OpenNETCF Consulting

HOL P303 - Using VirtualPathProviders in Padarn to Provide REST Web Services

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 continue to understand how easy it is to develop web service-backed applications using Padarn and configure your server to in many ways act like a standard ASP.NET 2.0 instance. By developing RESTful web services under Padarn, it is feasible to expose industry-standard interfaces to server resources as well as remote data stores.

Lab Objective

Upon completion of this lab, you will be familiar with Padarn's virtual path provider support, which enables REST web services.

In this HOL, you will perform the following exercises:

- ✓ Learn how to modify the ocfhttpd.exe.config file to virtual path provider support
- ✓ Create your first virtual path provider DLL (resident in typical code-behind location)
- ✓ Expose several RESTful web services that stem from our RPC service discussion

1.1. Background

Throughout this lab, we will make frequent reference to a web service design philosophy called REST. Notice we say "style" and not strict standard. This is important and something we will take advantage of in moving forward. REST stands for *Representational State Transfer*. In layman's terms, this means every unique URL relates to a particular object or resource. You can read properties from that object using standard HTTP GET, modify using HTTP PUT, or delete using HTTP DELETE. Because of the limitations of maximum URL length and the disadvantages of passing sensitive requests through clear text address bar boxes, HTTP POST is most often used. Please review HOL P201 - Creating an RPC Web Service on Padarn – for more details about HTTP web methods and the strict W3 organization rules around those actions.

REST is important to understand because it's much more than a passing novelty. It's also platform and browser agnostic because the majority of the heavy lifting is performed on the server. All the client needs is compliance with HTTP standards that have been around for many years. REST is quite ubiquitous today – Yahoo, Flickr, bloglines, technorati, del.icio.us, Amazon, and eBay all use REST in some form or fashion. Another web service style called Simple Object Access Protocol (SOAP) is supported by Google and many large enterprises. SOAP proxies are often natively understood by smart devices since IDE support has been available since Visual Studio 2005 for Windows Mobile/CE products. Both REST and SOAP have been around for a number of years, though between the two technologies, REST is newer.

Many custom-build web service implementers targeting Microsoft platform mobile devices might run immediately to SOAP since it's possible to have J2EE on the backend and Compact Framework on the client side, for example, and have Visual Studio extract Web Service Definition Language (WSDL) feed into the client application automatically. The downside here is that SOAP, although compressible, is transmitted in a large XML document, and mobile devices are generally (relatively) slow at parsing heavily nested XML documents on the fly, especially during critical, synchronous operations. On the flip side, although REST is lightweight and only a minimal amount of data is exchanged, the parsing can be much trickier as the return result does not show up on the client as a nicely structured object or set of objects, with types native to the platform/language being used (e.g. a C# struct). It's up to the client to very carefully parse the result set and provide proper error handling in case formatting is not complete.

As REST is a URI-based methodology, it leverages really simple tools for debugging: a web browser. SOAP, on the other hand, generally requires either a SOAP toolkit or an IDE like Visual Studio. Both REST and SOAP web services can return XML, but only REST can return simple, flattened strings that are extremely quick to process. Just how big of a difference exists in transmission size? Since SOAP requires an XML wrapper for every request and response, even for a simple transaction, there is an order of magnitude difference! And again, since REST leverages HTTP web methods, which are well understood by most developers, the learning curve is quite insignificant.

One advantage of SOAP is that its usage of an XML specification ensures a strict contractual agreement between client and server, so that if data types ever change, the specification would be modified and could be ingested before the SOAP web service is called. REST web services generally public a usage website, human legible, that aides a developer. We've created such a helpful document in code for this HOL. Another disadvantage of SOAP is that all requests proceed as RPCs over standard HTTP ports. This makes it hard for a firewall administrator to discern between a simple query and a potentially dangerous data modification. REST, on the other hand, uses the HTTP standard web methods, so an HTTP GET can be treated as safe unconditionally but an HTTP DELETE might be restricted to only certain IP addresses.

RPC, as discussed in HOL 201, is a generalized means by which two discrete processes (often on different host machines) can interact with one another. RPC-style web services can still be found across the web but in many places are being overtaken by REST-style interfaces. REST is simple in that you must perform individual actions on individual sets of objects. RPC allows for command-based operations to be performed on a filtering of entities. The guidelines behind these commands are less strict, so while there are only four simple actions within REST, there are an infinite number within RPC. And in RPC, each command can do a complex sequence of events that must be carried out on the same server.

In addition to the difference in what actions can do, RPC exposes a unique set of objects and functions, which must be known prior to usage. API discovery is thus somewhat limited. With REST, however, a clearer set of resources is established and it's not necessary to have a complete view of all resources available for viewing/manipulation by the client. Instead, discovery is done by sequential hyperlinking.

For the remainder of this explanation, we will focus on REST. REST actually was stamped as a common descriptor after a Ph.D. dissertation from Roy Fielding was published. He summarizes quite well:

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

Even though REST is not a standard but more an architectural concept, it does leverage several key web standards: HTTP, URL/URI, standard document resource representations like XML, HTML, JPG, etc., and it uses MIME types like text/xml, image/gif, etc. Let's take a look at how it's used in common application. If we're using the example of a library, to get the collection of books available, we might use the following:

http://www.domain.org/library/show

The format of our URI request is simply hostname/controller/action/parameters.format

Both parameters and format are optional (implementation specific) but could come in the form of SearchTerm.xml. The term controller corresponds to object or resource. We're not specifying how the content will be returned, and we're not specifying the format (yet). We might get back something like:

```
<?xml version="1.0"?>
<p:Library xmlns:p="http://www.domain.org"
         xmlns:xlink="http://www.w3.org/1999/xlink">
      <Book id="Title 1" xlink:href="http://www.domain.org/library/Title 1"/>
      <Book id="Title 2" xlink:href="http://www.domain.org/library/Title 2"/>
      <Book id="Title 3" xlink:href="http://www.domain.org/library/Title 3"/>
      <Book id="Title 4" xlink:href="http://www.domain.org/library/Title 4"/>
</p:Library>
```

An alternative to the XML representation might be an HTML response. If we wanted to show only one book or filter our query based upon a string match, we could use the following:

http://www.domain.org/library/show/<Search Query>

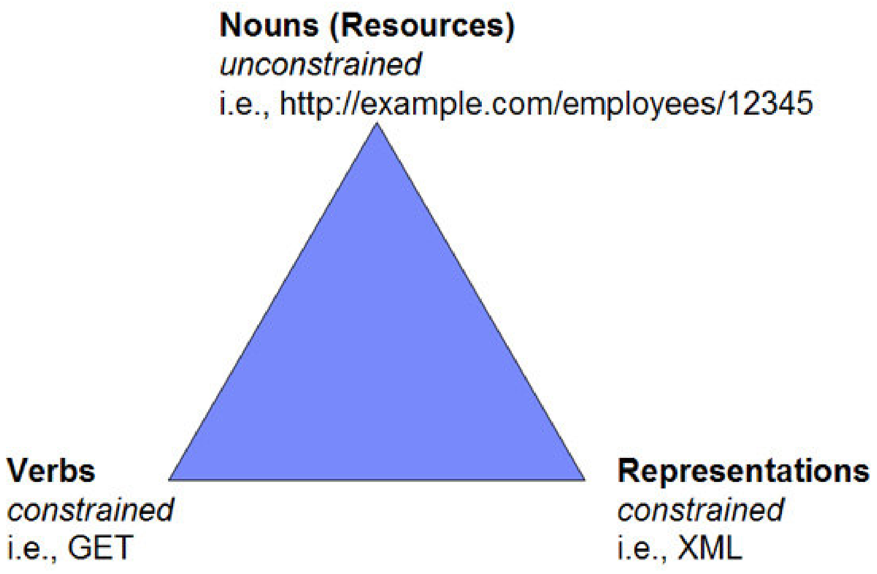
It's up to the server-side implementation to parse that search string and do whatever data store look-up is needed to service the request. It would make sense that if only one item were found, it would return some details about that entity. Again, either an XML snippet might be returned or alternatively, some HTML. Sample XML would be:

```
<Book>
      <Title>Sample Title</Title>
      <Author>Author Name</Author>
      <ISBN>1400044480</ISBN>
</Book>
```

Because the schema used is likely not published anywhere (though it could be!), there needs to be a strict agreement between client and server to ensure parsing is done properly. Sometimes it's necessary to iterate over the result before performing any real processing to ensure any changes to the API are detected.

The above illustrates a very simple operation: fetching of data. This is equivalent to HTTP GET. To modify, create, or delete objects, it's necessary to use the other HTTP web methods. Once again, it is important to review HOL P201, as that lab steps you through several real-world examples of how a web browser pushes form data to Padarn or any standards-compliant web server. For the purposes of REST, there are four actions that can be taken:

- show: This handles a GET request for the representation of one resource instance.
- create: This handles a POST request for creating a new resource instance
- update: This handles a PUT request for updating an existing resource
- destroy: This handles a DELETE request on a resource instance.



Courtesy of IBM.com, here is the "REST Triangle"

Show and destroy actions are illustrated in the HOL code. Create and update are out of scope. To wrap up our discussion of REST, here are some important characteristics when investigating REST versus other web service styles.

- Client-Server - A pull-based interaction. Consuming components, whether a web browser or web client, pull representations.
- Stateless - Each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server. So if we wanted to do authentication, like in earlier HOLs, we'd need to pass in an authentication token every time. This will improve server scalability; different servers can be used to handle different requests in a session.
- Uniformity - All resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).

- Cache - To improve network efficiency, responses must be capable of being labeled as cacheable or non-cacheable. See HOL 302 for details on this.
- Named resources - The system is comprised of resources which are named using a URL.
- Interconnected resource representations - The interexchange of the objects are formed with RLs, thereby enabling a client to progress from one state to another. REST does not require a separate resource discovery mechanism due to the use of hyperlinks in representations.
- Layered components - Intermediaries, such as proxy servers, cache servers, firewall rules, etc, can be inserted between clients to boost security, speed, and scalability.
- Long-term viability – REST is an improvement over RPC and other styles because of its capability (similar to HTML) to evolve without breaking backwards- or forwards-compatibility and its ability to add support for new content types as they are defined without dropping or reducing support for older content types, meaning APIs can grow without breaking older clients.

There are several key design considerations to bear in mind when thinking through an API set for any application. In no particular order, it's key to:

1. Figure out what objects need to be acted upon. These will act as your resources mentioned above. We care about libraries, which contain elements of type book.
2. URLs should terminate with an identifier or fully-qualified object instance name, rather than the CGI- or generic RPC-style calling convention. In other words, you want http://www.domain.com/library/show/Name rather than http://www.domain.com/library/getBook?id=Name
3. Within your implementation and depicted in your published API guide, you'll need to figure out which objects will be read-only (and thus only accept HTTP GET) and which objects will be modifiable (and thus will also accept HTTP PUT, HTTP POST, and HTTP DELETE). HTTP GET should never change the object being requested! Read-only is read-only.
4. If returning a result set in HTML or XHTML format, make sure a user can drill-down levels easily with hyperlinks. We do this in our code demo.

1.2. Virtual Paths

Let's take a step back and look at the URL above. Notice how <hostname>/library/show/ID doesn't point to an item on the web server's disk. Rather, it's a virtual file path. Padarn v1.3.5 enables virtual file providers, a feature also available in ASP.NET 2.0 and later, to allow clients to point to resources instead of just discrete files. To make this happen, a virtual path provider must be established in the Padarn configuration file (ocfhttpd.exe.config) and then must be deployed with your solution, generally in the \Windows\Inetpub\bin folder. On the full ASP.NET 2.0 framework, it's possible to have URLs passed in by a browser map to SQL data or even a ZIP file. There's nothing stopping you from linking to a SQL Server Compact Edition database, storage card resource, or pretty much anything else visible to the Padarn hosting process. In fact, virtual paths can point to anything but:

- Assemblies (DLLs or EXEs)
- Global Configuration files
- Special folder-resident files

This is because the items above are established before what's called the virtual path provider is loaded. It might be unclear why a virtual path provider is needed in the first place. After all, Padarn supports virtual folders without any additional effort, meaning <hostname>/Admin might actually point to \Windows\inetpub\folderA\Admin, not the intuitive server path of \Windows\inetpub\Admin. A virtual path provider is created to translate a user request for a particular virtual file (like a REST resource) into a request that can be understood by Padarn (such as show all books in a library).

A virtual path provider necessarily is modeled off of ASP.NET's `VirtualPathProvider` class. It's up to you to override the following functions:

- `DirectoryExists` - Gets a value that indicates whether a directory exists in the virtual file system.
- `FileExists` - Gets a value that indicates whether a file exists in the virtual file system. This is often used to determine whether the user sees a 404 error before further processing is performed.
- `GetDirectory` - Gets a virtual directory from the virtual file system.
- `GetFile` - Gets a virtual file from the virtual file system. FileExists uses this function to determine if a 404 error should be returned to the user. It might initialize some internal data stores if this is the first time the virtual path provider has been used to fetch/set data.
- `GetFileHash` - Gets a hash for the specified virtual file. This function can be implemented to ensure caching reliability – if this hash function is improperly coded, it's possible that despite caching being enabled globally through the Padarn configuration file, the resource being requested might be rebuilt from scratch every time.

Padarn makes it very easy to use a virtual path provider by doing registration on your behalf so long as the *ocfhttpd.exe.config* global configuration file is set correctly. We'll take a look at the applicable XML section later on in this HOL. An important design goal of Padarn is to make deployment as simple as possible and to allow even the most specific fine-tuning to be done in one place (the configuration file) to allow the hosting process and all static content to be deployed in one CAB. Future enhancements to Padarn will merely require the redeployment of OpenNETCF.Web.dll and the configuration file.

Two additional classes need to be discussed at this time: `VirtualFile` and `VirtualDirectory`. Since we're treating REST resources as `VirtualFile` instances, we'll focus on this class. Since this is an abstract class, the constructor and the Open method both need to be overridden. The constructor calls into the `VirtualPathUtility` class to read directory, extension, and filename information. The open method is where the resource servicing takes place – here, anything that will be displayed to the user is constructed as a Stream. The Stream returned is read-only and is seekable.

`VirtualDirectory` can be used to return the contents listing of a virtual path. The three properties that must be overridden by a `VirtualDirectory` subclass include: Files, Directories, and Children.

- Files - Returns an IEnumerable instance to enumerate all of the virtual files of the virtual directory.
- Directories - Returns an IEnumerable instance to enumerate all of the child virtual directories of the virtual directory.
- Children - Returns an IEnumerable instance to enumerate all of the virtual directories and virtual files of the virtual directory.

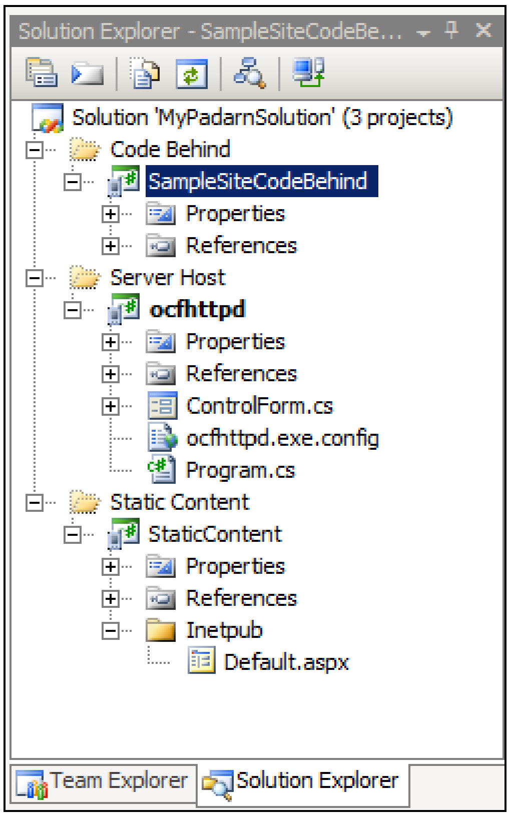
Both `VirtualFile` and `VirtualDirectory` are subclasses of `VirtualFileBase`. The IsDirectory function must be overridden, and this indicates whether the instance is a virtual file or virtual directory.

One great support point for Padarn is chaining of virtual path providers. It's simple to point Padarn to more than one virtual path provider class so that different resources can be handled by different points in your code. Any of the file or directory existence methods as well as get file or directory methods will look backwards in the chain to see if whatever is being searched on can be found in earlier-initialized providers. All of this is transparent to subclasses of the `VirtualPathProvider` class.

For the purposes of this HOL, we will only provide a single virtual path provider that handles servicing of REST-style requests for access to our library concept. In production, it might make sense to have one virtual path provider for administrative tasks, such as checking on the health of the Padarn host, and then provide another virtual path provider for whatever web services that Padarn is exposing. This keeps code clean and more easily deployed should updates be needed.

Exercise 1: Initial Solution Setup

Unlike in past HOLs, this lab does not make use of Active Server Page (ASPX) technology – this is because it will be our responsibility to tell Padarn how to parse incoming requests for our RESTful web services. All of our logic will live in the SampleSiteCodeBehind project and access to our services will take place through a web browser. Alternatively, any HTTP-enabled client can be written to achieve the same thing. To begin, please open the Visual Studio 2008 solution entitled "MyPadarnSolution.sln". Here is how the Solution Explorer displays the contents of this solution:



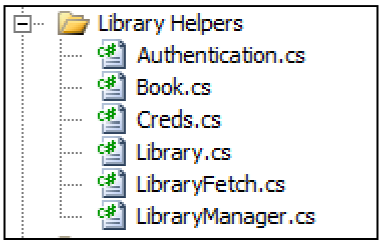
```
Solution 'MyPadarnSolution' (3 projects)
  Code Behind
    SampleSiteCodeBehind
      Properties
      References
  Server Host
    ocfhttpd
      Properties
      References
      ControlForm.cs
      ocfhttpd.exe.config
      Program.cs
  Static Content
    StaticContent
      Properties
      References
      Inetpub
        Default.aspx
```

The three projects listed are a sample Padarn web server launch utility (Smart Device project), an empty project to hold static content, and a shell for the code that will make up the virtual path provider along with its auxiliary classes.

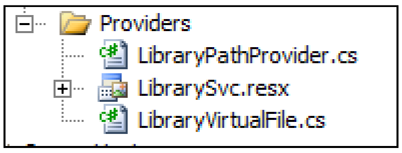
Next, you will use Visual Studio to create classes containing the code-behind logic for the RESTful web services. Unlike with full ASP.NET, registration is handled internally and merely requires configuration through the global *.config* file (see below). The Padarn server will create an instance of the main class and run the Open method of the `VirtualFile` class when a client browser (or Net client) accesses any file that doesn't point to static content on disk (like default.aspx).

The code-behind project consists of two groupings of files:

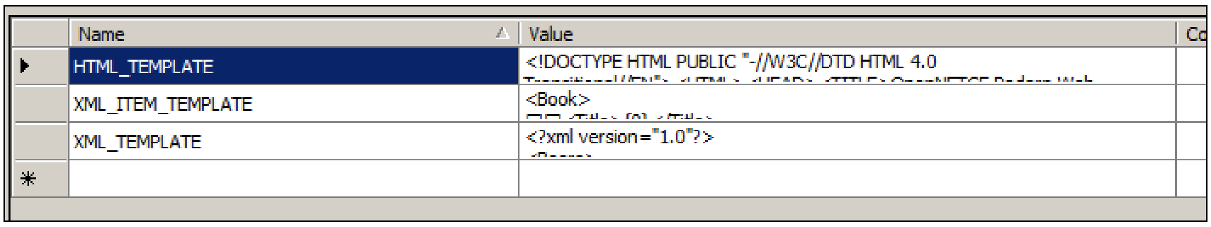
- Library Helpers: These were first created in HOL 201 for our RPC web service surrounding Library functionality. Logic exists in here to authenticate a user, search a library, and ensure that all library interaction continues only for as long as the credentials remain valid. We will not be using authentication checking for the purposes of this HOL.


Library Helpers
Authentication.cs
Book.cs
Creds.cs
Library.cs
LibraryFetch.cs
LibraryManager.cs

- Providers: This is where we will define our `LibraryPathProvider`, and `LibraryVirtualFile` classes.

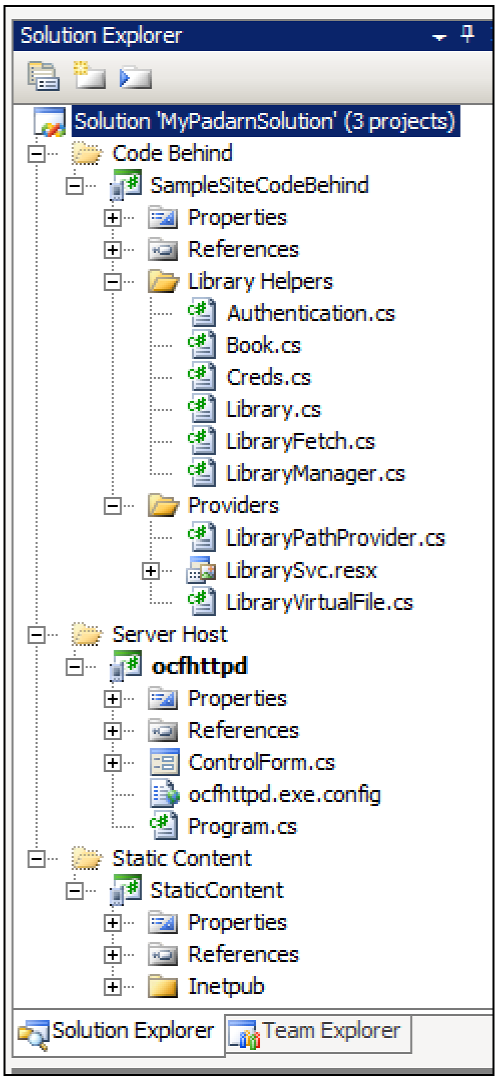

Providers
LibraryPathProvider.cs
LibrarySvc.resx
LibraryVirtualFile.cs

We also include a resource file called *LibrarySvc.resx* to include the HTML and XML formatting templates that we'll use to return content as whichever format is specified by the client.



Name	Value
HTML_TEMPLATE	<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
XML_ITEM_TEMPLATE	<Book>
XML_TEMPLATE	<?xml version="1.0"?>

When all is said and done, this is what our solution will look like:



```
Solution Explorer
Solution 'MyPadarnSolution' (3 projects)
  Code Behind
    SampleSiteCodeBehind
      Properties
      References
      Library Helpers
        Authentication.cs
        Book.cs
        Creds.cs
        Library.cs
        LibraryFetch.cs
        LibraryManager.cs
      Providers
        LibraryPathProvider.cs
        LibrarySvc.resx
        LibraryVirtualFile.cs
  Server Host
    ocfhttpd
      Properties
      References
      ControlForm.cs
      ocfhttpd.exe.config
      Program.cs
  Static Content
    StaticContent
      Properties
      References
      Inetpub
Solution Explorer | Team Explorer
```

Exercise 2: The code-behind logic for virtual path provider

The majority of the Library Helper code (under the "Library Helpers" folder in the SampleCodeBehind project) has been discussed in HOL 201. We have added a single class called LibraryManager to illustrate that our code can be easily extended to support multiple libraries. We'll only assume we have one library, indexed by the word "default", for now.

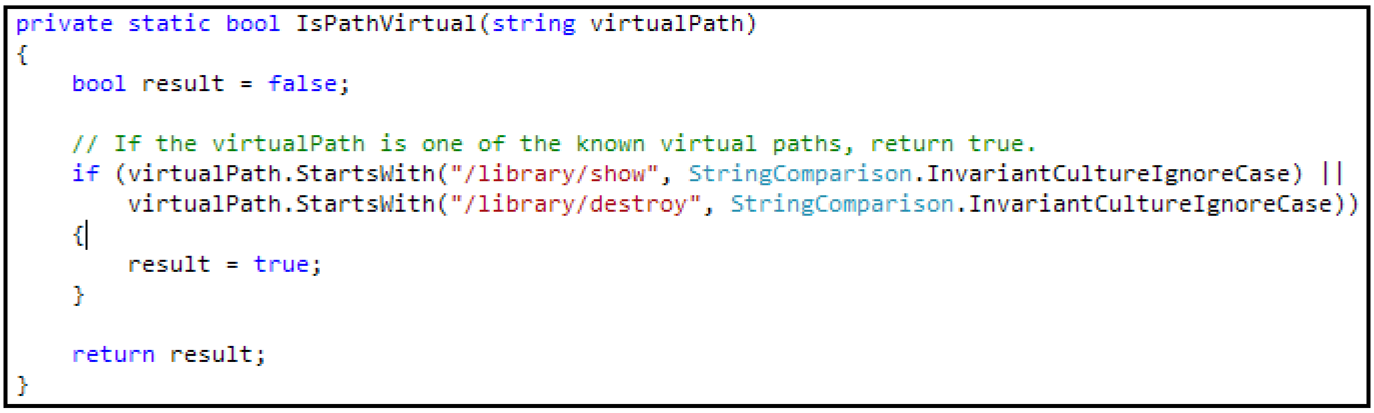
```
public class LibraryManager
{
    static Dictionary<string, Library> m_libraryList = new Dictionary<string, Library>();

    public static Library GetLibrary(string name)
    {
        if (String.IsNullOrEmpty(name))
        {
            name = "Default";
        }

        if (!String.IsNullOrEmpty(name) && !m_libraryList.ContainsKey(name))
        {
            Library newLib = new Library();
            newLib.SetupLibrary();
            m_libraryList.Add(name, newLib);
        }

        return m_libraryList[name];
```

We will use a simple Dictionary structure to maintain a collection of Library objects. Notice that the GetLibrary call ensures a properly-initialized instance is returned, so the client code is simplified. The only other change to the Library class worth describing here is to draw reference to how lookup is being performed, both with fetching and removal.

```
public ReadOnlyCollection<Book> RetrieveMatchingBooks(string title, string author, string ISBN)
{
    List<Book> bookList = new List<Book>();

    // Search through each book in the library's collection and do partial matching
    foreach (Book b in _bookCollection)
    {
        if (!String.IsNullOrEmpty(title) &&
            b._title.IndexOf(title, StringComparison.CurrentCultureIgnoreCase) != -1
            || !String.IsNullOrEmpty(author)
            && b._author.IndexOf(author, StringComparison.CurrentCultureIgnoreCase) != -1
            || !String.IsNullOrEmpty(ISBN)
            && b._ISBN == Int32.Parse(ISBN))
        {
            bookList.Add(b);
        }
    }

    return bookList.AsReadOnly();
}
```

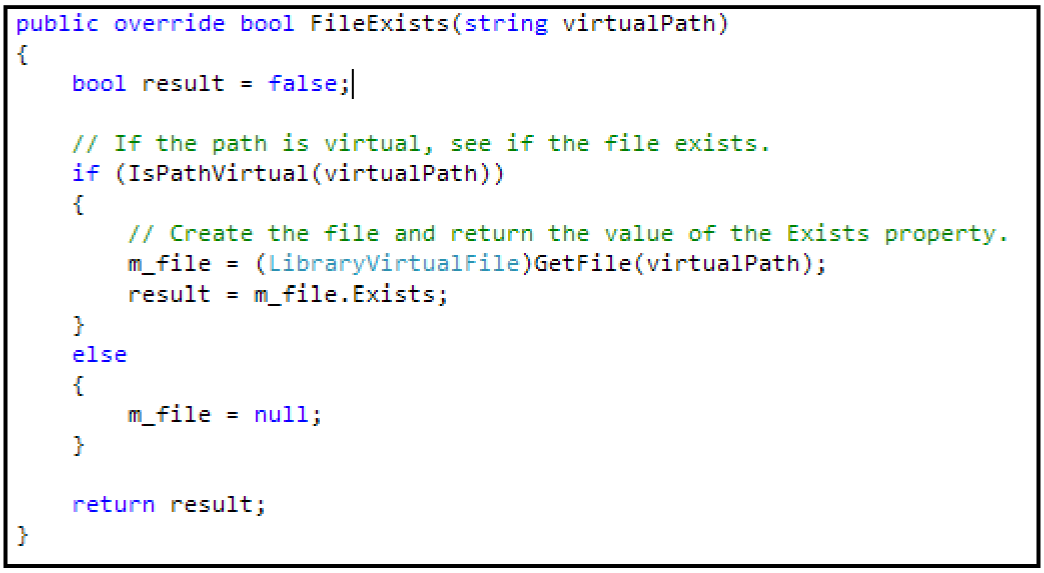
We are doing a partial-match search (case insensitive) and will return the collection of books as read-only. We do this to impart a sense of control over who has the ability to make modifications since during an update or delete, we'd like want to perform validation and not have arbitrary changes to the underlying data structure take effect. Now for the rest of the code.

1. We already mentioned the inclusion of the *LibrarySvc.resx* resource file. This is used so that if we care to change the output format of our REST service, it can be done and not impact the C# code.
2. We need to create a `VirtualPathProvider` subclass. Three functions should be defined: `IsPathVirtual`, `FileExists`, and `GetFile`. `IsPathVirtual` ensures that the execution path will continue to the next virtual provider configured if false is returned. `FileExists` ensures the resource being requested is valid and can be serviced by the virtual path provider. `GetFile` does the actual data pulling to return content back to the user through the defined `VirtualFile` class.



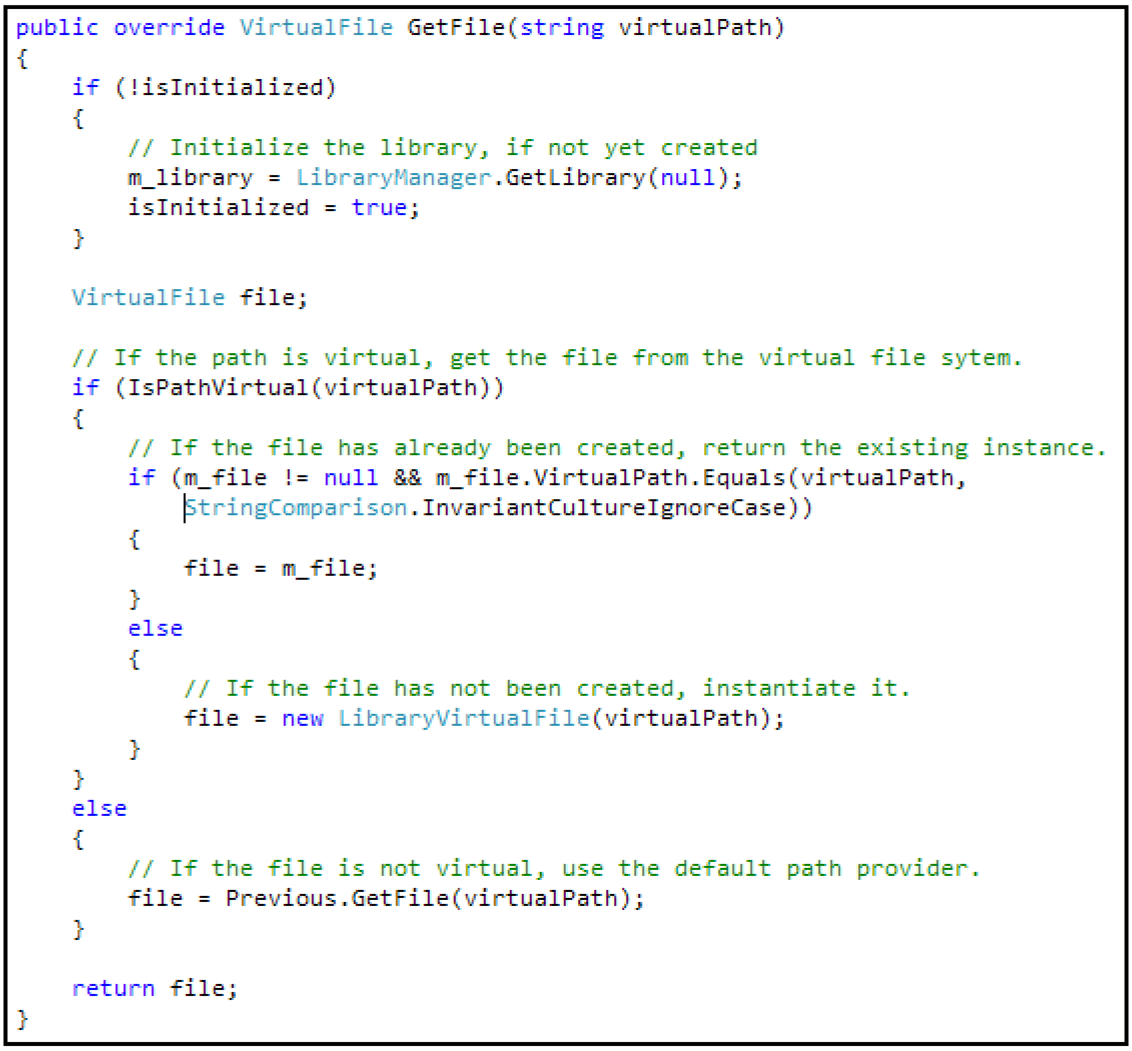
```
private static bool IsPathVirtual(string virtualPath)
{
    bool result = false;

    // If the virtualPath is one of the known virtual paths, return true.
    if (virtualPath.StartsWith("/library/show", StringComparison.InvariantCultureIgnoreCase) ||
        virtualPath.StartsWith("/library/destroy", StringComparison.InvariantCultureIgnoreCase))
    {
        result = true;
    }

    return result;
}
```

We are looking for only two actions as valid input: show and destroy.



```
public override bool FileExists(string virtualPath)
{
    bool result = false;

    // If the path is virtual, see if the file exists.
    if (IsPathVirtual(virtualPath))
    {
        // Create the file and return the value of the Exists property.
        m_file = (LibraryVirtualFile)GetFile(virtualPath);
        result = m_file.Exists;
    }
    else
    {
        m_file = null;
    }

    return result;
}
```

Only if the `GetFile` method, defined below, returns data do we allow this function to return true and thus proceed with the request of the REST service (otherwise we throw a 404 to the user).



```
public override VirtualFile GetFile(string virtualPath)
{
    if (!isInitialized)
    {
        // Initialize the library, if not yet created
        m_library = LibraryManager.GetLibrary(null);
        isInitialized = true;
    }

    VirtualFile file;

    // If the path is virtual, get the file from the virtual file sytem.
    if (IsPathVirtual(virtualPath))
    {
        // If the file has already been created, return the existing instance.
        if (m_file != null && m_file.VirtualPath.Equals(virtualPath,
            StringComparison.InvariantCultureIgnoreCase))
        {
            file = m_file;
        }
        else
        {
            // If the file has not been created, instantiate it.
            file = new LibraryVirtualFile(virtualPath);
        }
    }
    else
    {
        // If the file is not virtual, use the default path provider.
        file = Previous.GetFile(virtualPath);
    }

    return file;
}
```

The `GetFile` method calls into the various library routines and does some initialization, unless that work has already been done, in which case we return pre-existing library objects.

3. The “heavy lifting” of the virtual path provider takes place in `LibraryVirtualFile`, our subclass of `VirtualFile`. We perform the basic action here of servicing the user’s request, figuring out what to return based upon the parsed virtual path (file, extension, and directory), requested format, and of course the unique identifier. All data will be returned as a `MemoryStream`. This is done because there’s no value in persisting data to disk, although so long as an `IO.Stream` is returned, the function’s requirement is met. Actions we take:
    a. Check for extension of request: html (default) or xml.

```
switch (m_extension)
{
    case ".html":
        HttpContext.Current.Response.ContentType = "text/html";
        renderFormat = Format.Xhtml;
        break;
    case ".xml":
        HttpContext.Current.Response.ContentType = "text/xml";
        renderFormat = Format.Xml;
        break;
    default:
        HttpContext.Current.Response.ContentType = "text/html";
        renderFormat = Format.Xhtml;
        break;
}
```

b. Determine whether this is a show or destroy request and set appropriate enumeration instance.

```
if (m_virtualPath.IndexOf("destroy") != -1)
    requestedAction = Action.Destroy;
else
    requestedAction = Action.Show;
```

c. Figure out which book within the collection is being accepted. If "all" is requested, then return entire collection.

```
string bookTitle = m_fileName.StartsWith("all") ?
    String.Empty : OpenNETCF.Web.HttpUtility.UrlDecode(fileNameArray[0]);
```

d. Remove any book requested to be deleted from collection.

```
if (requestedAction == Action.Destroy && !String.IsNullOrEmpty(bookTitle))
{
    LibraryManager.GetLibrary(null).RemoveMatchingBooks(bookTitle, null, null);
    bookTitle = String.Empty;
}
```

e. Return collection output in requested format.

```
switch (renderFormat)
{
    case Format.Xhtml:
        stream = GetLibraryInfoAsHtml(bookTitle);
        break;
    case Format.Xml:
        stream = GetLibraryInfoAsXml(bookTitle);
        break;
}

return stream;
```

4. HTML output is achieved using a few OpenNETCF.Web.HTML classes, which are conveniently used for any rich document output. The general flow of the `GetLibraryInfoAsHtml` function is as follows.

   a. Get applicable collection of read-only books

```
ReadOnlyCollection<Book> foundBooks = (String.IsNullOrEmpty(title) ?
    LibraryManager.GetLibrary(null).RetrieveAllBooks() :
    LibraryManager.GetLibrary(null).RetrieveMatchingBooks(title, null, null));
```

   b. Create a row for each found book. Use hyperlinking for each valid book.

```
foreach (Book b in foundBooks)
{
    Row dataRow = new Row(CssClass);
    Hyperlink href = new Hyperlink(b._title, String.Format("/library/show/{0}/", b._title));
    dataRow.Cells.Add(href);
    dataRow.Cells.Add(new RawText(b._author.ToString()));

    // Only show ISBN if this is a details click
    if (!String.IsNullOrEmpty(title))
        dataRow.Cells.Add(new RawText(b._ISBN.ToString()));

    dataTable.Rows.Add(dataRow);
}
```

   c. Insert generated HTML string into HTML template defined by our resource file and return as a stream.

```
string htmlString = String.Format(LibrarySvc.HTML_TEMPLATE, dataTable.OuterHtml);

MemoryStream stream = new MemoryStream();
byte[] buffer = Encoding.ASCII.GetBytes(htmlString);
stream.Write(buffer, 0, buffer.Length);
stream.Position = 0;

return stream;
```

5. XML output is a bit simpler. We query our books in the same fashion as with HTML output, but instead of creating HTML rows, we build up a list of XML elements using `StringBuilder` again using a template defined in our resource file.

```
foreach (Book b in foundBooks)
{
    items.AppendFormat(System.Globalization.CultureInfo.InvariantCulture,
        LibrarySvc.XML_ITEM_TEMPLATE, b._title, b._author, b._ISBN);
}
```

That's it! You can debug this HOL and watch as the request enters the `LibraryPathProvider` and funnels through to the `LibraryVirtualFile` class to exit through its Open method, returning HTML or XML output to the user.

Exercise 3: Configuring Padarn to use Virtual Path Providers

Padarn v1.3.5 and later versions support virtual path providers. Because this setting exists in the global settings for your Padarn instance, it might be sometimes necessary to provide multiple provider references. As a result, Padarn supports chaining of virtual path providers, just like ASP.NET 2.0. Be warned – because the Open function of each provider must be executed until one returns true, there could be a bit of a performance hit if multiple providers with complex initialization logic are included.

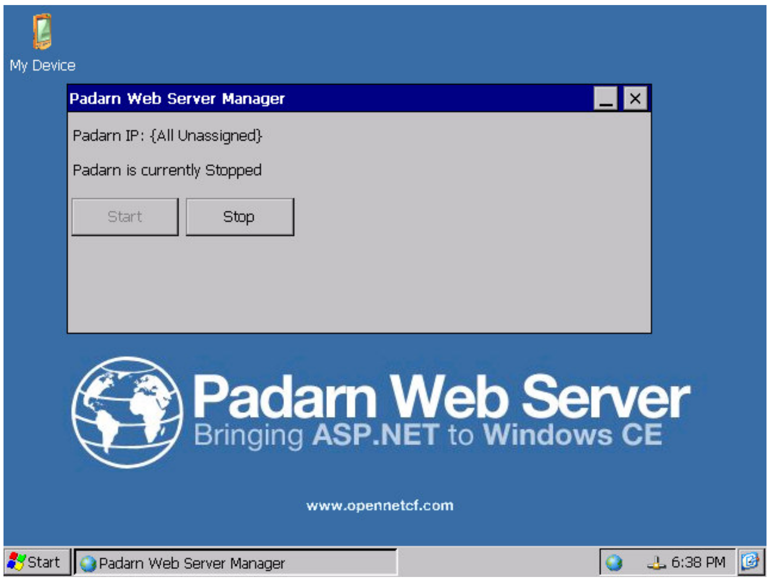
```
    <!--
    Configure a VirtualPathProvider

    For each provider, add a new Provider element, specifying
    the fully-qualifed name of the VirtualPathProvider type.

  -->
    <VirtualPathProviders>
        <Provider Type="SampleSite.Providers.LibraryPathProvider, SampleSite" />
    </VirtualPathProviders>
```

As is usually the case, a Virtual Path Provider section requires one element per provider. You merely specify the type as "Namespace.Class, DLLName". In our case, our `VirtualFile` subclass can be found in the SampleSite.Providers namespace with class name `LibraryPathProvider`. This class is built as part of the *SampleSite.dll*.

Exercise 4: Verify Virtual Path Provider Behavior

In order to validate that our caching settings have taken hold, we need to ensure a valid instance of Padarn is up and running.

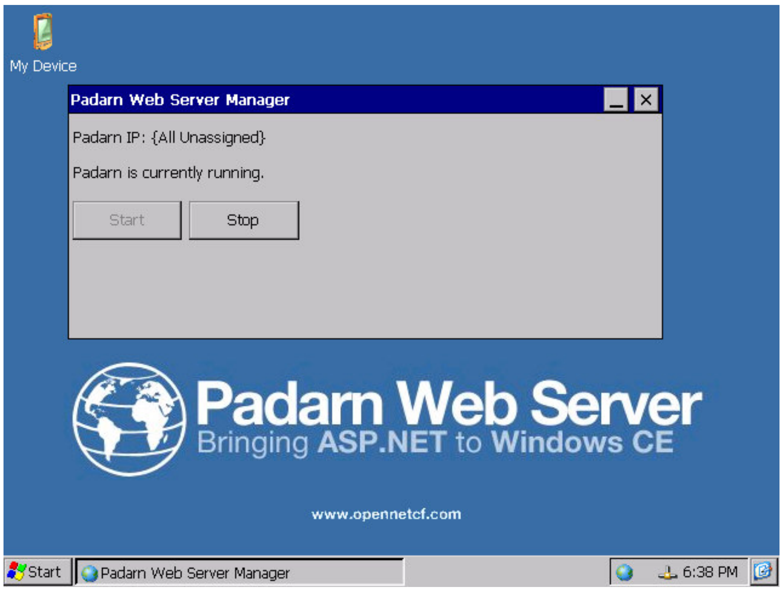
1. Initiate Padarn through the MyPadarnSolution Visual Studio 2008 solution. Right-click on the ocfhttpd project and select **Debug -> Start Without Debugging** (or press Ctrl + F5). For this demonstration, we will be using the Windows Mobile 5.0 Pocket PC R2 Emulator.


My Device
Padarn Web Server Manager
Start
Stop
Padarn Web Server
Bringing ASP.NET to Windows CE
www.opennetcf.com
Start
Padarn Web Server Manager
6:38 PM

Padarn IP: {All Unassigned}

Padarn is currently Stopped

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


My Device
Padarn Web Server Manager
Start
Stop
Padarn Web Server
Bringing ASP.NET to Windows CE
www.opennetcf.com
Start
Padarn Web Server Manager
6:38 PM

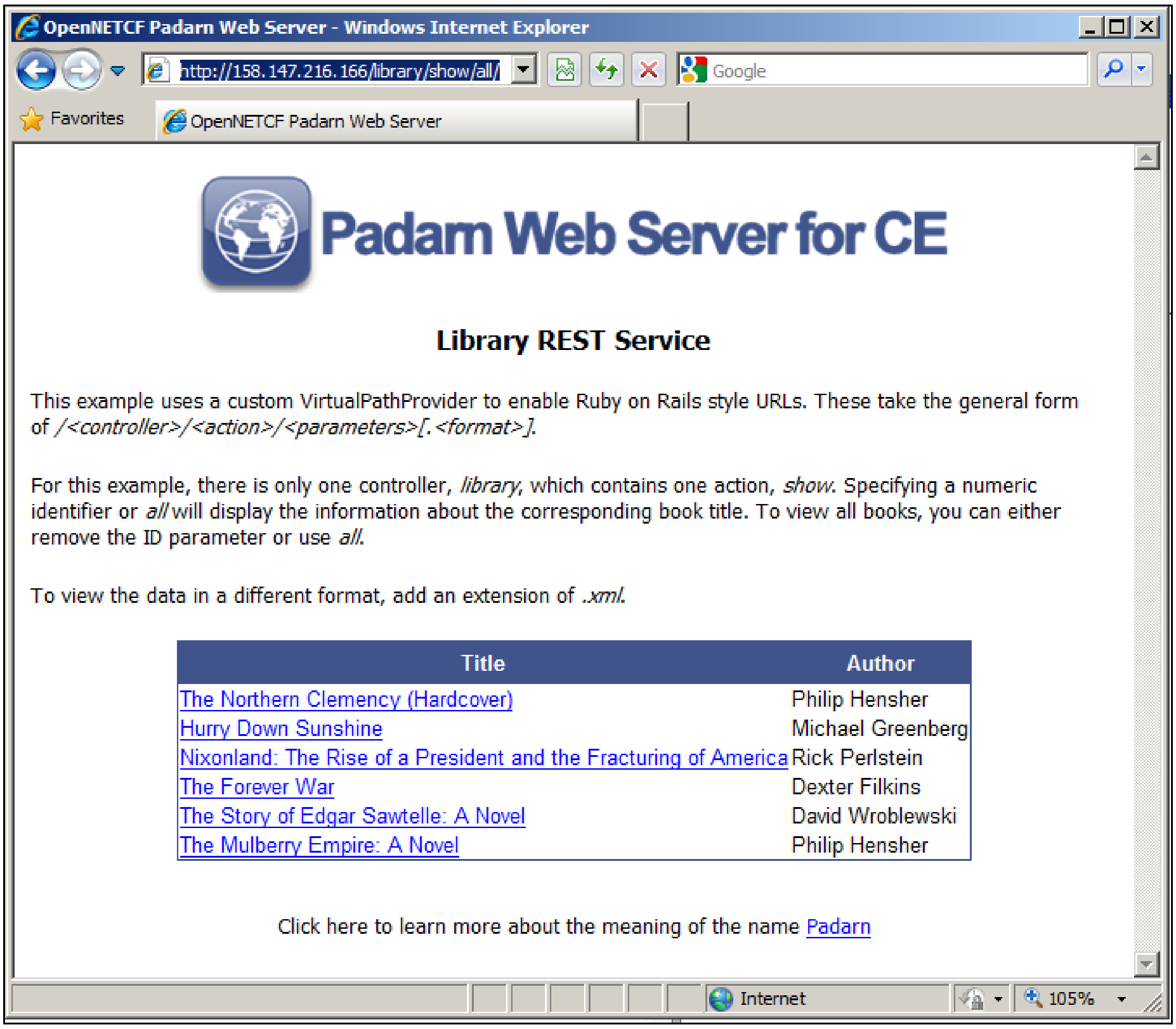
Padarn IP: {All Unassigned}

Padarn is currently running.

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. To get started, let's see what happens when we request all books in the default (HTML format). Our URL we enter will be http://<Padarn IP>/library/show/all.


OpenNETCF Padarn Web Server - Windows Internet Explorer
http://158.147.216.166/library/show/all/
Google
Favorites
OpenNETCF Padarn Web Server
Padarn Web Server for CE
Internet
105%

Library REST Service

This example uses a custom VirtualPathProvider to enable Ruby on Rails style URLs. These take the general form of */<controller>/<action>/<parameters>[.<format>]*.

For this example, there is only one controller, *library*, which contains one action, *show*. Specifying a numeric identifier or *all* will display the information about the corresponding book title. To view all books, you can either remove the ID parameter or use *all*.

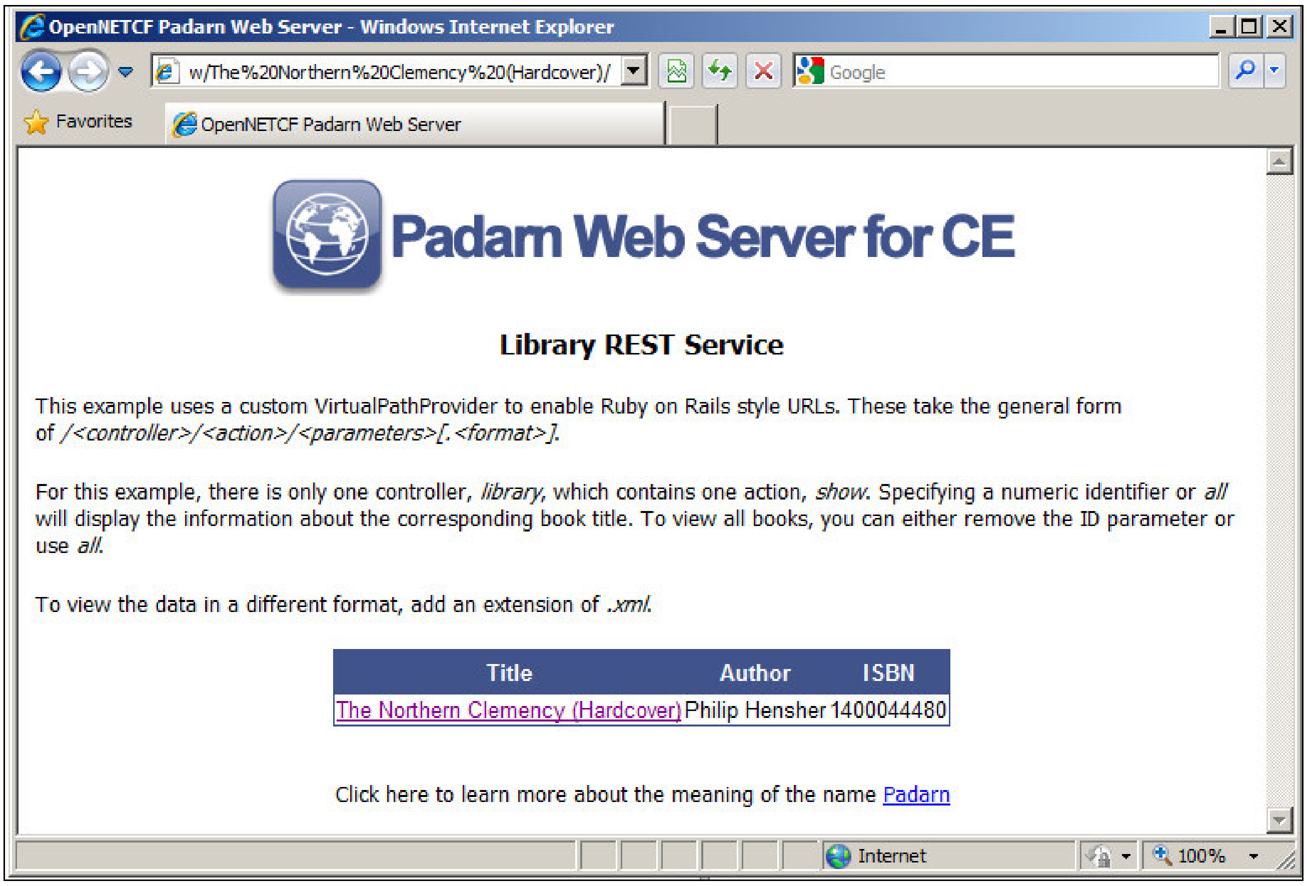
To view the data in a different format, add an extension of *.xml*.

Title	Author
The Northern Clemency (Hardcover)	Philip Hensher
Hurry Down Sunshine	Michael Greenberg
Nixonland: The Rise of a President and the Fracturing of America	Rick Perlstein
The Forever War	Dexter Filkins
The Story of Edgar Sawtelle: A Novel	David Wroblewski
The Mulberry Empire: A Novel	Philip Hensher

Click here to learn more about the meaning of the name Padarn

Notice the nice formatting as well as an explanation of the purpose of this REST service.

5. By clicking on any of the individual links, you can drill down into the details. This hyperlinking is an important aspect of RESTful web services.


OpenNETCF Padarn Web Server - Windows Internet Explorer
w/The%20Northern%20Clemency%20(Hardcover)/
Google
Favorites
OpenNETCF Padarn Web Server
Padarn Web Server for CE
Internet
100%

Library REST Service

This example uses a custom VirtualPathProvider to enable Ruby on Rails style URLs. These take the general form of */<controller>/<action>/<parameters>[.<format>]*.

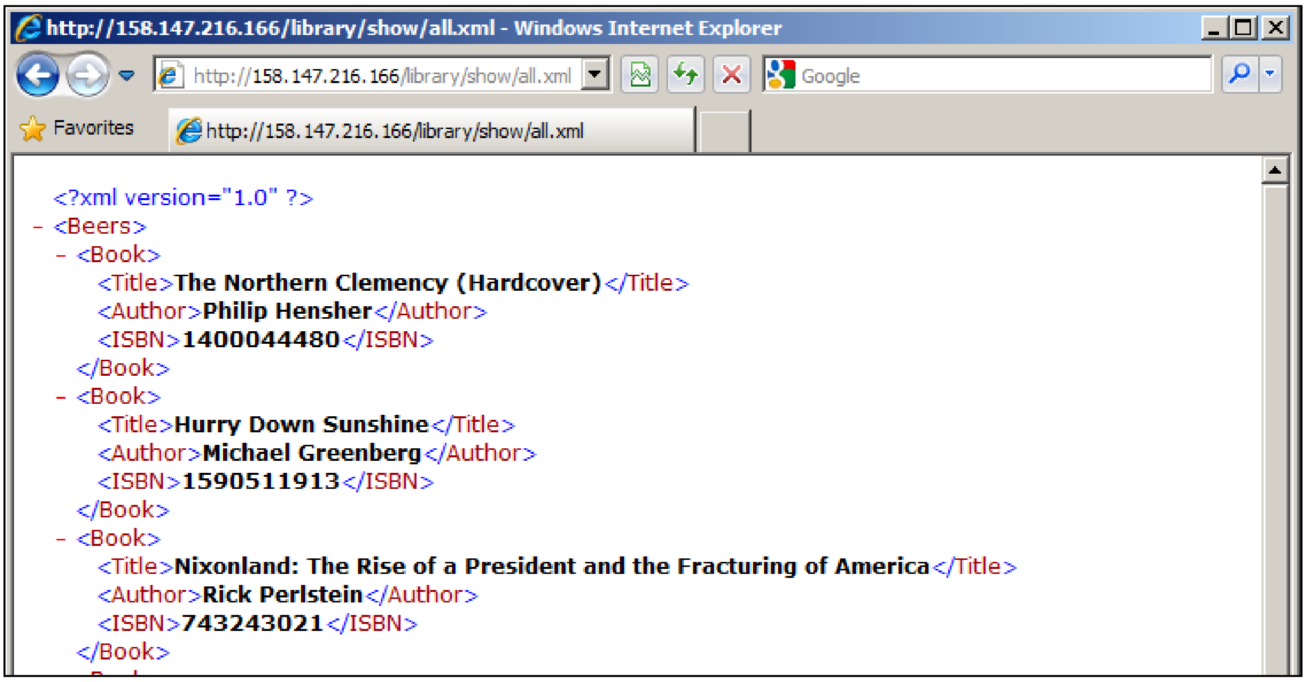
For this example, there is only one controller, *library*, which contains one action, *show*. Specifying a numeric identifier or *all* will display the information about the corresponding book title. To view all books, you can either remove the ID parameter or use *all*.

To view the data in a different format, add an extension of *.xml*.

Title	Author	ISBN
The Northern Clemency (Hardcover)	Philip Hensher	1400044480

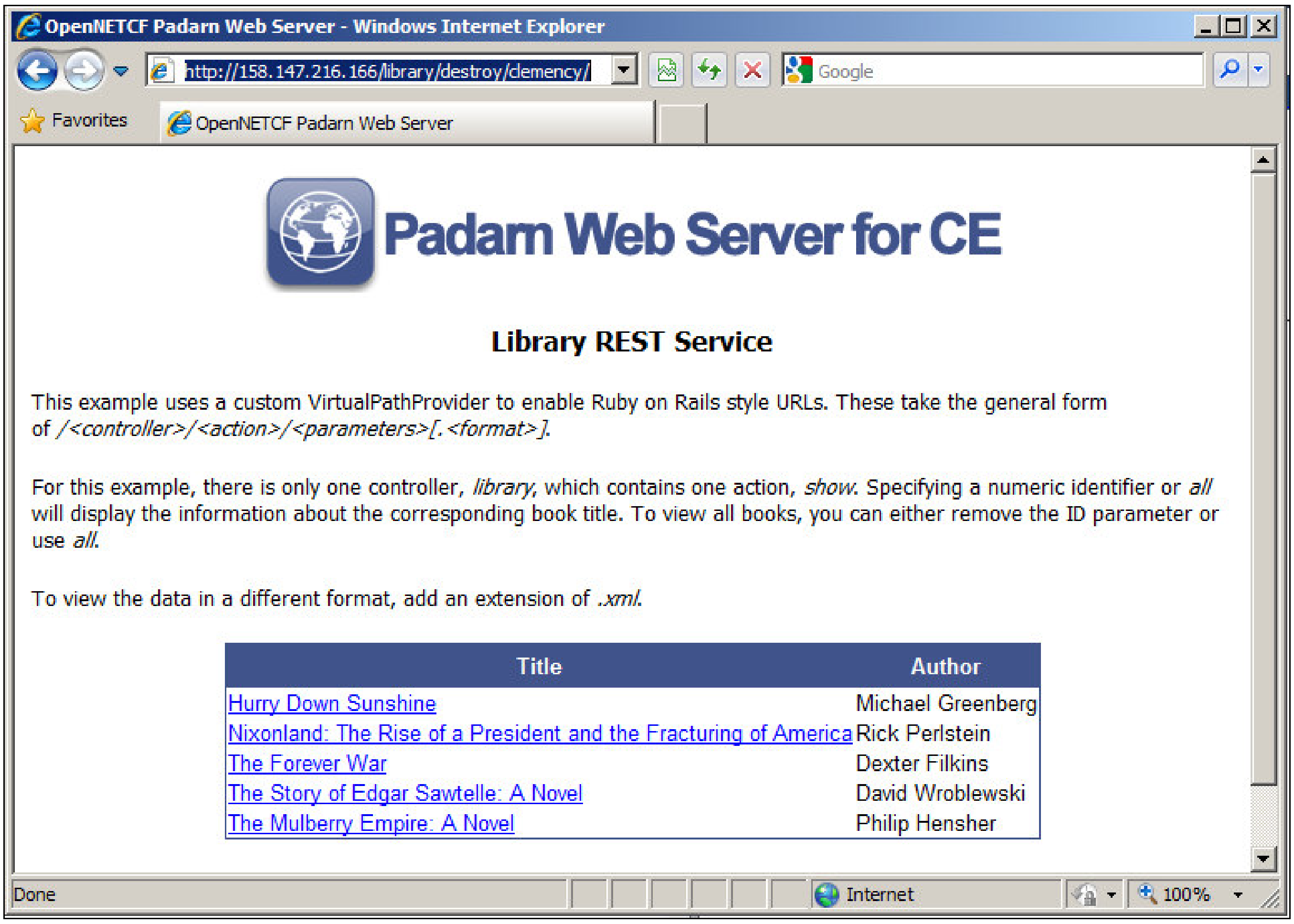
Click here to learn more about the meaning of the name Padarn

6. Now let's request the library contents and request an XML output. We'll use a different URL, http://<Padarn IP>/library/show/all.xml.


http://158.147.216.166/library/show/all.xml - Windows Internet Explorer
http://158.147.216.166/library/show/all.xml
Google
Favorites
http://158.147.216.166/library/show/all.xml

```
  <?xml version="1.0" ?>
- <Beers>
  - <Book>
      <Title>The Northern Clemency (Hardcover)</Title>
      <Author>Philip Hensher</Author>
      <ISBN>1400044480</ISBN>
    </Book>
  - <Book>
      <Title>Hurry Down Sunshine</Title>
      <Author>Michael Greenberg</Author>
      <ISBN>1590511913</ISBN>
    </Book>
  - <Book>
      <Title>Nixonland: The Rise of a President and the Fracturing of America</Title>
      <Author>Rick Perlstein</Author>
      <ISBN>743243021</ISBN>
    </Book>
```

7. Finally, let's see what happens when we use the destroy action to remove any item that contains the text "Clemency" in the title. The URL would be http://<Padarn IP>/library/destroy/clemency.


OpenNETCF Padarn Web Server - Windows Internet Explorer
http://158.147.216.166/library/destroy/clemency/
Google
Favorites
OpenNETCF Padarn Web Server
Padarn Web Server for CE
Done
Internet
100%

Library REST Service

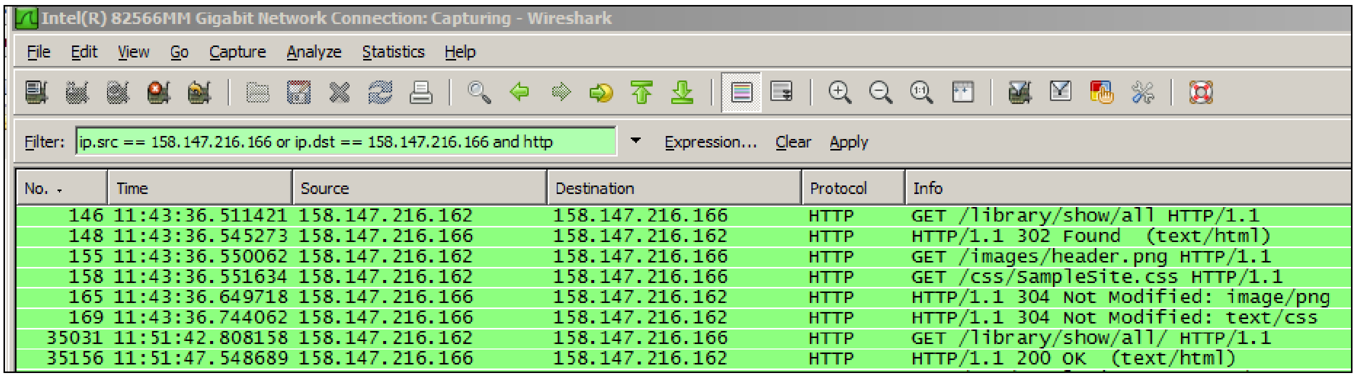
This example uses a custom VirtualPathProvider to enable Ruby on Rails style URLs. These take the general form of */<controller>/<action>/<parameters>[.<format>].*

For this example, there is only one controller, *library*, which contains one action, *show*. Specifying a numeric identifier or *all* will display the information about the corresponding book title. To view all books, you can either remove the ID parameter or use *all*.

To view the data in a different format, add an extension of *.xml*.

Title	Author
Hurry Down Sunshine	Michael Greenberg
Nixonland: The Rise of a President and the Fracturing of America	Rick Perlstein
The Forever War	Dexter Filkins
The Story of Edgar Sawtelle: A Novel	David Wroblewski
The Mulberry Empire: A Novel	Philip Hensher

Once the page fully loads, we can use a network sniffer tool such as Wireshark (information here) to determine all headers exchanged between Padarn and the web browser. Below is a brief snapshot of what we would expect to see on the server. The following events have taken place:

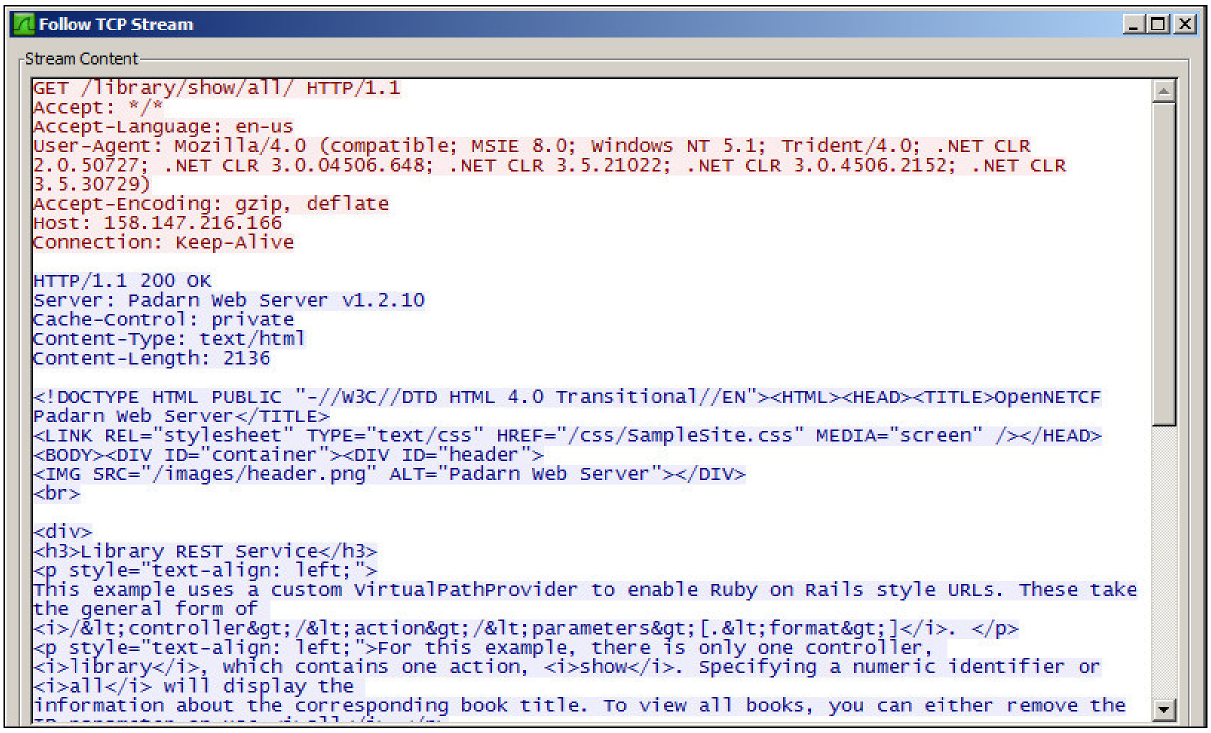

Intel(R) 82566MM Gigabit Network Connection: Capturing - Wireshark
File
Edit
View
Go
Capture
Analyze
Statistics
Help
Expression...
Clear
Apply

Filter: ip.src == 158.147.216.166 or ip.dst == 158.147.216.166 and http

No.	Time	Source	Destination	Protocol	Info
146	11:43:36.511421	158.147.216.162	158.147.216.166	HTTP	GET /library/show/all HTTP/1.1
148	11:43:36.545273	158.147.216.166	158.147.216.162	HTTP	HTTP/1.1 302 Found (text/html)
155	11:43:36.550062	158.147.216.162	158.147.216.166	HTTP	GET /images/header.png HTTP/1.1
158	11:43:36.551634	158.147.216.162	158.147.216.166	HTTP	GET /css/SampleSite.css HTTP/1.1
165	11:43:36.649718	158.147.216.166	158.147.216.162	HTTP	HTTP/1.1 304 Not Modified: image/png
169	11:43:36.744062	158.147.216.166	158.147.216.162	HTTP	HTTP/1.1 304 Not Modified: text/css
35031	11:51:42.808158	158.147.216.162	158.147.216.166	HTTP	GET /library/show/all/ HTTP/1.1
35156	11:51:47.548689	158.147.216.166	158.147.216.162	HTTP	HTTP/1.1 200 OK (text/html)

1. The content at /library/show/all is requested (HTTP GET)

2. Server returns a successful HTTP 302/Found response
3. Server indicates that web browser cache of header PNG is current
4. Server indicates that web browser cache of cascading style sheet is current
5. The REST resource, as requested, is downloaded to the browser.



```
GET /library/show/all/ HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022; .NET CLR 3.0.4506.2152; .NET CLR
3.5.30729)
Accept-Encoding: gzip, deflate
Host: 158.147.216.166
Connection: Keep-Alive

HTTP/1.1 200 OK
Server: Padarn Web Server v1.2.10
Cache-Control: private
Content-Type: text/html
Content-Length: 2136

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"><HTML><HEAD><TITLE>OpenNETCF
Padarn Web Server</TITLE>
<LINK REL="stylesheet" TYPE="text/css" HREF="/css/SampleSite.css" MEDIA="screen" /></HEAD>
<BODY><DIV ID="container"><DIV ID="header">
<IMG SRC="/images/header.png" ALT="Padarn Web Server"></DIV>
<br>

<div>
<h3>Library REST Service</h3>
<p style="text-align: left;">
This example uses a custom VirtualPathProvider to enable Ruby on Rails style URLs. These take
the general form of
<i>/&lt;controller&gt;/&lt;action&gt;/&lt;parameters&gt;[.&lt;format&gt;]</i>. </p>
<p style="text-align: left;">For this example, there is only one controller,
<i>library</i>, which contains one action, <i>show</i>. Specifying a numeric identifier or
<i>all</i> will display the
information about the corresponding book title. To view all books, you can either remove the
```

If we look at the entire TCP stream of an image capture, we see that cache-control setting is left at private and the standard HTML content follows after the Padarn header content.

Hands-on Lab Summary

Virtual path provider support is another step forward for the Padarn offering in that it allows for development, publication, and management of potentially sophisticated web services that connect to a simple Windows CE device in infinite ways – whether it's a storage card, SQL Server Compact Edition database, local file, or networked resource. Just like with ASP.NET 2.0, this is fully abstracted from the user, and the interaction is akin to browsing Wikipedia or one of millions of Internet sites with a virtualized access methodology (where pre-formed HTML is rarely pushed to the user).

Unlike with ASP.NET 2.0, though, the virtual path provider is registered automatically on your behalf so long as Padarn is properly configured using the guidelines above.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Made a changes to the Padarn *ocfhttpd.exe.config* to enable a single virtual path provider
- ✓ Re-used a library collection routine to fetch books based upon search criteria
- ✓ Built a `VirtualPathProvider` and `VirtualFile` subclass to implement a RESTful web service
- ✓ Walked through the basic show and destroy operations of our service as an end user

In summation, virtual path providers can and should be used to serve dynamic content to users where the local storage paths change or when the user can't be held accountable for recalling fully-resolved URLs.