

HOL P200 - Passing Data Between Padarn Pages

Table of Contents

Table of Contents	2
HOL Requirements	3
Summary.....	4
Lab Objective	4
Pre-Requisites.....	4
Introduction to HTTP Web Methods	4
GET and PUT In Detail.....	5
Working with Form Data in Padarn	7
Exercise 1: Adding the FormExample ASPX.....	8
Exercise 2: Stubbing out the Code-Behind Classes.....	10
Exercise 3: The static code-behind logic for FormExample page	11
Exercise 4: The dynamic code-behind logic for FormExample page	13
Exercise 5: Walking through the FormExample page.....	15
Hands-on Lab Summary	18

HOL Requirements

The following items are required to run this HOL:

- A Desktop PC running Microsoft Windows® XP or Windows Vista
- Microsoft Visual Studio 2008 Professional (or higher)
- A Web Browser
- A Padarn reference system or developer kit

While Padarn will run on almost any hardware that supports the Microsoft .NET Compact Framework 2.0 or higher, this lab assumes that you have one of the Padarn reference hardware platforms with an OpenNETCF-validated Windows CE image running on it. If you are using an alternate hardware or software configuration, the steps outlined in this Hands-on Lab may not be accurate for your environment.

Summary

In this lab, you will learn how to exchange data between pages of a Padarn-based web application using various classes and properties exposed by the OpenNETCF.Web namespace.

Lab Objective

Upon completion of this lab, you will be able to create a web front end for users to input data and the corresponding back end to retrieve values passed in and then perform basic processing on those values.

In this HOL, you will perform the following exercises:

- ✓ Create a simple web form using traditional HTML control components
- ✓ Establish the code-behind to process inputted values and display output HTML

Pre-Requisites

Prior to walking through this HOL, it is important that you are comfortable with the overall architecture of Padarn as well as have studied the HOL P1XX Series, which shed some light on configuration of Padarn and debugging best practices.

Introduction to HTTP Web Methods

Exchanging data between client and data is a fairly open ended problem that has had many historical approaches and even more options today. Data can be exchanged over anything from basic sockets to RPC (see HOLs P201 and P202), or even web services encoded using SOAP, JSON, or REST. The HTTP protocol provides clear direction for utilizing HTML language to facilitate a basic data exchange.

In this lab, we will continue our discussion from the RPC HOL set and explore the GET and POST methods in a bit more detail. We will make frequent references to these HOLs since they provide a good working example of these methods in action. Before continuing, it would make sense to define a key term used when discussing HTTP methods: *idempotence*.

"In computer science, the term idempotent is used to describe method or subroutine calls which can safely be called multiple times, as invoking the procedure a single time or multiple times results in the system maintaining the same state i.e. after the method call all variables have the same value as they did before.

Example: Looking up some customer's name and address are typically idempotent, since the system will not change state based on this. However, placing an order for a car for the customer is not, since running the method/call several times will lead to several orders being

placed, and therefore the state of the system being changed to reflect this."

Wikipedia, [Idempotence \(Computer Science\)](#)

Aside from error events or exceptions that take place, outside the control of the browser or requesting client, there are four web methods defined in the HTTP 1.1 specification that meet this definition: GET, HEAD, PUT, and DELETE. It should be noted that even if several methods are independently idempotent, several grouped together might not be. Such a case would exist if the results depend on a value that is modified at a later point in the same sequence of methods.

HEAD is identical to GET except there must be no message body returned. This is often used when the requester doesn't care about any processing to be done on the contents of the message and instead wishes to ensure the URI is valid, accessible, authorized, etc. This method can also be used to refresh a previous return result if a particular caching policy is in place.

DELETE asks the server to remove whatever resource is located at the URI passed in through the call. Even though the server might return that the command succeeded, there is no guarantee that the result code can be trusted by the client. A properly designed web application will be honest in returning a success code when the deletion can occur immediately at the time of request. A success code here would be 200 (if the `HttpResponse` will include status information), 202 if the request was accepted, or 204 if the request was accepted but the entity body contains no data. Other HTTP methods not covered in depth within this HOL are:

- Options. This method generally acts as a request by the client for information about what kind of communication channel exists to access a particular URI, and this can be done without an actual fetch of the resource at the URI.
- Trace. This method returns to the client the data received in the header during the request. It's generally best for tracing to be disabled at the server level in a production environment, since it is a debugging tool.
- Connect. This method relates to proxy tunneling and is beyond the scope of this article.

GET and PUT In Detail

At its simplest, an HTML form is a document that contains input elements. These input elements might be text areas, text fields (passwords or clear text), drop-down menus, radio buttons, checkboxes, etc. By themselves, these visual components don't perform any action. That's where the submit button comes into play – it allows the values set within the various input components to be submitted over an HTTP channel to the publishing server.

```
<form name="input" action="form_submit.aspx" method="get">  
Entry:  
<input type="text" name="entry" />  
<input type="submit" value="Submit" />  
</form>
```

Notice the form tag introducing. Once this snippet shows up in an HTML document, the value set in the text field of type text and will be passed through to the server with name = "entry", and value =

<User Entry>. Notice we are using the GET method here because we are retrieving from, rather than modifying, data on the server. A submission button will be provided along with the text input.

The HTML 1.1 specification indicates that the GET Method is used to describe when form data is to be encoded into a URL, while the POST Method contains data within the message body entity. There are reasons why there might be some confusion here: if the POST data is small enough, it will probably fit into the URL without error. However, since the server is responsible for decoding HTML character codes (for spaces, special characters, punctuation, etc.), it's advisable to use these two methods according to the definition whenever possible. Parsing a URL, of course, is usually easier but Padarn makes it very easy to read the input elements passed in through a Form. To put into more concrete terms: if the form request will leave no lasting, observable effect on the server, then it should be sent as a GET. Otherwise, the request should be of type PUSH.

Here's another reason why the distinction matters. In a modern web browser, if a user decides to hit the **Refresh** button on a page that is displayed as a result of a form submission, one of two things will happen:

- GET methods will generally refresh without any user interaction
- PUSH methods will generally tell the user that data must be refreshed (giving the ability to halt the transaction)

So in other words, the user might inadvertently cause another data change to take place if he accidentally requests the data a second time. Furthermore, GET requests are often times cached (see HOL 112 - Understanding Padarn's [HttpRequest](#) and [HttpResponse](#) Objects for more details on how caching rules are set), which can result in erroneous results being displayed to the user.

Back to our simple form example from above, let's discuss what happens when the user presses the **Submit** button on this mock form. First, a form data set is created. This data set is then encoded based upon how the web browser interprets the request. The *enctype* attribute is set based on the HTTP Method being utilized. For POST, the value is `multipart/form-data`, whereas `application/x-www-form-urlencoded` is sometimes used for both POST and GET. How does the form data set get transmitted? According to the HTML 4.0 specification:

- If the method is "get", the user agent takes the value of `action`, appends a '?' to it, then appends the form data set, encoded using the `application/x-www-form-urlencoded` content type. The user agent then traverses the link to this URI. In this scenario, form data are restricted to ASCII codes.
- If the `method` is "post", the user agent conducts an HTTP post transaction using the value of the `action` attribute and a message created according to the content type specified by the `enctype` attribute.

When a form is submitted using HTTP GET, the data is transmitted as a URI to the web server and the user will sometimes see the full URI requested in his web browser (`http://<Padarn Server IP>/default.aspx?action=doSomething`). When a form is submitted using HTTP POST, the URL

displayed to the user is the page (<http://<Padarn Server IP>/default.aspx>) rather than the full expression as submitted by the user. We will discuss how either scenario is parsed by Padarn in the examples to follow. Why should a developer shy away from using GET when it's so simple to parse?

- There are limitations on the length of URLs and requests with many fields can become unwieldy
- If form data contains non-ASCII characters, such as symbols or non-English characters, processing will often fail
- Though you can specify to hide fields from showing up in the URL, a user can always check out the source code of the resultant form submission and deduce the values sent

For much more detail on how Forms work in HTML 4.0 as well as what attributes are supported according to the specification, please visit the W3's page [explaining these concepts in detail](#).

Working with Form Data in Padarn

Thankfully, many of the details of the HTTP 1.1 specification for GET and POST Methods are handled on your behalf when using Padarn. `OpenNETCF.Web.HttpRequest` exposes a `Form` object via a getter property. The `Form` property is a `NameValueCollection` that has several conveniences: it strips away certain whitespace, it handles the splitting of URI containing the GET Method form data, and it will even ensure that a cross-scripting attempts aren't being made (by using redirection `'/'` characters, a web server can be tricked into sending the user to a rogue website).

The `Form` property uses strings as index keys and treats all stored values as strings. Casting must be done in the code-behind if the values are to be treated as numerical values.

The flow of a typical ASP.NET Active Server Page (ASP or ASPX extension in the browser address bar) is:

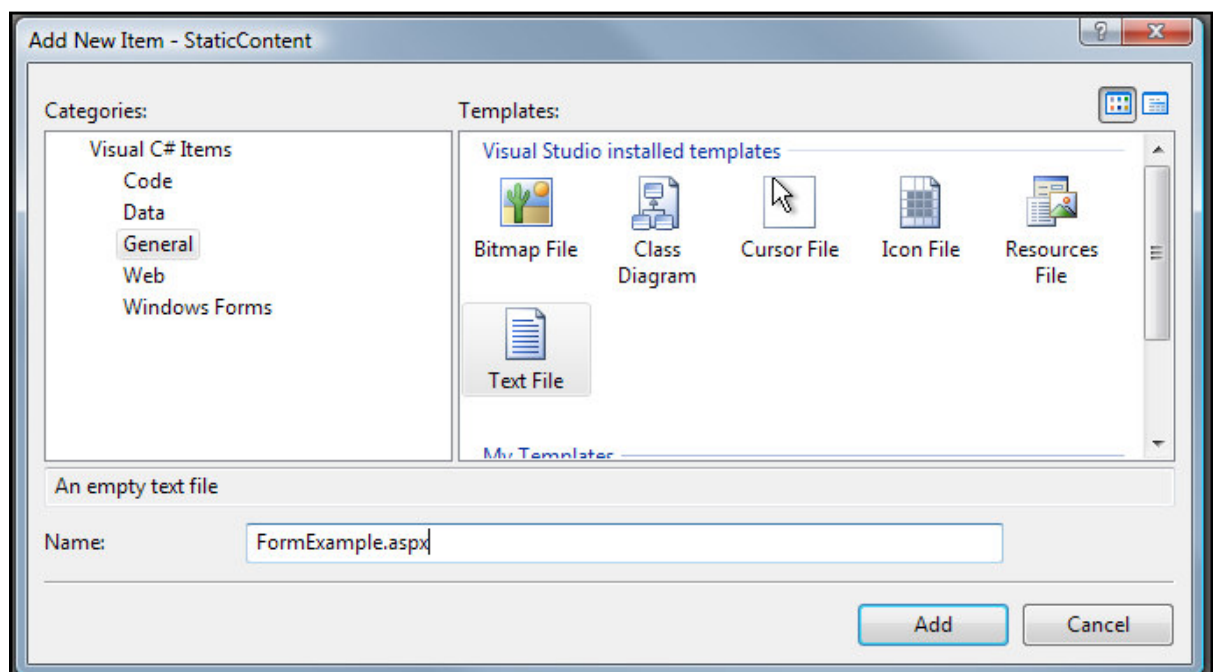
- User requests landing page <http://localhost.com/page.aspx>
- Server received a `Page_Load` function hit. Code behind sees no parameters passed in and the `Request` object is made available through the platform being used. All Padarn HOLs will use C# under .NET 2.0 as the target platform.
- With the `Request` object in hand, it is possible for the code-behind logic to parse those values and present new data to the user via the `Response` object. Both the `Request` and `Response` objects are implemented with easily-interpreted `NameValueCollection`s.

Exercise 1: Adding the FormExample ASPX

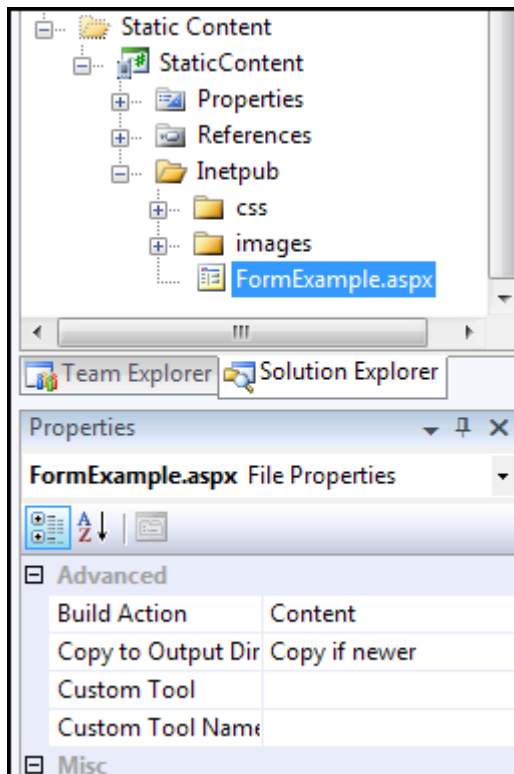
Whenever a user accesses a Padarn webpage, he will be accessing an HTML, ASPX, or other standard front-end ASP.NET-compatible document type. However, Padarn will immediately call into an appropriate code-behind assembly (DLL) that contains page load logic for each page. We've been using this model for each of our previous HOLs, and we'll continue in the same direction here.

To create the ASPX file:

1. In Visual Studio's **Solution Explorer** Pane, right-click on the **Inetpub** folder in the **StaticContent** project and select **Add -> New Item** to display the **Add New Item** dialog.
2. From the list of **Visual Studio Templates**, select **Text File**
3. In the **Name** textbox enter "FormExample.aspx"



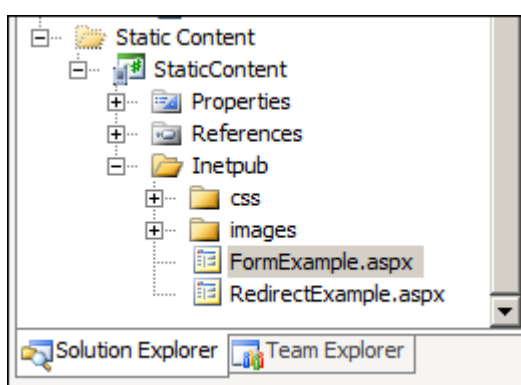
4. Click **Add** to add the new file to the project
5. In Visual Studio's Solution Explorer Pane, click on the newly-created FormExample.aspx file
6. In the Properties pane, set the Build Action for FormExample.aspx to "Content"
7. In the Properties pane, set the Copy to Output Directory property for FormExample.aspx to "Copy if Newer"



8. Paste the following code into the newly-created FormExample.aspx file (this will tell Padarn to look for the code-behind logic in a DLL called SampleSite.dll, using reflection to peer into class SampleSite.FormExample):

```
<%@ Page CodeBehind="SampleSite.dll" Inherits="SampleSite.FormExample" %>
```

9. Repeat the above procedure for an Active Service Page named RedirectExample.aspx. The code behind will live within the SampleSite.dll
10. The solution will now look like this:

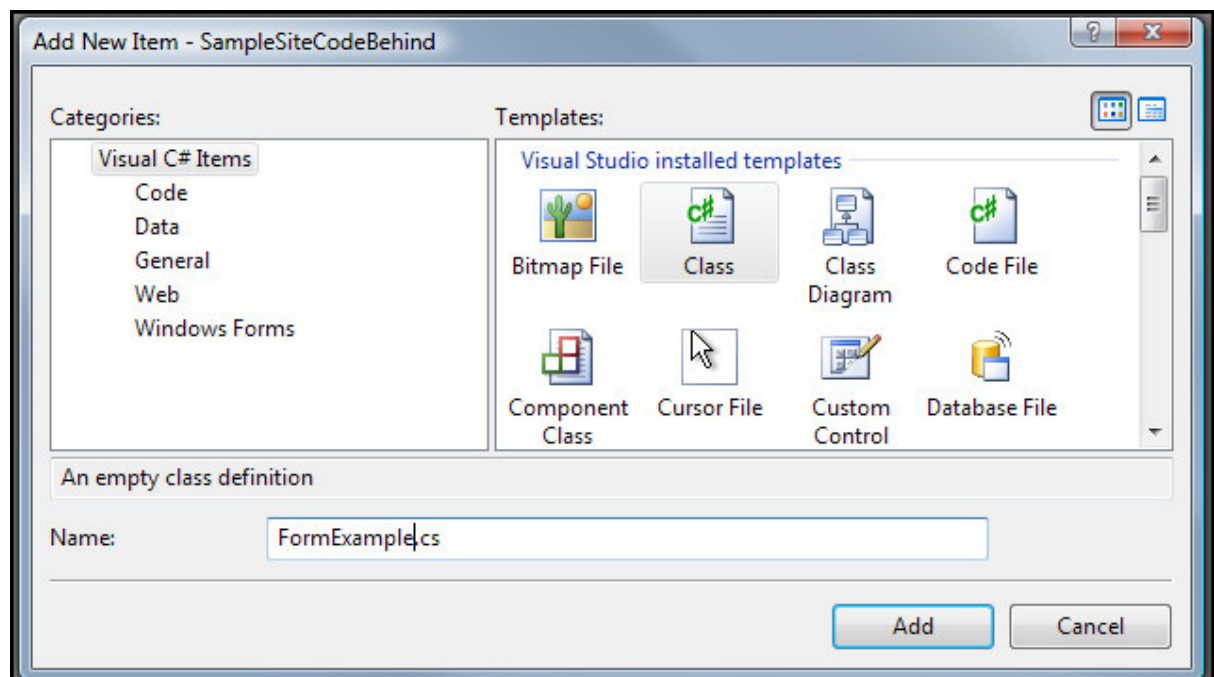


Exercise 2: Stubbing out the Code-Behind Classes

We will be adding to main classes to our code-behind logic layer. Both correspond to the Active Server Pages created above: FormExample.cs and RedirectExample.cs.

To create the main code-behind class:

1. In Visual Studio's Solution Explorer pane, right-click on the SampleSiteCodeBehind project and select **Add -> New Item** to display the Add New Item dialog.
2. From the list of Visual Studio Installed Templates select **Class**.
3. In the Name textbox, enter "FormExample.cs".



4. Click **Add** to add the new file to the project
5. Repeat the process for RedirectExample.cs.

Exercise 3: The static code-behind logic for FormExample page

A key illustration of intra-page data exchange within Padarn can be found in the mere display of a standard HTML input form to a user and processing values once the user submits that form. We will create a form that displays several HTML common elements: a text input box, a radio group (collection of radio buttons), and a checkbox. We'll include submit and reset buttons for good measure.

```
#region Constants

private const string VERB_FUNCTION = "function=";
private const string RADIO_CLASS = "RADIO_CLASS";
private const string TEXT_INPUT = "TEXT_INPUT";
private const string RADIO_INPUT = "RADIO_INPUT";

private const string ACTION = "action";

#endregion
```

We have defined some constants within FormExample.cs to help with the calls within the OpenNETCF.Web.HTML namespace, which has a variety of helpful classes to be used in building HTML documents on the fly.

```
#region Simple Properties

private bool IsHttpPost
{
    get { return (Request.RequestType == "POST"); }
}

private bool URIHasData
{
    get { return !String.IsNullOrEmpty(Request.RawQueryString); }
}

private bool FormHasData
{
    get { return Request.Form.HasKeys(); }
}

}
```

Next, we define some getters that help us understand the incoming request, whether it's a POST or GET Method type, whether the interaction with Padarn is via URI (value inputted to browser address bar), or whether the interaction with Padarn is via a web form.

We now jump into the Page_Load method, which is the first entry point of any code behind logic within Padarn, excluding a custom log routine (see HOL 301 - Implementing a Custom Padarn Log Provider).

```
if (!FormHasData && !URIHasData)
{
    CreateHTMLInput();
}
```

If the user has not inputted any form data or URL entry, we know the user is browsing to <http://<Padarn Server IP>/FormExample.aspx>. The goal here is to return to the user a very simple HTML page that contains our input form. We invoke our `CreateHTMLInput` function.

```
Document page = new Document();
page.Head = new DocumentHead("OpenNETCF Padarn Web Server - Form Test Input",
    new StyleInfo("css/SampleSite.css"));
page.Head.MetaTags.Add(new ContentType(CharSet.UTF8));
```

We will build a new `Document` object and establish the character set for good measure.

```
Form form = new Form("FormExample.aspx", FormMethod.Post);
```

We declare a new `Form` object so that we can associate this HTML document with an HTTP method. In this case, we will use `POST`, since we will set the stage for data processing to be carried out on the inputted values.

```
form.Add(new Input(TEXT_INPUT));
form.Add(new Button(new ButtonInfo(ButtonType.Submit, "Submit")));
form.Add(new Button(new ButtonInfo(ButtonType.Reset, "Reset"));
```

Add a text input box (named "TEXT_INPUT") and two command buttons.

```
RadioGroup rg = new RadioGroup(RADIO_INPUT);
rg.Items.Add(new RadioItem("Value 1", "Val1", true, RADIO_CLASS));
rg.Items.Add(new RadioItem("Value 2", "Val2", false, RADIO_CLASS));
form.Add(rg);
```

Add a radio group (named "RADIO_INPUT") and nest two radio buttons within that group. This is convenient because we can set one radio button to be selected when it is displayed.

```
form.Add(new RawText("Choice selected?:" +
    "<input type=\"checkbox\" name=\"CheckSelected\" value=\"Selected\"\" +
    "/>" +
    "<br />"));
```

We create a checkbox named "CheckSelected".

```
page.Body.Elements.Add(form);
Response.Write(page.OuterHtml);
Response.Flush();
```

Last, we add the form to the `Document` we've created, and then we write out the HTML elements to the `Response` object and perform a `Flush`, which sends the prepared document back to the user.

Exercise 4: The dynamic code-behind logic for FormExample page

The first part of the FormExample code-behind class was intended to provide the user with an HTML input form. The second part does work on the data returned by the user.

```
switch (functionName)
{
    case "submit":
    {
        string[] chunkedValues =
            OpenNETCF.Web.HttpUtility.UrlDecode(Request.RawQueryString).Split('&');
        Dictionary<string, string> parsedValues = new Dictionary<string, string>();

        foreach (string qualifier in chunkedValues)
        {
            string[] pair = qualifier.Split('=');
            if (pair.Length == 2)
            {
                parsedValues.Add(pair[0], pair[1]);
            }
        }

        string action =
            parsedValues.ContainsKey(ACTION) ? parsedValues[ACTION] : String.Empty;

        Response.Redirect("RedirectExample.aspx");

        break;
    }
}
```

Should the user enter FormExample.aspx via a URI (for example, <http://<Padarn Server IP>/FormExample.aspx?submit&action=doSomething>), then we will not have any data to retrieve from the `Request.Form` `NameValueCollection`. Rather, we will only get access to the URI parameters. This requires a bit more work. As you can see above, we'll need to chunk up the `RawQueryString` to understand what the user was trying to accomplish. In most cases, input over URI will be HTTP Get actions. HOL 201 and HOL 202 (RPC Web Services with Padarn) go into more detail about why this access method is advantageous.

This code snippet illustrates a rather useful capability of the `Response` property: it can be used to redirect the user to another page. *Note: the Form collection available to this page will not be passed onto the redirection page!* The code behind logic for `RedirectExample.aspx` does nothing more than present the user with action inputted by the user. See the example solution for details.

```
else if (FormHasData && IsHttpPost)
{
    /* (POST) User is passing in input through browser form - use Form object */

    // 1. Pull from Form
    string textInput = Request.Form[TEXT_INPUT] as string;
    string rdoVal = Request.Form[RADIO_INPUT] as string;
    string checkSelected = Request.Form["CheckSelected"] as string;

    // 2. Perform some server-side validation and processing on inputs

    // 3. Display results to user
    CreateHTML(textInput, rdoVal, checkSelected);
}
```

Generally, though, we'll be expecting incoming data as form data as would be the case with the HTML input form we've created in the previous exercise. The work here is minimal. We simply pull from `Request.Form` and gain immediate access to the inputted values. We're omitting the data manipulation work that would always be associated with an HTTP Post. Of course, you'll want to ensure the inputted values are in a valid format before attempting to use them.

After retrieving the values, we then proceed to display to the user what he entered in the second overloaded `CreateHTML` method.

Exercise 5: Walking through the FormExample page

Let's take a look at the HTML input page being served up through the FormExample code-behind as well as see what gets displays to the user after submitted it. In order to validate that our code-behind class work properly, we need to ensure a valid instance of Padarn is up and running; furthermore, you should validate that FormExample.aspx and RedirectExample.aspx (along with the various CSS and image files) have been deployed as static content to the Padarn host machine. Instructions for how to accomplish this are in HOL 100.

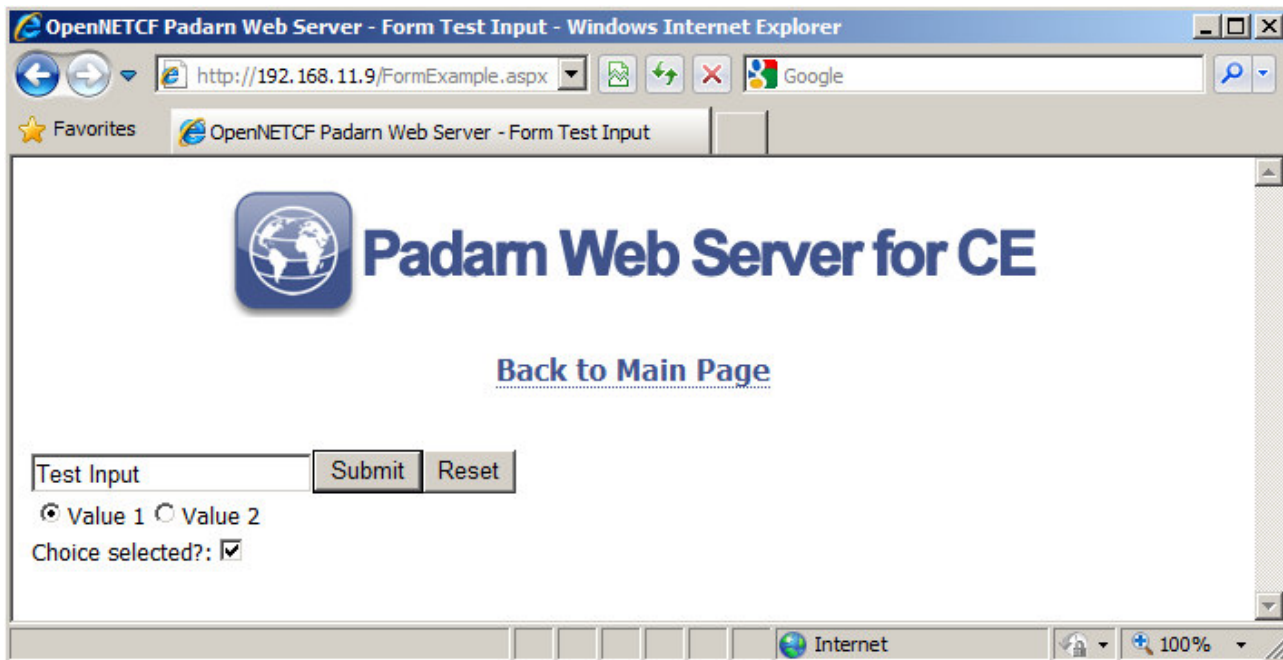
1. Initiate Padarn through the MyPadarnSolution Visual Studio 2008 solution. Right-click on the ochttpd project and select **Debug -> Start New Instance** (or press F5). For this demonstration, we will be using the Windows Mobile 5.0 Pocket PC R2 Emulator.



2. Press the **Start** button to initiate the Padarn instance.



3. From a web browser that can communicate with the Padarn instance (for example, a Windows XP laptop that is host to the Windows Mobile 5.0 emulator). The IP address of the Padarn instance is key for bring up any Padarn-served content. Various utilities exist for Windows CE that provide this information. For example, you could use the [OpenNETCF.Net.NetworkInformation](#) namespace.
4. Visit `http://<Padarn Server IP Address>/FormExample.aspx` in your favorite web browser.



OpenNETCF Padarn Web Server - Form Test Input - Windows Internet Explorer

http://192.168.11.9/FormExample.aspx

OpenNETCF Padarn Web Server - Form Test Input

 **Padarn Web Server for CE**

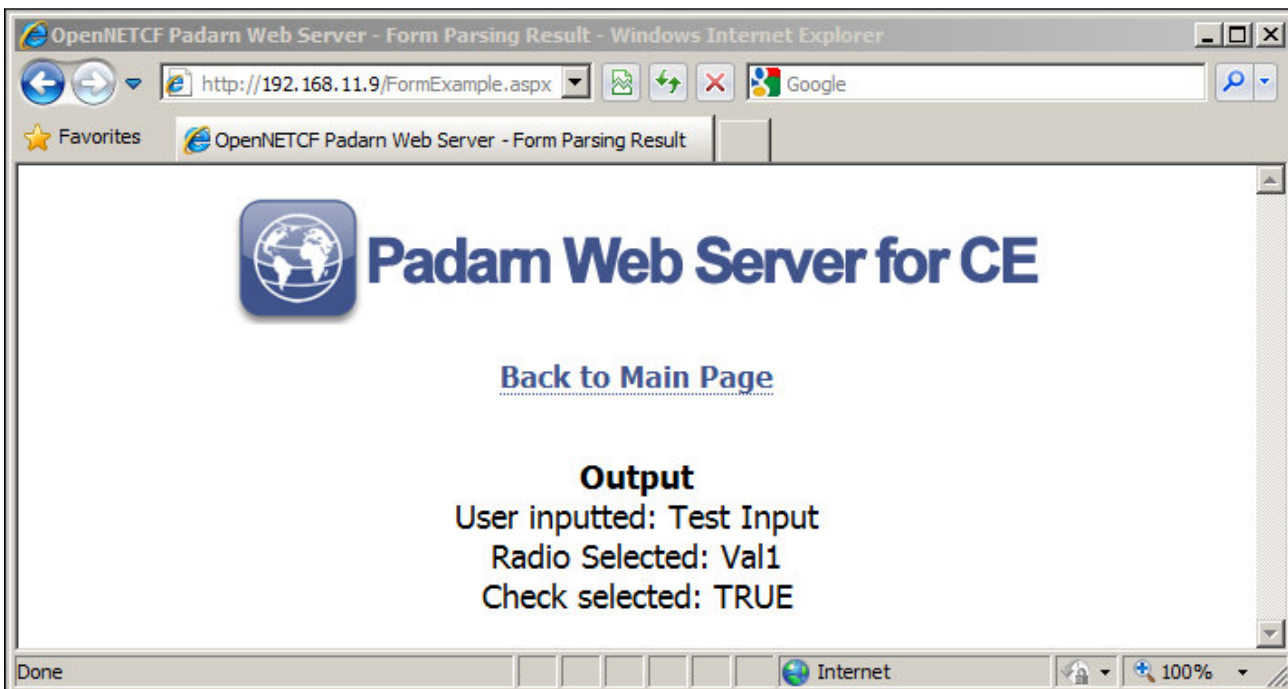
[Back to Main Page](#)

Test Input

☒ Value 1 ☐ Value 2

Choice selected?: ☒


5. Enter "Test Input" into the text box and press the "Submit" button.



OpenNETCF Padarn Web Server - Form Parsing Result - Windows Internet Explorer

http://192.168.11.9/FormExample.aspx

OpenNETCF Padarn Web Server - Form Parsing Result

 **Padarn Web Server for CE**

[Back to Main Page](#)

Output

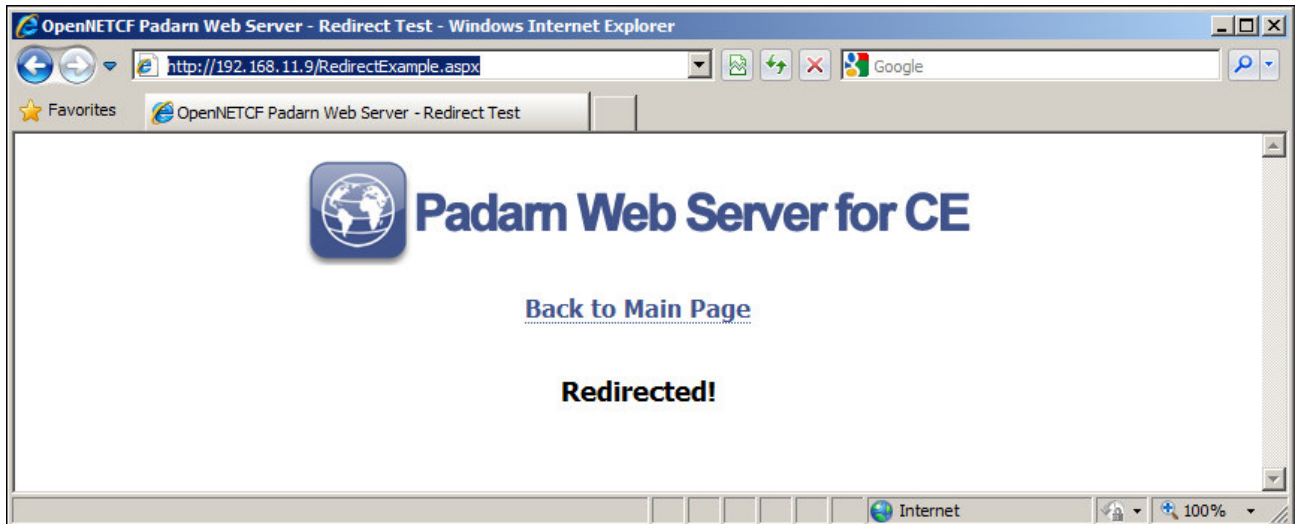
User inputted: Test Input

Radio Selected: Val1

Check selected: TRUE

Once the Page_Load function is called a second time, it will recognize the Request.Form has keys and so will display the input values to the Response.

6. Enter `http://<Padarn Server IP>/FormExample.aspx?submit&action=doSomething` in your web browser address bar.



You will be immediately redirected to RedirectExample.aspx with *no* Request data passed along pertaining to the initial request.

Hands-on Lab Summary

Stepping back to the fundamentals of the HTTP 1.1 specification, we are provided with clear guidance as to appropriate ways to exchange data between pages within a web application. Padarn provides a simple set of mechanisms to parse inputted values from a client, by way of HTTP GET, HTTP POST, or even URI submission.

We introduced the HOL with a discussion of the various HTTP Web Methods and concluded with a concrete example showing why GET and POST are so prevalent across websites.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Explored the basics of HTTP methods, including GET, HEAD, PUT, and DELETE
- ✓ Created logic to display an HTML input form
- ✓ Created logic to parse the values submitted in an HTML input form
- ✓ Implemented a basic redirection page for Padarn

After the concepts of this HOL are well understood by the reader, please proceed to the HOL 201 and 202, where we extend the HTTP methods discussed above to concretely implement a Remote Procedure Call set of services powered by Padarn.