

HOL P130 –Virtual Directories and File Uploading in Padarn

Table of Contents

Table of Contents	2
HOL Requirements	3
Summary.....	4
Lab Objective	4
Background.....	4
Exercise 1: Configuring the file receiver web resource	6
Exercise 2: Configuring Padarn for file uploading	12
Exercise 3: Preparing MyPadarnSolution for virtual directory support	14
Exercise 4: Creating code-behind for virtual directory listing	18
Exercise 5: Configuring Padarn for Virtual Directories	20
Exercise 6: Putting it all together!	21
Hands-on Lab Summary	25

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 leverage Padarn's ability to accept files submitted through a web client interface as well as how to configure convenient URL aliases called virtual directories. You will learn both how to configure Padarn as well as how to implement (in C#) the code necessary to demonstrate both capabilities.

Lab Objective

Upon completion of this lab, you will be familiar with how to modify the Padarn configuration file to enable virtual folders as well as how to create the code behind classes in support of file upload.

In this HOL, you will perform the following exercises:

- ✓ Understand the basics of HTTP POST with file data
- ✓ Learn how to receive uploaded files through C# code parsing the HTTP POST
- ✓ Learn how to modify the ocfhttpd.exe.config file to enable virtual folders
- ✓ Understand how Padarn authentication modes work in tandem with virtual folders

Background

File uploading - In HOL 201 and 202, we will cover how to implement and consume a Remote Procedure Call (RPC) web service. In these labs, we go into moderate detail as to how web servers interact with various client requests submitted over HTTP requests: HTTP POST, PUT, and GET all necessitate different actions. To upload a file to a server, there are several options including File Transfer Protocol (FTP, WebDAV, and various Peer-To-Peer (P2P) routines. Since Padarn supports many of the functions of an ASP.NET server, we will employ the `HttpPostedFile` functionality provided by Padarn. Here are the requirements for client and server:

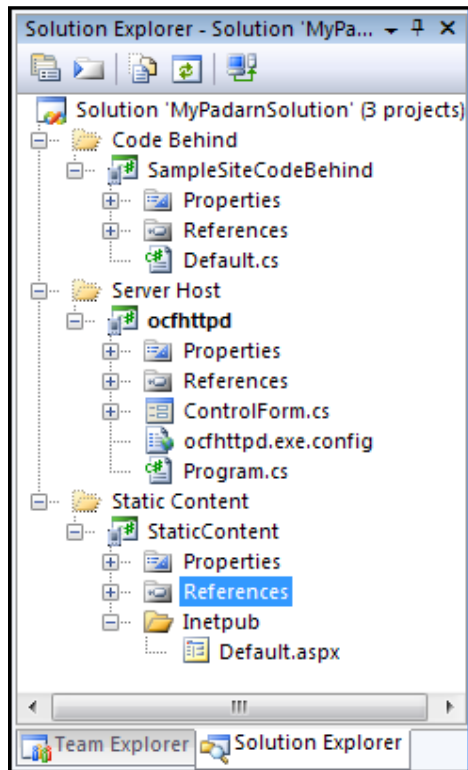
1. Client will be provided a form (a webpage form, in this case) with:
 - a. HTTP POST as form method
 - b. Content type as `multipart/form-data`
 - c. A single browse button (provided by the `OpenNETCF.Web.Html.Upload` class)
 - d. A single event button to perform the upload
2. Server will receive the standard Request object. It must:
 - a. Ensure the length of the `ContentLength` of the Request isn't 0
 - b. Verify the file data extracted from Request isn't null
 - c. Ensure local destination directory exists
 - d. Delete the local file with same file name if exists
 - e. Save file locally

Note: Padarn's configuration will establish the maximum size of the file transfer. Details will be provided in exercise 2.

Virtual directories – A virtual directory is a directory not actually contained within Padarn's home directory but appears as such to a client. A virtual directory consists of an alias and a mapping to a local directory. A client appends the alias to the URL of the Web site to browse the Web content in that virtual directory. We will use this concept to show how a directory listing webpage might be served, displaying the details of the files uploaded through the illustrated file upload support.

Exercise 1: Configuring the file receiver web resource

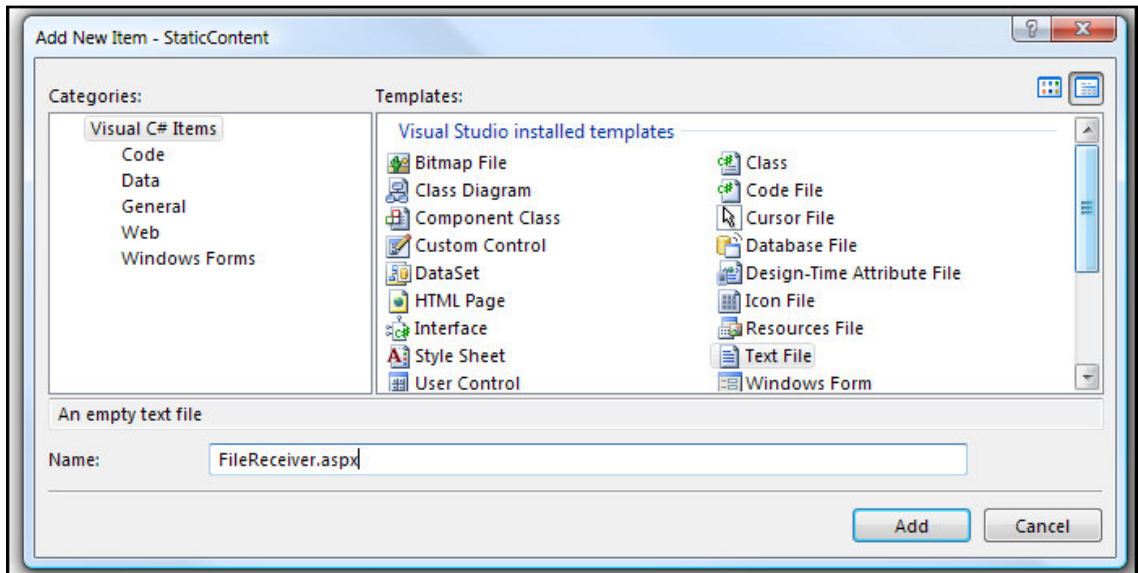
In this exercise, we will configure Padarn to serve a single Active Server Page (ASP) and we'll create some basic HTML as the test harness. To begin, please open the Visual Studio 2008 solution entitled "MyPadarnSolution.sln". Here is how the Solution Explorer displays the contents of this solution:



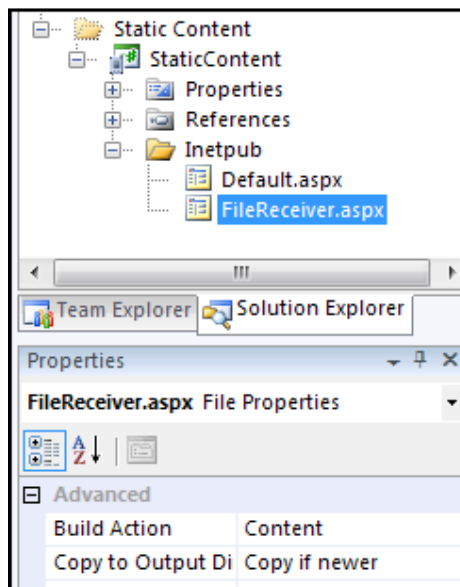
The two projects listed are a sample Padarn web server launch utility (Smart Device project) and an empty project to hold static content. We will now add a single Active Service Page (ASP) to the StaticContent project as our protected (and default) webpage for the Padarn instance.

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 "FileReceiver.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 Default.aspx file
6. In the Properties pane, set the Build Action for Default.aspx to "Content"
7. In the Properties pane, set the Copy to Output Directory property for Default.aspx to "Copy if Newer"



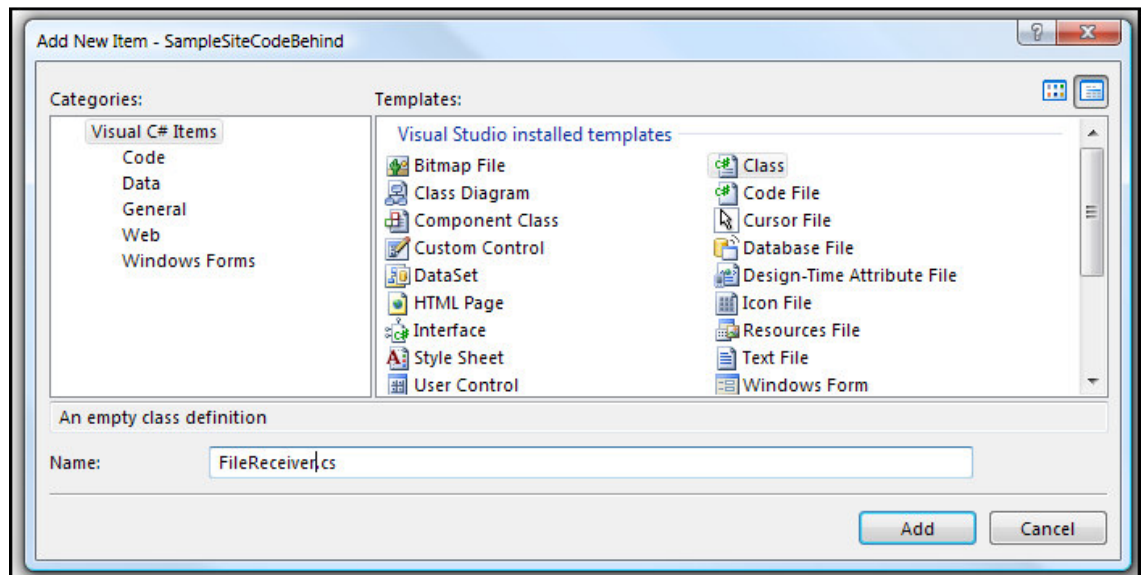
8. Paste the following code into the newly-created Default.aspx file:

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

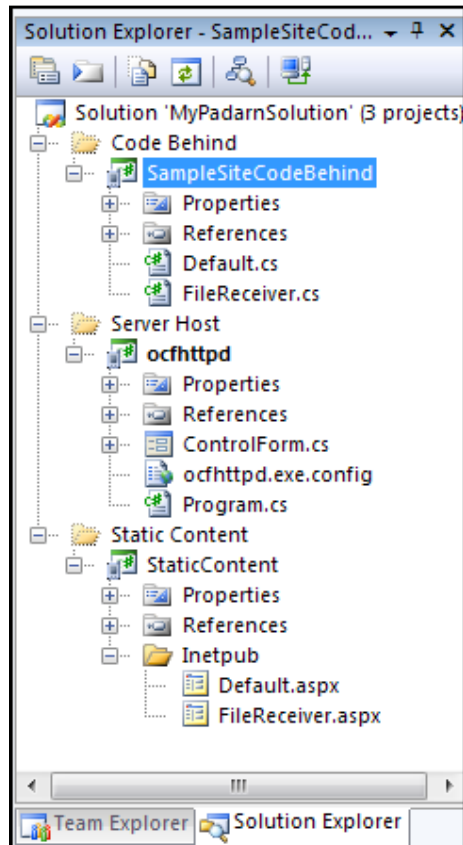
Next, you will use Visual Studio to create classes containing the code-behind logic for the file receiver landing page. The Padarn server will create an instance of the main class and run the Page_Load method of the class when a client browser (or Net client) accesses the FileReceiver.aspx page.

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 "FileReceiver.cs".



4. Click **Add** to add the new file to the project
5. The solution should now look the following:



At this point, we will create the code for the File Receiver class. This class will perform two basic functions: save a file locally if the incoming request is an HTTP POST or otherwise just display a basic page with a **Browse** and **Upload** button. To begin, create the basic structure of the page. This is all defined in FileReceiver.cs.

```
1 using System;
2 using System.Text;
3 using OpenNETCF.Web.UI;
4 using OpenNETCF.Web.Html;
5 using System.IO;
6 using System.Runtime.InteropServices;
7
8 namespace SampleSite
9 {
10     public class FileReceiver : Page
11     {
12         public const string VIRTUAL_DIRECTORY_LOCAL = @"\\Windows\\DumpSite\\";
13
14         protected override void Page_Load(object sender, EventArgs e)
15         {
16             Document doc = new Document(new DocumentHead("OpenNETCF File Upload Example"));
17
18             if (Request.HttpMethod == "POST")
19             {
20                 // We will save this file to the Virtual Directory folder
21                 if (this.Request.ContentLength > 0 && this.Request.Files.Count > 0
22                     && this.Request.Files[0] != null)
23                 {
24                 }
25             }
26             else
27             {
28             }
29
30             Response.Write(doc.OuterHtml);
31             Response.Flush();
32         }
33     }
34 }
```

We will be storing all files to the \\Windows\\DumpSite directory. Within the check for the POST method, we will perform the following tasks: check for existence of DumpSite directory (create if not), check for existence of incoming file (delete if so since we wish to always overwrite), verify sufficient local space (throw exception and include in response stream if so), and if all saves correctly, write upload time to response object. We use a P/Invoke to GetDiskFreeSpaceEx since there is no .NET means by which to check the amount of thread-accessible storage space free in a given location.

After verifying the contents of the Request object (see check in code above), we can use the [HttpPostedFile](#) class SaveAs method, which makes streaming to a local file store very easy! You can also take the from InputStream stream from the [HttpPostedFile](#) class and do with it as you please. We're now ready to save the file locally:

```
//Save file locally
this.Request.Files[0].SaveAs(fullPath);

FileInfo fiUploaded = new FileInfo(fullPath);

//Indicate success to user
doc.Body.Elements.Add(new RawText("Upload success"));
doc.Body.Elements.Add(new LineBreak());
doc.Body.Elements.Add(new RawText(String.Format("File upload completed at {0}",
    fiUploaded.CreationTime.ToLongTimeString())));
```

Be sure to notify the user if any error has been encountered, otherwise the client will be staring at a white screen without any further instructions.

```
catch
{
    //Indicate failure to user
    doc.Body.Elements.Add(new RawText("Upload failure"));
}
```

The second role of this code file is to display a UI by which a user can upload a single file to the server. The code to do this is simple:

```
protected override void Page_Load(object sender, EventArgs e)
{
    Document doc = new Document(new DocumentHead("OpenNETCF File Upload Example"));

    if (Request.HttpMethod == "POST")
    {
        //Code from above
    }
    else
    {
        //Utilitarian browse/upload form
        Form form = new Form("FileReceiver.aspx", FormMethod.Post, "upload", "upload");
        form.ContentType = "multipart/form-data";
        form.Add(new RawText("File to upload:"));
        form.Add(new Upload("upfile"));
        form.Add(new Button(new ButtonInfo(ButtonType.Submit, "Upload")));
        doc.Body.Elements.Add(form);
    }

    Response.Write(doc.OuterHtml);
    Response.Flush();
}
```

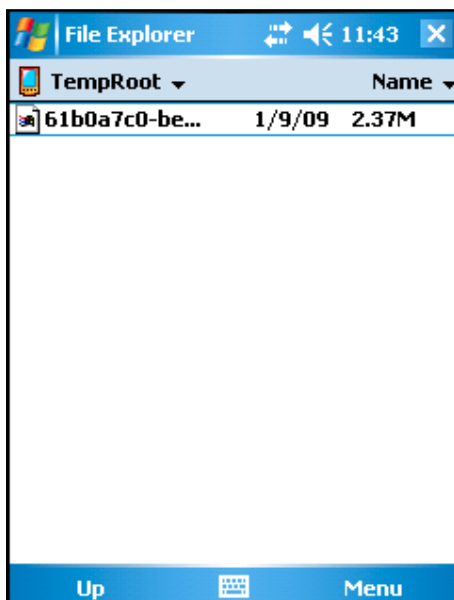
By leveraging the useful HTML wrappers from the `OpenNETCF.Web.Html` namespace, you can create a form page with little effort. Just establish the originating ASPX, an appropriate class name, a browse button with name, an upload button with name, and be sure to set the appropriate `ContentType`.

Exercise 2: Configuring Padarn for file uploading

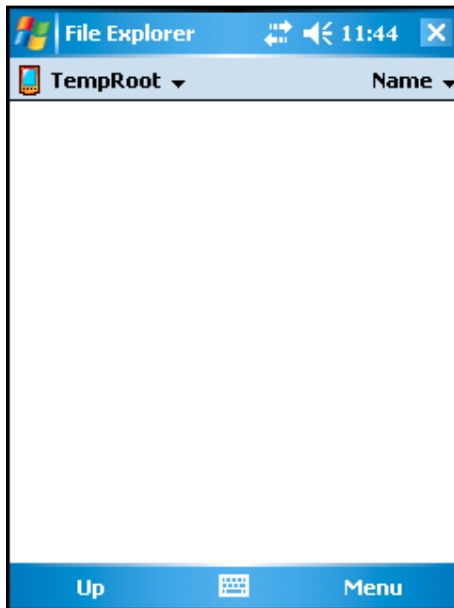
If Padarn is expecting file transfers, it is a good idea to protect the server against HTTP requests that could potentially overwhelm the host hardware. Although we verify that we have enough local space in the `FileReceiver` class above, we still run the risk of a request that is too intensive to handle. Padarn allows you to specify a temporary root location as well as bounds on the transmission size. By default, the default temporary root location is `\Windows\Temp\ASP.NET Temp Files`.

```
<WebServer
  LocalIP="0.0.0.0"
  DefaultPort="80"
  MaxConnections="20"
  DocumentRoot="\Windows\Inetpub\"
  Logging="true"
  LogFolder="\Temp\Logs"
  TempRoot="\TempRoot"
  LogExtensions="aspx;html;htm;zip"
  BrowserDefinitions="\Windows\Inetpub\config\browsers"
  UseSsl="false"
  CertificateName="\Windows\certificate\server.crt"
  CertificatePassword="padarn"
>
```

By setting the `TempRoot` element in the `WebServer` section, you can force Padarn to store temporary data files (any request content) to a Storage Card, for example. If this folder doesn't exist, during Padarn startup, it will be created. During transmission, you will see content appear in the directory of your choosing:



(Notice the size of the request!) After the request is processed, the folder is flushed:



Padarn supports two `httpRuntime` elements: `maxRequestLength` and `requestLengthDiskThreshold`.

- `maxRequestLength` specifies the limit for the input stream buffering threshold, in KB. This limit can be used to prevent denial of service attacks that are caused, for example, by users posting large files to the server.
- `requestLengthDiskThreshold` specifies the limit for the input stream buffering threshold, in bytes. This value should not exceed the `maxRequestLength` attribute. This value can usually be left to the default (256 bytes).

To prevent a very large file from being uploaded to padarn, we'll pare down the `maxRequestLength` value.

```
<httpRuntime
  maxRequestLength="1024"
  requestLengthDiskThreshold="256"
/>
```

Save the contents of the configuration file and be sure to deploy the entire solution to the device (the existing project properties should allow this to happen without further Visual Studio project setting changes). Select **Build -> Deploy Solution**.

Exercise 3: Preparing MyPadarnSolution for virtual directory support

Often times, virtual directories are used to provide clients with a folder/file listing of the mapped local folder on the server. However, Padarn does not currently support this, so we will implement a basic output of the contents of a virtual directory. Since the destination folder on the Padarn host machine specified above, \Windows\DumpSite, does not exist within the context of the Padarn home site, we'll need to do some work to allow the user to see the contents of the folder.

Create a landing page for virtual directory requests

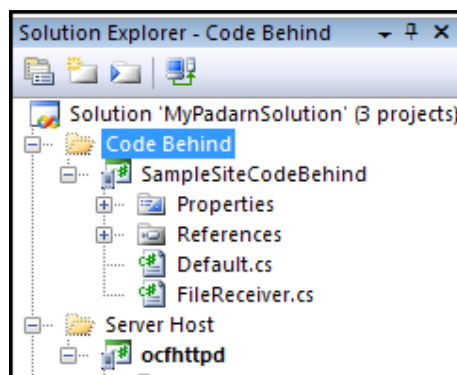
1. Create a new folder in the StaticContent project called DumpSite. Add a new text file called "Default.aspx" to the DumpSite folder.
2. Set the build action of this file to Content and Copy action to "Copy if Newer".
3. The code of "Default.aspx" should be:

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

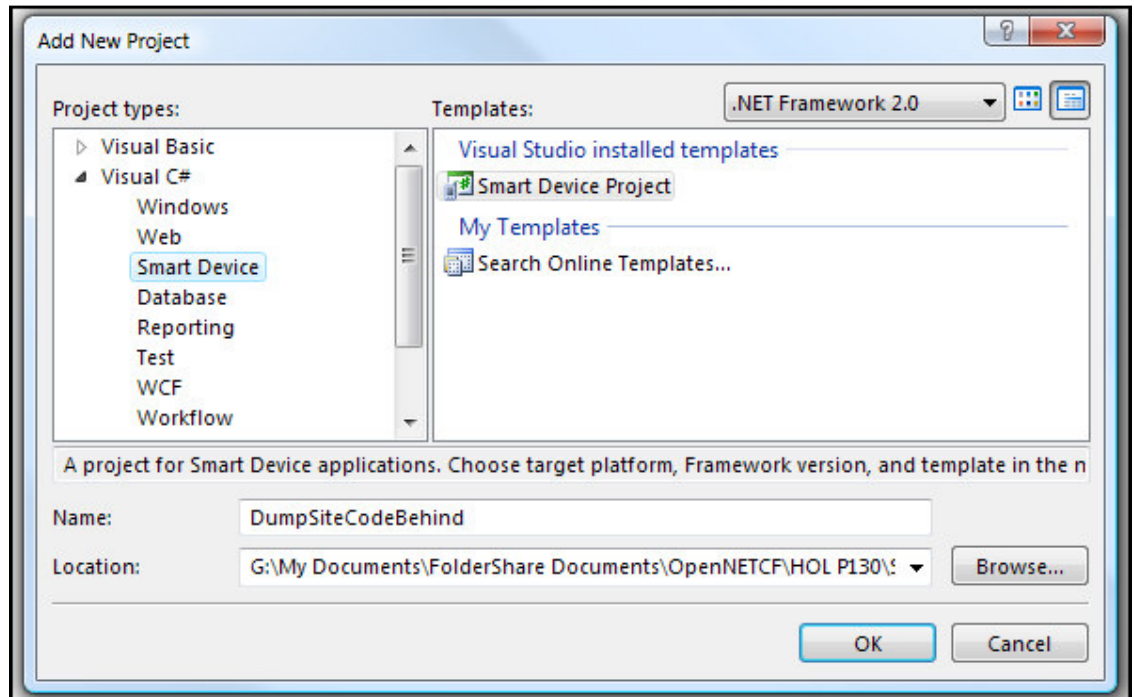
Check the example solution for details on the above.

Create the code-behind project for virtual directory requests.

1. From the Visual Studio **Project** menu item, select **Add New Solution Folder** and name the new folder **Code Behind**.

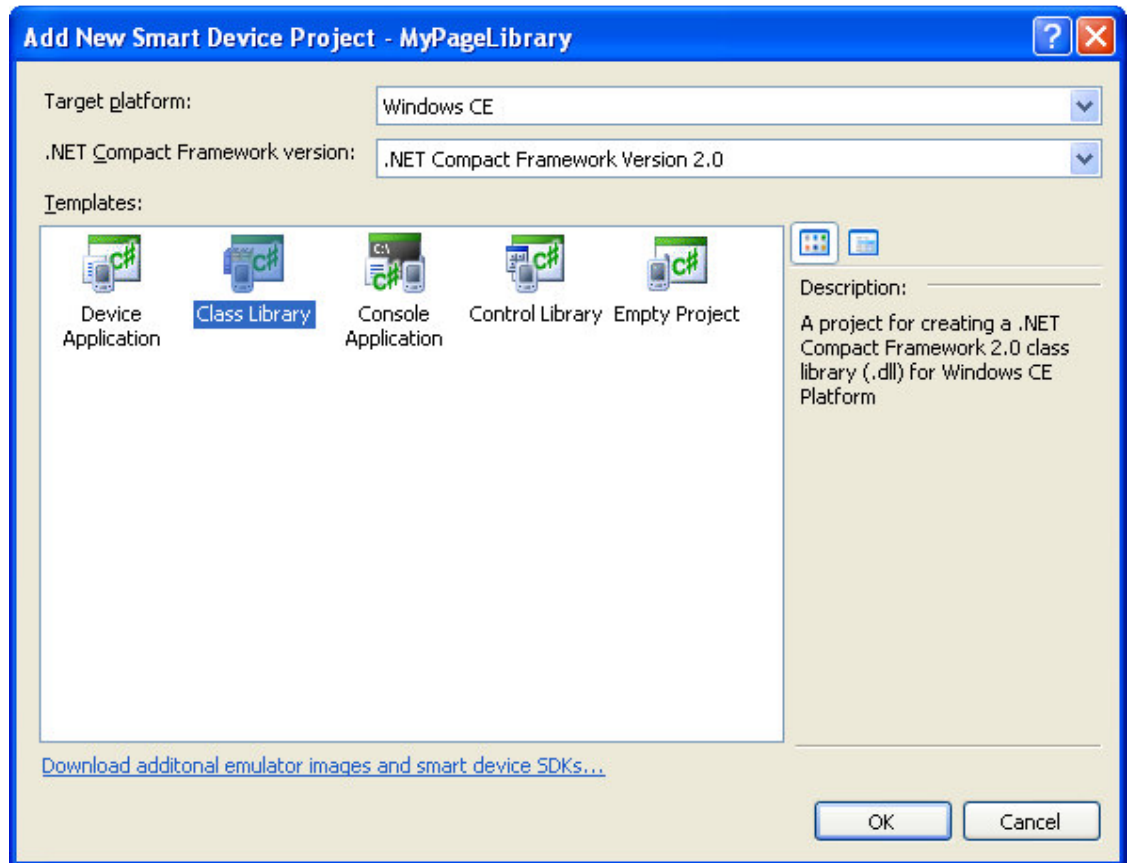


2. In Visual Studio's **Solution Explorer** pane, select the **Code Behind** Solution Folder.
3. In Visual Studio's **File** menu, select **Add** and then **New Project...** to launch the New Project Wizard.
4. In the **New Project** dialog box, under **Project Types**, expand **Visual C#**, and select the **Smart Device** project type.
5. Under **Visual Studio installed templates**, select **Smart Device Project**.
6. In the **Name** text box, type **DumpSiteCodeBehind**.
7. Click **OK** to move on to the **Add New Smart Device Project** Wizard.

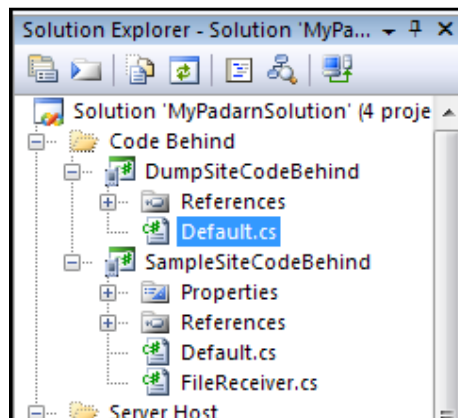


In the **Add New Smart Device Project Wizard**:

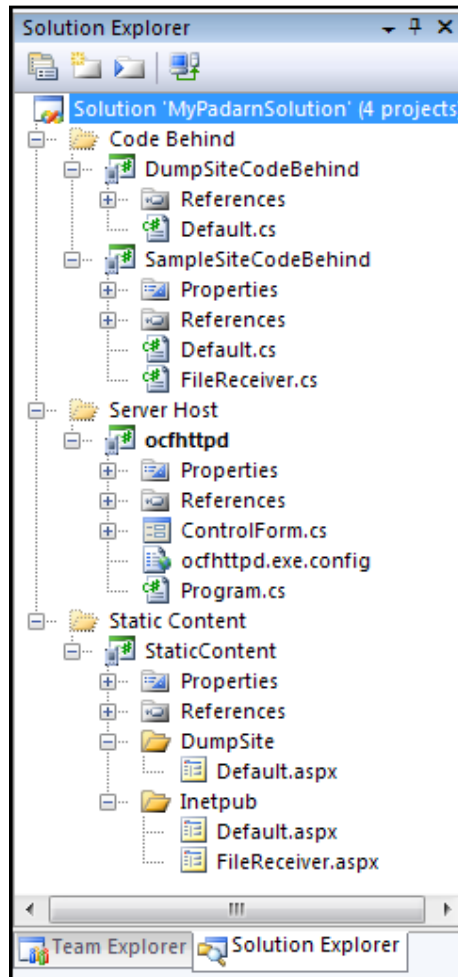
8. Select **Windows CE** from the **Target platform** dropdown.
9. Select **.NET Compact Framework Version 2.0** from the **.NET Compact Framework version** dropdown.
10. Select **Class Library** from the **Templates** list
11. Click **OK** to have the Wizard generate your code-behind library Project.



12. In the **Solution Explorer** pane, select the default **Class1.cs** file and rename it to **Default.cs**.



13. Verify your solution has the following layout:



Exercise 4: Creating code-behind for virtual directory listing

In this exercise, we'll step through the code of providing a detailed file output of the file receiver destination directory on the Padarn host machine (\\Windows\\DumpSite in our case). The majority of the effort will involve using the handy `OpenNETCF.Web.Html` wrappers to create a table within which to display file details. To begin, create the basic code structure in `Default.cs`:

```
1 using System;
2 using OpenNETCF.Web.UI;
3 using OpenNETCF.Web.Html;
4 using System.IO;
5
6 namespace SampleSite
7 {
8     public class Default : Page
9     {
10         public const string VIRTUAL_DIRECTORY_LOCAL = @"\\Windows\\DumpSite\\";
11         private const string CLASS_NAME = "DIRECTORY_LISTING";
12
13         protected override void Page_Load(object sender, EventArgs e)
14         {
15             try
16             {
17                 if (Directory.Exists(VIRTUAL_DIRECTORY_LOCAL))
18                 {
19                     foreach (string fileName in
20                         Directory.GetFiles(VIRTUAL_DIRECTORY_LOCAL))
21                     {
22                         //Loop through each file
23                     }
24                 }
25             }
26             catch
27             {
28                 //Display exception to Response object
29             }
30
31             // send the document html to the Response object|
32             // flush
33         }
34     }
35 }
```

You'll notice that once again, we're using the `Page_Load` method to perform all of our HTML activities. Since there will be no data submitted to the `Request` object for the `Page`, all of our effort will surround writing useful HTML to the `Response` (which will show up in the client as a table if a web browser).

1. Create the document object, establish containers to hold the main table, and display some friendly welcome text.

```
//Create the document
Document doc = new Document();

//Add a header to the document
doc.Head = new DocumentHead("OpenNETCF Dump File Directory Listing");

Div menuContainer = new Div();
menuContainer.ClassName = "centeredContainer";

Div menuItemContainer = new Div();
menuItemContainer.ClassName = "siteMenu";

Paragraph paragraph = new Paragraph();
paragraph.Elements.Add(new FormattedText("Listing of Dump Virtual Folder Contents:",
    TextFormat.Bold, "16"));
paragraph.Elements.Add(new LineBreak());

Table table = new Table(CLASS_NAME, null, Align.Left);

//Create header row
Row row = new Row();
row.Cells.Add(new RawText("#"));
row.Cells.Add(new RawText("Full Path"));
row.Cells.Add(new RawText("File Size (kb)"));
row.Cells.Add(new RawText("Upload date"));
table.Headers.Add(row);
```

2. For each file encountered within the local DumpSite folder, we'll show some file attributes.

```
//Create one row per file
DateTime dCreate = File.GetCreationTime(fileName);
FileInfo fiFile = new FileInfo(fileName);

row = new Row();
row.Cells.Add(new RawText(fileCount.ToString()));
row.Cells.Add(new RawText(fiFile.Name));
row.Cells.Add(new RawText((fiFile.Length / 1024).ToString()));
row.Cells.Add(new RawText(fiFile.CreationTime.ToString()));
table.Rows.Add(row);
```

If any file encountered is of type ASPX, we ignore it.

3. Finally, we add the paragraph and two containers to the document object, write this all to the Response object, and flush the Response.

```
menuItemContainer.Elements.Add(paragraph);
menuContainer.Elements.Add(menuItemContainer);
doc.Body.Elements.Add(menuContainer);

// send the document html to the Response object
Response.Write(doc.OuterHtml);

// flush
Response.Flush();
```

Exercise 5: Configuring Padarn for Virtual Directories

As mentioned above, virtual directories on a web server configuration consist of two elements: an alias (to be used by a client) and a mapping (a folder local to the host machine where the documents actually live). In Padarn, we set these two values in the `ocfhttpd.exe.config` file.

```
<VirtualDirectories> |
  <Directory
    VirtualPath="dump"
    PhysicalPath="\\Windows\\DumpSite\\"
    RequireAuthentication="false"
  />
</VirtualDirectories>
```

`VirtualPath` is the alias and `PhysicalPath` is the actual mapped folder. Notice we've elected to not impose authentication. If we did impose authentication (`RequireAuthentication="true"`), it would reference the Authentication Mode section discussed at length in HOL P110.

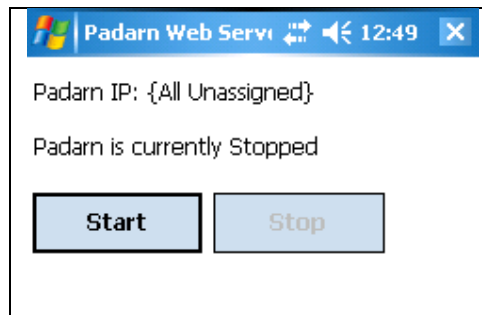
```
<Authentication Mode="Digest" Enabled="false" Realm="Padarn Test Site">
  <Users>
    <User Name="adminuser" Password="adminpass" />
  </Users>
</Authentication>
```

For the purposes of this example, we will not use authentication.

Exercise 6: Putting it all together!

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 `DumpSite\Default.aspx` and `FileReceiver.aspx` have been deployed as static content to the Padarn host machine. Instructions for how to accomplish this are in HOL 100.

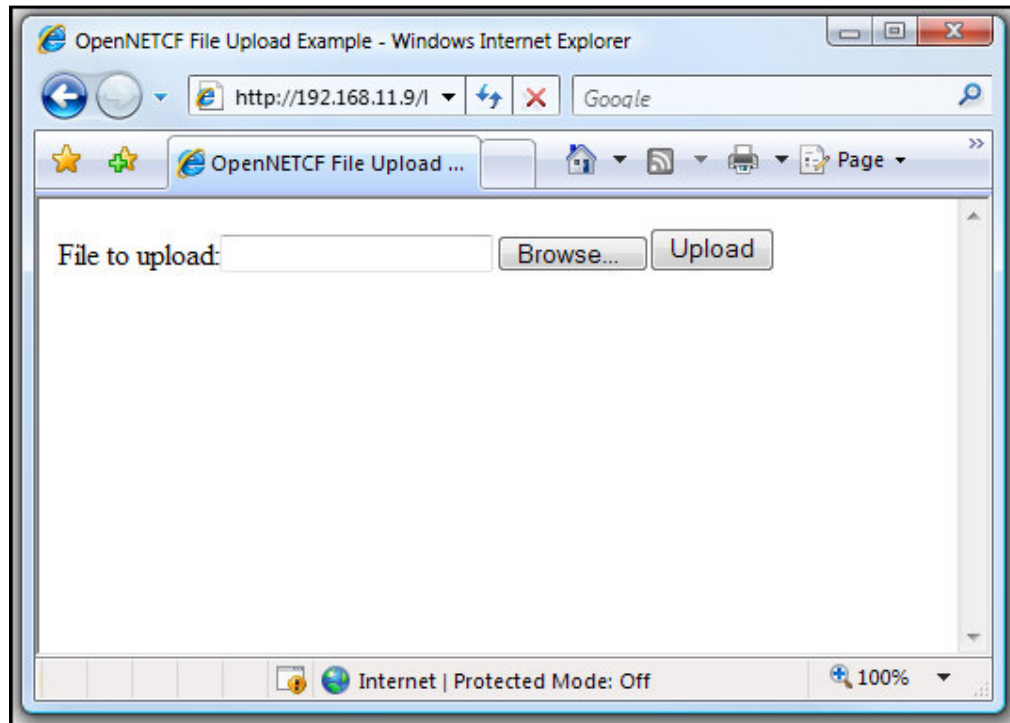
1. Initiate Padarn through the MyPadarnSolution Visual Studio 2008 solution. Right-click on the `ocfhttpd` project and select **Debug -> Start Without Debugging** (or press Ctrl + F5). For this demonstration, we will be using the Windows Mobile 5.0 Pocket PC R2 Emulator.



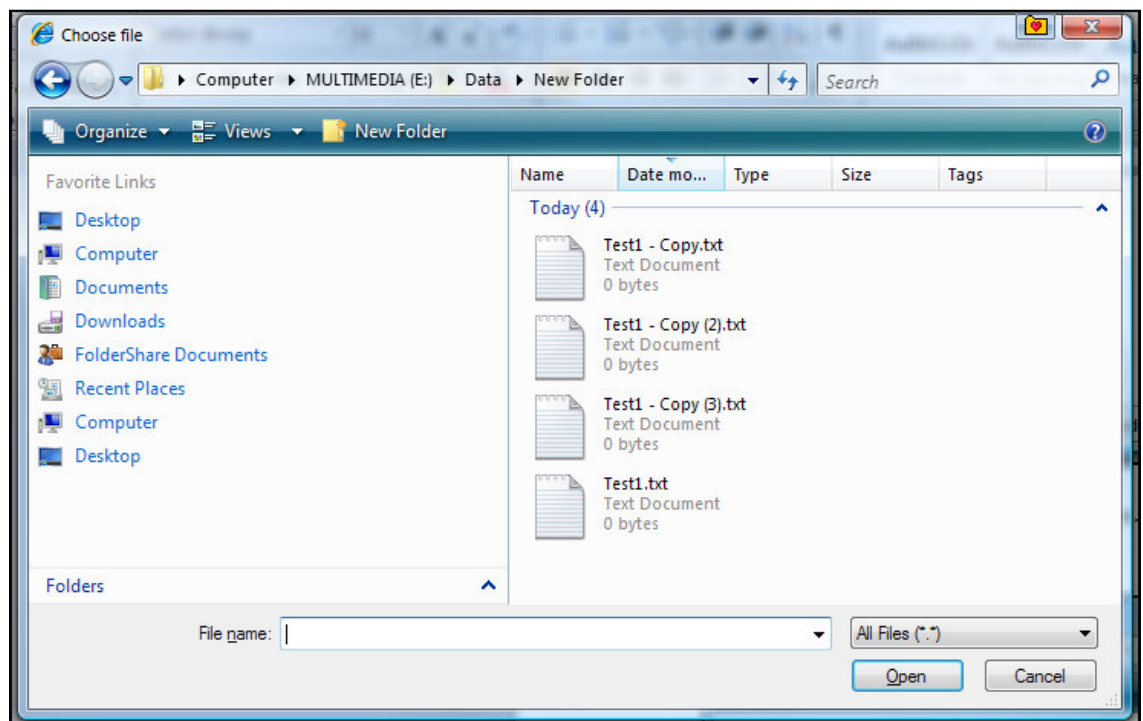
2. Press the **Start** button to initiate the Padarn instance.



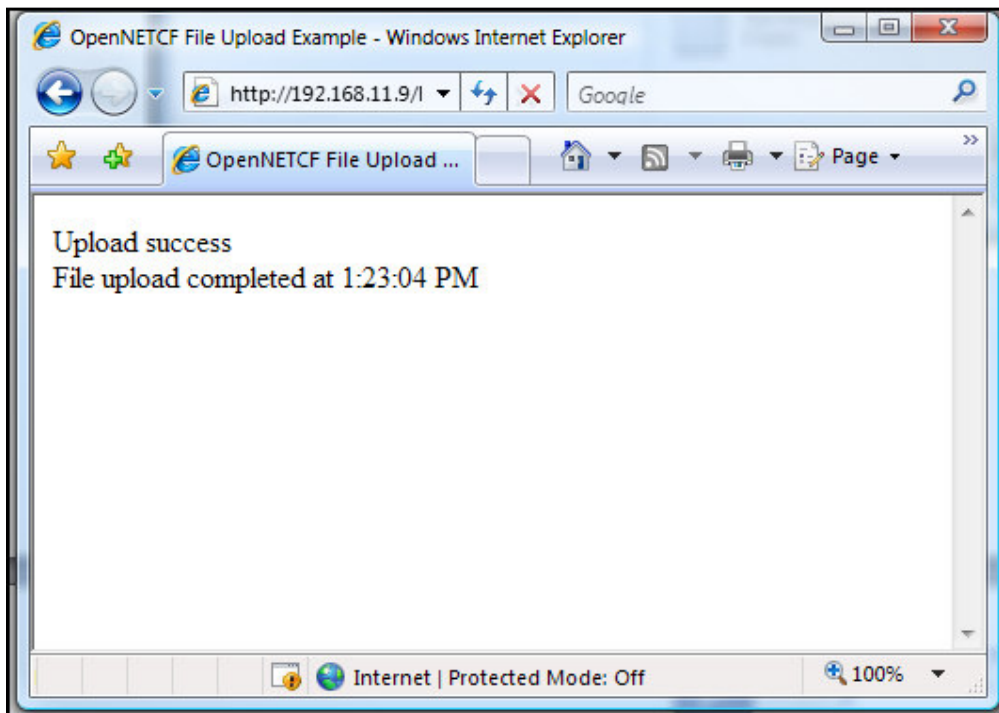
3. Right click on the `RPCExampleClient` project and select **Debug -> Start New Instance**
4. From a web browser that can communicate with the Padarn instance (for example, a Windows XP laptop that is host to the Windows Mobile 5.0 emulator). The IP address of the Padarn instance is key for bring up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



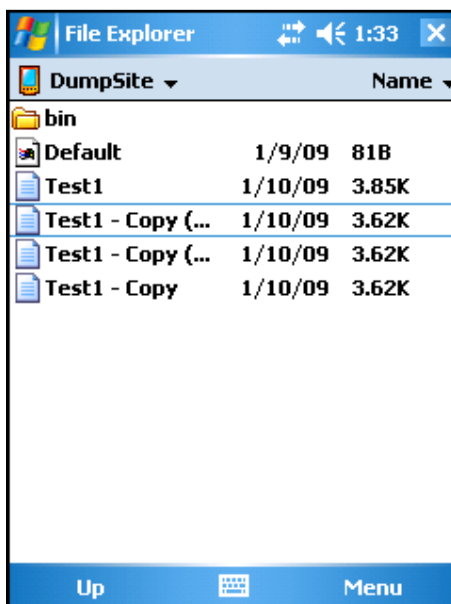
Here you see the standard file upload interface. Clicking **Browse** brings up the browser-implemented File – Open dialog:



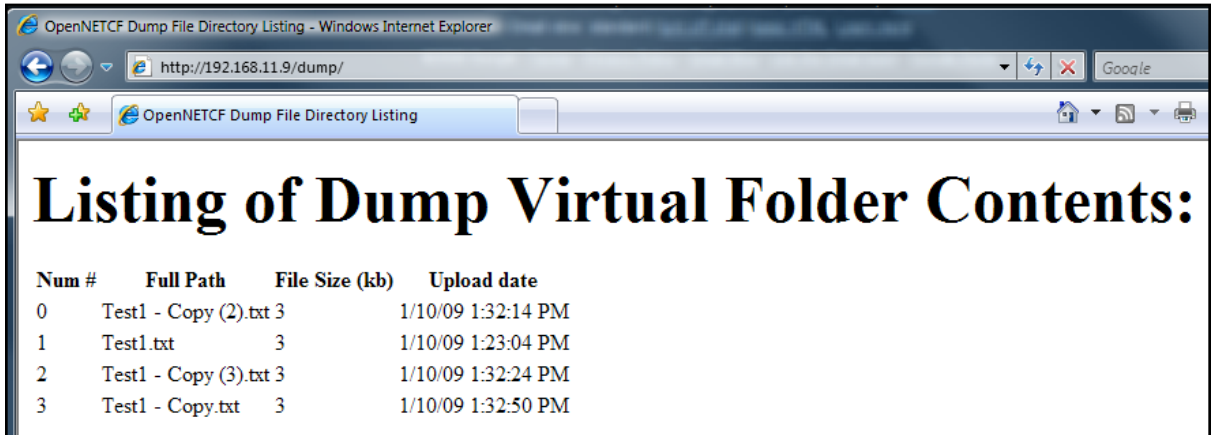
6. Upload several files. After each upload, you should see the Response screen.



On the Padarn machine, you'll see the files show up in File Explorer.



7. To test the DumpSite output, browse to the virtual directory on Padarn of <Padarn Server IP Address>/dump



Num #	Full Path	File Size (kb)	Upload date
0	Test1 - Copy (2).txt	3	1/10/09 1:32:14 PM
1	Test1.txt	3	1/10/09 1:23:04 PM
2	Test1 - Copy (3).txt	3	1/10/09 1:32:24 PM
3	Test1 - Copy.txt	3	1/10/09 1:32:50 PM

Your output mirrors the contents of the \Windows\DumpSite folder on the Padarn host machine

Hands-on Lab Summary

Due to OpenNETCF's strong desire to make Padarn as fully-compliant an ASP.NET web server as possible, it strives to include features that desktop developers might take for granted though are warmly welcomed by device developers. Two such features – file uploading and virtual directories – have come to Padarn v1.1.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Made simple change to the Padarn ocfhttpd.exe.config to enable virtual directories and customized maximum file upload size
- ✓ Learned about HTTP POST file upload as implemented by Padarn
- ✓ Create a code-behind class to display a summary of files uploaded to Padarn
- ✓ Walked through the end-user experience of uploading files to Padarn and seeing a summary through a web browser

In the HOL 2xx series, we take a closer look at consuming a broader set of Padarn services – namely Remote Procedure Calls (RPCs).