0);
    strip.setPixelColor(55, 0, 0, 0);
    strip.setPixelColor(56, 0, 0, 0);
    strip.setPixelColor(57, 0, 0, 0);
    strip.setPixelColor(58, 0, 0, 0);
    strip.setPixelColor(59, 0, 0, 0);
    strip.setPixelColor(60, 0, 0, 0);
    strip.setPixelColor(61, 0, 0, 0);
    strip.setPixelColor(62, 0, 0, 0);
    strip.setPixelColor(63, 0, 0, 0);
    strip.setBrightness(30);
    strip.show();
}

void doorClosed() {
    strip.setPixelColor(0, 0, 0, 0);
    strip.setPixelColor(1, 0, 0, 0);
    strip.setPixelColor(2, 0, 0, 0);
    strip.setPixelColor(3, 0, 0, 0);
    strip.setPixelColor(4, 0, 0, 0);
    strip.setPixelColor(5, 0, 0, 0);
    strip.setPixelColor(6, 0, 0, 0);
    strip.setPixelColor(7, 0, 0, 0);
    strip.setPixelColor(8, 0, 0, 0);
    strip.setPixelColor(9, 185, 30, 0);
    strip.setPixelColor(10, 0, 0, 0);
    strip.setPixelColor(11, 0, 0, 0);
    strip.setPixelColor(12, 0, 0, 0);
    strip.setPixelColor(13, 0, 0, 0);
    strip.setPixelColor(14, 185, 30, 0);
    strip.setPixelColor(15, 0, 0, 0);
    strip.setPixelColor(16, 0, 0, 0);
    strip.setPixelColor(17, 0, 0, 0);
    strip.setPixelColor(18, 185, 30, 0);
    strip.setPixelColor(19, 0, 0, 0);
    strip.setPixelColor(20, 0, 0, 0);
    strip.setPixelColor(21, 185, 30, 0);
    strip.setPixelColor(22, 0, 0, 0);
    strip.setPixelColor(23, 0, 0, 0);
    strip.setPixelColor(24, 0, 0, 0);
    strip.setPixelColor(25, 0, 0, 0);
    strip.setPixelColor(26, 0, 0, 0);
    strip.setPixelColor(27, 185, 30, 0);
    strip.setPixelColor(28, 185, 30, 0);
    strip.setPixelColor(29, 0, 0, 0);
    strip.setPixelColor(30, 0, 0, 0);
    strip.setPixelColor(31, 0, 0, 0);
    strip.setPixelColor(32, 0, 0, 0);
    strip.setPixelColor(33, 0, 0, 0);
    strip.setPixelColor(34, 0, 0, 0);
    strip.setPixelColor(35, 185, 30, 0);
    strip.setPixelColor(36, 185, 30, 0);
    strip.setPixelColor(37, 0, 0, 0);
    strip.setPixelColor(38, 0, 0, 0);
    strip.setPixelColor(39, 0, 0, 0);
    strip.setPixelColor(40, 0, 0, 0);
    strip.setPixelColor(41, 0, 0, 0);
    strip.setPixelColor(42, 185, 30, 0);
    strip.setPixelColor(43, 0, 0, 0);
    strip.setPixelColor(44, 0, 0, 0);
    strip.setPixelColor(45, 185, 30, 0);
    strip.setPixelColor(46, 0, 0, 0);
    strip.setPixelColor(47, 0, 0, 0);
    strip.setPixelColor(48, 0, 0, 0);
    strip.setPixelColor(49, 185, 30, 0);
    strip.setPixelColor(50, 0, 0, 0);
    strip.setPixelColor(51, 0, 0, 0);
    strip.setPixelColor(52, 0, 0, 0);
    strip.setPixelColor(53, 0, 0, 0);
    strip.setPixelColor(54, 185, 30, 0);
    strip.setPixelColor(55, 0, 0, 0);
    strip.setPixelColor(56, 0, 0, 0);
    strip.setPixelColor(57, 0, 0, 0);
    strip.setPixelColor(58, 0, 0, 0);
    strip.setPixelColor(59, 0, 0, 0);
    strip.setPixelColor(60, 0, 0, 0);
    strip.setPixelColor(61, 0, 0, 0);
    strip.setPixelColor(62, 0, 0, 0);
    strip.setPixelColor(63, 0, 0, 0);
    strip.setBrightness(30);
    strip.show();
}
```

Responsive Web App



Github Repo

Having a sensor that could input the status of the door and output a sign outside of it is one thing, but what about being able to check the status anywhere? Chris and I hit the pavement with setting up a simple responsive site using JavaScript, Node.JS, socket.io, and Heroku.

We chose to roll with Node.JS () to handle the Spark Core API requests on the back-end and Socket.io () to stream the results of the request to the front-end web browser client.

We're using Express () to serve the static client side assets to the browser. Socket.io binds to the node http module for broadcasting events.

Here's our server.js file. This file invokes http and Express. We're simply starting a static web server with Express to serve the client side JS/CSS/HTML. The node http module consumes the Express instance and we then export a reference to http for use in our app.

```
/*jslint node: true */

'use strict';

var http = require('http');
var express = require('express');
var app = module.exports.app = express();

app.use(express.static(__dirname));

var server = http.createServer(app);

module.exports = server;
```

Our app.js file is where the magic happens. Let's briefly explain each portion.

Remember server.js? It doesn't run by itself, instead, it's required and invoked in our app here and attached to the Socket.io node module. The app.js file is what node fires off to create our application.

```
/*jslint node: true */

'use strict';

var server = require('./server');
var request = require('superagent');

var io = require('socket.io').listen(server);

var port = process.env.PORT || 9002;

server.listen(port);
```

Now that our global references are all set, our Express server is serving client assets, and Socket.io is listening for client side stream requests we can finally start grabbing information from our Spark Core and sending events to the client. We've constructed an object called App which starts off by firing it's initialization method where we assign values to our variables for later use, and finally kicking off our startLoop method.

```
var App = function () {
    return this.init();
};

App.prototype.init = function () {
    this.bathStatus = 'spark core api endpoint url';

    return this.startLoop();
};
```

Here lies The infamous setInterval function which we're using to abuse the Spark Core API every two seconds.

```
App.prototype.startLoop = function () {
    setInterval(this.getResult.bind(this), 2000);
    return this;
};
```

Our getResult method fires a request using Superagent (). The startLoop() method is running this every two seconds. It makes an ajax request to the Spark Core API and fires off Socket.io events based on the results. If the result is 0 we emit a "Open" event and a "Close" event if the result is 1. We're listening for these events on the client.

```
App.prototype.getResult = function () {
    request
        .get(this.bathStatus)
        .on('error', function (err) {
            io.emit('error', err);
        }.bind(this))
        .end(function (res) {
            if(res.body.result === 0) {
                io.emit('open');

                return this;
            }

            if(res.body.result === 1) {
                io.emit('close');

                return this;
            }

            io.emit('error', res);
        }.bind(this));

    return this;
};
```

That's our NodeJS application.

We tried to keep it as simple as possible while also allowing us to extend it later on with other features such as recording events into a database.

Let's breifly cover the client side JavaScript used to listen for the events emitting from our NodeJS application. First we pull in the client side Socket.io module which will bind itself to the events emitting from the http server on the NodeJS side.

```
var socket = io();
```

The familiar constructor pattern is employed here in our client side code too. We construct the App and fire off it's initialization method which fires off our setupHandlers, createChildren, and enable methods.

```
var App = function () {
    return this.init();
};
```

```
App.prototype.init = function () {
    return this
        .setupHandlers()
        .createChildren()
        .enable();
};
```

In setupHandlers we are binding our event handlers to (this) to avoid scoping problems while we're working inside of the socket() scope. The createChildren method is caching our DOMElement selectors for use in our App later. Finally, now that everything is in place we can kickoff the Socket.io connection to the http server and start listening for the events. Each event has it's own method so we can control what happens when we hear a specific event.

```
App.prototype.setupHandlers = function () {
    this.onOpenMessageHandler = this.onOpenMessage.bind(this);
    this.onCloseMessageHandler = this.onCloseMessage.bind(this);
    this.onErrorMessageHandler = this.onErrorMessage.bind(this);

    return this;
};

App.prototype.createChildren = function () {
    this.box = document.querySelector('.box');
    this.icon = document.querySelector('.icon');

    return this;
};

App.prototype.enable = function () {
    socket.connect();

    socket.on('open', this.onOpenMessageHandler);
    socket.on('close', this.onCloseMessageHandler);
    socket.on('error', this.onErrorMessageHandler);

    return this;
};
```

When a subscribed event occurs one of the following methods is invoked. Everything that we've covered so far comes down to these three methods. Our onOpenMessage and onCloseMessage uses the cached selectors from createChildren to add and remove CSS classes in the DOM. Our onErrorMessage handles any errors from our NodeJS application and logs the data to the console.

```
App.prototype.onOpenMessage = function () {
    this.box.classList.remove('box_isActive');
    this.icon.classList.remove('icon_isActive');

    this.box.classList.add('box_isInActive');
    this.icon.classList.add('icon_isInActive');

    return this;
};

App.prototype.onCloseMessage = function () {
    this.box.classList.remove('box_isInActive');
    this.icon.classList.remove('icon_isInActive');
    this.box.classList.add('box_isActive');
    this.icon.classList.add('icon_isActive');

    return this;
};

App.prototype.onErrorMessage = function (result) {
    console.error('Ermah G. Herde: ', result);

    return this;
};
```

There you have it!

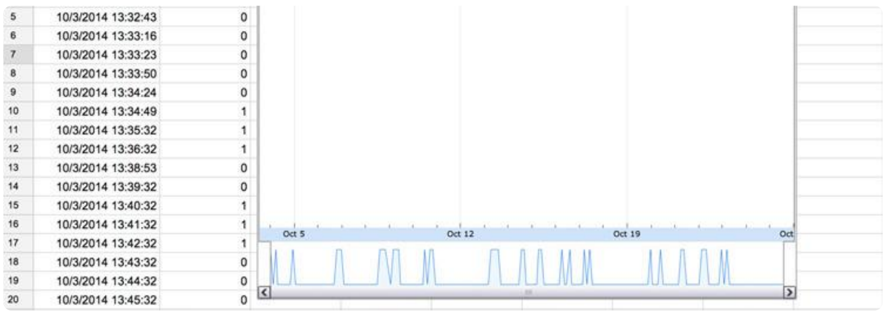
All of the code that we've covered for NodeJS and the client side JS boils down to just changing a CSS class in the DOM. Nothing more, nothing less. All of this code and more is open sourced on our Github repo, you should give it a pull:

Github Repo

Our Results & Ideas

Logging The Data

We recommend this article () which explains how you can log your data periodically (every minute) to a Google Spreadsheet for analysis. This will give you a never ending spreadsheet with patterns of usage history, useful in our case to make some visualization of frequency. One could easily replace this with a real database and output the data on the web with a small amount of effort.



5	10/3/2014 13:32:43	0
6	10/3/2014 13:33:16	0
7	10/3/2014 13:33:23	0
8	10/3/2014 13:33:50	0
9	10/3/2014 13:34:24	0
10	10/3/2014 13:34:49	1
11	10/3/2014 13:35:32	1
12	10/3/2014 13:36:32	1
13	10/3/2014 13:38:53	0
14	10/3/2014 13:39:32	0
15	10/3/2014 13:40:32	1
16	10/3/2014 13:41:32	1
17	10/3/2014 13:42:32	1
18	10/3/2014 13:43:32	0
19	10/3/2014 13:44:32	0
20	10/3/2014 13:45:32	0

What you can do with this type of data is get a minute by minute log of the activity of your sensor from the internet. What we found from a month of door activity is interesting; the hours spent per day/week/month in use, the times of day it's available, and readings that almost indicates someone's living in the building. We can't tell if it's reducing or creating more time in the bathroom; yet.

Our Findings

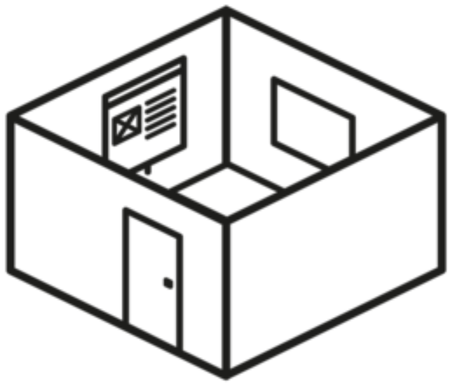
With over 100 web designers and developers sitting near this bathroom, we saw an immense amount of traffic around lunch time. Activity at 3AM and false positives (someone closes the door behind them) were fun anamolies to see as well. The most impactful part of this information at a company that tracks billable hours is asking the question:

> How many hours per day, on average, are spent in the private bathroom?

Surely a company can't stop employees from using the bathroom to increase efficiency. However they can help increase visibility to what's available when you need to take a seat. Five hours a day on average were spent with this bathroom door closed; one of the many bathrooms in the area.

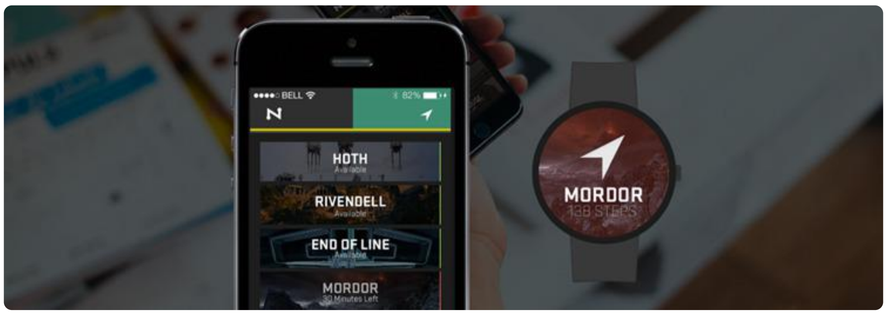
> That's a lot of time flushed down the toilet

Is a natural response by managers, but what isn't being considered is the time spent leading up to occupancy. Every day we'd watch dozens of people get rejected by a locked door, but since this device helps be more effective. We don't have enough data at this time to determine if the bathroom indicator is creating more or less time spent using it.



Sure, this gag wears thin with people don't like to talk about bathrooms or the notion of people pooping. If we shift the conversation to a different type of room that does have critical impact on a business; we can talk about people squatting in conference rooms. Conference rooms are a vital resource, we have plenty of tools to book time in them, but we still fail to find a place to meet.

How many conference rooms does your company have? The Nerdery has 37 in one office and they're always booked at certain times in the day. Many times those rooms are booked for recurring meetings most people don't attend anymore. By using a different type of sensor, we could apply our same concept from the bathroom. Determining whether someone is in the room or not is all we need to know, not who it is or when they're done. Simply the status.



Our next project will be solving the problem of conference rooms in businesses in the same way. We hope that this inspires you to solve a small problem of your own using a internet of things style device.