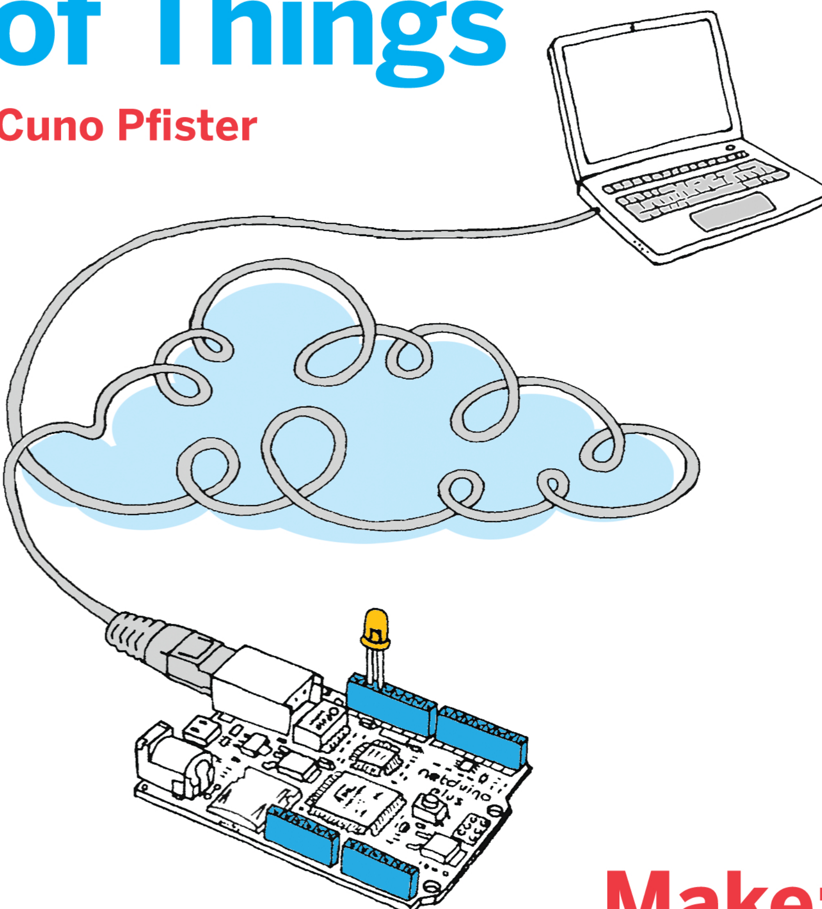
Make: PROJECTS

CONNECTING SENSORS AND MICRO-CONTROLLERS TO THE CLOUD

Getting Started with the Internet of Things

Cuno Pfister



O'REILLY®

Make:
makezine.com

Getting Started with the Internet of Things

Cuno Pfister

O'REILLY®

BEIJING • CAMBRIDGE • FARNHAM • KÖLN • SEBASTOPOL • TOKYO

Getting Started with the Internet of Things

by Cuno Pfister

Printed in the United States of America.

Published by O'Reilly Media, Inc.
1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Print History: May 2011: First Edition.

Editor: Brian Jepson
Production Editor: Jasmine Perez
Copyeditor: Marlowe Shaeffer
Proofreader: Emily Quill
Compositor: Nancy Wolfe Kotary
Indexer: Angela Howard
Illustrations: Marc de Vinck
Cover Designer: Marc de Vinck

ISBN: 978-1-4493-9357-1

[LSI]

Preface

One of the most fascinating trends today is the emergence of low-cost *microcontrollers* that are sufficiently powerful to connect to the Internet. They are the key to the *Internet of Things*, where all kinds of devices become the Internet's interface to the physical world.

Traditionally, programming such tiny *embedded* devices required completely different platforms and tools than those most programmers were used to. Fortunately, some microcontrollers are now capable of supporting modern software platforms like .NET, or at least useful subsets of .NET. This allows you to use the same programming language (C#) and the same development environment (Visual Studio) when creating programs for small embedded devices, smartphones, PCs, enterprise servers, and even cloud services.

So what should you know in order to get started? This book gives one possible answer to this question. It is a *Getting Started* book, so it is neither an extensive collection of recipes (or design patterns for that matter), nor a reference manual, nor a textbook that compares different approaches, use cases, etc. Instead, its approach is "less is more," helping you to start writing Internet of Things applications with minimal hassle.

The Platforms

The *.NET Micro Framework* (NETMF) provides Internet connectivity, is simple and open source (Apache license), has hardware available from several vendors, and benefits from the huge .NET ecosystem and available know-how. Also, you can choose between Visual Studio (including the free Express Edition) on Windows, and the open source Mono toolchain on Linux and Mac OS X.

There is an active community for NETMF at *http://www.netmf.com/Home.aspx*. The project itself is hosted at *http://netmf.codeplex.com/*.

Netduino Plus (*http://www.netduino.com/netduinoplus*) is an inexpensive NETMF board from *Secret Labs* (*http://www.secretlabs.com*). This board makes Ethernet networking available with a price tag of less than $60. It has the following characteristics:

- A 48 MHz Atmel SAM7 microcontroller with 128 KB RAM and 512 KB Flash memory
- USB, Ethernet, and 20 digital I/O pins (six of which can be configured optionally for analog input)
- Micro SD card support
- Onboard LED and pushbutton
- Form factor of the Arduino (*http://www.arduino.cc/*); many Arduino *shields* (add-on boards) can be used
- .NET Micro Framework preprogrammed into Flash memory
- All software and hardware is open source

There is an active community for the Netduino Plus (and NETMF) at *http://forums.netduino.com/*. All the examples in this book use the Netduino Plus.

How This Book Is Organized

The book consists of three parts:

- Part I, Introduction

  The first part tells you how to set up the development environment and write and run a "Hello World" program. It shows how to write to output ports (for triggering so-called *actuators* such as LED lights or motors) and how to read from input ports (for *sensors*). It then introduces the most essential concepts of the Internet of Things: HTTP and the division of labor between clients and servers. In the Internet of Things, devices are programmed as clients if you want them to push sensor data to some service; they are programmed as servers if you want to enable remote control of the device over the Web.

» Part II, Device as HTTP Client

The second part focuses on examples that send HTTP requests to some services—e.g., to push new sensor measurements to the Pachube service (*http://www.pachube.com*) for storage and presentation.

» Part III, Device as HTTP Server

The third part focuses on examples that handle incoming HTTP requests. Such a request may return a fresh measurement from a sensor, or may trigger an actuator. A suitable server-side library is provided in order to make it easier than ever to program a small device as a server.

» Appendix A, Test Server

This contains a simple test server that comes in handy for testing and debugging client programs.

» Appendix B, .NET Classes Used in the Examples

This shows the .NET classes that are needed to implement all examples, and the namespaces and assemblies that contain them.

» Appendix C, Gsiot.Server Library

This summarizes the interface of the helper library `Gsiot.Server` that we use in Part III.

Who This Book Is For

This book is intended for anyone with at least basic programming skills in an object-oriented language, as well as an interest in sensors, micro-controllers, and web technologies. The book's target audience consists of the following groups:

» Artists and designers

You need a prototyping platform that supports Internet connectivity, either to create applications made up of multiple communicating devices, or to integrate the World Wide Web into a project in some way. You want to turn your ideas into reality quickly, and you value tools that help you get the job done. Perhaps you have experience with the popular 8-bit Arduino platform (*http://www.arduino.cc/*), and might even be able to reuse some of your add-on hardware (such as shields and *breakout boards*) originally designed for Arduino.

» Students and hobbyists

  You want your programs to interact with the physical world, using mainstream tools. You are interested in development boards, such as the Netduino Plus, that do not cost an arm and a leg.

» Software developers or their managers

  You need to integrate embedded devices with web services and want to learn the basics quickly. You want to build up an intuition that ranges from overall system architecture to real code. Depending on your prior platform investments, you may be able to use the examples in this book as a starting point for feasibility studies, prototyping, or product development. If you already know .NET, C#, and Visual Studio, you can use the same programming language and tools that you are already familiar with, including the Visual Studio debugger.

  To remain flexible, you want to choose between different boards from different vendors, allowing you to move from inexpensive prototypes to final products without having to change the software platform. To further increase vendor independence, you probably want to use open source platforms, both for hardware and software. To minimize costs, you are interested in a platform that does not require the payment of target royalties, i.e., per-device license costs.

If your background is in the programming of PCs or even more powerful computers, a fair warning: embedded programming for low-cost devices means working with very limited resources. This is in shocking contrast with the World Wide Web, where technologies usually seem to be created with utmost inefficiency as a goal. Embedded programming requires more careful consideration of how resources are used than what is needed for PCs or servers. Embedded platforms only provide small subsets of the functionality of their larger cousins, which may require some inventiveness and work where a desired feature is not available directly. This can be painful if you feel at home with "the more, the better," but it will be fun and rewarding if you see the allure of "small is beautiful."

What You Need to Get Started

This book focuses on the interaction between embedded devices and other computers on the Internet, using standard web protocols. Its examples mostly use basic sensors and actuators, so it is unnecessary to buy much additional hardware besides an inexpensive computer board. Here is a list of things you need to run all the examples in this book:

» A Netduino Plus board (*http://www.netduino.com/netduinoplus*)

» A micro USB cable (normal male USB-A plug on PC side, male micro USB-B plug on Netduino Plus side), to be used during development and for supplying power

» An Ethernet router with one Ethernet port available for your Netduino Plus

» An Internet connection to your Ethernet router

» An Ethernet cable for the communication between Netduino Plus and the Ethernet router

» A potentiometer with a resistance of about 100 kilohm and through-hole connectors

» A Windows XP/Vista/7 PC, 32 bit or 64 bit, for the free Visual Studio Express 2010 development environment (alternatively, you may use Windows in a virtual machine on Mac OS X or Linux, or you may use the Mono toolchain on Linux or Mac OS X)

NOTE: There are several sources where you can buy the hardware components mentioned above, assuming you already have a router with an Internet connection:

» Maker SHED (*http://www.makershed.com/*)

  » Netduino Plus, part number MKND02
  » Potentiometer, part number JM2118791

» SparkFun (*http://www.sparkfun.com/*)

  » Netduino Plus, part number DEV-10186

- Micro USB cable, part number CAB-10215 (included with Netduinos for a limited time)
- Ethernet cable, part number CAB-08916
- Potentiometer, part number COM-09806

For more sources in the U.S. and in other world regions, please see *http://www.netduino.com/buy/?pn=netduinoplus*.

It is also possible to add further sensors and actuators.

Conventions Used in This Book

The following typographical conventions are used in this book:

- *Italic*

  Indicates new terms, URLs, email addresses, filenames, and file extensions.

- `Constant width`

  Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, statements, and keywords.

- ***`Constant width bold`***

  Shows commands or other text that should be typed literally by the user.

- *`Constant width italic`*

  Shows text that should be replaced with user-supplied values or by values determined by context.

NOTE: This style signifies a tip, suggestion, or general note.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Getting Started with the Internet of Things*, by Cuno Pfister. Copyright 2011 Cuno Pfister, 978-1-4493-9357-1."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/0636920013037

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://oreilly.com

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

My thanks go to Brian Jepson, Mike Loukides, and Jon Udell, who made it possible to develop this mere idea into an O'Reilly book. It was courageous of them to take on a book that uses a little-known software platform, bets on a hardware platform not in existence at that time, and addresses a field that is only now emerging. Brian not only edited and contributed to the text, he also tried out all examples and worked hard on making it possible to use Mac OS X and Linux as development platforms.

I would like to thank my colleagues at Oberon microsystems for their support during the gestation of this book. Marc Frei and Thomas Amberg particularly deserve credit for helping me with many discussions, feedback, and useful code snippets. Their experience was invaluable, and I greatly enjoyed learning from them. Marc's deep understanding of REST architecture principles and its implementation for small devices was crucial to me, as was Thomas's insistence on "keeping it simple" and his enthusiasm for maker communities like those of Arduino and Netduino. Both showed amazing patience whenever I misused them as sounding boards and guinea pigs. I could always rely on Beat Heeb for hardware and firmware questions, thanks to his incredible engineering know-how, including his experience porting the .NET Micro Framework to several different processor architectures.

Corey Kosak's feedback made me change the book's structure massively when most of it was already out as a Rough Cut. This was painful, but the book's quality benefited greatly as a result.

I have profited from additional feedback by the following people: Chris Walker, Ben Pirt, Clemens Szyperski, Colin Miller, and Szymon Kobalczyk. I am profoundly grateful because their suggestions definitely improved the book.

The book wouldn't have been possible without the Netduino Plus, and Chris Walker's help in the early days when there were only a handful of prototype boards. Whenever I had a problem, he responded quickly, competently, and constructively. I have no idea when he finds time to sleep.

Last but not least, many thanks go to the team at Microsoft—in particular Lorenzo Tessiore and Colin Miller—for creating the .NET Micro Framework in the first place. Their sheer tenacity to carry on over the years is admirable, especially that they succeeded in turning the platform into a true open source product with no strings attached.

12/Handling Actuator Requests

To change the state of a resource, a web client can send PUT requests. A PUT request contains a representation of the desired new state of the resource. In this chapter's example, an LED's state (on/off) is controlled through a web service, as illustrated in Figure 12-1.

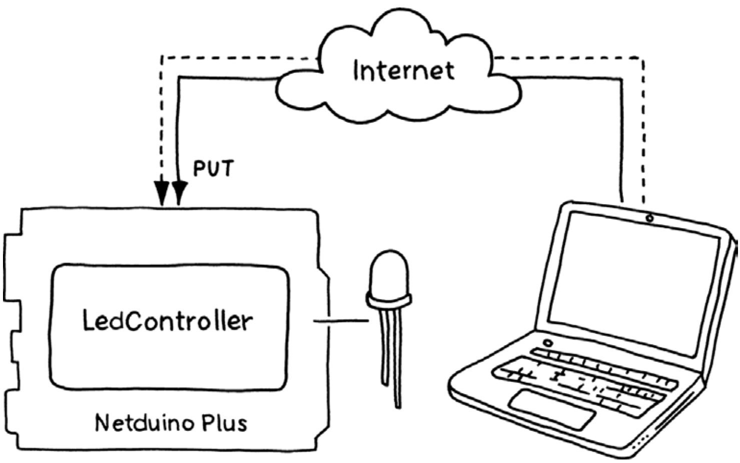


Figure 12-1. Architecture of LedController

`LedController` shows how to handle PUT requests; thus, it is a server program. Unfortunately, you cannot directly use a web browser as a client for sending PUT requests because web browsers are focused on GET requests. Later in this chapter you will see how you can write your own client program (in both C# and JavaScript versions) for testing the server.

NOTE: If you don't mind learning your way around tools like cURL (*http://curl.haxx.se/docs/*) or the Poster add-on for Firefox (*https://addons.mozilla.org/en-US/firefox/addon/poster/*), you can initiate PUT requests with these as well.

For example, with the cURL command-line utility—which is usually installed by default on Mac OS X and Linux—you could use a command like this to turn the LED on (be sure to change the URI to match your configuration):

```
curl -X PUT -d true \

  http://try.yaler.net/gsiot-FFMQ-TTD5/led/target
```

From HTTP Resources to Controlling Things

The resource managed in this example has the meaning "desired state of the LED on the board." Such a resource that accepts *target values* (or *setpoints*) is called a *manipulated variable*. When a server receives a PUT request for a manipulated variable resource, it takes the setpoint value contained in the request message body and feeds it to an actuator. In this example, the actuator is simply an LED.

A server that supports a manipulated variable may or may not support GET requests, in addition to PUT requests, for this resource. A GET request may simply return the most recent PUT value.

URIs of Manipulated Variables

By convention, manipulated variables in this book are called `/name/target`; in this case `/led/target`:

```
http://192.168.5.100/led/target
```

In more complicated applications than this example here, it may not be certain that putting a target value will really have the desired physical effect. For example, if you send a PUT request to a manipulated variable for a valve, with "closed" as the desired state, there may be mechanical reasons why this desired state is not achieved (e.g., the valve may have become mechanically blocked). In such situations, it might make sense to additionally provide a measured variable (sensor) for the valve. This would result in two separate resources: one for the actuator and one for the sensor:

```
http://192.168.5.100/valve/target
http://192.168.5.100/valve/actual
```

The distinction between these two resources reflects the physical reality of a device that has both a sensor (producing the actual value) and an actuator (changing state based on the target value). You may also provide a more abstract combined resource. For example, "state of the fountain in my garden" returns the actual value of the fountain's valve in response to a GET request, and accepts a target value for the valve as part of a PUT request:

```
http://192.168.5.100/fountain-state
```

You can play with the resources until you find the most suitable design for your application. People like different ways to "see" into a system. For example, your parents may only be interested in a temperature given in degrees Celsius, whereas you may be interested in the raw values returned by the sensor—especially if your parents complain that the temperature values cannot be correct. Maybe the sensor is defective, or the algorithm that translates raw sensor values to human-readable engineering units is buggy. Then, it helps to provide both the raw value and the processed value as resources.

LedController

The structure of `LedController` (Example 12-1) is very similar to that of Example 11-1, `VoltageMonitor`.

Example 12-1. LedController

```
using Gsiot.Server;
using SecretLabs.NETMF.Hardware.NetduinoPlus;

public class LedController
{
    public static void Main()
    {
        var ledActuator = new DigitalActuator
```

```
            {
                OutputPin = Pins.ONBOARD_LED
            };

            var webServer = new HttpServer
            {
                RelayDomain = "gsiot-FFMQ-TTD5",
                RelaySecretKey =
                    "o5fIIZS5tpD2A4Zp87CoKNUsSpIEJZrV5rNjpg89",
                RequestRouting =
                {
                    {
                        "PUT /led/target",
                        new ManipulatedVariable
                        {
                            FromHttpRequest =
                                CSharpRepresentation.TryDeserializeBool,
                            ToActuator = ledActuator.HandlePut
                        }.HandleRequest
                    }
                }
            };

            webServer.Run();
        }
    }
```

The main differences between the two examples are that Example 12-1 uses an instance of `DigitalActuator` (`ledActuator`) instead of `AnalogSensor`, and an instance of `ManipulatedVariable` (created using C#'s initializer syntax that was explained in Chapter 10) instead of `MeasuredVariable`.

A `ManipulatedVariable` instance has a delegate property `FromHttpRequest` for the conversion from an HTTP message body to a setpoint object, and a `ToActuator` delegate property for applying the setpoint to an actuator.

`FromHttpRequest` must be compatible with this delegate type:

```
    delegate bool Deserializer(RequestHandlerContext context,
        out object content);
```

and `ToActuator` must be compatible with this delegate type:

```
delegate void PutHandler(object o);
```

Library method `CSharpRepresentation.TryDeserializeBool` is compatible with `Deserializer`, so it can be assigned to `ToHttpResponse`. The method `ledActuator.HandlePut` is compatible with `PutHandler`, so it can be assigned to `ToActuator`.

Inside Gsiot.Server's DigitalActuator Class

The library class `DigitalActuator` is implemented in namespace `Gsiot.Server`, as shown in Example 12-2.

Example 12-2. DigitalActuator

```
public class DigitalActuator
{
    public Cpu.Pin OutputPin { get; set; }

    OutputPort port;

    public void Open()
    {
        port = new OutputPort(OutputPin, false);
    }

    public void HandlePut(object setpoint)
    {
        if (port == null) { Open(); }
        port.Write((bool)setpoint);
    }
}
```

The purpose of this class is to provide a common interface for actuators—namely, a method that consumes new setpoints and is compatible with the delegate type `PutHandler`, and with a "declarative" initialization mechanism like the one of `HttpServer`.

C#: Protecting You from Dangerous Conversions

A variable declared with type `object` accepts anything assigned to it. It is often used in libraries, which should be independent of the exact types that will occur in the various applications that use those libraries. In our case, it is the `Gsiot.Server` library and the `setpoint` parameter of `HandlePut`.

If you know that at some point in your program, a variable of type `object` must contain a value of a particular type, you can cast it safely in the following way:

```
object setpoint = false;        // setpoint now contains a bool value
                                // bool ledSetpoint = (bool)setpoint;
                                // setpoint interpreted as bool value
```

Unlike some other languages, C# will never allow you to proceed with an erroneous type cast on objects. Such type casts will either generate error messages at compile time or exceptions at runtime. In the above example, the check is performed at runtime because the compiler has no way of knowing what you might assign to `ledSetpoint`. By contrast, the following code results in an error at *compile time*:

```
object setpoint = false;        // setpoint now contains a bool value

int boilerSetpoint = setpoint;  // illegal, flagged by compiler
```

The following code results in an exception at *runtime*:

```
object setpoint = false;        // setpoint now contains a bool value

int boilerSetpoint = (int)setpoint;     // throws an exception!
```

As a friend likes to say: every beer bottle is a bottle, but not every bottle is a beer bottle. Similarly, every boiler setpoint is a setpoint (which is an object in turn), but not every setpoint is a boiler setpoint. The C# type system helps to catch many programming mistakes either at compile time or at runtime—and the earlier, the better.

Inside Gsiot.Server's ManipulatedVariable Class

The library class `ManipulatedVariable` is implemented in namespace `Gsiot.Server`, as shown in Example 12-3.

Example 12-3. ManipulatedVariable

```
public class ManipulatedVariable
{
    public Deserializer FromHttpRequest { get; set; }
    public PutHandler ToActuator { get; set; }

    public void HandleRequest(RequestHandlerContext context)
    {
        object setpoint;
        if (FromHttpRequest(context, out setpoint))
        {
            // setpoint may be null
            ToActuator(setpoint);
            context.ResponseStatusCode = 200;   // OK
        }
        else
        {
            context.ResponseStatusCode = 400;   // Bad Request
        }
    }
}
```

The purpose of this request handler for manipulated variables is to separate the request processing from the representation used in the request (`FromHttpRequest`) and from the way new setpoints are consumed (`ToActuator`).

Test Client in C#

To test your `LedController` server with a client that runs on a computer, use the test client given in Example 12-4, which sends a PUT request to the server. You need to adapt the constant `uri` to the address of your device.

The representation sent to the server is contained in constant `message`. See what happens if you send the value as given below, or if you change it to `false` or some unsupported value.

NOTE: This code won't run on a Netduino Plus. You'll have to run it on Windows using .NET, or on Mac OS X or Linux using Mono. Mono is an open source implementation of .NET that runs on several platforms.

Example 12-4. LedControllerClient test client in C#

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading;

public class LedControllerClient
{
    public static void Main()
    {
        const string method = "PUT";
        const string uri =
            "http://try.yaler.net/gsiot-FFMQ-TTD5/led/target";
        const string type = "text/plain";
        const string message = "true";   // ignored for GET requests

        HttpWebRequest request = CreateRequest(method, uri, type,
            message);
        try
        {
            using (var response = (HttpWebResponse)request.
                GetResponse())
            {
                LogResponse(response);
            }
        }
        catch (Exception e)
        {
            Console.Write(e.ToString());
            Thread.Sleep(Timeout.Infinite);
        }
    }

    static HttpWebRequest CreateRequest(string method,
        string uri, string type, string body)
    {
        var request = (HttpWebRequest)WebRequest.Create(uri);

        // request line
        request.Method = method;
```

```
        if ((body != null) && (method != "GET"))
        {
            byte[] buffer = Encoding.UTF8.GetBytes(body);

            // request headers
            request.ContentType = type;
            request.ContentLength = buffer.Length;

            // request body
            using (Stream stream = request.GetRequestStream())
            {
                stream.Write(buffer, 0, buffer.Length);
            }
        }
        return request;
    }

    static void LogResponse(HttpWebResponse response)
    {
        // response status line
        Console.WriteLine("HTTP/" + response.ProtocolVersion + " " +
            response.StatusDescription);

        // response headers
        string[] headers = response.Headers.AllKeys;
        foreach (string name in headers)
        {
            Console.WriteLine(name + ": " + response.Headers[name]);
        }

        // response body
        var buffer = new byte[response.ContentLength];
        Stream stream = response.GetResponseStream();
        int toRead = buffer.Length;
        while (toRead > 0)
        {
            // already read: buffer.Length - toRead
            int read = stream.Read(buffer, buffer.Length - toRead,
                toRead);
            toRead = toRead - read;
        }
```

```
            char[] chars = Encoding.UTF8.GetChars(buffer);
            Console.WriteLine(new string(chars));
            Thread.Sleep(Timeout.Infinite);
        }
    }
```

The test client writes the server's response to a console window and then waits for you to press Ctrl-C to quit it.

Embed a JavaScript Test Client on the Netduino

Web browsers are convenient HTTP clients because they are available on practically any platform, and also because they can download new programs (*scripts*) without extra installation hassles. The trick is that script code can be embedded in HTML pages, so ordinary HTTP GET requests are sufficient as download mechanisms for JavaScript programs. Since JavaScript can issue PUT requests, you can click buttons on a web page to turn your LEDs on and off!

And since the Netduino Plus is functioning as a web server, you can serve this JavaScript directly from your .NET Micro Framework code!

To include some JavaScript in an HTML document, add a `<script>` XML element with the code shown in Example 12-5. (Example 12-6 shows the complete example.)

Example 12-5. LedController test client in JavaScript, embedded in HTML

```
<html>
  <head>
    <script type="text/javascript">
      var r;

      try {
        r = new XMLHttpRequest();
      } catch (e) {
        r = new ActiveXObject('Microsoft.XMLHTTP');
      }
```

```
      function put (content) {
        r.open('PUT', '/gsiot-FFMQ-TTD5/led/target');
        r.setRequestHeader("Content-Type", "text/plain");
        r.send(content);
      }
    </script>
  </head>
  <body>
    <p>
      <input type="button" value="Switch LED on"
        onclick="put('true')"/>
      <input type="button" value="Switch LED off"
        onclick="put('false')"/>
      <input type="button" value="Bah" onclick="put('bah')"/>
    </p>
  </body>
</html>
```

This script creates a new `XMLHttpRequest` object `r` (short for "request") or an equivalent *ActiveX* object for Internet Explorer 6 or newer. This object has a method `open` that takes the HTTP method and the request URI as parameters, and a method `setRequestHeader` for adding request headers. It also has a method `send`, which sends the HTTP request back to your server (your Netduino Plus).

NOTE: `XMLHttpRequest` can send any kind of representation, not just XML as its name suggests.

The object `r` is used in the function `put`, which takes the request message content as a parameter and sends it back in an HTTP PUT message to the same server from which the JavaScript came.

The body of the HTML page produces three buttons: Switch LED on, Switch LED off, and Bah. When you click on them, they call the `put` function with the arguments `"true"`, `"false"`, or `"bah"`. In the first case, the request is meant to switch on the Netduino Plus's onboard LED. In the second case, the request is meant to switch off the Netduino Plus's onboard LED. In the third case, the request is meant to provoke an error situation (see the debug console for what happens when you click on it).

The resulting web page looks like Figure 12-2.

Figure 12-2. Simple web page for controlling an LED

The entire program is given in Example 12-6, which encodes the script from Example 12-5 in one large string. Instead of loading the HTML from a file, the Netduino Plus will serve it up out of its memory in the body of the `HandleLedTargetHtml` handler.

Example 12-6. LedControllerHtml with embedded JavaScript

```
using Gsiot.Server;
using SecretLabs.NETMF.Hardware.NetduinoPlus;

public class LedControllerHtml
{
    public static void Main()
    {
        var ledActuator = new DigitalActuator
        {
            OutputPin = Pins.ONBOARD_LED
        };

        var webServer = new HttpServer
        {
            RelayDomain = "gsiot-FFMQ-TTD5",
            RelaySecretKey =
                "o5fIIZS5tpD2A4Zp87CoKNUsSpIEJZrV5rNjpg89",
            RequestRouting =
            {
                {
                    "PUT /led/target",
                    new ManipulatedVariable
                    {
                        FromHttpRequest =
                            CSharpRepresentation.TryDeserializeBool,
                        ToActuator = ledActuator.HandlePut
                    }.HandleRequest
                },
                {
                    "GET /led/target.html",
```

```
                    HandleLedTargetHtml
                }
            }
        };

        webServer.Run();
    }

    static void HandleLedTargetHtml(RequestHandlerContext context)
    {
        string requestUri = context.BuildRequestUri("/led/target");
        var script =
            @"<html>
                <head>
                  <script type=""text/javascript"">
                    var r;
                    try {
                      r = new XMLHttpRequest();
                    } catch (e) {
                      r = new ActiveXObject('Microsoft.XMLHTTP');
                    }
                    function put (content) {
                      r.open('PUT', '" + requestUri + @"');
                      r.setRequestHeader(""Content-Type"",
                        ""text/plain"");
                      r.send(content);
                    }
                  </script>
                </head>
                <body>
                  <p>
                    <input type=""button"" value=""Switch LED on""
                      onclick=""put('true')""/>
                    <input type=""button"" value=""Switch LED off""
                      onclick=""put('false')""/>
                    <input type=""button"" value=""Bah""
                      onclick=""put('bah')""/>
                  </p>
                </body>
             </html>";
        context.SetResponse(script, "text/html");
    }
}
```

In this example, note that two resources are supported:

```
{
    "PUT /led/target",
    … request handler …
},
{
    "GET /led/target.html",
    … request handler …
}
```

Another noteworthy aspect of the example is the use of *verbatim strings*. A verbatim string starts with an `@` sign and is followed by a `"` character. It ends at the first `"` character that isn't doubled. To allow `"` characters in a verbatim string, two subsequent `"` characters are interpreted as a single `"` character. In a verbatim string there may be carriage returns, line feeds, tabulator characters, etc., that don't need an escape sequence like normal strings. This can make verbatim strings more readable in some cases. Here is an example of a verbatim string:

```
string s = @"Hello ""World"" again";
```

It is equivalent to this regular string:

```
string s = "Hello \"World\" again";
```

What You Should Know About HTTP PUT

To change the state of a device's actuator, you send it HTTP PUT messages. Like GET, PUT is defined as being *idempotent*, meaning that issuing the same PUT request multiple times has the same effect on the server's resources as issuing it only once—assuming no one else changes the same resource. This is particularly relevant in one situation: suppose your client program has sent a PUT request, but it does not get back a response. After a while, the client will time out. What should happen then? If the request had been lost on its way to the server, your client could simply try again and send the PUT request a second time.

But what if the request had been received by the server, was processed correctly, and only the response message got lost somewhere on the way back to your client? Sending the PUT request again would cause the resource to be manipulated a second time. What could be a huge problem is no problem at all if you design your PUT request handlers to be idempotent, in which case simply sending the same PUT request again is harmless. This is the beauty of RESTful web services with HTTP. Distributed systems, where clients and servers operate on different machines and are connected through sometimes-unreliable connections, are notoriously difficult to program correctly. The reason is that unlike single programs on single computers, distributed systems suffer from *partial failures*: one component dies, but the other components continue without knowing what exactly happened. This makes it nearly impossible to recover from failures in such a way that all components are guaranteed to have consistent states again.

The idempotent way in which HTTP GET and PUT (and DELETE) are defined reduces this problem enormously: if a client suspects a problem with a request, it simply repeats it. It doesn't need to find out the current resource state of the server, and it doesn't need to correct it. On the other hand, a server simply responds to a request it receives from a client, and then forgets about this client. It doesn't need to keep track of the client's application state. Whether a client really receives a response message or has died, or whether the message was lost somewhere on the network, need not concern the server. This decoupling of the clients' application states and the servers' resource states is sometimes called *statelessness*.

In practice, this means that almost anything can be a resource—except commands. For example, if you control a loudspeaker's volume with an HTTP server, you can send it a PUT request with a representation of the desired state, e.g., `"70%"`. This is idempotent. You can send it as often as you want; seventy percent remains seventy percent. By contrast, commands are not always idempotent; e.g., `"increase volume by one notch"` would not be idempotent. Often, a URI name that contains a verb betrays such a mistake, e.g., `/loudspeaker/increaseVolume`.

Multithreading

Buffer

An instance of class `Buffer` provides a threadsafe way of communication between actors (see Chapter 13). A buffer instance basically acts as a variable whose current value can be read and written:

```
public sealed class Buffer
{
    public void HandlePut(object o);
    public object HandleGet();
}
```

» `void HandlePut(object o)`

This method puts `o` into the buffer. The new value in the buffer replaces the old one. At most one value is buffered; there is no queuing of multiple values. The method performs the necessary locking to enable safe use of the buffer from multiple threads. Object `o` may be `null`.

» `void HandleGet()`

This method gets the current buffer state, *without* changing it. The method performs the necessary locking to enable safe use of the buffer from multiple threads. The result may be `null`.

Index

Symbols

D

E

F

G

H

I

J

L

M

N

Q

R

S

T

U

V

W

X

Y

About the Author

Dr. Cuno Pfister studied computer science at the Swiss Federal Institute of Technology in Zürich (ETH Zürich). His PhD thesis supervisor was Prof. Niklaus Wirth, the designer of the Pascal, Modula-2, and Oberon programming languages. Dr. Pfister is the Managing Director of Oberon microsystems, Inc., which has worked on various projects related to the Internet of Things, from mobile solutions to a large hydropower-plant monitoring system with 10,000 sensors.

Colophon

The cover, heading, and body font is BentonSans, and the code font is Bitstreams Vera Sans Mono.