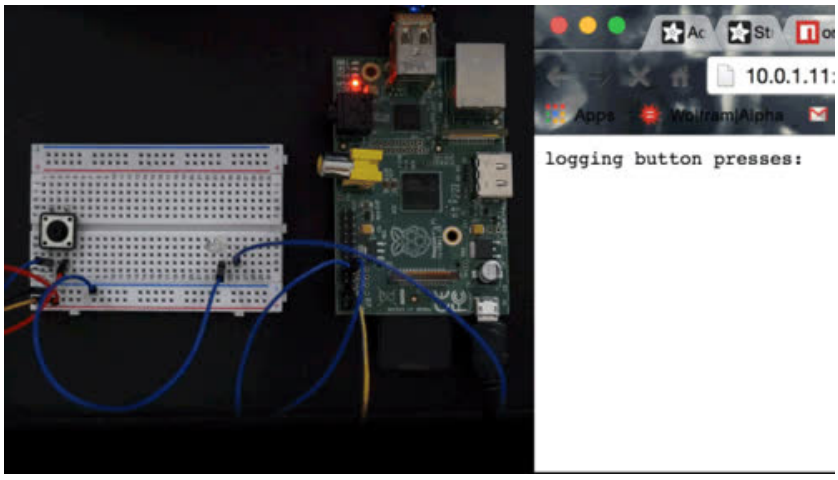
Node.js Embedded Development on the Raspberry Pi

Created by Todd Treece



Last updated on 2017-03-09 10:12:03 PM UTC

Guide Contents

Guide Contents 2
Why node.js? 3
- npm 3
- Events 4
- Streams 4

Connecting via SSH 5
- Hardware Connections 5
- Finding Your Raspberry Pi's IP Address 5
- Connecting via SSH 7

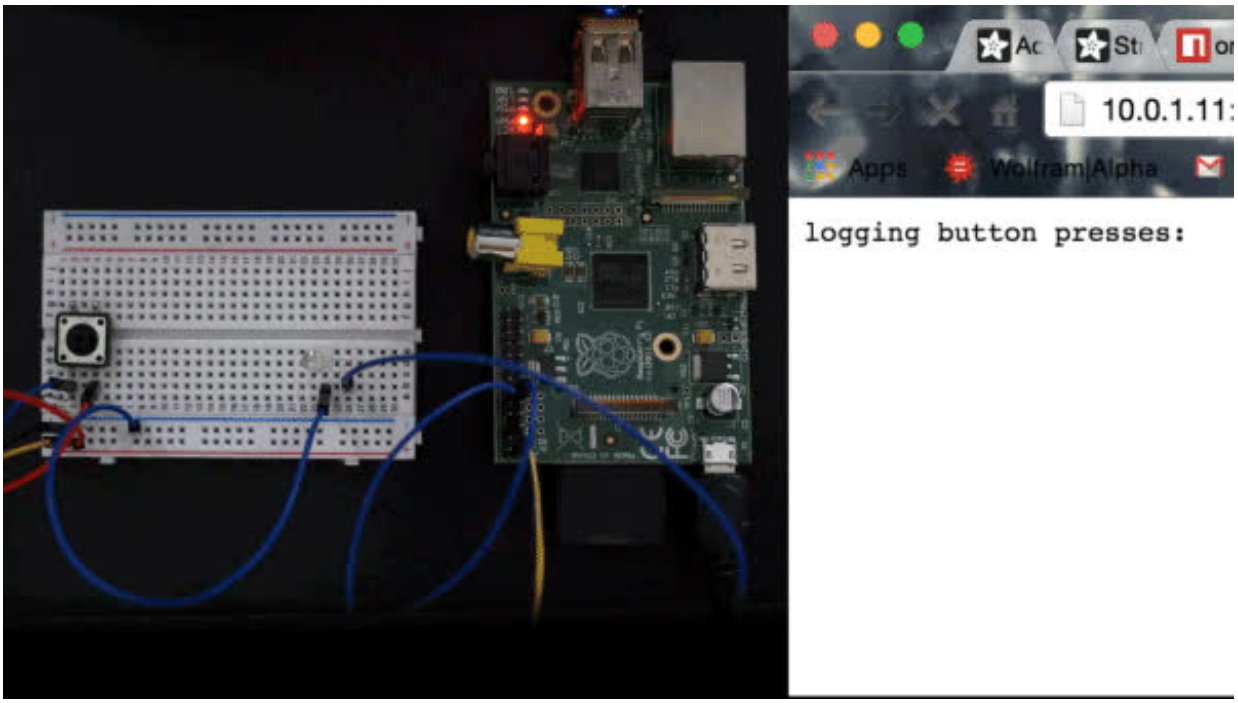
Installing node.js 10
- Adding the Package Repository 10

Events & Callbacks 11
- Callbacks 11
- Arduino & node.js Examples 13

Streams 17
- Unix Pipeline 17
- Node.js Streams 18
- Chunked Transfer Stream 20

Wrapping Things Up 22
- package.json 22
- Dependencies 23
- Example Package 23
- Publishing Packages 24

Why node.js?



Why is it worth exploring node.js (http://adafru.it/dIP) development in an embedded environment? JavaScript is a widely known language that was designed to deal with user interaction in a browser. Thanks to node.js, we can now use JavaScript outside of the browser, and even interact with hardware on devices such as the Raspberry Pi.

Here's a brief overview of some of the features that make node.js great for embedded development.

npm

npm (http://adafru.it/eht) is one of the best things to happen to node.js. npm is the package manager for node. It makes managing third-party node dependencies a breeze, and makes it easy for mere mortals like myself to publish node.js packages.

Why is this important? Community. Because npm is so great at managing dependencies, you can include open source libraries in your project hassle-free. Since it's easy to include community libraries, the community has grown exponentially, and the number of available packages is mesmerizing. Have you ever tried to install an Arduino library? No offense to Arduino, but it's not the most user friendly experience. Thankfully, they are working on adding a package manager (http://adafru.it/ehu) to the Arduino IDE.

Events

Events are a great way to deal with user interaction. What's an event? Maybe it would be easiest to give you a few examples of events. Toggling a switch, clicking a mouse, and pressing a key on your keyboard are all examples of events.

The great thing about using event listeners is that you can write a chunk of code that will be called whenever an event happens. No longer do you have to constantly check the state of a button to see if the state has changed in a loop. Instead, the button will let you know when it was pressed!

Streams

The node.js stream API combines the power of events, with the power of theUnix pipeline (http://adafru.it/ehv). If you know how to pipe commands in Unix environments, you know it's a game changer. We'll dig deeper into streams after taking a look at how to setup the node.js environment on a Raspberry Pi.

Connecting via SSH

The Raspberry Pi was designed to make it very easy to get started by connecting a monitor, keyboard, and mouse. But, what if you don't have access to the necessary peripherals or don't want to disconnect them from your main workstation? The answer? Secure Shell (http://adafru.it/ehw) (SSH). SSH is a network protocol that can be used for secure remote command-line login, and thankfully it's enabled by default if you are using the Raspbian Distro (http://adafru.it/d6i).

Hardware Connections

To connect via SSH, you only need connect your Raspberry Pi to power & to your network via an ethernet cable. For this guide, I have loaded a fresh install of the *2014-09-09* release of Raspbian on my SD card, but if you have an older release of Raspbian already installed, it should work as well.

Finding Your Raspberry Pi's IP Address

You can plug in an Ethernet cable and the Pi will automagically DHCP and connect to the internet. Then you have to actually connect to the Pi. To find the IP address of

your Raspberry Pi, there's a lot of options.

- If you have an HDMI monitor, connect it to the Pi, on boot it will print out it's IP address right before the console login: prompt
- If you have a console cable, you can use our tutorial to watch the Pi boot and print out the IP address (http://adafru.it/kgF)

If you have neither, you can find the device using your router's control panel, or you can use nmap (http://adafru.it/ehy) to search for devices that are accepting connections on TCP port 22, which is the port SSH uses. I am not going to assume that you have access to your router's control panel, so I will show you how to use *nmap* to locate your Raspberry Pi's IP address.

If you are running Windows, head over to nmap.org for theirWindows installation instructions (http://adafru.it/ehz). If you are running Mac OS X, you can install using the executable installer (http://adafru.it/ehA), or by using a package manager like homebrew (http://adafru.it/df3).

```
brew install nmap
```

If you are running a Linux distribution on your workstation, use your favorite package manager to install nmap, or visit the nmap.org downloads page (http://adafru.it/ehB) for instructions on installing from source.

```
sudo apt-get install nmap
```

Once you have nmap installed, you are now ready to search for the IP address. Using the terminal on Linux & Mac OS, or the command prompt on Windows, enter the following command.

```
nmap -p 22 --open -sV 10.0.1.*
```

If your local network is setup to use **192.168.1.*** addresses, modify the command to match your network setup.

```
nmap -p 22 --open -sV 192.168.1.*
```

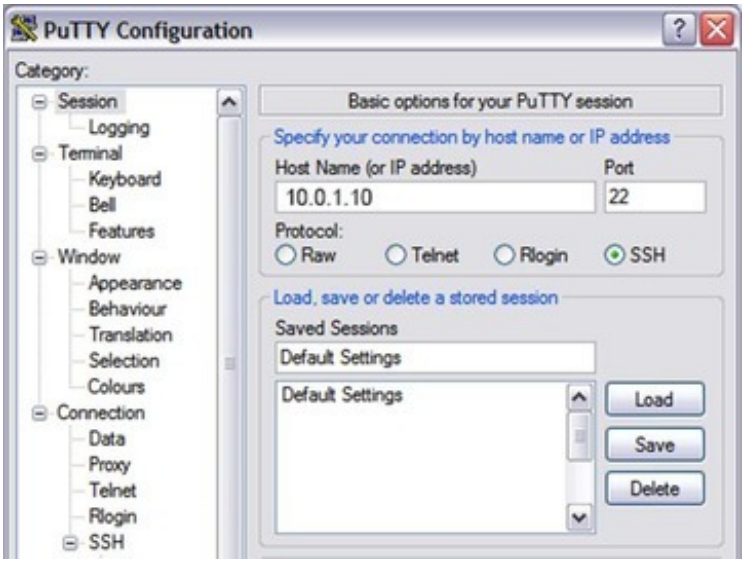
The output of nmap might seem intimidating, but here are a couple hints that should make it easier to identify your Raspberry Pi. Look for a line that contains something like "***OpenSSH 6.0p1 Debian***". Once you find that, navigate your way up a few lines to a line that starts with "***Nmap scan report for...***". The IP address that follows that statement should be the IP of your Raspberry Pi! In the example below, you can see that the IP address I'm looking for is **10.0.1.10**.

```
$ nmap -p 22 --open -sV 10.0.1.*
```

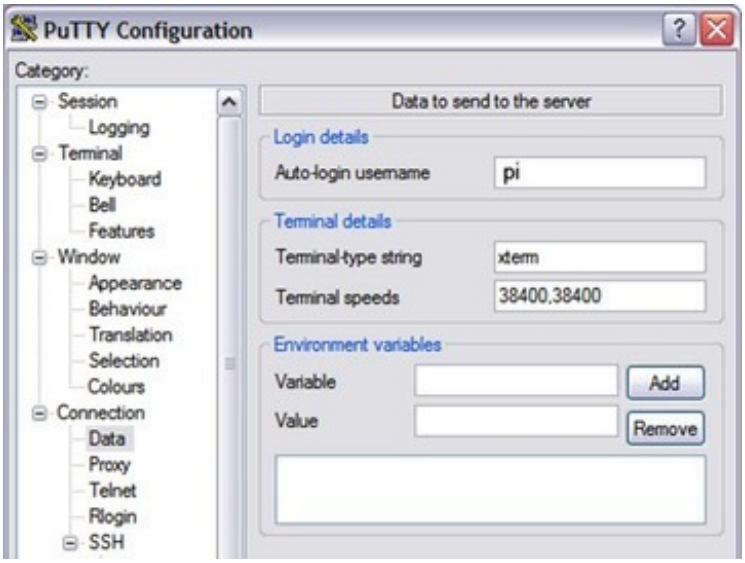
```
Starting Nmap 6.47 ( http://nmap.org ) at 2014-12-17 11:47 EST
Nmap scan report for 10.0.1.10
Host is up (0.0056s latency).
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 6.0p1 Debian 4+deb7u2 (protocol 2.0)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 255 IP addresses (3 hosts up) scanned in 3.39 seconds
```

Connecting via SSH

If you are using Windows, you will need to install a SSH client like PuTTY (http://adafru.it/aUb) to connect to your Raspberry Pi, but if you use Linux or Mac OS X, you are ready to connect. The default user for logging into a fresh Raspbian install is **pi**, and the default password is **raspberry**. If you are using PuTTY on Windows, enter the connection details into the session configuration window.

- 

Replace the **10.0.1.10** address with the IP address of your Raspberry Pi.

- 

Under the *Connection -> Data* menu, enter **pi** into the *Auto-login username* field, and click the *Open* button at the bottom of the window.

If you are using Linux or Mac OS X, enter the following command into your terminal.

```
ssh pi@10.0.1.10
```

You will then be prompted to verify that you want to connect to the host. Type **yes** to confirm, and hit the *Enter/Return* key on your keyboard. You should only be prompted to confirm that you want to connect the first time you try connecting.

```
$ ssh pi@10.0.1.10
The authenticity of host '10.0.1.10 (10.0.1.10)' can't be established.
RSA key fingerprint is e8:e0:f0:09:7e:a4:81:42:44:30:65:f0:4f:b4:a6:a5.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.1.10' (RSA) to the list of known hosts.
```

After that, you will be prompted for the password for the **pi** user. Enter the default password of **raspberry** and hit the Enter/*Return* key. If you have entered everything correctly, you should be logged into your Raspberry Pi via SSH!

```
pi@10.0.1.10's password:
Linux raspberrypi 3.12.28+ #709 PREEMPT Mon Sep 8 15:28:00 BST 2014 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Dec 17 14:01:23 2014 from 10.0.1.7
pi@raspberrypi ~ $
```

Once you get this all working and you can connect to the Pi, its a good idea to install

Zeroconf/Bonjour (http://adafru.it/lPE)so you can just **ssh to raspberrypi.local**

Installing node.js

Now that you are connected to your Pi via SSH, you can move on to installing the latest stable version of node.js. Thanks to the node-arm project (http://adafru.it/ehD) and Adafruit's Raspberry Pi Package Repository (http://adafru.it/obo), installing the latest version of node.js on a Raspberry Pi is very simple.

Adding the Package Repository

The first step will be to add the apt.adafruit.com package repository to your Pi's /etc/apt/sources.list file. You can do this by running the following command.

```
curl -sLS https://apt.adafruit.com/add | sudo bash
```

Next, we will install the latest version of node.js using*apt-get*.

```
sudo apt-get install node
```

If everything went as expected, we can check the installed version of node.js by running ***node -v***. At the time of this writing, the latest stable version of node.js was*v0.12.0.*

```
pi@raspberrypi ~ $ node -v
v0.12.0
```

Now that we have node.js installed, we can look at the strengths of node.js development on a Raspberry Pi.

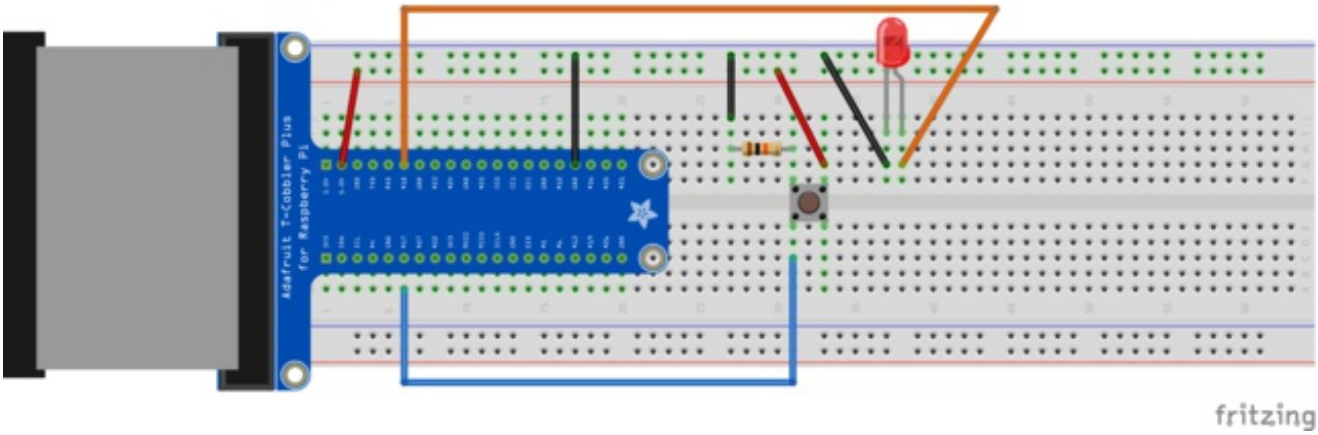
Events & Callbacks

If you are used to programming hardware in an Arduino environment, this section might seem a bit foreign to you. Unless you are using interrupts, code execution happens in logical order. When dealing with events & callbacks, specific chunks of code gets called as events happen. If you are used to developing with Javascript, then you should feel right at home.

Callbacks

We've already discussed events earlier in the guide, but what's a callback? A callback is a function that is passed as a parameter to another function so it can be used at any point in the future.

What on earth does this have to do with events & event listeners? When dealing with asynchronous events, your callback function defines the actions that happen in response to events. If you want a LED to light up when a button press happens, then you would write the code that turns on the LED inside of a callback. Let's take a look at an example using the Adafruit T-Cobbler to connect the button and LED.



We will show you how to download and run the example code in the last section of the guide, but if you would like to follow along and create the example files yourself, you will need to install the onoff (http://adafru.it/elV) npm package.

```
npm install onoff
```

Next, you can use **nano** text editor to create the test file. If you need help getting started with nano, check out this handy guide (http://adafru.it/obp).

```
nano test.js
```

Next, paste the following example into nano, and save the file.

```
// button is attached to pin 17, led to 18
var GPIO = require('onoff').Gpio,
    led = new GPIO(18, 'out'),
    button = new GPIO(17, 'in', 'both');

// define the callback function
function light(err, state) {

  // check the state of the button
  // 1 == pressed, 0 == not pressed
  if(state == 1) {
    // turn LED on
    led.writeSync(1);
  } else {
    // turn LED off
    led.writeSync(0);
  }

}

// pass the callback function to the
// as the first argument to watch()
button.watch(light);
```

From there, you can run the example file using the **node** command. Go ahead run the test file and press the button to make sure that your LED lights up.

```
node test.js
```

Let's break down what is happening in the example code. **button.watch(***callback***)** is a function that you can call if you would like to watch a button for changes in state. When a change event happens, the **watch()** function calls the *callback* function that was passed to it as its only parameter. In this case, the watch function will call the **light()** function when the button changes state.

When **watch()** notices that the button state changed, the **light()** function is called with two parameters. The first parameter has any *errors* that may have occurred, and the second parameter gives the current button *state*. When I wrote the function definition for the light method, I called the first parameter **err**, and the second parameter **state**, but you could name them whatever you would like. It's best to name them after what they represent if possible.

How did I know that the callback would be called with those two parameters? The documentation for the onoff GPIO library's **watch()**

function (http://adafru.it/ehl) demonstrates that it will call the callback with *errors* as the first parameter and the *button value/state* as the second.

Callbacks don't have to be defined separately, and then passed to the functions that will call them. You can define the callback and pass it to the watch function all in the same step. The only drawback to this method is that your callback can't be used by any other buttons. Here's an example of defining your callback & passing it to **watch()** in the same step.

```
// button is attached to pin 17, LED to 18
var GPIO = require('onoff').Gpio,
    led = new GPIO(18, 'out'),
    button = new GPIO(17, 'in', 'both');

// pass the callback function to the
// as the first argument to watch() and define
// it all in one step
button.watch(function(err, state) {

  // check the state of the button
  // 1 == pressed, 0 == not pressed
  if(state == 1) {
    // turn LED on
    led.writeSync(1);
  } else {
    // turn LED off
    led.writeSync(0);
  }

});
```

Now that you have a handle on events & callbacks, let's compare how you would approach things in node.js vs how you would approach things in Arduino.

Arduino & node.js Examples

Let's take a look at how you would approach a couple simple tasks in Arduino, and then compare that with how you would tackle the same tasks using node.js.

Let's start with a classic example. Blinking a LED! First up, Arduino.

```
// LED pin
int led = 13;

void setup() {
  // initialize the LED pin as an output
  pinMode(led, OUTPUT);
}
```

```
void loop() {
  // turn the LED on
  digitalWrite(led, HIGH);
  // delay for one second
  delay(1000);
  // turn the LED off
  digitalWrite(led, LOW);
  // delay for one second
  delay(1000);
}
```

What's going on here? As the sketch runs the main loop, it flips the LED on and off separated by delays of one second. Now, let's look at the same example using node.js on a Raspberry PI.

```
// export GPIO 18 as an output.
var GPIO = require('onoff').Gpio,
    led = new GPIO(18, 'out');

// start a timer that runs the callback
// function every second (1000 ms)
setInterval(function() {
  // get the current state of the LED
  var state = led.readSync();
  // write the opposite of the current state to the LED pin
  led.writeSync(Number(!state));
}, 1000);
```

What's going on here? The same thing, but the difference is that the[Arduino delay](http://adafru.it/ehJ) function blocks all other code from executing for one second for each call to **delay(1000)**, and the [node.js setInterval](http://adafru.it/ehK) timer does not.

Why does this matter? Let's say you wanted to blink a green LED on and off every second, and you wanted to control the state of a red LED with a momentary button. Let's take a look at another Arduino example sketch.

```
const int buttonPin = 2;
const int redPin =  10;
const int greenPin = 11;

void setup() {
  // initialize the LED pins as outputs
  pinMode(redPin, OUTPUT);
  pinMode(greenPin, OUTPUT);
  // initialize the button pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop() {
```

```
  // read the state of the pushbutton value
  int buttonState = digitalRead(buttonPin);
  // turn the red LED on or off depending on
  // the button state.
  digitalWrite(redPin, buttonState);

  // turn the green LED on
  digitalWrite(greenPin, HIGH);
  // delay for one second
  delay(1000);
  // turn the green LED off
  digitalWrite(greenPin, LOW);
  // delay for one second
  delay(1000);

}
```

Since we are using Arduino's **delay** function, the button presses could be missed because the loop has to make it through both calls to **delay(1000)** before checking the button state.

Note: *I know some of the more experienced Arduino users can think of a ways to avoid using delay(), and solve this problem with interrupts or by checking the current millis() against previous millis() + 1000. This is just a simple example designed to highlight how things can easily get a lot more complicated when you don't have a way to call code asynchronously.*

Now let's look at the same thing on a Raspberry Pi using node.js.

```
// export GPIO 17 as the red LED output, GPIO 18 as
// the button input, and GPIO green as the button input.
var GPIO = require('onoff').Gpio,
    green = new GPIO(21, 'out'),
    red = new GPIO(17, 'out'),
    button = new GPIO(18, 'in', 'both');

// watch the button for changes, and pass
// the button state (1 or 0) to the red LED
button.watch(function(err, state) {
  red.writeSync(state);
});

// start a timer that runs the callback every second
setInterval(function() {
  // get the current state of the LED
  var state = green.readSync();
  // write the opposite of the current
  // green LED state to the green LED pin
  green.writeSync(Number(!state));
}, 1000);
```

Next, let's look at how to make things even simpler with node.js streams!

Streams

Now that we know how to use events, we can dig into one of my favorite parts of the node.js core: readable & writeable streams.

First, we'll walk through some examples of piping in Unix-like environments, since it would be good to know what node.js streams are emulating.

Unix Pipeline

Let's look at the contexts of an example text file by using the cat (http://adafru.it/ehL) command to output the contents of the file to stdout (http://adafru.it/ehM).

```
$ cat names.txt
Gordon
Mike
Brennen
Casey
Toni
Pamela
Nick
Joe
Randy
Ben
Jenny
Tyler
Pete
Chris
Rob
Dave
Jeff
Erik
Paul
```

Now, let's assume we want to reduce the list of names to names with less than four characters. We can pipe the output of **cat** to sed (http://adafru.it/ehN) (with a simple regex (http://adafru.it/ehO)) to accomplish this.

```
$ cat names.txt | sed '/.\{4\}/d'
Joe
Ben
Rob
```

Now that we have filtered the list to get the names we were looking for, let's sort the names in reverse dictonary order by piping the output of **sed** to sort (http://adafru.it/ehP).

```
$ cat names.txt | sed '/.\{4\}/d' | sort --dictionary-order --reverse
Rob
Joe
Ben
```

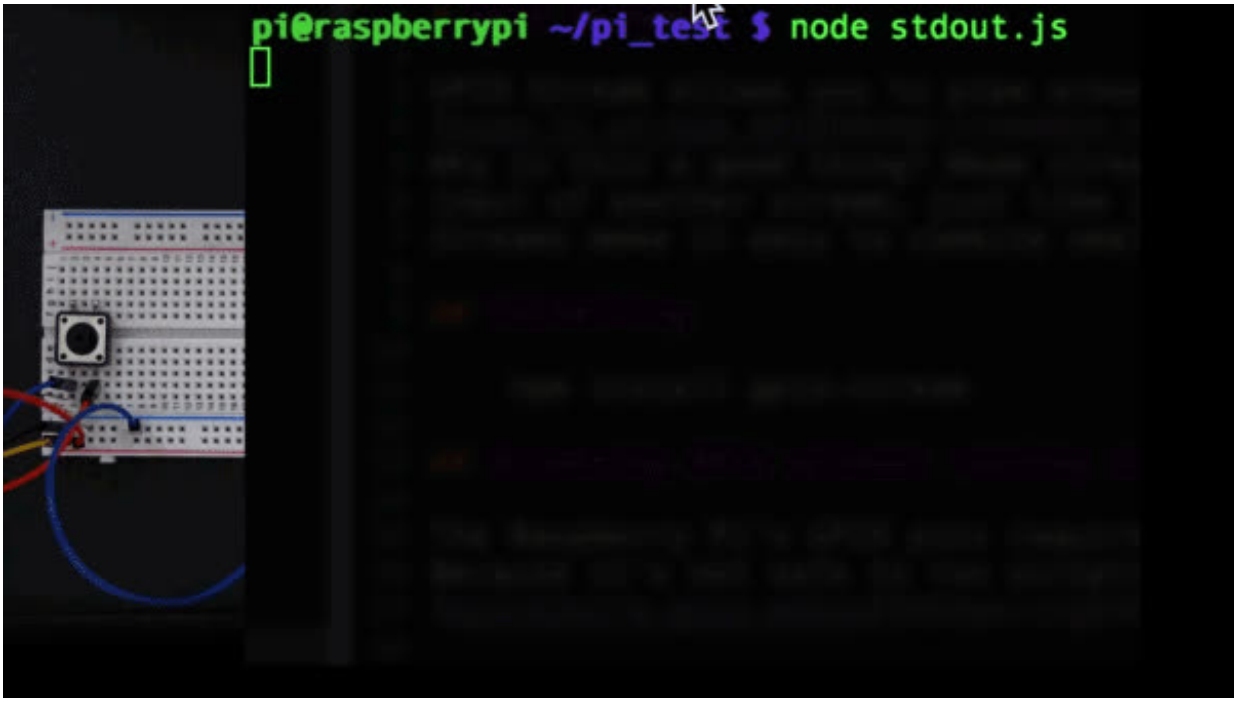
Pretty neat, huh? By piping the output of one simple program to the input of another simple program, you can accomplish complex tasks easily & quickly without having to have one behemoth program that reads files, filters, sorts, etc. If you'd like to learn more specifically about piping in Unix environments, check out the standard IO section (http://adafru.it/ehQ) of Brennen Bearnes' userland book (http://adafru.it/ehR).

Node.js Streams

Now that you have an idea what inspired the stream API, let's try something similar using the node.js stream **pipe()** command.

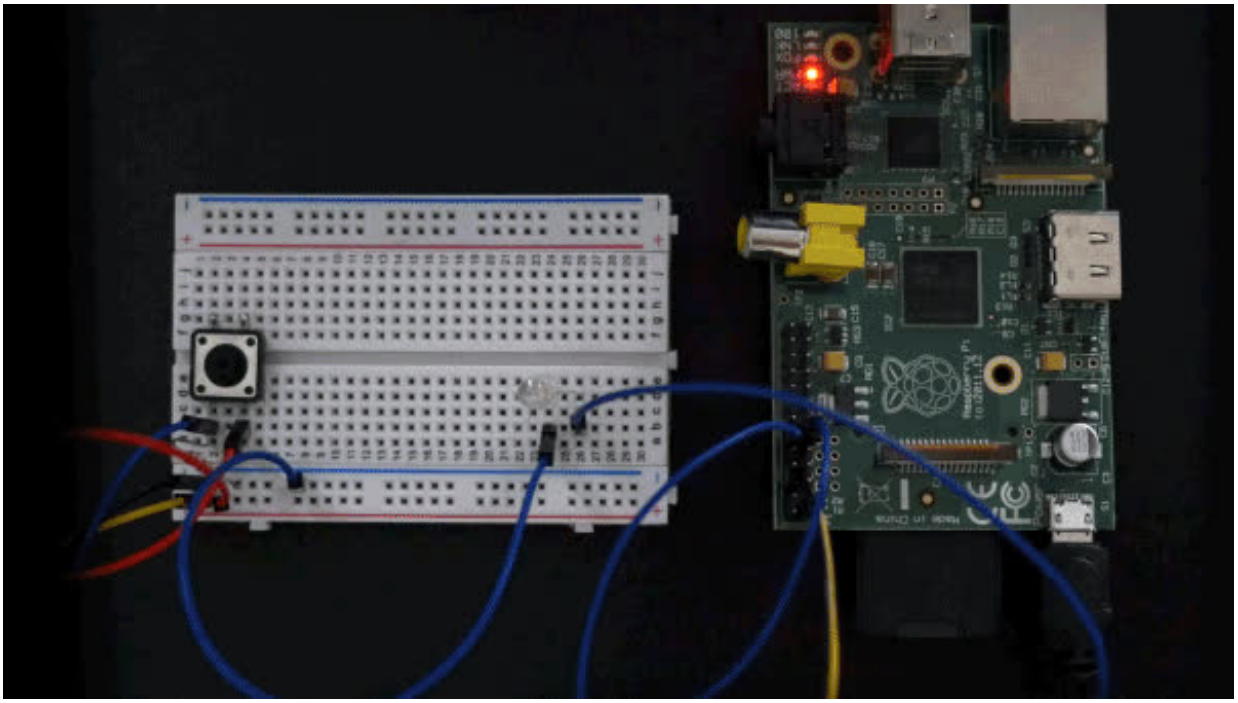
Let's say we want to view the state of the button from our SSH connection. Since stdout (http://adafru.it/ehM) is a writable stream, we can pipe the output of the button directly to *stdout*. Let's assume we have a tactile button attached to pin 17 on our Raspberry Pi.

```
var GpioStream = require('gpio-stream'),
    button = GpioStream.readable(17);

// pipe the button presses to stdout
button.pipe(process.stdout);
```



That was pretty simple. What if we wanted to redirect the output of our button to light up a LED on pin 18?

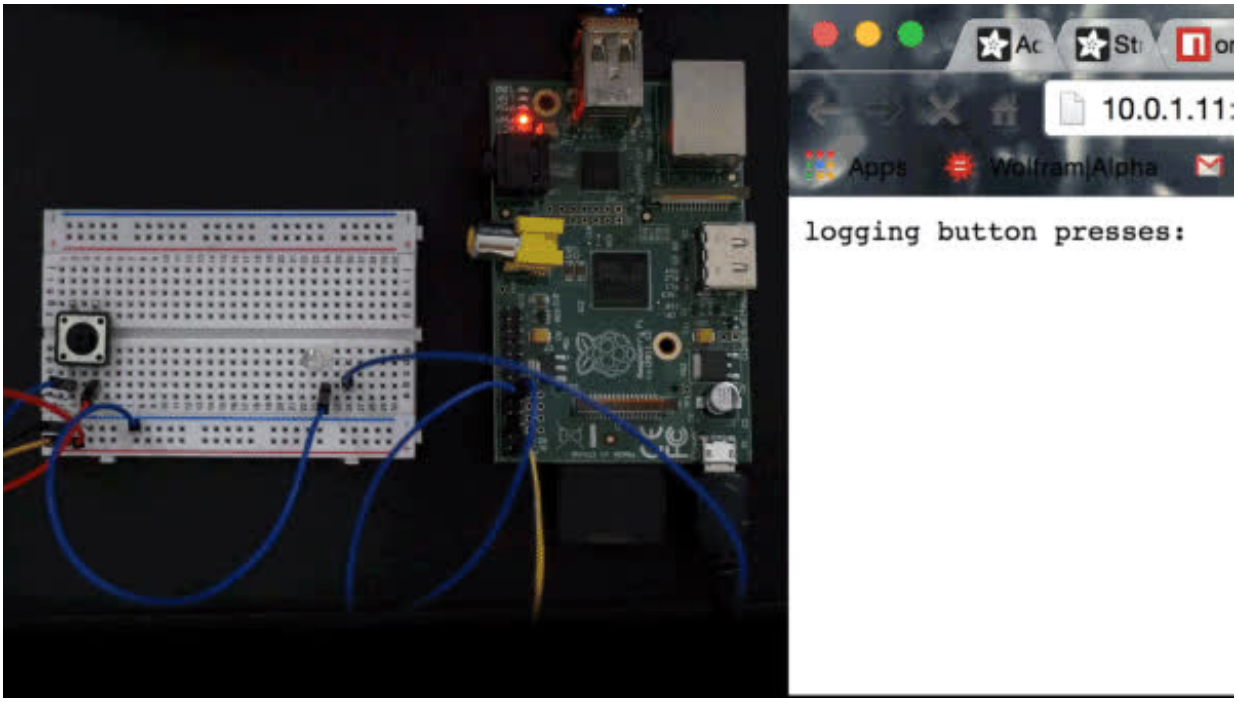
```
var GpioStream = require('gpio-stream'),
    button = GpioStream.readable(17),
    led = GpioStream.writable(18);

// pipe the button presses to the LED
button.pipe(led);
```



Chunked Transfer Stream

Now that you have a couple simple examples under your belt, let's try something a bit more interesting. What if we wanted to output our button presses to the LED & a web browser? Since the node.js HTTP server response is a writable stream, we can pipe the button presses to the LED, and then to the HTTP response object. Your browser can receive the presses on the fly using chunked transfer encoding (http://adafru.it/ehS). All of that with ~10 lines of code!

```
var GpioStream = require('gpio-stream'),
    http = require('http'),
    button = GpioStream.readable(17),
    led = GpioStream.writable(18);

var stream = button.pipe(led);

http.createServer(function (req, res) {
  res.setHeader('Content-Type', 'text/html');
  res.write('<pre>logging button presses:\n');
  stream.pipe(res);
}).listen(8080);
```



You could pipe the button presses to a file, a web service, or pretty much anything you can think of thanks to the endless number of stream packages available in npm (http://adafru.it/ehT). The button example is a simple one, but this same concept can be applied to a wide range of sensors.

Speaking of npm, let's examine how to use npm to manage third party packages. We'll even look at how you can create your own npm package and publish it for everyone to use!

Wrapping Things Up

As I described earlier, npm (http://adafru.it/ehT) is one of the best things about developing in a node.js environment. Because we used a few third party dependencies in the examples, let's take a look at how you would wrap up all of the dependencies and include them in your project by creating a *package.json* file.

package.json

The *package.json* file is what npm uses as a guide when downloading and installing dependencies for your project. The project title, description, version, authors, and third party dependencies are all pieces of information you would find in a standard package file.

Let's start by creating a folder for our project.

```
mkdir ~/pi_streams_example && cd ~/pi_streams_example
```

Now, we can use **npm init** to create our *package.json* file.

```
pi@raspberrypi ~/pi_streams_example $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (pi_streams_example)
version: (1.0.0)
description: An example that demonstrates how to use streams with GPIO on a Raspberry Pi
entry point: (index.js)
test command:
git repository:
keywords: raspberry, pi, streams, gpio
author: Todd Treece <todd@uniontownlabs.org>
license: (ISC)
About to write to /home/pi/pi_streams_example/package.json:

{
  "name": "pi_streams_example",
  "version": "1.0.0",
```

```
  "description": "An example that demonstrates how to use streams with GPIO on a Raspberry Pi",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "raspberry",
    "pi",
    "streams",
    "gpio"
  ],
  "author": "Todd Treece <todd@uniontownlabs.org>",
  "license": "ISC"
}


Is this ok? (yes) yes
```

Dependencies

That took care of creating the basics, but what about adding dependencies? We have a couple of third party dependencies to add to our package: onoff and gpio-stream.

```
npm install --save onoff gpio-stream
```

By using **npm install** with the **--save** flag, npm will automatically install & save the dependencies to the package.json file that we just created. All of your dependencies will now be installed in a folder called *node_modules* inside your project folder. The best part is that you do not have to worry about including dependencies when you share your code. All someone has to has to do is run **npm install**, and all of the dependencies you listed in your *package.json* file will automatically be installed via npm for them!

Example Package

Now that we have looked at how to create a package, let's look at an example package. I have created a git repository (http://adafru.it/eia)that contains examples from this guide. Run the following command to clone the repo to your Pi.

```
git clone https://github.com/adafruit/Pi_Node_Example.git ~/pi_examples && cd ~/pi_examples
```

Now, to install the dependencies, all you have to do is run**npm install**.

```
npm install
```

To run the examples, you can type **node** followed by the name of the example file.

```
node stream_stdout.js
```

Publishing Packages

When you create your own package, and decide you want to publish your package to npm, it's as simple as running **npm publish**, and your code will be available for the world to enjoy!

npm publish