

HOL P202 – Consuming a Padarn-hosted RPC Web Service

Table of Contents

Table of Contents	2
HOL Requirements	3
Summary.....	4
Lab Objective	4
Pre-Requisites.....	4
Exercise 1: Adding the Login and Search form	5
Exercise 2: Adding the Helper class	9
Exercise 3: Implementing the Helper class.....	10
Exercise 4: Hooking the client up to the Helper class	14
Exercise 5: Trying out the client	16
Hands-on Lab Summary	19

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 utilize a Remote Procedure Call (RPC) exposed as a web service from a simple .NET client that can run on a desktop platform. There is nothing stopping you from implementing this code as a native Windows Mobile class, JavaScript function, or really any other language and platform combination that understands simple HTTP GET and POST methods.

Lab Objective

Upon completion of this lab, you will be familiar with the process of configuring a basic network-enabled client to pass requests through an RPC service via the [HttpWebRequest](#) class.

In this HOL, you will perform the following exercises:

- ✓ Create a WinForm to handle authentication and another for search results
- ✓ Use the System.Net.HttpWebRequest class to generate POST content and encode search parameters
- ✓ Use generic .NET constructs to parse the return values from the RPC web service

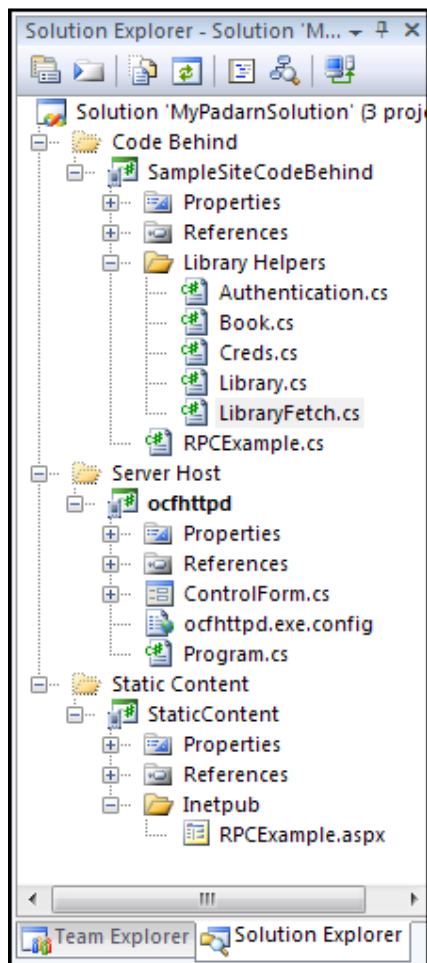
Pre-Requisites

This lab builds on the code created through HOL 201. Although the necessary components from HOL 201 are included in this lab's example solution, it's advisable to go through the explanations in HOL 201 to better understand the various functions provided by the sample RPC web service.

In HOL 201, we illustrated 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). This lab steps you through how a client would access a Library and search its contents.

Exercise 1: Adding the Login and Search form

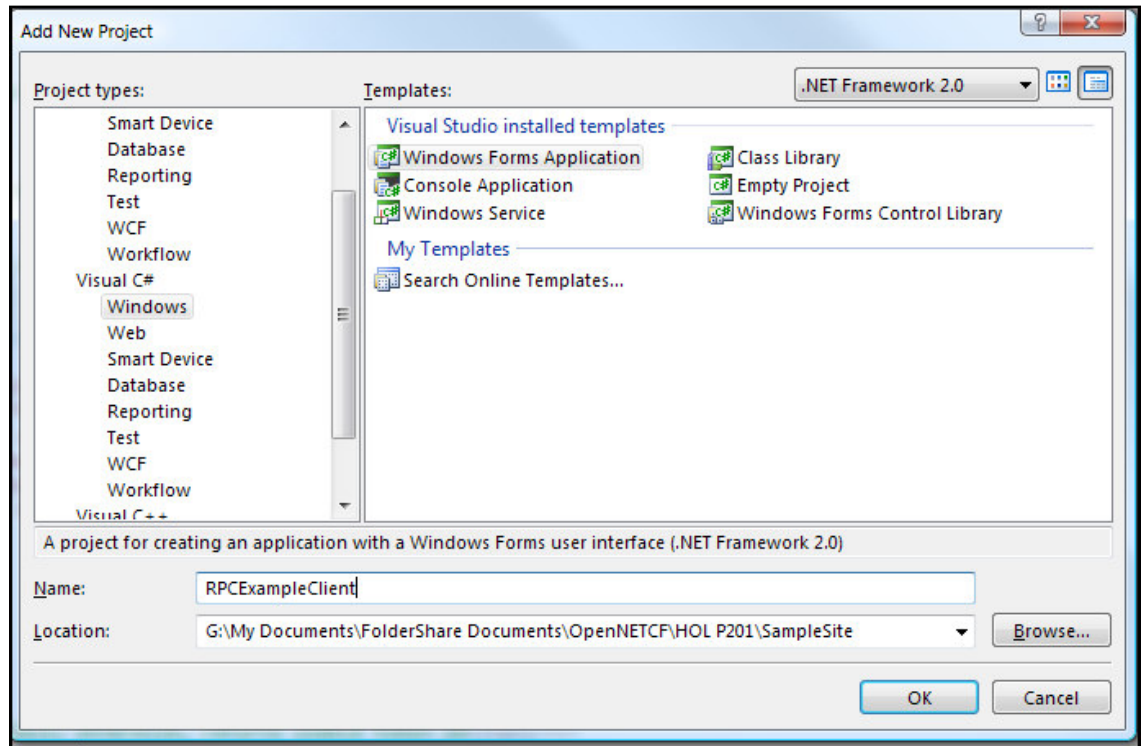
To begin, please open the Visual Studio 2008 solution entitled “MyPadarnSolution.sln”. Here is how the Solution Explorer displays the contents of this solution:



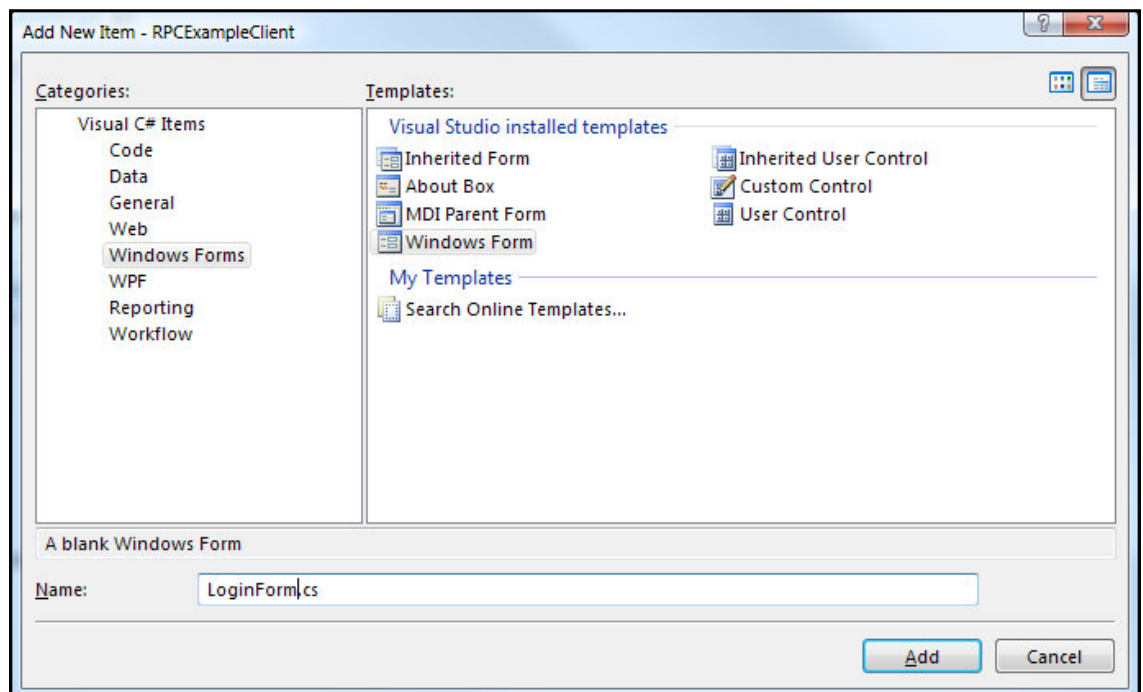
The three projects listed are the code-behind logic for the RPC web service (Smart Device project), a sample Padarn web server launch utility (Smart Device project), and finally the RPC web service itself contained within a single Active Server Page (Smart Device project). We will now add a *desktop* client to the solution.

To create the desktop project:

1. In Visual Studio’s **Solution Explorer** Pane, right-click on the root solution node and select **Add -> New Project**
2. From the list of Visual Studio Templates, select Visual C#: Windows Form Application.
3. Name the project “RPCExampleClient”. Be sure to select .NET Framework 2.0 for maximum compatibility should you choose to share this client with your peers.

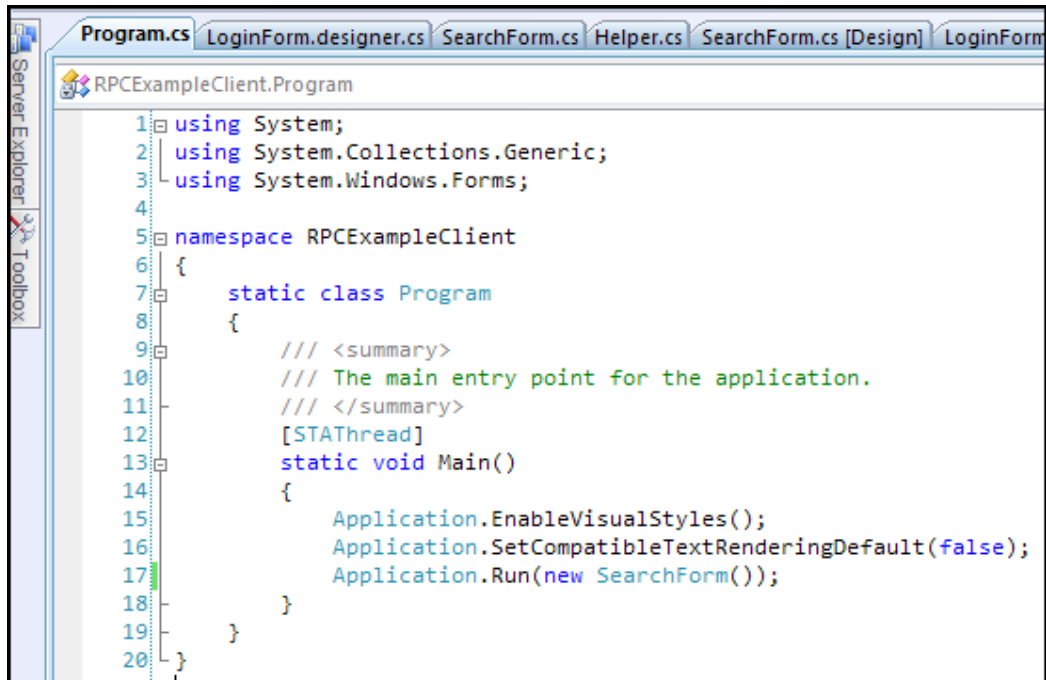


4. From within the project, delete Form1.cs (created by default).

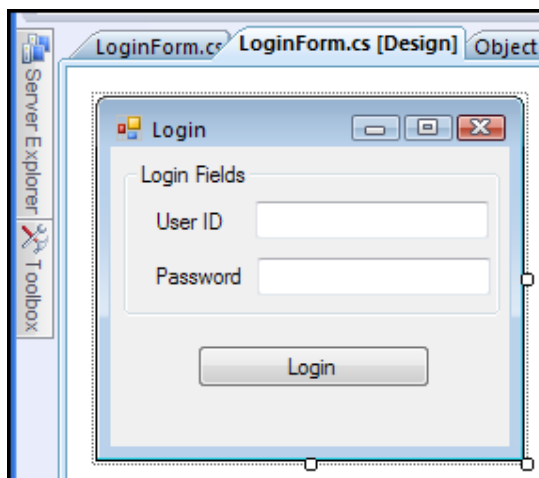


5. Right-click "RPCExampleClient" from Solution Explorer and **Add New Item**. Choose Visual C# Items – Windows Forms – Windows Form. Call this first form LoginForm.cs. Click **Add**.

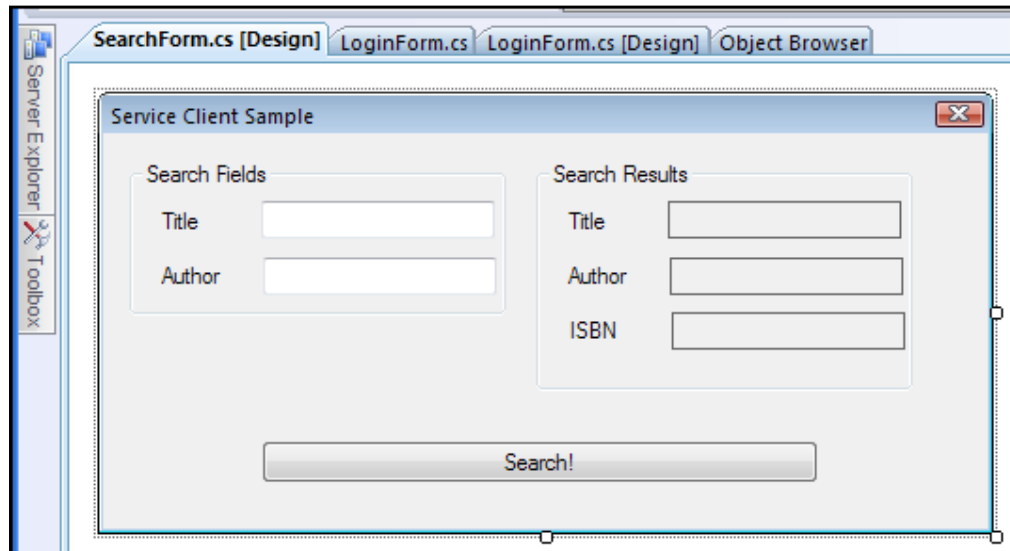
6. Right-click “RPCExampleClient” from Solution Explorer and **Add New Item**. Choose Visual C# Items – Windows Forms – Windows Form. Call this first form LoginForm.cs. Click **Add**.
7. Open up the Program.cs file and ensure that the startup form is SearchForm



8. Now, let's review the controls that need to be placed on each of the two forms.



The LoginForm will have two textboxes (user ID and password) and a Login button. This form will be treated as a child of the SearchForm so that the user is required to login prior to begin searching.



The SearchForm has two textboxes upon which our search query is based (title and author), three labels that will display the search result, and a search button.

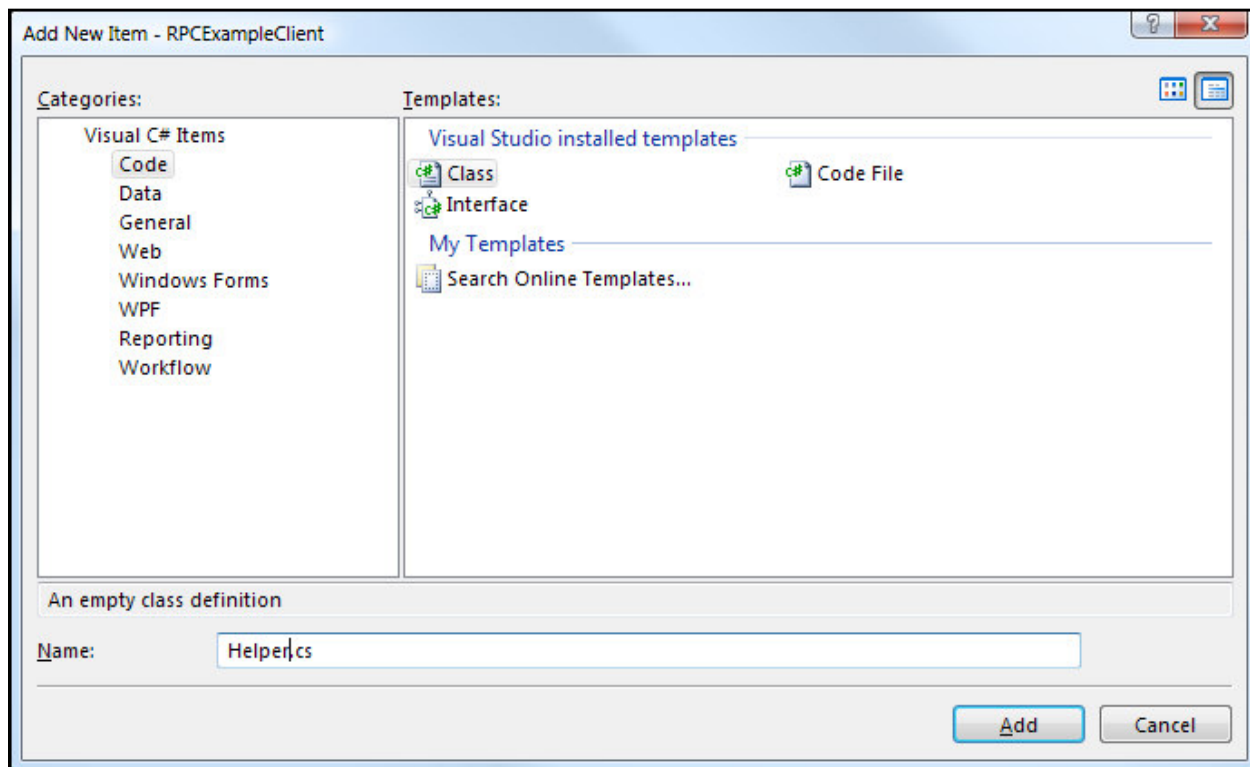
Please see example solution for specifics on how these two forms are constructed using Visual Studio 2008's designer.

Exercise 2: Adding the Helper class

The majority of the RPC sample client logic will exist in the Helper class. This class will submit an [HttpRequest](#) instance to the RPC URI and will handle the parsing of the resultant XML, either notifying the user of an error or by placing the results in the appropriate SearchForm labels.

To create the CS file:

1. Right-click “RPCExampleClient” from Solution Explorer and **Add New Item**. Choose Visual C# Items – Code – Class. Call this class “Helper.cs”. Click **Add**.



Exercise 3: Implementing the Helper class

Our first key function to implement is the logic that interacts with the RPC web service. Let's define several constants.

```
8 namespace ExampleServiceClient
9 {
10     class Helper
11     {
12         private const string SERVER_ADDRESS = "192.168.11.9";
13         private const string SERVICE_URI_FORMAT = "http://{0}/RPCExample.aspx?{1}";
14         private const string UNABLE_TO_CONNECT = "Unable to connect to the remote server";
15         private const string BOOK_NOT_FOUND = "Book not found in library";
16
17         private const string VERB_AUTHENTICATE = "Authenticate";
18         private const string VERB_QUERY = "Query";
19
20         private static string loginToken = String.Empty;
21         private string searchResult = String.Empty;
22
23         public static bool IsAuthenticated
24         {
25             get { return !String.IsNullOrEmpty(loginToken); }
26         }
27     }
28 }
```

Several constants need to be defined. First, the IP address of the server is key for having any interactions with Padarn. Various utilities exist for Windows CE that provide this information. For example, you could use the [OpenNETCF.Net.NetworkInformation](#) namespace. Next, the path to the web service needs to be defined. Several strings are defined in code that should be stored into the project's resource files so as to be localized where appropriate. These messages will be shown to the user. Finally, we define our two functions that the web service understands. We store the authentication token locally as well as the result string from any authorized search queries. For the SearchForm to determine whether it can dismiss the LoginForm, it uses the IsAuthenticated property.

We must establish some plumbing around the web service interaction. As mentioned earlier, we use an `HttpRequest` object to handle the initial communication, then create an `XmlDocument` object to read the entire response. Two important notes: `XmlDocument` is not as efficient as other XML processing technologies though it is flexible and we have no performance requirement for this sample; most web requests should be performed in an asynchronous way (typically carried about by a worker thread) such that the UI remains responsive throughout the operation and the user can cancel at any point without waiting for the timeout period.

Let's take a look at the code chunks necessary to facilitate an RPC call against Padarn.

```
private static string RPCCallResult(bool isPost, string url, byte[] contentBytes, string
    nodeName)
{
    //1. Establish web request object
    //2. Open Request and write message to it
    //3. Make sure valid response came back
    //4. Parse response - expecting a Guid back
    //5. Read result text for our target.
}
```

1. The `HttpRequest` is what allows us to submit data to a web service endpoint. Our usage of this class is no different than submitting user input from a webpage form.

```
HttpRequest request = (HttpRequest)WebRequest.Create(url);
request.Method = isPost ? "POST" : "GET";
request.ContentLength = contentBytes != null ? contentBytes.Length : 0;
request.ContentType = "application/x-www-form-urlencoded";
```

If this function is used as part of a GET method, then we do not include any content in the message body and instead use the URL to specify the information we're requesting. If the function is used as part of a POST method, we specify in our request header that the method is POST and the length of the content is the byte-converted length of the message. In both cases, we need to tell Padarn that we're sending our URL as HTML encoded (compliant with RFC 1738).

2. Given the full URL of the Padarn web server including the sample RPC destination, we are ready to write the POST content to the request object (only if POST) and wait for the server response (if POST or GET).

```
if (isPost)
{
    using (Stream s = request.GetRequestStream())
    {
        s.Write(contentBytes, 0, contentBytes.Length);
    }
}

HttpResponse response = (HttpResponse)request.GetResponse();

if (response.StatusCode != HttpStatusCode.OK)
{
    response.Close();
    return String.Empty;
}
```

3. Since in HOL 201 we saw that the result will be stored in simple XML format (the node name will be "<VERB>Result"), we use `XmlDocument` to read the result so long as we didn't time out. If the request times out, we throw a generic Exception back to the caller with the timeout string constant as the message value.
4. Now that we have the `RPCCallResult` function defined, we can get one step closer to wiring up our sample client by creating in functions that will be used directly by the WinForms.

```
public static bool CallRemoteAuthenticateMethod(string userName, string userPassword)
{
    //1. Establish full URL with action verb
    string url = String.Format(SERVICE_URI_FORMAT, SERVER_ADDRESS,
        VERB_AUTHENTICATE);

    //2. Establish POST message body
    string contentString = String.Format("userName={0}&userPW={1}",
        userName, userPassword);
    byte[] contentBytes = Encoding.ASCII.GetBytes(contentString);

    loginToken = RPCCallResult(true, url, contentBytes, "/authenticateResult");
    if (!String.IsNullOrEmpty(loginToken))
    {
        if (loginToken.Equals(UNABLE_TO_CONNECT))
        {
            throw new Exception(UNABLE_TO_CONNECT);
        }
    }

    return !String.IsNullOrEmpty(loginToken);
}
```

Authenticate: As you can see, we use basic string manipulation to fill in the userName and userPW fields required by the RPC call. Because this is a POST method, we prepare a byte-converted representation of the command along with the expected resultant XML node name.

```
public static string CallRemoteQueryMethod(string title, string author)
{
    //1. Establish full URL with action verb
    string url = String.Format(SERVICE_URI_FORMAT, SERVER_ADDRESS, VERB_QUERY);

    //2. Establish GET URL
    string urlFullString =
        HttpUtility.UrlEncode(String.Format("token={0}&author={1}&title={2}",
            loginToken, author, title));

    //3. Run auxiliary HttpWebRequest function
    string queryResult = RPCCallResult(false, url + "&" + urlFullString, null,
        "/queryResult");

    if (!String.IsNullOrEmpty(queryResult) && queryResult.Equals(UNABLE_TO_CONNECT))
    {
        throw new Exception(UNABLE_TO_CONNECT);
    }

    //4. Return
    return queryResult;
}
```

Query: This function is simpler in that it does not need to create a byte-converted representation of the command we are submitting to the RPC web service. Instead, we merely

submit the encoded command as part of the URL. We then call our `RPCCallResult` method and tell it to run the command as a `GET` method along with the expected resultant XML node name.

Exercise 4: Hooking the client up to the Helper class

We're now ready to wire up the two forms we created at the beginning of the lab. The Helper class will allow us to authenticate the user to the Library as well as perform a book search on behalf of an authenticated user. Any exceptions caught by the Helper class should be logged and is outside the scope of this lab; here, we merely display message boxes informing the user of issues. To wire:

1. Define a button-click handler for LoginForm's login button. The submitted user name and password will be sent to `Helper.CallRemoteAuthenticateMethod`. Should an error arise, display it to the user. Clear out the password if authentication failed but the web service returned a valid result. Be sure to perform first-level validation on the user name and password
2. The SearchForm requires interaction with the Helper class in two ways: to ensure user is validated and to perform the search web service call. When the LoginForm calls `Helper.CallRemoteAuthenticateMethod` and succeeds, LoginForm will close itself and activate SearchForm. SearchForm's OnActivate should check if `Helper.IsAuthenticated` is true and if so, allow the user to enter search query particulars.
3. The SearchForm has a utility function that parses an RPC-style result string into a dictionary of values that can be used to fill the labels. Here is how that function looks:

```
private void ParseSearchResult(string resultValue)
{
    if (String.IsNullOrEmpty(resultValue))
    {
        lblAuthor.Text = lblISBN.Text = lblTitle.Text = String.Empty;
        System.Windows.Forms.MessageBox.Show("Book not found");
    }
    else
    {
        //Individual parameters will be separated by semi-colon
        string corrected = resultValue.Replace("\r\n", ";");
        string[] chunkedValues = resultValue.Split(';');
        Dictionary<string, string> parsedValues = new Dictionary<string, string>();

        foreach (string qualifier in chunkedValues)
        {
            string[] pair = qualifier.Split('=');
            if (pair.Length == 2)
            {
                //With this code, we only consider the first match
                if (!parsedValues.ContainsKey(pair[0]))
                    parsedValues.Add(pair[0], pair[1]);
            }
        }

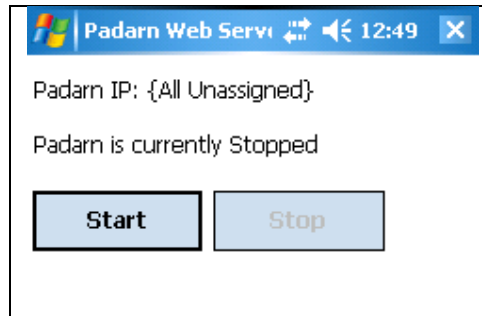
        lblTitle.Text = parsedValues["Title"];
        lblAuthor.Text = parsedValues["Author"];
        lblISBN.Text = parsedValues["ISBN"];
    }
}
```

We are now ready to see the interaction between RPC web service and the client we just created.

Exercise 5: Trying out the client

In order to validate that our .NET client works properly, we need to ensure a valid instance of Padarn is up and running; furthermore, you should validate that RPCExample.aspx has been deployed as static content to that Padarn's instance inetpub folder. Instructions for how to accomplish this are in HOL 100.

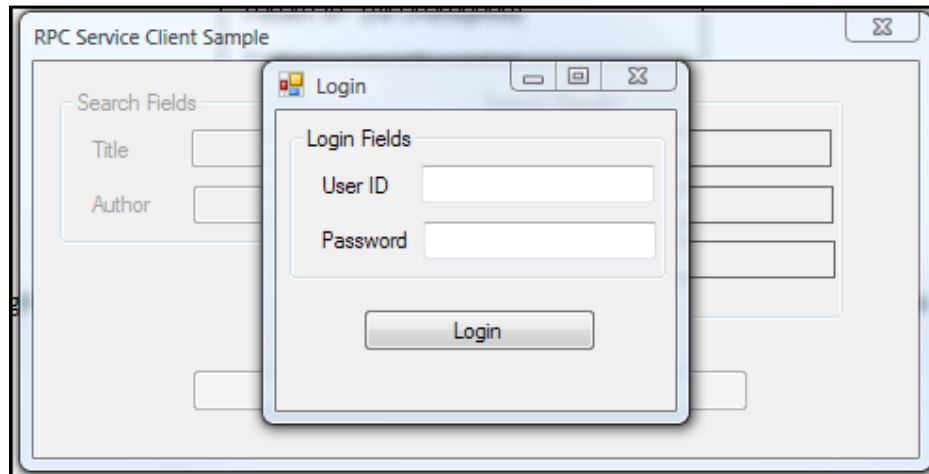
1. Initiate Padarn through the MyPadarnSolution Visual Studio 2008 solution. Right-click on the ocfttpd project and select **Debug -> Start Without Debugging** (or press Ctrl + F5). For this demonstration, we will be using the Windows Mobile 5.0 Pocket PC R2 Emulator.



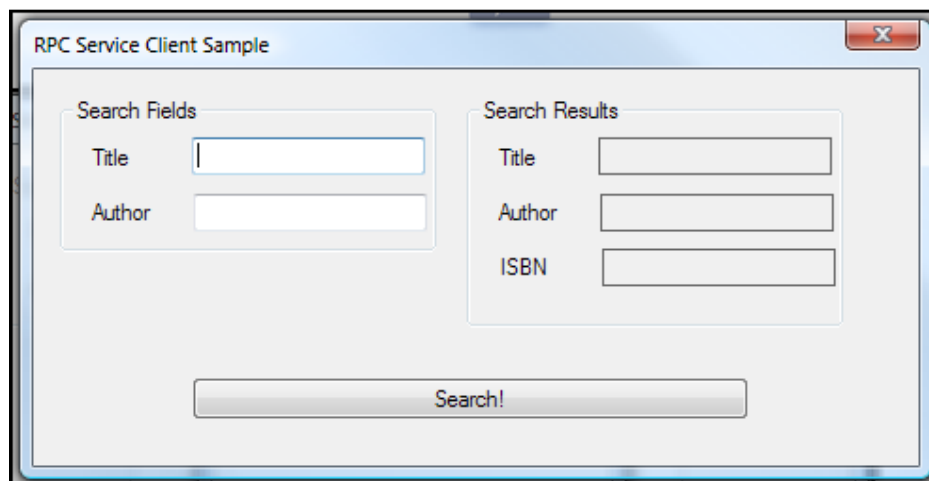
2. Press the **Start** button to initiate the Padarn instance.



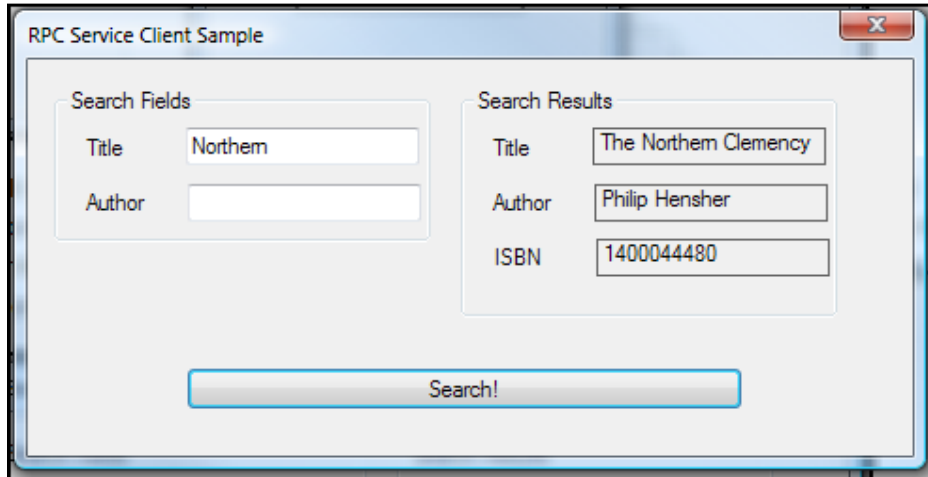
3. Right click on the RPCExampleClient project and select **Debug -> Start New Instance**



You will be prompted for a user name and password. Enter User ID = "TestUser01" and Password = "TestPW01". Press the **Login** button. Assuming your client is configured to the proper IP address of your Padarn server machine and the Padarn instance is still running, the Login screen will disappear and you'll be able to enter values into the SearchForm.

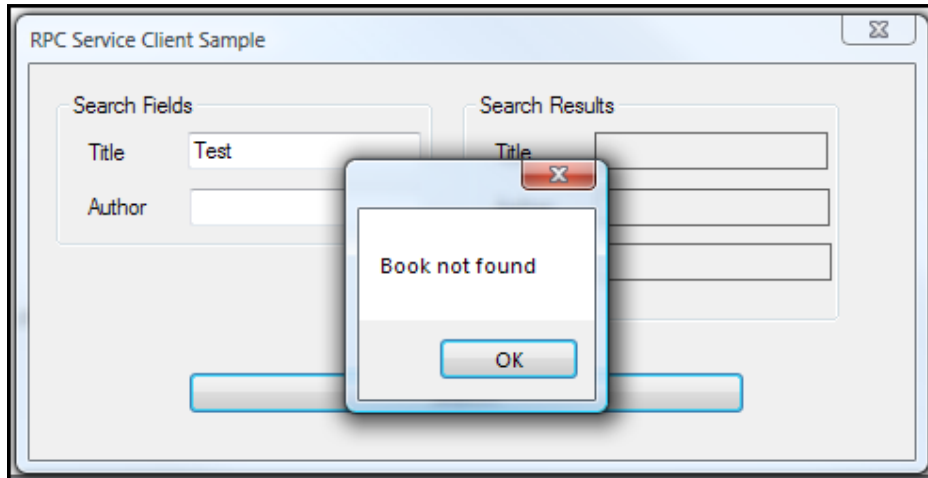


Now you can enter just a book author (partial or full name), book title (partial or full title), or both author and title. There are infinite improvements to this form that are out of scope for this hands on lab: the title or author fields could support auto-complete based upon past queries, the title or author could take in regular expressions or Boolean search characters, etc.



The screenshot shows a window titled "RPC Service Client Sample". It contains two main sections: "Search Fields" and "Search Results". In the "Search Fields" section, the "Title" field is filled with "Northern" and the "Author" field is empty. In the "Search Results" section, the "Title" field is filled with "The Northern Clemency", the "Author" field is filled with "Philip Hensher", and the "ISBN" field is filled with "1400044480". A "Search!" button is located at the bottom center of the window.

So long as you enter valid search parameters, the results from the RPC web service will appear in the appropriate labels. If no book is found, you'll see the following:



The screenshot shows the same "RPC Service Client Sample" window, but with a modal dialog box in the foreground. The dialog box is titled "Book not found" and has an "OK" button. In the background, the "Search Fields" section shows the "Title" field filled with "Test" and the "Author" field empty. The "Search Results" section is empty.

It is important to realize that the parsing logic used throughout this lab is suitable for small projects and likely would need optimization to handle heavy volume.

Hands-on Lab Summary

With minimal effort, we were able to create a practical front-end to an extensible Remote Procedure Call-enabled web service hosted within a Padarn instance. Although the client was implemented as a .NET Framework 2.0 WinForm, the same logic could exist within a basic webpage, a Windows Mobile DLL, or many other formats. Although simplifications were made in this lab that would require future optimization for greater scalability, it should be clear that Padarn leverages much of the ASP.NET development model and as such makes client implementations straightforward.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Created a .NET Framework 2.0 WinForm client to call Padarn-hosted RPC functions
- ✓ Added validation to client to ensure unnecessary network traffic is minimized
- ✓ Implemented an RPC-style parser that facilitates remote interaction with said model
- ✓ Walked through the process of starting Padarn, using the client, and seeing results come across the network

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.)