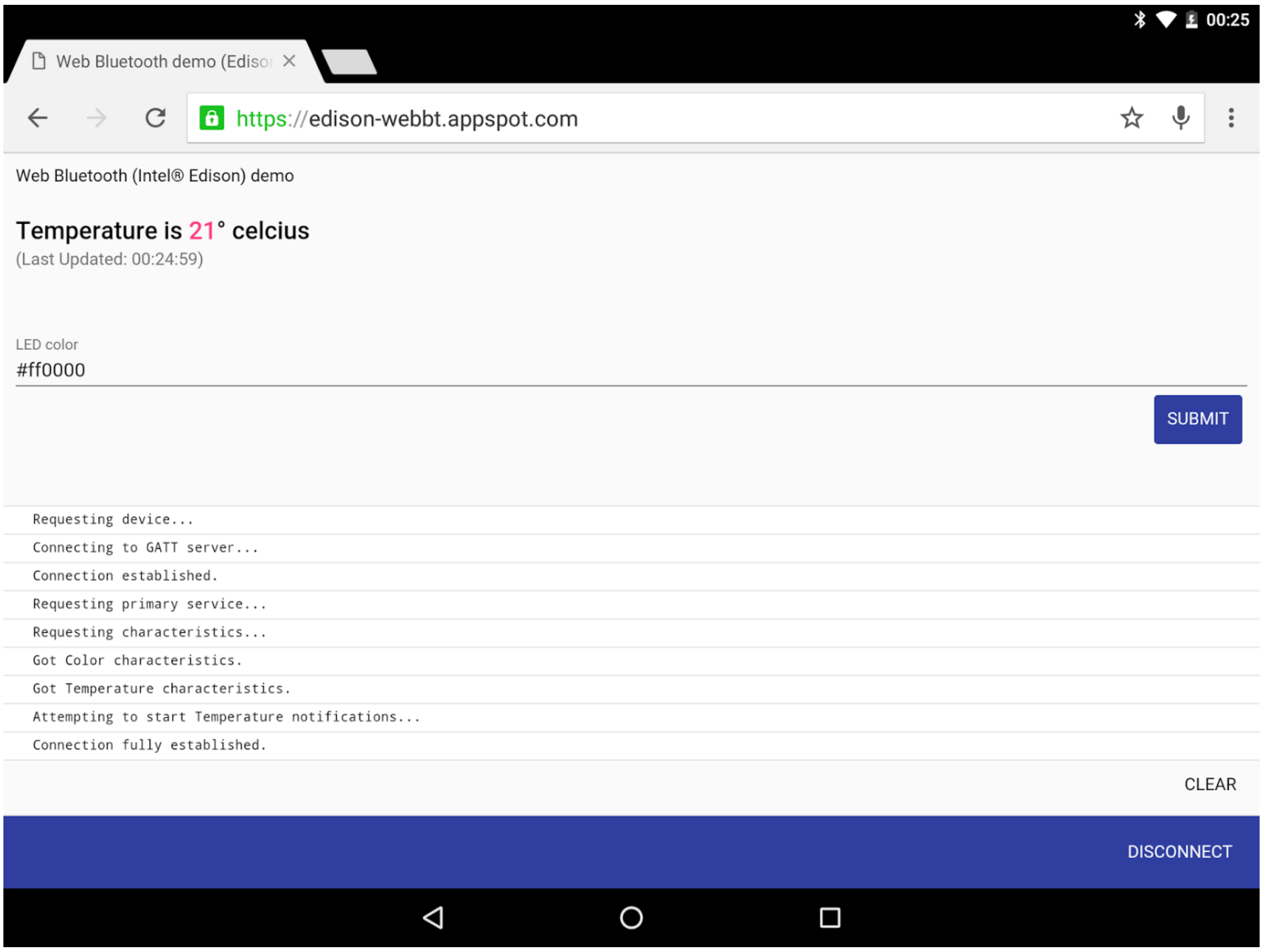
Creating a Web-Enabled IoT Device with Intel Edison

By Kenneth Christiansen
Kenneth is an Engineer at Intel

The Internet of Things is on everyone's lips these days, and it makes tinkerers and programmers like me very excited. Nothing is cooler than bringing your own inventions to life and being able to talk to them!

But IoT devices that install apps that you rarely use can be annoying, so we take advantage of upcoming web technologies such as the Physical Web and Web Bluetooth to make IoT devices more intuitive and less intrusive.



Web Bluetooth (Intel® Edison) demo

Temperature is 21° celcius
(Last Updated: 00:24:59)

LED color
#ff0000

```
Requesting device...
Connecting to GATT server...
Connection established.
Requesting primary service...
Requesting characteristics...
Got Color characteristics.
Got Temperature characteristics.
Attempting to start Temperature notifications...
Connection fully established.
```

Web and IoT, a match to be

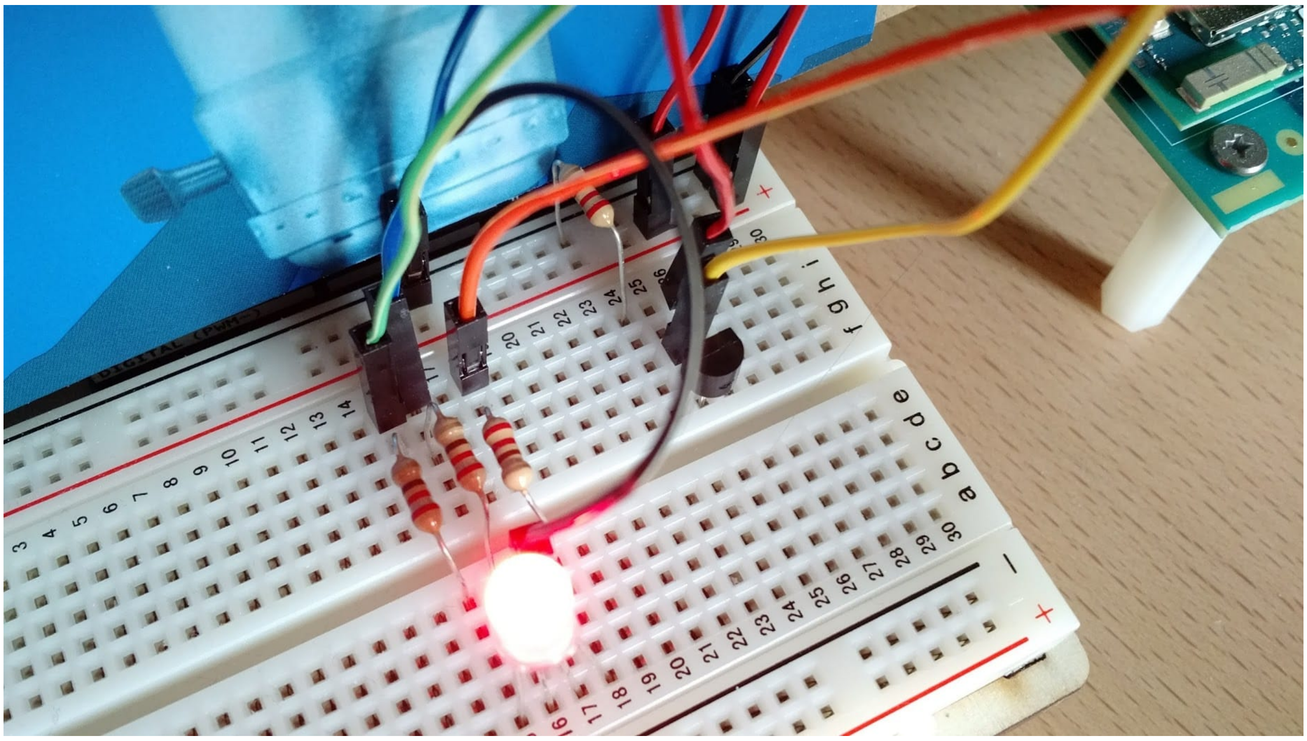
There are still a lot of hurdles to overcome before Internet of Things can be a huge success. One obstacle is companies and products that require people to install apps for each device they purchase, cluttering users' phones with a multitude of apps that they *rarely* use.

For this reason, we are very excited about the Physical Web project, which allows devices to broadcast a URL to an online website, in a *non-intrusive way*. In combinations with emerging web technologies such as Web Bluetooth, Web USB and Web NFC, the sites can connect directly to the device or at least explain the proper way of doing so.

Although we focus primarily on Web Bluetooth in this article, some use cases might be better suited for Web NFC or Web USB. For example, Web USB is preferred if you require a physical connection for security reasons.

The website can also serve as a Progressive Web App (PWA). We encourage readers to check out Google's explanation of PWAs. PWAs are sites that have a responsive, app-like user experience, can work offline and can be added to the device home screen.

As a proof of concept, I have been building a small device using the Intel® Edison Arduino breakout board. The device contains a temperature sensor (TMP36) as well as an actuator (colored LED cathode). The schematics for this device can be found at the end of this article.



The Intel Edison is an interesting product because it can run a full Linux* distribution. Therefore I can easily program it using *Node.js*. The installer lets you install the Intel* XDK which makes it easy to get started, although you can program and upload to your device manually as well.

Note: It is possible to use Brillo* or Ostro* OS instead of the default OS software. If you do, follow the Brillo or Ostro OS documentation to get a Node.js application running on the device.

For my Node.js app, I required three node modules, as well as their dependencies:

- `eddystone-beacon`
- `parse-color`
- `johnny-five`

The former automatically installs `noble`, which is the node module that I use to talk via Bluetooth Low Energy.

Note: It is important to not list `noble` as a dependency in the `package.json` file, as you need to use the same `noble` instance as `eddystone-beacon`, for them to work together. You can find more info here

The `package.json` file for the project looks like this:

```
{
  "name": "edison-webbluetooth-demo-server",
  "version": "1.0.0",
  "main": "main.js",
  "engines": {
    "node": ">=0.10.0"
  },
  "dependencies": {
    "eddystone-beacon": "^1.0.5",
    "johnny-five": "^0.9.30",
    "parse-color": "^1.0.0"
  }
}
```

Announcing the website

Beginning with version 49, Chrome on Android supports Physical Web, which allows Chrome to see URLs being broadcasted by devices around it. There are some requirements the developer must be aware of, like the need for the sites need to be publicly accessible and use HTTPS.

The Eddystone protocol has an 18 byte size limit on URLs. So to make the URL for my demo app work (https://webbt-sensor-hub.appspot.com/), I need to use a URL shortener.

Broadcasting the URL is quite simple. All you need to do it import the required libraries and call a few functions. One way of doing this is by calling `advertiseUrl` when the BLE chip is turned on:

```
var beacon = require("eddystone-beacon");
var bleno = require('eddystone-beacon/node_modules/bleno');

bleno.on('stateChange', function(state) {
  if (state === 'poweredOn') {
    beacon.advertiseUrl("https://goo.gl/9FomQC", {name: 'Edison'});
  }
}
```

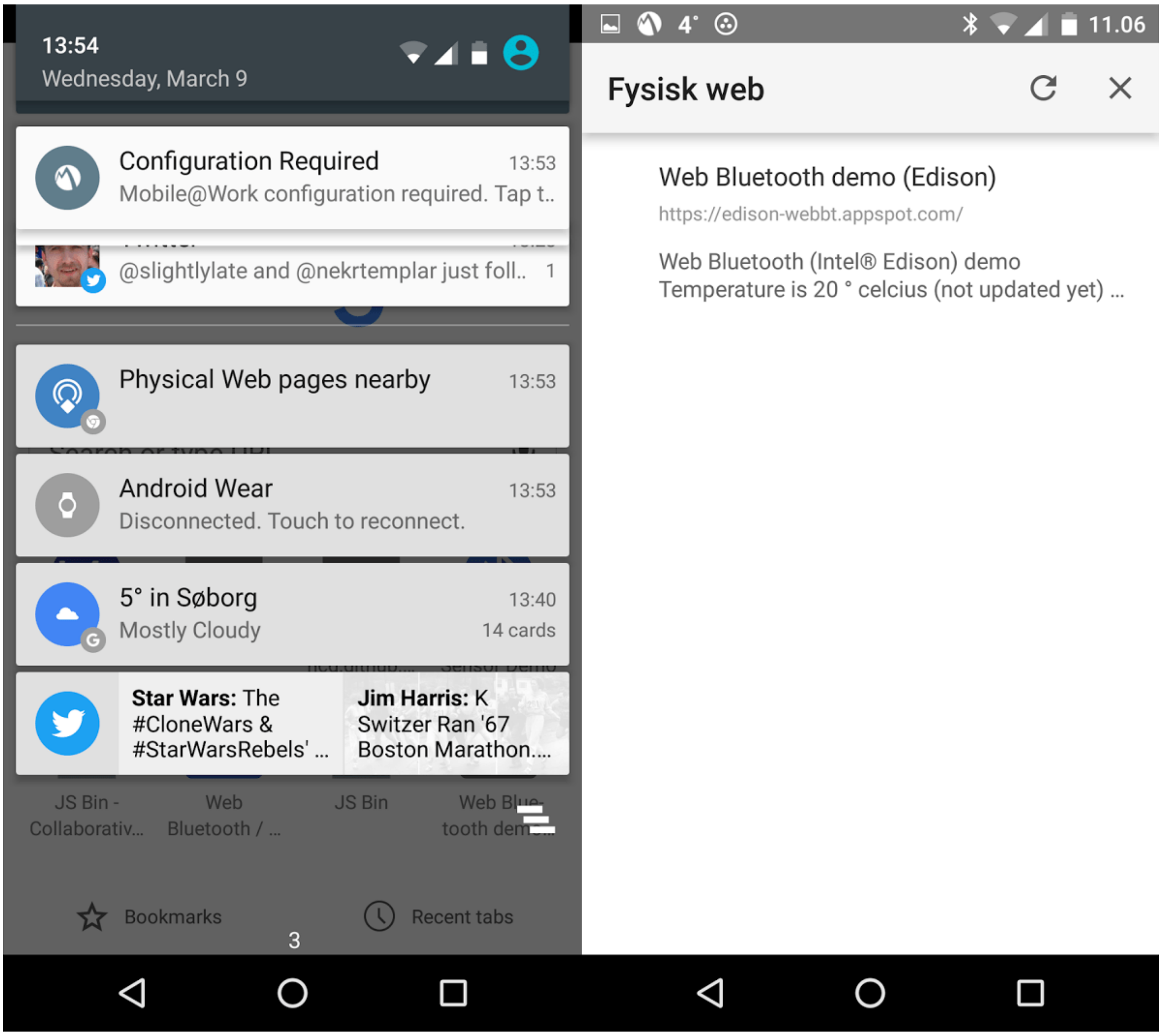
That really couldn't be easier. You see in the image below that Chrome finds the device nicely.

Note: The Physical Web enabled devices only show up on your phone when Bluetooth is turned on, and you launch Chrome (currently only works with Beta). Additionally, you have to opt into the feature first time you launch Chrome beta.

Communicating with the sensor/actuator

We use Johnny-Five* to talk to our board enhancements. Johnny-Five has a nice abstraction for talking to the TMP36 sensor.

Below you can find the simple code for being notified of temperature changes as well as setting the initial LED color.


13:54
Wednesday, March 9

Configuration Required 13:53
Mobile@Work configuration required. Tap t..

@slightlylate and @nekrtemplar just foll.. 1

Physical Web pages nearby 13:53

Android Wear 13:53
Disconnected. Touch to reconnect.

5° in Søborg 13:40
Mostly Cloudy 14 cards

Star Wars: The #CloneWars & #StarWarsRebels' ...

Jim Harris: K Switzer Ran '67 Boston Marathon....

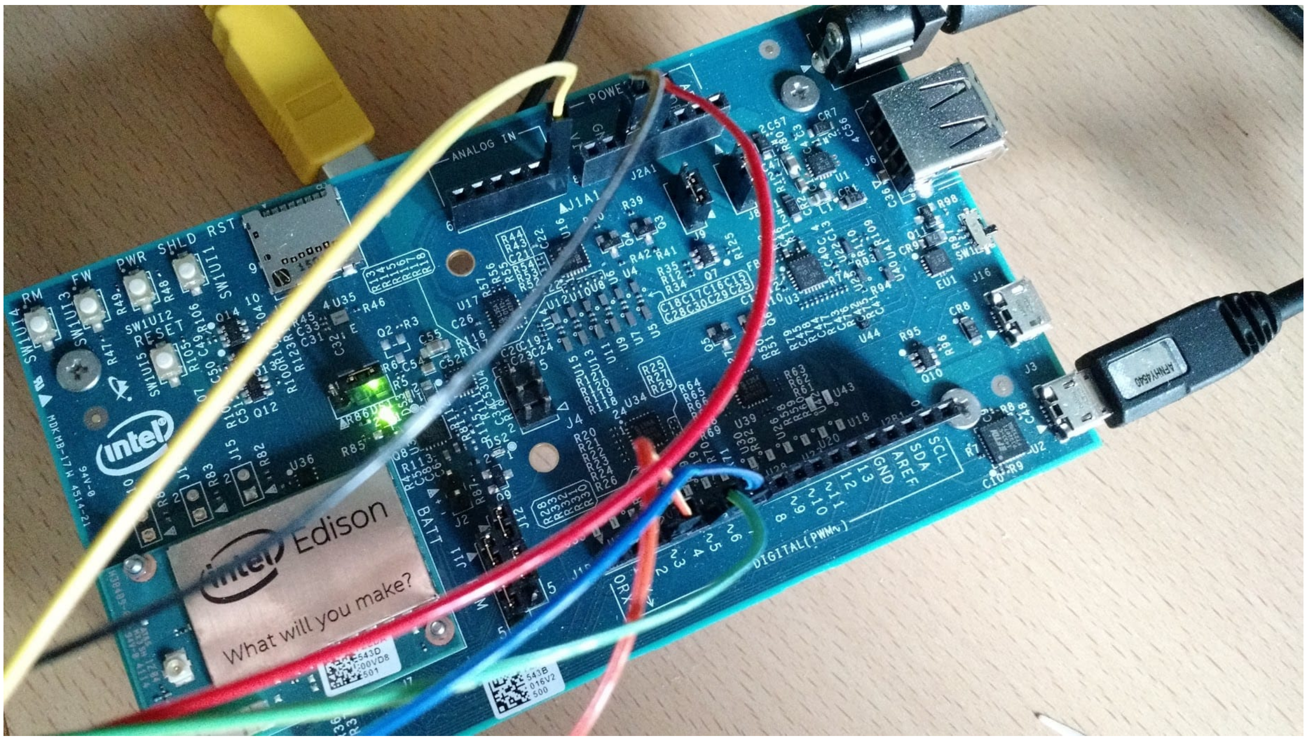
4° 11.06

Fysisk web

Web Bluetooth demo (Edison)
https://edison-webbt.appspot.com/
Web Bluetooth (Intel® Edison) demo
Temperature is 20 ° celcius (not updated yet) ...

You can ignore the above `*Characteristic` variables for now; these will be defined in the later section about interfacing with Bluetooth.

As you might notice in the instantiation of the Thermometer object, I talk to the TMP36 via the analog `A0` port. The voltage legs on the color LED cathode are connected to digital pins 3, 5 and 6, which happen to be the pulse-width modulation (PWM) pins on the Edison Arduino breakout board.



Talking to Bluetooth

Talking to Bluetooth couldn't be much easier than it is with `noble`.

In the following example, we create two Bluetooth Low Energy characteristics: one for the LED and one for the temperature sensor. The former allows us to read the current LED color and set a new color. The latter allows us to subscribe to temperature change events.

> Initially I had some problems with the Bluetooth connection being unstable, not working on every startup, or bailing out with a Frame Reassemble failure while connecting.
>
> If that happens, run the `rfkill block bluetooth` command, followed by `rfkill unblock bluetooth` over the serial connection to make it work again. The startup issue went away when I started powering the device from a power supply instead of using USB for power.
>
> If you encounter Frame Reassemble failures, reduce how often you send temperature change events until you no longer encounter the failure.
>
> Generally you should always use external power when using Bluetooth or when you connect something like a servo to your board.

With `noble`, creating a characteristic is quite easy. All you need to do is to define how the characteristic communicates and define a UUID. The communication options are read, write, notify, or any combination thereof. The easiest way to do this is to create a new object and inherit from `bleno.Characteristic`.

Note: I am not using ES2016 here as the Edison SDK currently uses an older version of Node.js.

With the newly launched Ostro Project which supports the Edison, that is no longer the case. If you use Brillo as part of the Brillo Early Access Program, then it is possible to compile and install a recent version of Node.js.

The resulting characteristic object looks like the following:

```
var TemperatureCharacteristic = function() {
  bleno.Characteristic.call(this, {
    uuid: 'fc0a',
    properties: ['read', 'notify'],
    value: null
  });

  this._lastValue = 0;
  this._total = 0;
  this._samples = 0;
  this._onChange = null;
};

util.inherits(TemperatureCharacteristic, bleno.Characteristic);
```

We are storing the current temperature value in the `this._lastValue` variable. We need to add an `onReadRequest` method and encode the value for a "read" to work.

```
TemperatureCharacteristic.prototype.onReadRequest = function(offset, callba
  var data = new Buffer(8);
  data.writeDoubleLE(this._lastValue, 0);
  callback(this.RESULT_SUCCESS, data);
};
```

For "notify" we need to add a method to handle subscriptions and unsubscription. Basically, we simply store a callback. When we have a new temperature reason we want to send, we then call that callback with the new value (encoded as above).

```
TemperatureCharacteristic.prototype.onSubscribe = function(maxValueSize, up
  console.log("Subscribed to temperature change.");
  this._onChange = updateValueCallback;
  this._lastValue = undefined;
};

TemperatureCharacteristic.prototype.onUnsubscribe = function() {
  console.log("Unsubscribed to temperature change.");
  this._onChange = null;
};
```

As values can fluctuate a bit, we need to smooth out the values we get from the TMP36 sensor. I opted to simply take the average of 100 samples and only send updates when the temperature changes by at least 1 degree.

```
TemperatureCharacteristic.prototype.valueChange = function(value) {
  this._total += value;
  this._samples++;

  if (this._samples < NO_SAMPLES) {
    return;
  }

  var newValue = Math.round(this._total / NO_SAMPLES);

  this._total = 0;
  this._samples = 0;

  if (this._lastValue && Math.abs(this._lastValue - newValue) < 1) {
    return;
  }

  this._lastValue = newValue;

  console.log(newValue);
  var data = new Buffer(8);
  data.writeDoubleLE(newValue, 0);

  if (this._onChange) {
    this._onChange(data);
  }
};
```

That was the temperature sensor. The color LED is simpler. The object as well as the "read" method are shown below. The characteristic is configured to allow for "read" and "write" operations and has a different UUID than the temperature characteristic.

```
var ColorCharacteristic = function() {
  bleno.Characteristic.call(this, {
    uuid: 'fc0b',
    properties: ['read', 'write'],
    value: null
  });
  this._value = 'ffffff';
  this._led = null;
};

util.inherits(ColorCharacteristic, bleno.Characteristic);

ColorCharacteristic.prototype.onReadRequest = function(offset, callback) {
  var data = new Buffer(this._value);
  callback(this.RESULT_SUCCESS, data);
};
```

To control the LED from the object, I add a **`this._led`** member which I use to store the Johnny-Five LED object. I also set the color of the LED to its default value (white, aka **`#ffffff`**).

```
board.on("ready", function() {
  ...
  colorCharacteristic._led = led;
  led.color(colorCharacteristic._value);
  led.intensity(30);
  ...
}
```

The "write" method receives a string (just like "read" sends a string), which can consist of a CSS color code (For example: CSS names such as **`rebeccapurple`** or hex codes such as **`#ff00bb`**). I use a node module called parse-color to always get the hex value which is what Johnny-Five expects.

```
ColorCharacteristic.prototype.onWriteRequest = function(data, offset, witho
  var value = parse(data.toString('utf8')).hex;
  if (!value) {
    callback(this.RESULT_SUCCESS);
    return;
  }

  this._value = value;
  console.log(value);

  if (this._led) {
    this._led.color(this._value);
  }
  callback(this.RESULT_SUCCESS);
};
```

All of the above will not work if we don't include the *bleno* module. **`eddystone-beacon`** will not work with *bleno* unless you use the **`noble`** version distributed with it. Luckily doing that is quite simple:

```
var bleno = require('eddystone-beacon/node_modules/bleno');
var util = require('util');
```

Now all we need is for it to advertise our device (UUID) and its characteristics (other UUIDs)

```
bleno.on('advertisingStart', function(error) {
    ...
    bleno.setServices([
      new bleno.PrimaryService({
        uuid: 'fc00',
        characteristics: [
          temperatureCharacteristic, colorCharacteristic
        ]
      })
    ]);
});
```

Creating the client web app

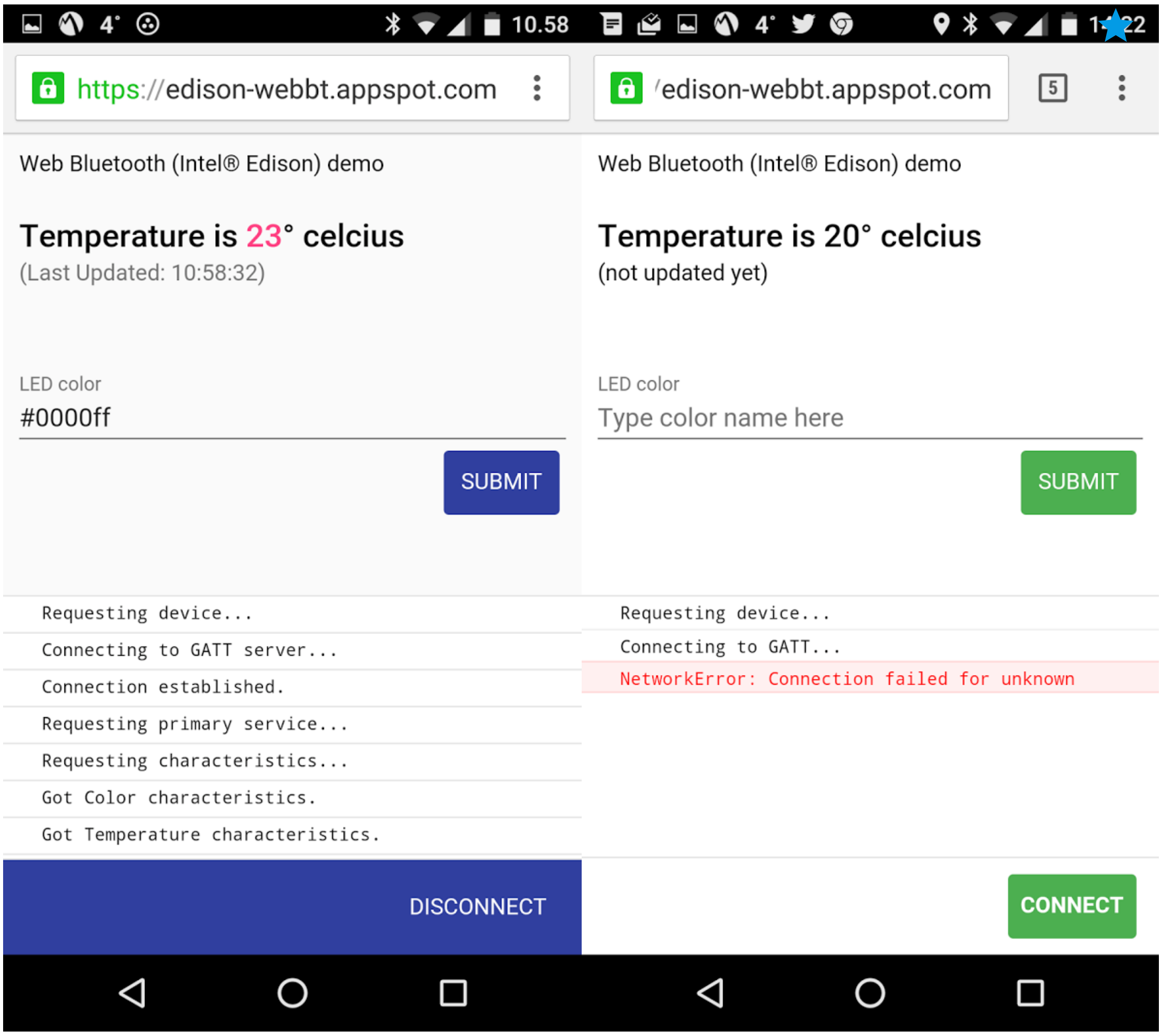
Without getting into too many defails about how the non-bluetooth parts of the client app work, we can demonstrate a responsive user interface created in Polymer* as an example. The resulting app is shown below:

The right side shows an earlier version, that showcases a simple error log that I added to ease the development.

Web Bluetooth makes it easy to communicate with Bluetooth Low Energy devices, so let's look at a simplified version of my connection code. If you don't know how promises work, check out this resource before reading further.

Connecting to a Bluetooth device involves a chain of promises. First we filter for the device (UUID: `FC00`, name: `Edison`). This displays a dialog to allow the user to select the device given the filter. Then we connect to the GATT service and get the primary service and associated characteristics, and then we read the values and set up notification callbacks.

Note: To make successive read/writes in the promise chain happen property, it is best practice to avoid fetching the characteristics *in parallel* with something like `Promise.all([p1, p2])`.


10.58

https://edison-webbt.appspot.com

Web Bluetooth (Intel® Edison) demo

Temperature is 23° celcius
(Last Updated: 10:58:32)

LED color
#0000ff

SUBMIT

```
Requesting device...
Connecting to GATT server...
Connection established.
Requesting primary service...
Requesting characteristics...
Got Color characteristics.
Got Temperature characteristics.
```

DISCONNECT

edison-webbt.appspot.com

Web Bluetooth (Intel® Edison) demo

Temperature is 20° celcius
(not updated yet)

LED color
Type color name here

SUBMIT

```
Requesting device...
Connecting to GATT...
NetworkError: Connection failed for unknown
```

CONNECT

The simplified version of our code below only works with the latest Web Bluetooth API and therefore thus requires Chrome Dev (M49) on Android.

```
navigator.bluetooth.requestDevice({
  filters: [{ name: 'Edison' }],
  optionalServices: [0xFC00]
})

.then(device => device.gatt.connect())

.then(server => server.getPrimaryService(0xFC00))

.then(service => {
  let p1 = () => service.getCharacteristic(0xFC0B)
  .then(characteristic => {
    this.colorLedCharacteristic = characteristic;
    return this.readLedColor();
   });
  let p2 = () => service.getCharacteristic(0xFC0A)
  .then(characteristic => {
    characteristic.addEventListener(
      'characteristicvaluechanged', this.onTemperatureChange);
    return characteristic.startNotifications();
  });

  return p1().then(p2);
})

.catch(err => {
  // Catch any error.
})

.then(() => {
  // Connection fully established, unless there was an error above.
});
```

Reading and writing a string from a **`DataView`** / **`ArrayBuffer`** (what the WebBluetooth API uses) is just as easy as using **`Buffer`** on the Node.js side. All we need to use is **`TextEncoder`** and **`TextDecoder`**:

```
readLedColor: function() {
  return this.colorLedCharacteristic.readValue()
  .then(data => {
    // In Chrome 50+, a DataView is returned instead of an ArrayBuffer.
    data = data.buffer ? data : new DataView(data);
    let decoder = new TextDecoder("utf-8");
    let decodedString = decoder.decode(data);
    document.querySelector('#color').value = decodedString;
  });
},

writeLedColor: function() {
  let encoder = new TextEncoder("utf-8");
  let value = document.querySelector('#color').value;
  let encodedString = encoder.encode(value.toLowerCase());

  return this.colorLedCharacteristic.writeValue(encodedString);
},
```

Handling the **`characteristicvaluechanged`** event for the temperature sensor is also quite easy:

```
onTemperatureChange: function(event) {
  let data = event.target.value;
  // In Chrome 50+, a DataView is returned instead of an ArrayBuffer.
  data = data.buffer ? data : new DataView(data);
  let temperature = data.getFloat64(0, /*littleEndian=*/ true);
  document.querySelector('#temp').innerHTML = temperature.toFixed(0);
},
```

Summary

That was it folks! As you can see, communicating with Bluetooth Low Energy using Web Bluetooth on the client side and Node.js on the Edison is quite easy and very powerful.

Using the Physical Web and Web Bluetooth, Chrome finds the device and allows the user to easily connect to it without installing seldom-used applications that the user may not want, and which may update from time to time.

Demo

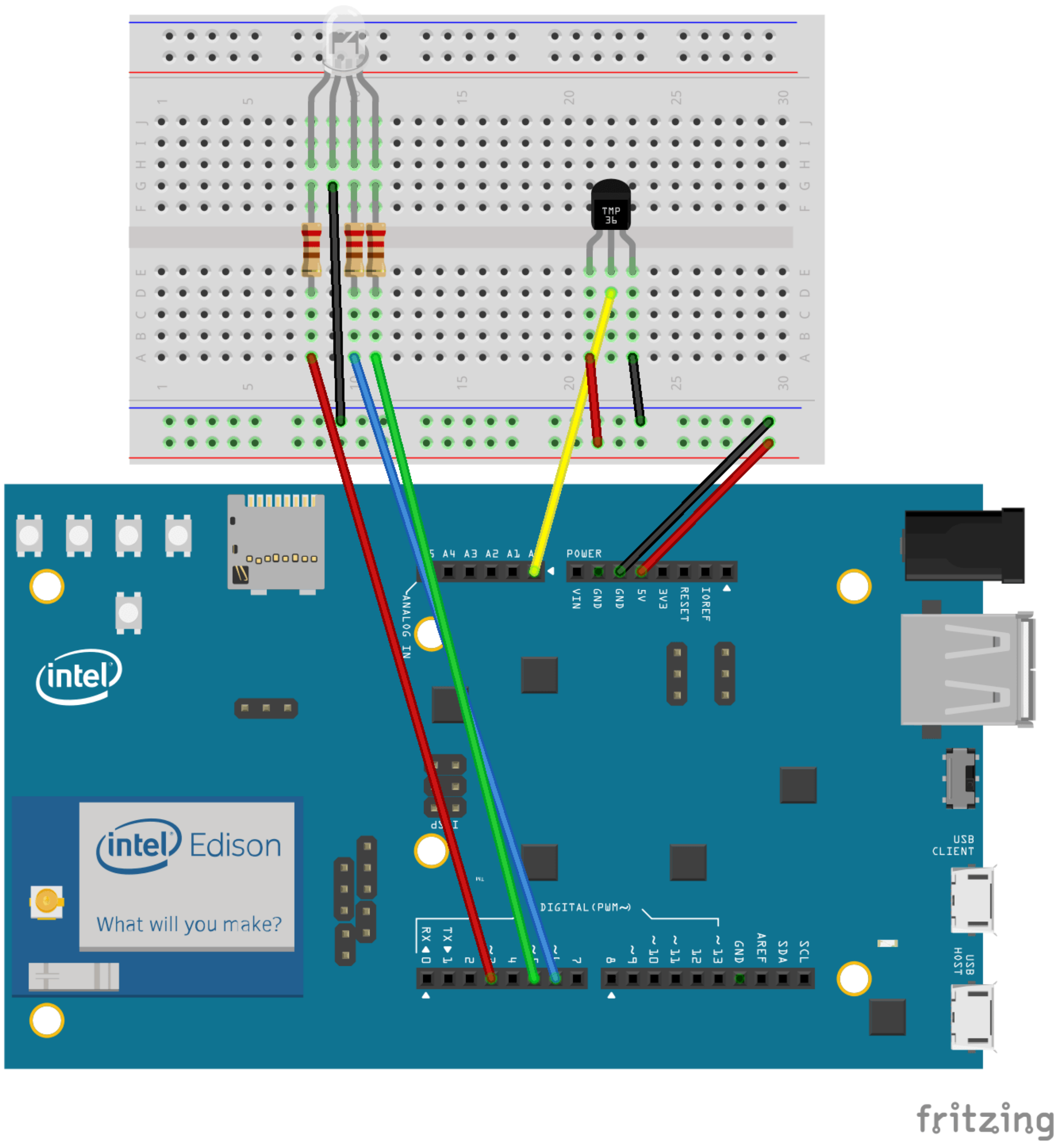
You can try the client to get inspired about how can you create your own web apps to connect to your custom Internet of Things devices.

Source code

The source code is available here. Feel free to report issues or send patches.

Sketch

If you are really adventurous and want to reproduce what I have done, refer to the Edison and breadboard sketch below:



Last updated July 2, 2018.