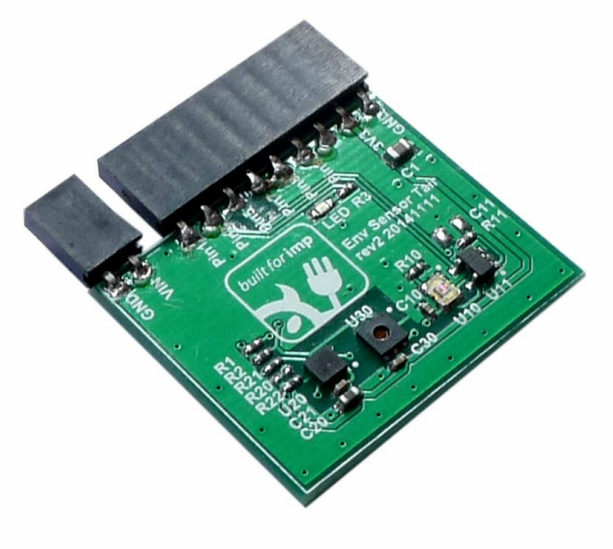
Electric Imp Tails Project: Weather Station

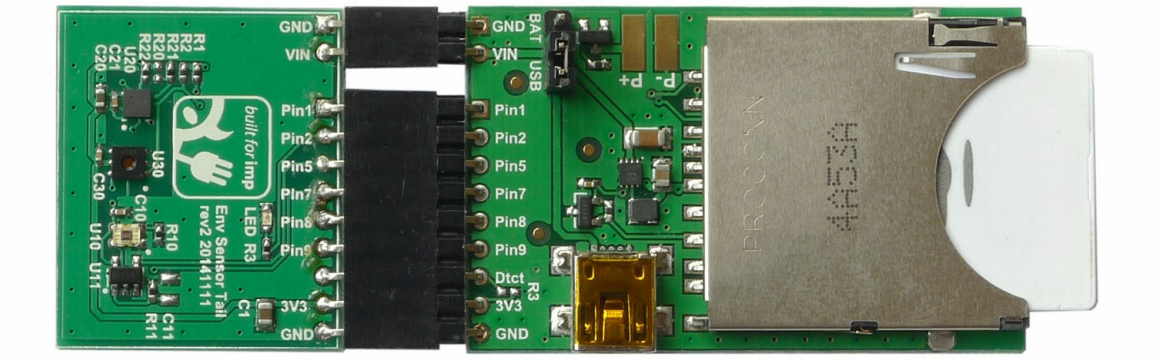
The sensors integrated into the Env Tail provide all we need to create a rudimentary weather station. The device doesn't need to do much at all: just take regular readings and, in response to the user asking for a more immediate update, take one-off readings too.



To make this work, we need to build some functionality into the device's agent – its online gopher that mediates communications between the device and the wider Internet. The agent is actually a micro web server, a feature we can make use of to provide a basic weather station UI: a readout of the current temperature, humidity and air pressure, and a button to request updated values.

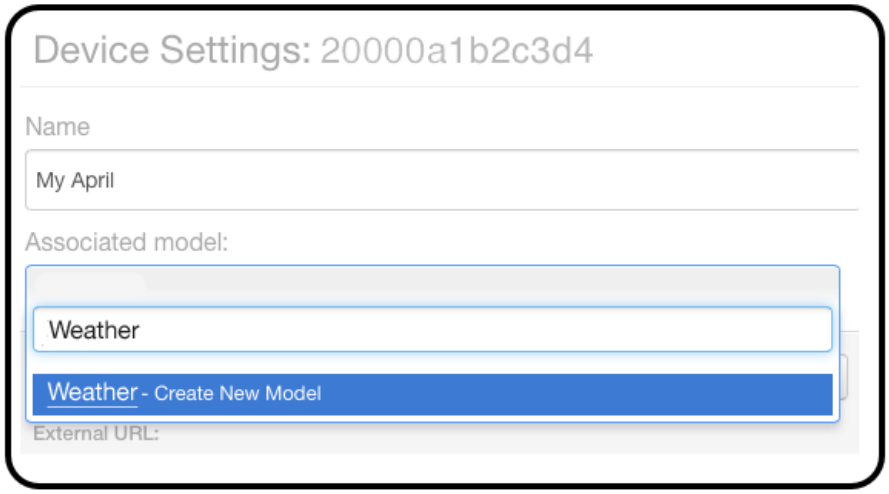
Step 1: Assemble the Hardware

If you haven't done so already, clip the Env Tail onto your April dev board. Slip in the imp001 card too, and connect the mini USB cable to a power supply and then to the April.



Step 2: Program the Project

Open the Electric Imp IDE in a web browser. You'll see your device listed on the left-hand side under 'Unassigned Devices'. Click on the gearwheel icon to the right of this to display the 'Device Settings' window. Here you can give the device a more friendly name, such as 'My April'. Click on the pop-up menu under 'Associated Model:' and in the empty space that appears, type in 'Weather' (without the single quotes). When you've done, click on 'Save Changes'.



Your device should now disappear from 'Unassigned Devices' and reappear under 'Weather' in the 'Active Models' section. If you can't see your device, just click on the disclosure triangle to the left of 'Weather' to reveal it. Can't see 'Weather'? Click on the disclosure triangle to the left of 'Active Models'.

Click on 'My April' and you'll see 'Agent', 'Device' and 'Device Logs' panels appear in the space on the right-side of the screen. This is where you enter your programs: one for the device, another for its online agent. Both blocks of code together comprise a model – Electric Imp terminology for an Internet of Things app.

The code you need is listed below; copy and paste it into the IDE's agent and device code panels. Make sure you paste it correctly. The agent code's first line should read:

```
    // Agent Code
```

Agent Code

```
// Agent Code

// HTML output
local baseTop = @"<!DOCTYPE html>
<html>
<head>
<title>Weather Station</title>
<meta http-equiv='refresh' content='120'>
<style>
h2 { font-family:Verdana;color:#3B6BB2; }
p { font-family:Verdana;padding-bottom:10px; }
b { color:#3B6BB2; }
</style>
</head>
<body>
<table style='width:100%;border:0px;' cellpadding='20'><tr><td align='center'>
<h2 align='center'>Weather Station</h2>
<table style='width:40%;border:1px solid #3B6BB2;border-collapse:collapse;' cellpadding='20'><tr><td valign='top'>"

local baseBottom = @"</td></tr></table></td></tr></table></body>
</html>";

local html = null;

local lastReading = {};
lastReading.pressure <- 1013.25;
lastReading.temp <- 22;
lastReading.day <- true;
lastReading.humid <- 0;

// Add a function to post data from the device to your stream

function manageReading(reading) {
    // Note: reading is the data passed from the device, ie.
    // a Squirrel table with the key 'temp'
    server.log("PostReading called");

```

```
    // Create HTML strings
    local tempString = "<p><b>Temperature</b> " + format("%.2f", reading.temp) + "&deg;C</p>";
    local humidString = "<p><b>Humidity</b> " + format("%.2f", reading.humid) + "%</p>";
    local pressString = "<p><b>Pressure</b> " + format("%.2f", reading.pressure) + "hPa &ndash; ";

    local diff = reading.pressure - lastReading.pressure;

    if (diff > 0) {
        pressString = pressString + "rising</p>"
    } else {
        pressString = pressString + "falling</p>";
    }

    html = baseTop + tempString + humidString + pressString + baseBottom;
    lastReading = reading;
}

function webServer(request, response) {
    // Serve up the HTML page with the weather data
    try {
        if (html == null) manageReading(lastReading);
        response.send(200, html);
    } catch (err) {
        response.send(500, "Agent Error: " + err);
    }
}

// Register the function to handle requests from a web browser
http.onrequest(webServer);

// Register the function to handle data messages from the device
device.on("reading", manageReading);
```

weather.agent.nut hosted with ❤ by **GitHub** **view raw**

Device Code

```
// Device Code
#require "Si702x.class.nut:1.0.0"
#require "APDS9007.class.nut:1.0.0"
#require "LPS25H.class.nut:1.0.0"

// Establish a global variable to hold environmental data
data <- {};
data.temp <- 0;
data.humid <- 0;
data.pressure <- 0;
data.day <- true;
data.lux <- 0;

// Instance the Si702x and save a reference in tempHumidSensor
hardware.i2c89.configure(CLOCK_SPEED_400_KHZ);
local tempHumidSensor = Si702x(hardware.i2c89);

// Instance the LPS25H and save a reference in pressureSensor
local pressureSensor = LPS25H(hardware.i2c89);
pressureSensor.enable(true);

// Instance the APDS9007 and save a reference in lightSensor
local lightOutputPin = hardware.pin5;
lightOutputPin.configure(ANALOG_IN);

local lightEnablePin = hardware.pin7;
lightEnablePin.configure(DIGITAL_OUT, 1);

local lightSensor = APDS9007(lightOutputPin, 47000, lightEnablePin);

// Configure the LED (on pin 2) as digital out with 0 start state
```

```
local led = hardware.pin2;
led.configure(DIGITAL_OUT, 0);

// This function will be called regularly to take the temperature
// and log it to the device's agent

function getReadings() {
    // Flash the LED
    flashLed();

    // Get the light level
    local lux = lightSensor.read();

    // Day or night?
    if (lux > 300) {
        data.day = true;
    } else {
        data.day = false;
    }

    // Get the pressure. This is an asynchronous call, so we need to
    // pass a function that will be called only when the sensor
    // has a value for us.
    pressureSensor.read(function(pressure) {
        data.pressure = pressure;

        // Now get the temperature and humidity. Again, this is an
        // asynchronous call: we need to a pass a function to be
        // called when the data has been returned. This time
        // the callback function also has to bundle the data
        // and send it to the agent. Then it puts the device into
        // deep sleep until it's next time for a reading.
        tempHumidSensor.read(function(reading) {
            data.temp = reading.temperature;
            data.humid = reading.humidity;

            // Send the data to the agent
            agent.send("reading", data);

            // Put the imp to sleep for five minutes BUT
            // only do so when impOS has done all it needs to
            // do and has gone into an idle state
            imp.onidle(function() { server.sleepfor(300); } );
        });
    });
}

function flashLed() {
    // Turn the LED on (write a HIGH value)
    led.write(1);

    // Pause for half a second
    imp.sleep(0.5);

    // Turn the LED off
    led.write(0);
}

// Take a temperature reading as soon as the device starts up
// Note: when the device wakes from sleep (caused by line 86)
// it runs its device code afresh - ie. it does a warm boot
getReadings();
```

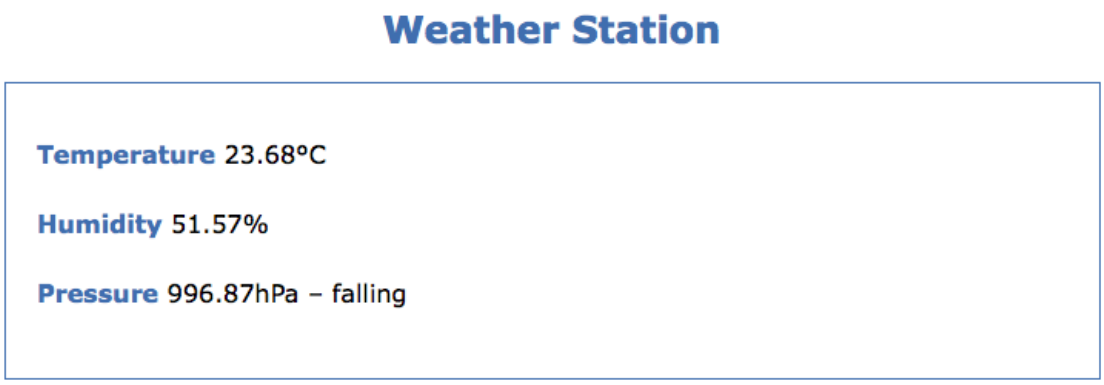
weather.device.nut hosted with ❤ by **GitHub** **view raw**

Step 3: Run the Code

The code you pasted into the IDE is ready to run, so click on the 'Build and Run' button. You should see the Tail flash its LED and then do so

every five minutes. Every time it does, it signals that the imp has taken readings from the Tail's sensors. The readings are packaged up and sent to the device's agent.

The agent generates the HTML code of a basic web page and incorporates the passed readings into it. You can view this page by clicking on the agent URL at the head of the Agent Code pane in the IDE. This opens a new window or tab in your browser with the Weather Station UI.



The UI is automatically refreshed every two minutes; the device takes a reading every five minutes.

On receipt of the readings, the agent updates the HTML it returns to the web browser. It checks whether the request for a reading was triggered by a press of the Refresh button – indicated by a saved HTTP response – and sends back fresh HTML if that was the case.

Step 4: What Next?

Well done – you've built a simple weather station. Here are some ideas to help you extend its functionality:

- Modify the UI to show whether it is night or day at the station site
  - The Tail already sends 'true' or 'false' according to whether it thinks there is enough light to indicate it is day.
  - Use these values to update the UI by incorporating extra HTML.
- Change the update frequency of the readings
  - Line 86 in the device code controls the number of seconds the device will sleep between readings.
  - For more information see server.sleepfor().
- Use the data to make weather predictions
  - The agent code checks whether the air pressure is rising or falling – this gives a crude mechanism for weather forecasting.
  - Rising pressure indicates better weather is on the way.
  - Falling pressure indicates worsening conditions.
- Try some of the other Tails projects
  - Visit the Env Tail page for more applications you can explore.

PLATFORM

BUSINESS SOLUTIONS

CUSTOMERS

DEV CENTER
Dev Kits
Getting Started
API Reference
Developer Guides
Hardware Reference
Manufacturing

ABOUT US
Jobs
FAQ
Media Coverage
Blog
Contact

electric imp

© 2015 Electric Imp, Inc.