HOL P201 - Creating an RPC Web Service On Padarn

Table of Contents

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 create a Remote Procedure Call (RPC), exposed as a web service that provides several functions and can be consumed by either a desktop or mobile client and with or without a frontend. A later lab will walk through the steps needed to leverage the RPC web service.

Lab Objective

Upon completion of this lab, you will be able to generate an RPC web service that provides one or several functions that can be accessed by anyone making an HTTP POST request.

In this HOL, you will perform the following exercises:

- ✓ Leverage an existing Smart Device Solution containing all necessary projects for a Padarn Solution
- ✓ Use the OpenNETCF.Web.HttpRequest class to parse POST content and read incoming parameters
- ✓ Use standard ASP.NET code-behind techniques to perform remoting tasks on behalf of the requester

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 HOL P100 (Creating a Padarn Web Solution), as it explains how to start a Padarn web server instance and acts as the starting point for the content to follow in this lab.

Introduction to RPC

RPC is a form of inter-process communication, which generally takes place between two different computers such as a server and a client machine. The benefit of RPC is that a less powerful machine can send simple commands to a more powerful one and not be bothered with the details of the transaction to follow; rather, following some form of authorization, the client machine gives the server some basic instructions and can be presented with a return code indicating success or failure. Also of importance is that although the requester is likely not given the details of the transaction, the server will have properly logged any errors encountered by the process, which can be brought to an administrator's attention.

It is important to note that RPC's are dependent on the integrity of the network connecting the server and client machines; should the network become unresponsive (timeout) or completely severed (port failure), the client will need to treat this as exceptions and handle without crashing.

It is fairly straightforward to expose an RPC as a web service. This has several advantages over other mediums since modern day programming languages, from Java to C++ to JavaScript all have methods by which to interact with remote resources without having to create a lot of code. Most web frameworks make it easy to invoke the four HTTP verbs: POST, GET, PUT, and DELETE. However, because only POST and GET are supported by HTML 4.01/XHTML 1.0, we will focus on verbs.

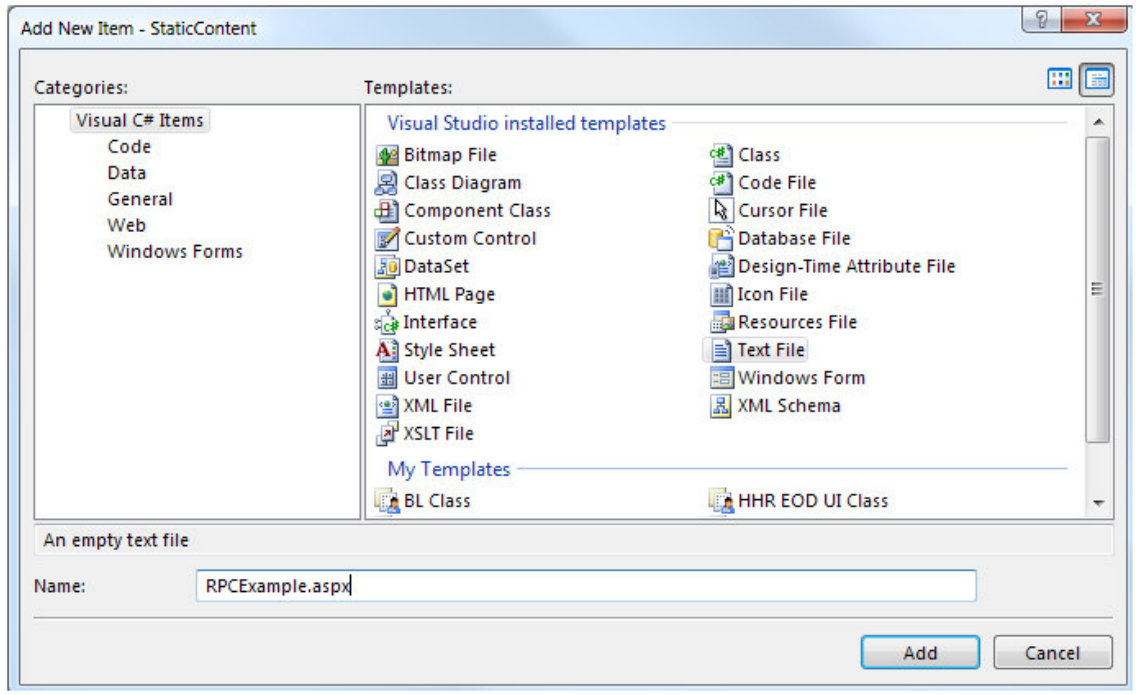
In this HOL, we will illustrate how a Library might be exposed as a series of RPC's, where each command performs an action pertaining to the Library's collection of books. As you can imagine, you might first need to show yourself as a valid Library user (POST), and then begin pursuing the collection of books (GET). It is important to realize that RPC does not expose a human-readable XML contract to a client like a SOAP web service would. As such, the supported functions of an RPC web service must be made available (usually published as an API guide) and agreed upon by all consumers.

Exercise 1: Adding the RPC ASPX

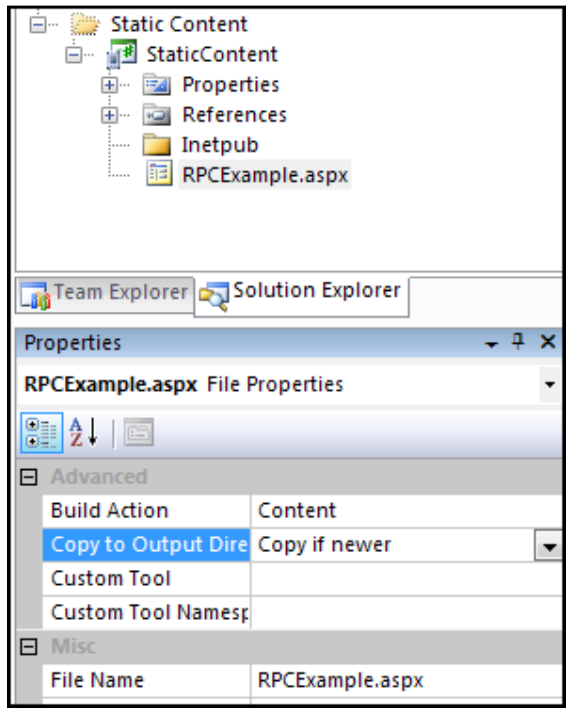
HOL P101 explained how to invoke an instance of the Padarn web server by calling into the WebServer class and calling the Start and Stop methods off the instance returned. Also, this HOL provided a launch form that can be used to easily turn on and off the Padarn web server. We will rely on these pieces (along with the configuration) to expose our RPC web service. First, we must publish a Microsoft Active Server Page (ASP) to the Padarn solution so that we can begin calling its functionality. You will use Visual Studio to create the ASPX file that describes to the Padarn server the code-behind assembly and class that it should load when a client browses to the page.

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 "RPCSample.aspx"



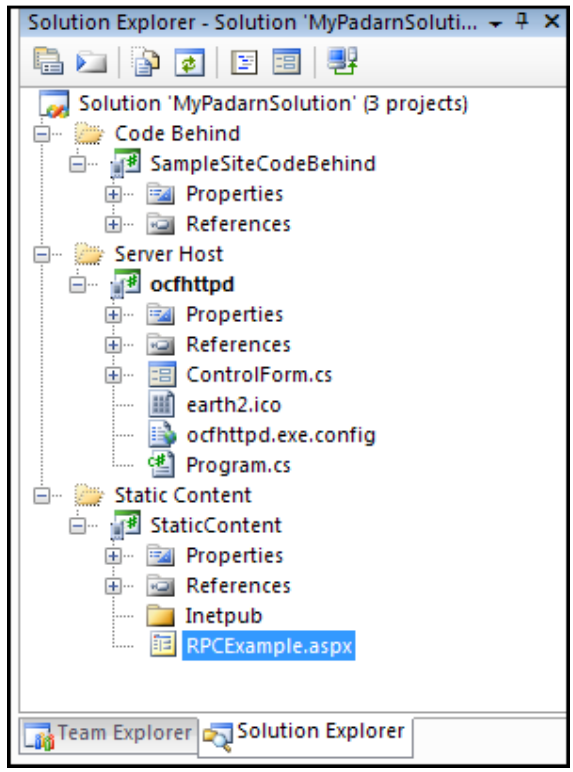
4. Click **Add** to add the new file to the project
5. In Visual Studio's Solution Explorer Pane, click on the newly-created RPCExample.aspx file
6. In the Properties pane, set the Build Action for RPCExample.aspx to "Content"
7. In the Properties pane, set the Copy to Output Directory property for RPCExample.aspx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



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

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

9. The solution will now look like this:

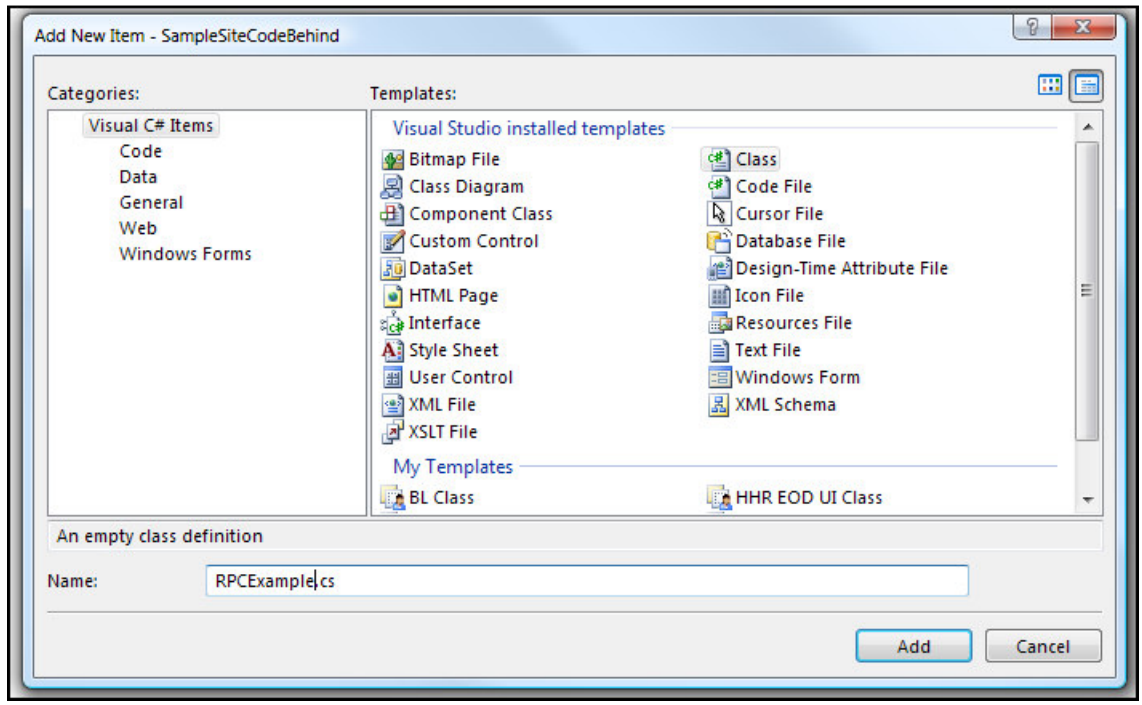


Exercise 2: Creating the Code-Behind Classes

In this exercise, you will use Visual Studio to create classes containing the code-behind logic for the sample RPC service. 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 RPCExample.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 "RPCExample.cs".

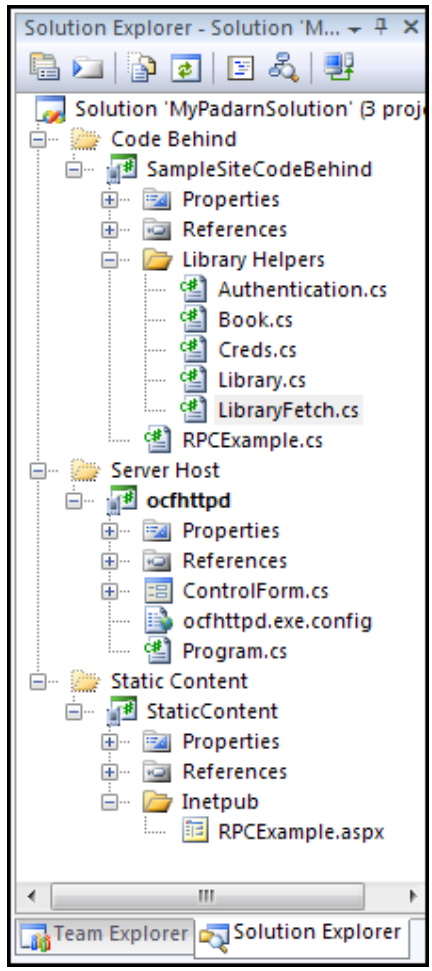


4. Click **Add** to add the new file to the project

To create the auxiliary code-behind classes:

1. In Visual Studio's Solution Explorer pane, right-click on the SampleSiteCodeBehind project and select **Add -> New Folder**. Call this folder "Library Helpers". This will hold a group of single purpose classes that will be leveraged by the main code-behind class
2. Add a total of five class files named "Authentication.cs", "Book.cs", "Creds.cs", "Library.cs", and "LibraryFill.cs" using the **Add -> New Item** function five times.

3. Since the default behavior of a Class file is to compile into the assembly, we need not alter the properties of any of the files created in this section. The solution should now look like the following:



Exercise 3: Creating the Code-Behind Structure

Before we can consider exposing a piece of functionality as a web service, we must have some kind of server logic coded and tested. For the purposes of this HOL, we will use a simple Library scenario. In this scenario, authorized users (users with library cards) will be able to access the properties of books. To read the properties of books, a user must identify himself (show his card), and then on any subsequent attempt to query books, he'll leverage the fact that he's already been checked in and won't have to go through the full handshake process again until he leaves the library. Obviously, in most libraries guests can access books but not check them out.

Let's first define our `Library` in code (localed in Library Helpers\Library.cs).

```
public partial class Library
{
        private List<Creds> _credentialStore;
        private List<Book> _bookCollection;

        public Library()
        {
                // Pre-fill credential store and book collection
                // Code lives in Library Helpers\LibraryFetch.cs
        }
        public string Login(string userName, string userPW)
        {
                return Authentication.Login(ref _credentialStore, userName, userPW);
        }

        public bool IsValidAuthenticationToken(String guidInput)
        {
                return Authentication.IsValidAuthenticationToken(_credentialStore,
                guidInput);
        }

        public string RetrieveBookProperties(string title, string author, string ISBN)
        {
                StringBuilder retVal = new StringBuilder();

                foreach (Book b in _bookCollection)
                {
                        // If match, add to retVal
                }

                return retVal.ToString();
        }
}
```

As you can see, our Library has a collection of authorized users and a collection of books. The login process requires a user name and password (typically would be passed through a typical credential prompt dialog) and a token is returned as a session ID. Though it's out of scope for this HOL, the session ID can be invalidated at any time by the server and thus require the user to re-login. Finally, we have a function that searches the book collection for any of the matching parameters passed in. If any of the three match, a hit is made. The example solution has code that compiles and runs for the above pseudocode.

Notes: Credentials would be fetched locally from an encrypted data store (or would call another RPC to get Active Directory or other Kerberos-based authentication server).

Next, let's define our `Authentication` class in code (located in Library Helpers\Authentication.cs).

```
public class Authentication
{
	internal static string Login(ref List<Creds> credsStore, string userName, string
		userPW)
	{
		// If user exists, see if a valid token exists for user and return it
		// If user exists but doesn't have valid token, create one and return it
		// If user doesn't exist, fail
	}

	internal static bool IsValidAuthenticationToken(List<Creds> credsStore, String
		guidInput)
	{
		// Check to see if token corresponds to valid logged in user
	}
}
```

Our `Creds` structure is a simple extension of the `System.Net.NetworkCredential` class (located in Library Helpers\Creds.cs).

```
public struct Creds
{
	internal NetworkCredential credentials;
	internal Guid sessionID;

	public Creds(string userNameIn, string userPWIn)
	{
		credentials = new NetworkCredential(userNameIn, userPWIn);
		sessionID = Guid.Empty;
	}
}
```

Finally, let's take a look at the `Book` class (located in Library Helpers\Book.cs).

```
public class Book
{
	internal string _title;
	internal string _author;
	internal int _ISBN;

	public Book(string title, string author, int ISBN)
	{
		//Remark: Would want to perform any format validation here
		_title = title;
		_author = author;
		_ISBN = ISBN;
	}
}
```

Obviously, this is a huge simplification of a book object and in the case of a real library, the user would want to read the contents of the book and likely get reviews or even a summary of the book. At this point, we're ready to start coding the `Page_Load` function of the code-behind class.

Exercise 4: Constructing the Page_Load Method

Even though we have defined a function that sounds like web browser parlance, any time a Padarn resource is accessed, it must first come through the `Page_Load` function. This makes coding simpler since at this point we can either act on the incoming request as if it's an HTTP method (GET or POST), or we can display content that has no bearing on the request. In this HOL, we will create the code that will allow us to respond to a call into RPCExample. The actual consumer will be illustrated in HOL 202 : Consuming a Padarn-hosted RPC Web Service.

We will respond to two verbs in RPCExample: authenticate and query. Authenticate will take in a user name and password in the format of

```
function=authenticate&username=<User>&userpw=<PW>
```

Since it is likely that the authenticate method will modify internal state and possible create new records on the server, we will treat this as a HTTP PUT. As such, the user name and password values will show up within the `HttpRequest` object. Each OpenNETCF.Web.UI.Page object has an `HttpRequest` and an `HttpResponse` object, matching the basic functionality of an ASP.NET page. In the case of HTTP POST, we are concerned about both objects. In the case of HTTP GET, we only need to deal with the `HttpResponse`.

Here is how intercept requests for authentication:

```
protected override void Page_Load(object sender, EventArgs e)
{
    string incomingParameters = Request.RawQueryString.ToLower();
    string[] actionList = incomingParameters.Split('&');

    // Read out functionName

    switch (functionName)
    {
        case "authenticate":
        {
                // Only allow HTTP POST for this method.
                if (!this.IsHttpPost)
                        return;

                // Format of POST content should be 'userName=XYZ&userPW=ABC'
                string userName = Request.Form["userName"];
                string userPW = Request.Form["userPW"];

                string GUID = myLibrary.Login(userName, userPW);

                // Format of POST content should be 'userName=ABC&userPW=DEF'
                WriteResult("authenticate", GUID);
        }
        break;
```

A few observations here: we mandate that the authenticate request be submitted as an HTTP POST. Furthermore, we leverage the `Form` object to read out the variables submitted in the request. We use a static `Library` object to authenticate the user then generate a local GUID. Finally, we use a

function called WriteResult to communicate the function result to the service consumer. WriteResult creates an XML document with a single XML node. See source contained within this HOL for details.

The next function is our query method. Since this function will not be modifying any server data, we might choose to implement this as an HTTP GET. HTTP GET methods do not contain request message bodies, so we will need to parse the incoming URL to obtain the qualifiers needed to carry out the lookup.

Here is how we do that (this code appears on the same level as the authenticate method):

```
case "query":
{
        //Use OpenNETCF HttpUtility to strip out HTML characters
        string[] chunkedValues =
        OpenNETCF.Web.HttpUtility.UrlDecode(Request.RawQueryString).Split('&');
        Dictionary<string, string> parsedValues = new Dictionary<string, string>();

        //Parse non-verb chunk of decoded URL and store into dictionary

        string token = parsedValues["token"];
        string title = parsedValues["title"];
        string author = parsedValues["author"];
        string ISBN = parsedValues["ISBN];

        if (myLibrary.IsValidAuthenticationToken(token))
        {
                //Only run query if valid token
                string queryResult = myLibrary.RetrieveBookProperties(title, author, ISBN);
                WriteResult("query", queryResult);
        }
        else
        {
                WriteResult("query", "Not authenticated");
        }
}
break;
```

Although the code snippet above is missing error handling, you can see we are taking an encoded URL (which has HTML escape character codes like %20 for <Space> or %23 for '#'), decoding it, then pushing the contents of the Request URL into a dictionary. From there, we run through our GUID-based authentication scheme and if that passes, then we call into the book search function. The book search function will write out to the responses object the zero-to-many results, which would be displayed in a web browser as text or can be parsed and displayed in a windowed application.

You will notice that the query operation has no enforced preconditions to allow it to run. In other words, because a token (GUID) has been passed in, the call can be processed regardless of whether the network had been disrupted at any point between host and client. This is an important advantage of RPC.

Hands-on Lab Summary

Remote Procedure Calls are a universally-accepted form of inner-process communication that works especially well across slow networks. Although we did not show how these two basic actions would be invoked via a client, we will do so in the next HOL. The code in this lab will allow you to quickly and easily perform any function supported by the host (typically a Windows CE-powered device in the case of Padarn) from a remote machine. Just about any TCP/IP-compliant computing device has the ability to send HTTP methods across a given network, so the power here is pretty huge.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Added a new Active Server Page to your website solution
- ✓ Added a new code-behind class to your website solution
- ✓ Walked through the class structure of a real world-based object model
- ✓ Implemented an RPC-style parser that facilitates remote interaction with said model

Although RPC is not the only available option for machine-to-machine inner process communication, it is the simplest and least bandwidth-intensive style supported by Padarn at this time. In the future, other methods might be made available (such as SOAP, JSON, etc.)