

# **HOL P112 - Understanding Padarn's HttpRequest and HttpResponse Objects**

## Table of Contents

Table of Contents .....	2
HOL Requirements .....	3
Summary.....	4
Lab Objective .....	4
Background.....	4
Request Object .....	5
Response Object.....	13
Exercise 1: Adding our HeaderWork ASPX .....	15
Exercise 2: Traversing HTTP Headers .....	18
Exercise 3: Inspecting browser capabilities of client.....	19
Exercise 4: Forming the Response.....	20
Exercise 5: Putting it all together! .....	21
Hands-on Lab Summary .....	23

## 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 the OpenNETCF Padarn implementation's of ASP.NET Response and Request objects allow your web applications, through code-behind logic, can interact with a web browser to return appropriate results. A wealth of information about how and what a client request is demanding from the server can be parsed from the HTTP Header included with every request; the [HttpRequest](#) contains accessors that make it more convenient for accessing these values.

## Lab Objective

Upon completion of this lab, you will be familiar with the exchange of data between client and server by way of a Page\_Load action.

In this HOL, you will perform the following exercises:

- ✓ Become comfortable with standard HTTP header contents
- ✓ Read about the OpenNETCF [HttpRequest](#) and [HttpResponse](#) classes
- ✓ See live Immediate Window output from various properties of these two classes
- ✓ Check out sample code that parses the contents of an HTTP header
- ✓ Look at the Browser property of [HttpRequest](#) and see how it relates to browser definition files

## Background

So far in this tutorial series, you've seen how to lock down a Padarn server instance with various security routines – digest authentication as well as Secure Socket Layer (SSL) protocol. Here, we will continue our conversation about general web technologies, starting from the basics of the HTTP 1.1 specification, available in its full glory at [W3.org](http://W3.org).

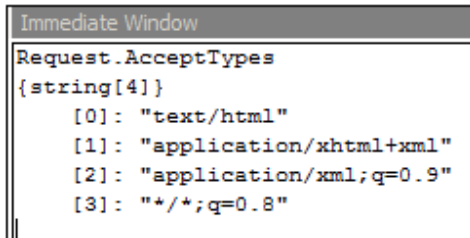
ASP.NET provides a strictly defined model of interaction between server and client browser. While in the past, most browsers were comfortable in making static requests for pre-built HTML, images, and other document types, ASP.NET provides for a much more dynamic inter-exchange that allows a the client to better steer the content to be returned and rendered by the browser. Two of the most important constructs within ASP.NET are the Request and Response objects. The flow of a typical ASP.NET Active Server Page (ASP or ASPX extension in the browser address bar) is:

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

## HttpRequest Object Discussion

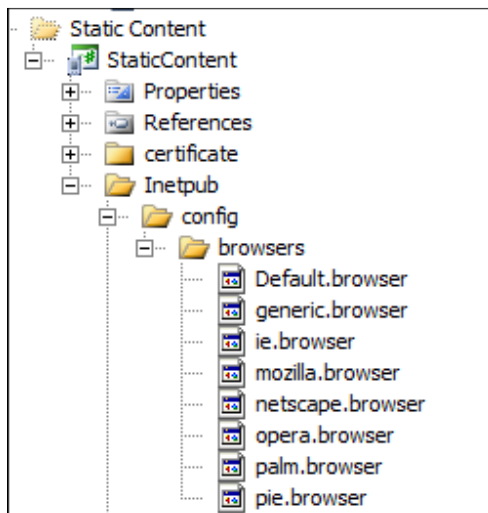
At this point, it'd be appropriate to take a closer look at what the Request object can contain. Here is a listing of the more important properties supported by Padarn (1.1 Series).

1. **AcceptTypes.** This gets a string array of client-supported MIME accept types. This is useful to help determine what content type the user might be expected, based upon whether the request is made via a full web browser or a simple HTTP web service consumer. Here is sample output for what you might get:



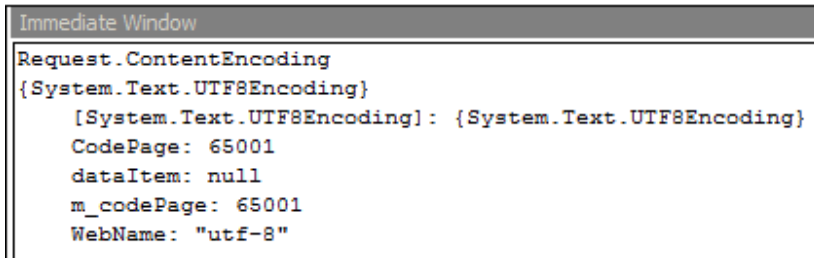
```
Immediate Window
Request.AcceptTypes
{string[4]}
[0]: "text/html"
[1]: "application/xhtml+xml"
[2]: "application/xml;q=0.9"
[3]: "*/*;q=0.8"
```

2. **Browser.** This property is dependent on Padarn's recognition of a particular client browser type. As part of the sample code bundled with the HOL, a variety of standard web browser definition files have been included. They can be found in the Static Content project under the \Inetpub\config\browser.



Although it's beyond the scope of this HOL to provide details on the structure of this document, there are a few commonalities that should be included: an identification region that provides a regular expression to determine whether a particular definition file should be loaded based upon the userAgent string passed in by the user, followed by basic browser versioning and OS-related features, followed by individual capability sections for all known versions of the browser series. For example, you'd typically only require one browser definition file for all IE versions, from v1.0 to v8.0. We will leverage the Browser property later on to see the detected capabilities of a user's browser and present the output through a Padarn webpage.

3. **ContentEncoding.** This specifies the HTTP character set of the message body being sent to Padarn. When present, this indicates what encoding schemes might have been applied to the entity-body, giving directions for what needs to be done to properly decode and obtain the media type referenced by the **ContentType** header field. If multiple encodings have been applied to an entity, the content codings must be listed in the order in which they were applied. Here is sample output from the **ContentEncoding** property:



```
Immediate Window
Request.ContentEncoding
{System.Text.UTF8Encoding}
[System.Text.UTF8Encoding]: {System.Text.UTF8Encoding}
CodePage: 65001
dataItem: null
m_codePage: 65001
WebName: "utf-8"
```

4. **ContentLength.** This field indicates the size of the entity-body sent to the server. This value should pretty much always be set since it's generally computable based upon the type of MIME content. If this is a non-form based request, for example if the page is being requested without any input data whatsoever, then the length will be 0. Otherwise, a positive value is to be expected.
5. **ContentType.** Simply put, this is the media type contained within the entity-body sent to the web server. Again, if the page is being requested with no data whatsoever, then the value here will be null.
6. **Cookies.** Although cookies do not show up in the HTTP 1.1 header definition, they are of critical importance in production-ready web applications and Padarn has made it straightforward to get, set, and delete client cookie files. **Cookies** is a [NameObjectCollectionBase](#) and exposes all of the key properties of cookie data, specifically the **Keys** string array. [HOL 203 \(Setting Client Browser Cookies from a Padarn Page\)](#) goes into detail explaining how cookie data is exchanged as well as how to leverage a client's cookie data to redirect and maintain state between sessions even when client and server are disconnected and "forget" about one another. Please take the time to review this HOL. Cookies support in Padarn closely matches the model provided in ASP.NET 2.0.
7. **Files.** Padarn supports batch file uploads to simplify the act of transmitting large amounts of data and storing onto the server that content. Rather than streaming files as raw binary data, files can be transmitted in a more appropriate format and can be saved to the local file store without any heavy lifting or usage of the various [StreamWriter](#) classes in .NET. [HOL 103 \(Virtual Directories and File Uploading in Padarn\)](#) goes into the details of file uploading through Padarn.
8. **Form.** Form data is the heart of data exchange between ASP.NET server and client. Whether the user is filling out a textbox, clicking on a submit button, or requesting a refresh of a dynamic view, the browser will pass along variables through the **Request** and result in a **Form** object being made available to the receiver with those values in place. It's important to note that only

POST methods will include Form data (this is because when using GET, the variables are stored within the redirect URL).

If the GET method is used, a [NameValueCollection](#) will be available containing a series of name/value pairs. HOL 200 - Passing Data Between Padarn Pages - will go into further detail about the Form property.

9. Headers. The headers of an [HttpRequest](#) provide another [NameValueCollection](#) for string-based access to key attributes of a browser request. Here is a list of typical HTTP headers that might show up in a Padarn web application:

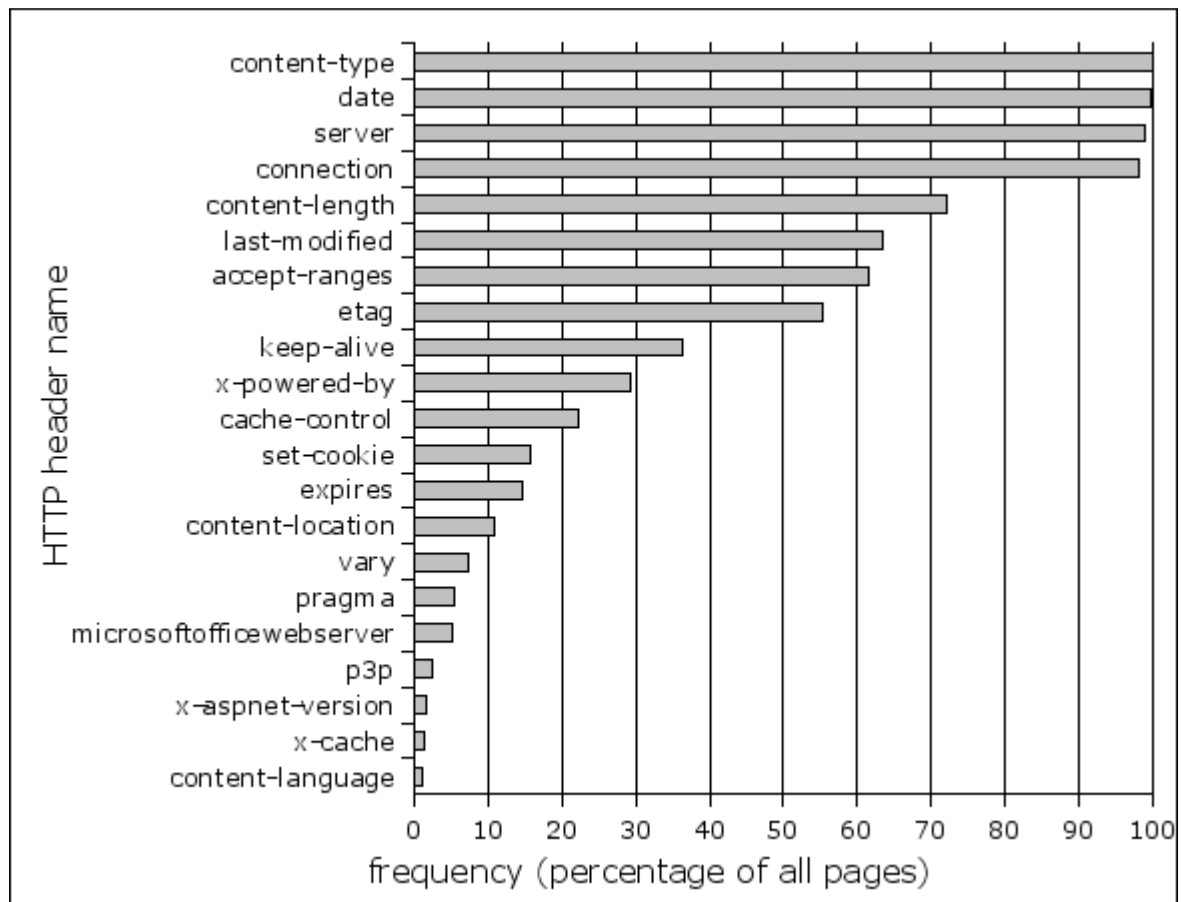
Header	Description	Example
Accept	This field contains a semicolon-separated list of representation schemes which will be accepted in the response to this request. This field is more easily accessible via <code>OpenNETCF.Web.HttpRequest.AcceptTypes</code> . This field is more easily accessible via <code>OpenNETCF.Web.HttpRequest.AcceptTypes</code> .	Accept: text/plain
Accept-Encoding	Acceptable encodings for this response	Accept-Encoding: x-compress; x-zip
Accept-Language	Acceptable languages for this response	Accept-Language: da
Authorization	Authentication credentials for HTTP authentication. The first word of this field will indicate the scheme in use. Scheme might be basic, user, or Kerberos. We show the details (through WireShark) of a secure transaction in HOL P111 (Securing a Padarn Site with SSL). A user agent that wants to authenticate itself with a server generally after receiving a 401 response does so by including an Authorization request-header field with the request. Realm information can also be included with the authorization field.	Authorization: Basic QWxhZGRpbjpvGVuHNIc2FtZQ==
Cache-Control	This allows web publishers to define how pages should be handled by caches. They can be broken down into the following general categories: <ul style="list-style-type: none"> <li>• Restrictions on what pages are cacheable</li> <li>• Restrictions on what may be stored by a cache</li> <li>• Modifications of the basic expiration mechanism</li> <li>• Controls over cache revalidation and reload</li> <li>• Control over transformation of entities</li> </ul>	Cache-Control: max-age=3600, must-revalidate
Cookie	Cookies are exchanged frequently between client and server in conjunction with authorization requests. Please visit HOL P203 (Setting Client Browser Cookies from a Padarn Page) for more details. This field is more easily accessible via	Cookie: \$Version=1; UserId=JohnDoe



	OpenNETCF.Web.HttpRequest.Cookie.	
Content-Encoding	See ContentEncoding (bullet 3) above	Content-Encoding: gzip
Content-Length	See ContentLength (bullet 4) above	Content-Length: 3495
Content-Type	See ContentType (bullet 5) above	Content-Type: application/x-www-form-urlencoded
Date	The date and time that the message was sent	Date: Tue, 15 Mar 2009 11:12:11 GMT
Expect	Indicates that particular server behaviors are required by the client	Expect: 100-continue
Host	The domain name of the server, mandatory since HTTP/1.1	Host: 192.168.11.1
If-Match	Only perform the action if the client supplied entity matches the same entity on the server. This is mainly for methods like PUT to only update a resource if it has not been modified since the user last updated it.	If-Match: "737060cd8c284d8af7ad3082f209582d"
If-Modified-Since	Allows a <i>304 Not Modified</i> to be returned if content is unchanged	If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
If-Range	If the entity is unchanged, send me the part(s) that I am missing; otherwise, send me the entire new entity	If-Range: "737060cd8c284d8af7ad3082f209582d"
If-Unmodified-Since	Only send the response if the entity has not been modified since a specific time.	If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT
Range	Request only part of an entity.	Range: bytes=500-999

Referrer	This is the address of the previous web page from which a link to the currently requested page was followed.	Referrer: http://192.168.11.1/MainSite
User-Agent	<p>A user agent string helps a server uniquely identify a client browser. It generally contains three tokens:</p> <ul style="list-style-type: none"><li>• Compatibility flag. This shows that the browser in use is compatible with a series of browsers.</li><li>• Version token. Provides a major version number for the browser in use.</li><li>• Platform token. Indicates the OS version (minor/major) of the client host.</li></ul>	User-Agent: Mozilla/5.0 (Linux; X11)

The frequency with which these headers appears on common webpages might be surprising. Below is a graph that shows just that (source: triin.net).



10. **HttpMethod**. This field describes as a string the HTTP method being used by the Request. Further detail will be provided for the two main methods (GET, POST) along with the other, less used methods (PUT, DELETE, HEAD). In brief:
  - a. GET – Retrieve data where identifier is contained within the URI
  - b. HEAD – Like GET but doesn't contain a body, only headers
  - c. PUT – Specifies that the data in the body should be stored where the URI specifies
  - d. DELETE – Requests that the data found at the URI be deleted
  - e. POST – Creates a new object linked to what's found at the URI.
11. **IsAuthenticated**. Has the user been authenticated by digest or basic authentication means (supported by Padarn in v1.1 Series)
12. **IsLocal**. Indicates whether the HTTP request originated from the local machine (i.e. the Padarn server) or a remote client
13. **IsSecureConnection**. Specifies whether the Request is being passed over an SSL channel.
14. **QueryString** – When a client browser submits an HTTP GET request to Padarn, the important content will appear in the URI submitted. Using the **QueryString** property, it's

possible to get a [NameValueCollection](#) accessor, which effectively splits the URI into individual fields. For example, if the user were to submit the following URL through his web browser address bar:

http://<Padarn Server IP>/default.aspx?value1=A&value2=B&value3=3

The following will show up within the Page\_Load function:

```
Immediate Window
Request.QueryString
{OpenNETCF.Web.x9223e5ea3f525cc9}
  [OpenNETCF.Web.x9223e5ea3f525cc9]: {OpenNETCF.Web.x9223e5ea3f525cc9}
    base {System.Collections.Specialized.NameObjectCollectionBase}: {OpenNETCF.Web.x9223e5ea3f525cc9}
    _allKeys: null
    AllKeys: {string[3]}
Request.QueryString.AllKeys
{string[3]}
  [0]: "value1"
  [1]: "value2"
  [2]: "value3"
```

Instead of resorting to manual manipulation of the URI, it's possible to just read directly from QueryString via

```
string valueIDVal = Request.QueryString["value1"] as string;
```

15. QueryString – When a client browser submits an HTTP GET request to Padarn, the important content will appear in the URI submitted. Using the QueryString property, it's possible to get a [NameValueCollection](#)
16. RawQueryString – Similar to QueryString except the value retrieved here is everything after the resource component of the passed URI. For example, using the same URL as before, you would see the following in the immediate window:

```
Immediate Window
Request.RawQueryString
"value1=A&value2=B&value3=3"
```

**Important Note:** RawQueryString will pass any HTML character codes (such as %20 for space, %21 for exclamation mark, %22 for double quotes, etc.) to your code behind logic. QueryString, on the other hand, will decode those special codes on your behalf.

17. RequestType. Same as HttpMethod (see above)
18. UserHostAddress. Very simple (and very useful!). This is the string representation of the host address of the remote client requesting the Padarn webpage. This is especially useful when doing filtering on incoming requests at the Page\_Load level.

## HttpResponse Object Discussion

In the simplest terms, the Response object contains all the data that the client's browser will require to render a page. While this object contains collections of data, it also provides methods for generating HTML content to display to the user. Two of the most commonly used methods are Write and Redirect. Write allows code behind to generate a full HTML document (which we often do in our HOL code behind demos). Redirect sends an HTML 302 code in the Response header, which forces the browser to jump to a different page.

At this point, we'll walk through the various properties and methods of the Response object, defined in OpenNETCF.Web.[HttpRequest](#).

1. **Cache.** This property allows for the [HttpCachePolicy](#) to be tweaked in two basic ways: the length of time for which the response remains valid can be set to any positive [TimeSpan](#) and the content being returned can be set to no-store. No-store's purpose is to prevent the inadvertent release or retention of sensitive information. This directive applies to the entire message, and may be sent either in a response or in a request. If sent in a request, a cache must not store any part of either this response or the request that elicited it. Most of the HOL demos do not use either of these two [HttpCachePolicy](#) settings so that the results are predictable. In a production environment, it's often important to make use of these.
2. **ContentLength.** Allows the code behind to specify the length of the entity body.
3. **ContentType.** As with the incoming Response object, we can set the outgoing HTTP MIME type of the outgoing stream with this string property. Here are some popular MIME categories: text (e.g. text/html ), image (e.g. image/gif), multipart (e.g. multipart/form-data), audio (e.g. audio/x-wav), and application (e.g. application/zip).
4. **Cookies.** This is the [HttpCookieCollection](#) (for retrieval only) that will be sent with your Request. You must use the SetCookie method to add Cookies. See below for details.
5. **HeaderEncoding.** This is the character set encoding of the header only. By default, UTF-8 is chosen, but this value can be overridden with ASCII, for example. Unicode (UTF-16) is not allowed and will throw an exception if used.
6. **Cookie methods:** AppendCookie and SetCookie allow you to add a cookie to the user's cookie store and modify an existing user's cookie, respectively. Full details of working with cookies are available in HOL P203 (Setting Client Browser Cookies from a Padarn Page).
7. **AppendHeader.** Just as the Request object provides access to a [NameValueCollection](#) of header fields, the Response object allows you to add name/value pairs for fields needing to go back to the client. No exception will be thrown if duplicates of a name/value pair are passed in. It's up to the client to properly render given those redundant values.
8. **Write methods:** BinaryWrite, Write, and WriteLine allow you to copy data into the entity body, whether it's a byte array, a string, or a string with carriage return. These functions are used to write out HTML content among other things to return to the user. For sample usage of these functions, please visit any of the HOLs posted to [opennetcf.com](#) and inspect any of the code behind files.
9. **Flush:** This method pushes all Response data to the client. Once the Response is received by a web browser, it will render to the user's screen.

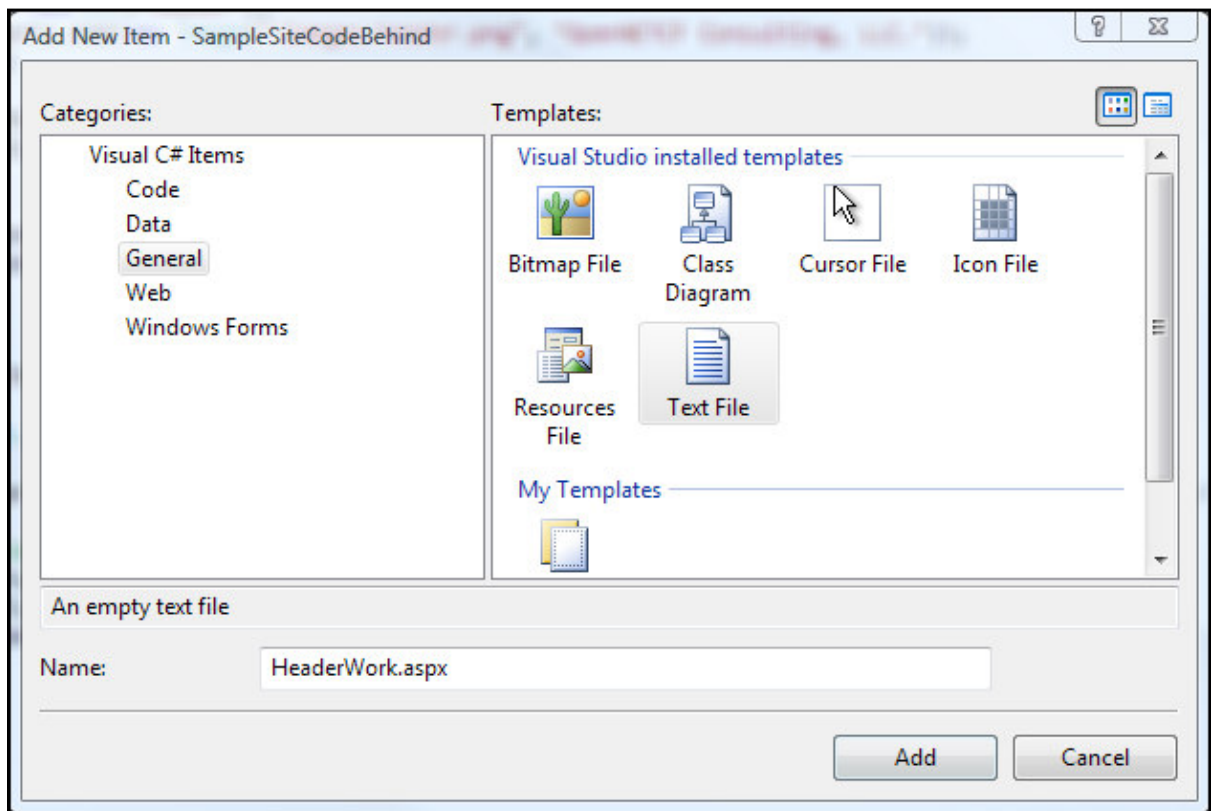
10. **Redirect:** Using this method will send the user to a new URI (ASPX, document, etc.). This is useful for passing a user from a page requiring input to a page displaying the results.

## Exercise 1: Adding our HeaderWork ASPX

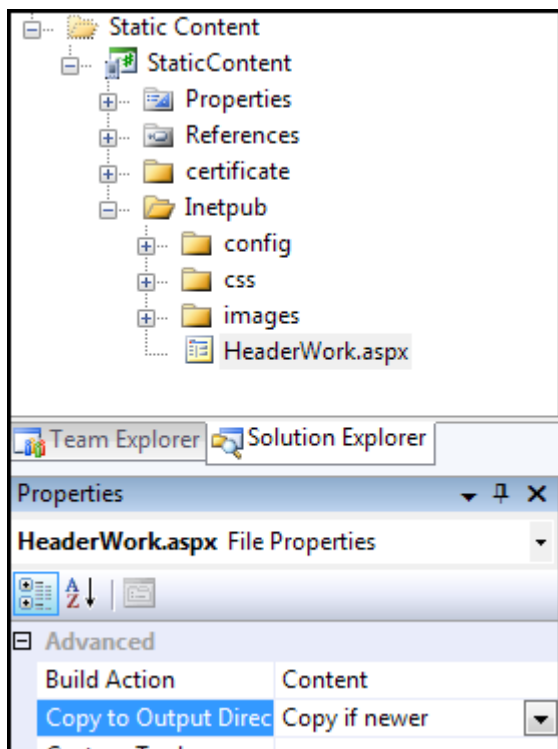
Throughout much of this HOL series, we will reuse a solution that contains three basic building blocks: a process host for the Padarn instance, a project for holding the Active Server Pages (ASPXs), JavaScripts, images, CSS file, and other static content, and finally a code behind project that contains the C# code that drives the functionality behind what will be displayed to the user.

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 "HeaderWork.aspx"



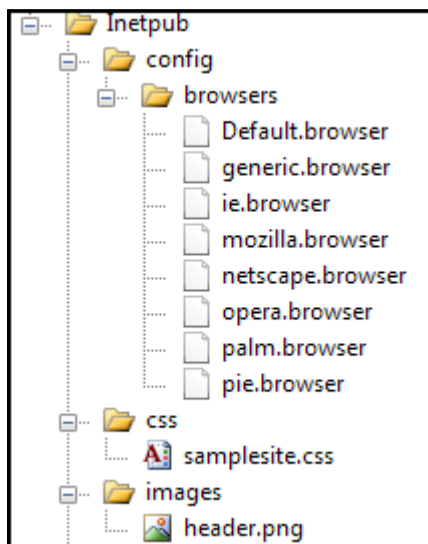
4. Click **Add** to add the new file to the project
5. In Visual Studio's Solution Explorer Pane, click on the newly-created HeaderWork.aspx file
6. In the Properties pane, set the Build Action for HeaderWork.aspx to "Content"
7. In the Properties pane, set the Copy to Output Directory property for HeaderWork.aspx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



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

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

The HOL comes with some extras that we won't discuss in great detail.

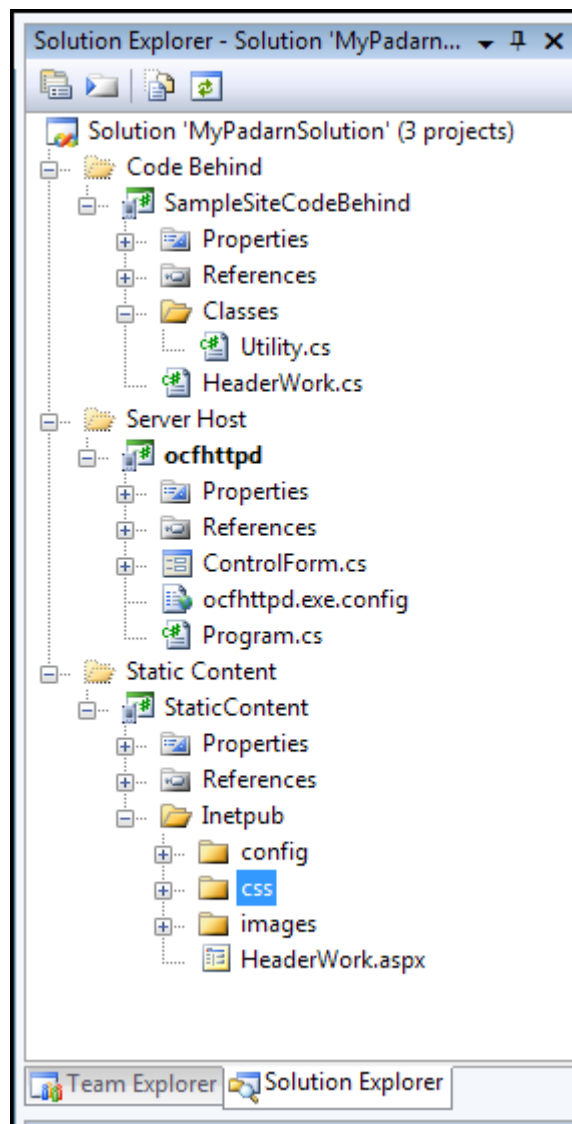


- Browser definition files. These each have the extension of .browser and abide by the common browser definition schema. Details of this schema can be [found on MSDN](#). All of these files deploy as static content automatically by Visual Studio 2008.
- CSS. This file defines the general font, spacing, and table layouts for all webpages subsuming this file.



- Code Behind – Utility Class (Utility.cs). Helper function that illustrates how to create standardized headers and footers throughout your site.
- Header image.

The solution will now look like this:



---

## Exercise 2: Traversing HTTP Headers

Padarn makes it quite straightforward to peer into all headers brought from the client request. By accessing the `Request.Headers` property, you'll be given read-only access to all header strings as discussed above in a [NameValueCollection](#). Below is code (which lives in our `HeaderWork.cs` code-behind file) that will first create an HTML table, then define a header row, then look throughout each header string and create a row per header item.

```
Table tableHeader = new Table(CLASS_NAME, "1000", Align.Center);

// Details section #1: What's in the header?
Row rowHeader = new Row();
rowHeader.Cells.Add(new RawText("Key"));
rowHeader.Cells.Add(new RawText("Value"));
tableHeader.Headers.Add(rowHeader);

// Go through entire NameValueCollection that makes up the request header
foreach (string key in Request.Headers.Keys)
{
    rowHeader = new Row();
    rowHeader.Cells.Add(new FormattedText(key, TextFormat.None, "2"));
    rowHeader.Cells.Add(new FormattedText(Request.Headers[key],
        TextFormat.None, "2"));
    tableHeader.Rows.Add(rowHeader);
}
```

We will see the output of this content later on in this lab.

## Exercise 3: Inspecting browser capabilities of client

Because of the wide variety of mobile and desktop browsers now available, each with its own strengths and weaknesses, it's up to the web application developer to make appropriate decisions on how to interact with that user. Some browsers might have security features missing in others. Some browsers might support a higher version of HTML (such as HTML 5, new in Firefox 3.5) that are absent in others. While Padarn isn't designed to serve demanding, animated/Flash content to end users, there's nothing stopping you from hosting this type of material.

OpenNETCF provides a class within the OpenNETCF.Web namespace called [HttpBrowserCapabilities](#). It is a class derived from the [HttpCapabilitiesBase](#) class, which has well over 100 boolean, string, and array-returning functions to help understand exactly what features of a browser have been enabled. This class largely follows from the .NET Framework 2.0. [See MSDN](#) for further details on the desktop implementation.

Continuing our progress in the HeaderWork.cs code-behind file, we first use reflection to pull all properties and methods from the [HttpCapabilitiesBase](#) class at runtime. Though this does take a performance hit, it makes our code much cleaner for demonstration purposes.

```
Type myBrowserCapabilities = (typeof(OpenNETCF.Web.HttpBrowserCapabilities));  
  
// Get the public methods  
MethodInfo[] myArrayMethodInfo = myBrowserCapabilities.GetMethods();
```

Next, we'll look at each [MethodInfo](#) object and create a row if its [ReturnType](#) is Boolean, string, or Version.

```
foreach (MethodInfo method in myArrayMethodInfo)  
{  
    if (method.ReturnType == typeof(bool) && method.Name.StartsWith("get_"))  
    {
```

We will "invoke" the property, which is equivalent to calling the getter.

```
bool valueBool = (bool)method.Invoke(Request.Browser, null);
```

Next, we create a row out of this data.

```
rowBrowser = new Row();  
rowBrowser.Cells.Add(new FormattedText(method.Name.Replace("get_", String.Empty),  
    TextFormat.None, "2"));  
rowBrowser.Cells.Add(new FormattedText(valueBool ? "YES" : "NO",  
    TextFormat.None, "2"));  
tableBrowser.Rows.Add(rowBrowser);
```

---

## Exercise 4: Forming the Response

The majority of our focus up to this point has been on the `HttpRequest` object. We now turn our attention to the `HttpResponse` object. In this lab, we are only using it to return to the user a constructed HTML page. We wish to display to the user the output of either the header details or browser capabilities and in either case, let the user know when the request came in, what the client host IP address is (using `Request.UserHostAddress`), and how long the transaction took to carry out.

```
Paragraph paragraph = new Paragraph(new FormattedText(String.Format(
    "Header details for request made on {0} by {1} and took {2} ms",
    startLoad.ToString(), Request.UserHostAddress,
    swPageLoadTime.ElapsedMilliseconds), TextFormat.Bold, "4"), Align.Center);
```

Now all we have to do is write the HTML document to the `HttpResponse`'s stream and submit it back to the user via the `Flush` command.

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

// Flush the response
Response.Flush();
```

## Exercise 5: 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 HeaderWork.aspx and all the browser definition files have been deployed as static content to the Padarn host machine.

Instructions for how to accomplish this are in HOL 100.

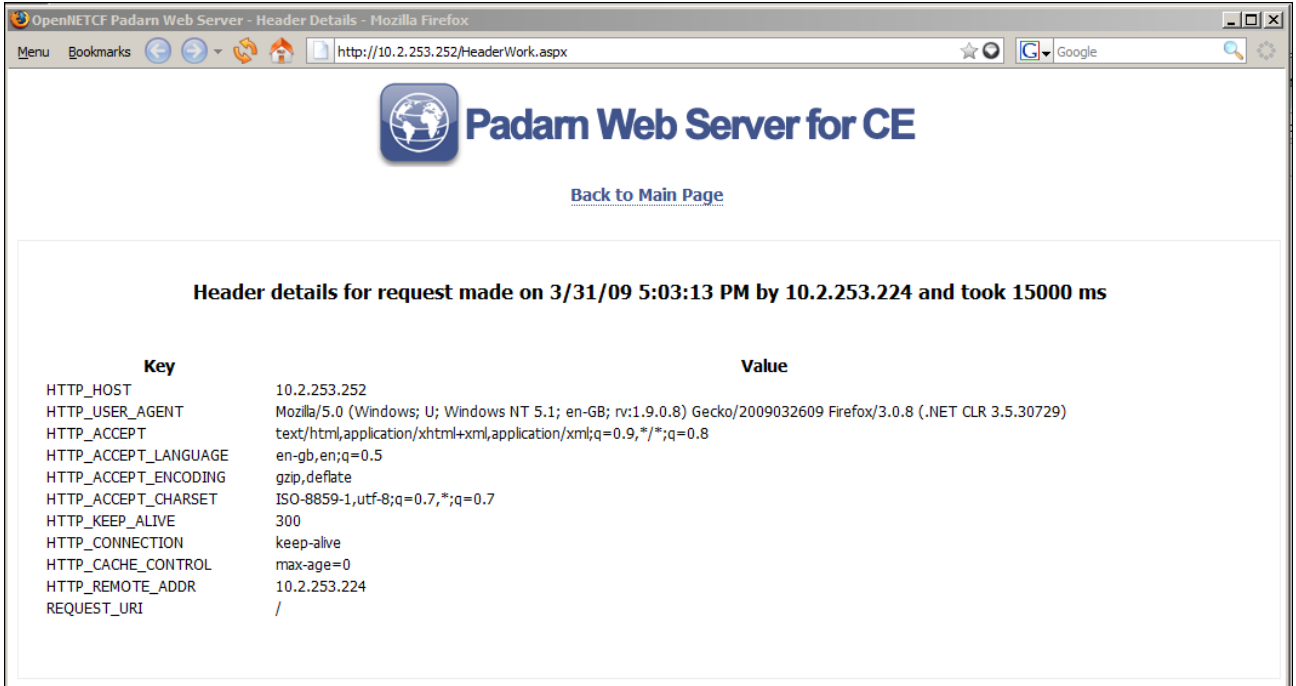
1. Initiate Padarn through the MyPadarnSolution Visual Studio 2008 solution. Right-click on the ocfhttpd project and select **Debug -> Start Without Debugging** (or press Ctrl + F5). For this demonstration, we will be using the Windows Mobile 5.0 Pocket PC R2 Emulator.



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



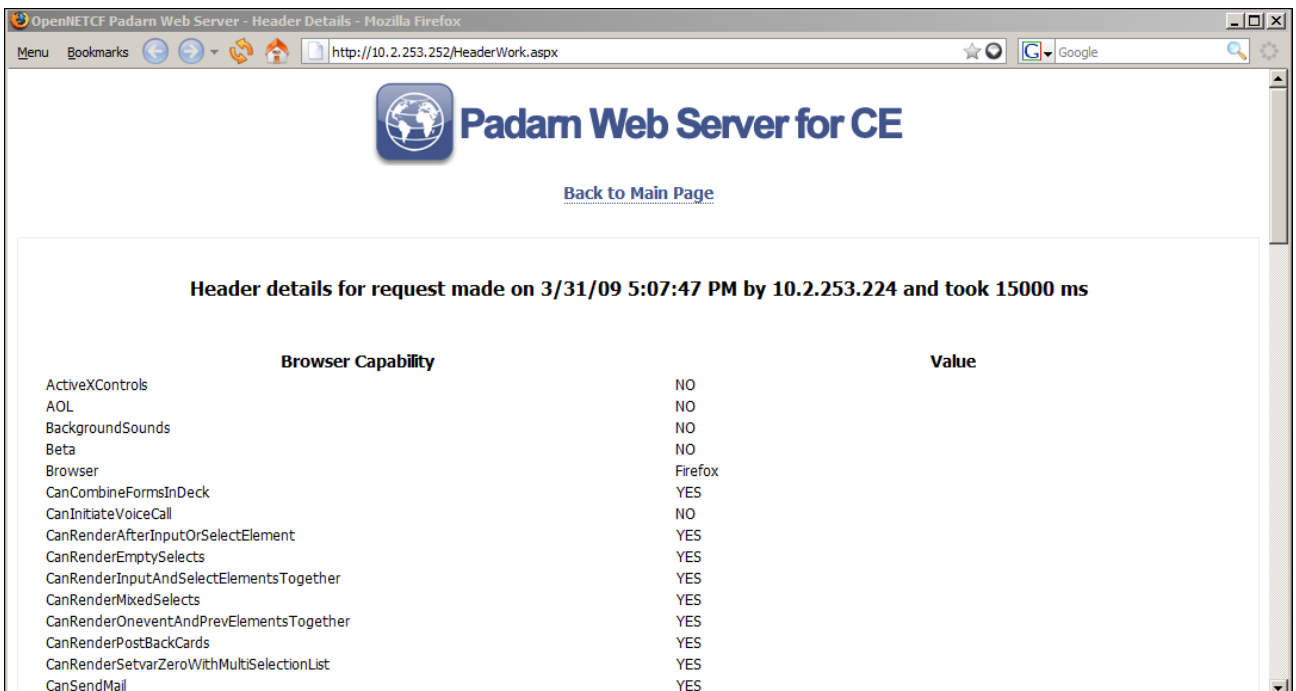
3. Right click on the ocfhttpd (under Server Host) project and select **Debug -> Start New Instance**
4. From a web browser that can communicate with the Padarn instance (for example, a Windows XP laptop that is host to the Windows Mobile 5.0 emulator). The IP address of the Padarn instance is key for 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>/HeaderWork.aspx in your favorite web browser.



**Header details for request made on 3/31/09 5:03:13 PM by 10.2.253.224 and took 15000 ms**

Key	Value
HTTP_HOST	10.2.253.252
HTTP_USER_AGENT	Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.9.0.8) Gecko/2009032609 Firefox/3.0.8 (.NET CLR 3.5.30729)
HTTP_ACCEPT	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_LANGUAGE	en-gb,en;q=0.5
HTTP_ACCEPT_ENCODING	gzip,deflate
HTTP_ACCEPT_CHARSET	ISO-8859-1,utf-8;q=0.7,*;q=0.7
HTTP_KEEP_ALIVE	300
HTTP_CONNECTION	keep-alive
HTTP_CACHE_CONTROL	max-age=0
HTTP_REMOTE_ADDR	10.2.253.224
REQUEST_URI	/

Here is what header details will be displayed over an unsecured access path to our Padarn server.



**Header details for request made on 3/31/09 5:07:47 PM by 10.2.253.224 and took 15000 ms**

Browser Capability	Value
ActiveXControls	NO
AOL	NO
BackgroundSounds	NO
Beta	NO
Browser	Firefox
CanCombineFormsInDeck	YES
CanInitiateVoiceCall	NO
CanRenderAfterInputOrSelectElement	YES
CanRenderEmptySelects	YES
CanRenderInputAndSelectElementsTogether	YES
CanRenderMixedSelects	YES
CanRenderOneventAndPrevElementsTogether	YES
CanRenderPostBackCards	YES
CanRenderSetvarZeroWithMultiSelectionList	YES
CanSendMail	YES

Here is the top portion of the many browser capabilities reported back by reading the appropriate browser definition file bundled with Padarn.

## Hands-on Lab Summary

To fully grasp the significance of Padarn's value within a production system, it's necessary to observe the close parallels to the ASP.NET 2.0 model that exist. By reusing skills, developers, and online resources focused around the desktop, Microsoft-originating version, it's possible to save a great deal of time and effort in bringing web applications to life that are hosted by extremely minimal hardware configurations.

Padarn provides a clean and structure access into the HTTP header submitted from any modern client browser to the server itself through the [HttpRequest](#) object. As the labs demonstrated, the information is easy to parse and act upon based upon authentication level, cache routine, browser capabilities, cookie enablement, and more. [HttpResponse](#), on the other hand, makes it straightforward to modify cookie data, set the expected content length/type, and command a client to immediately display an HTML document.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Learned about HTTP headers as defined by the HTTP 1.1 protocol
- ✓ Discovered the properties available in the [HttpRequest](#) and [HttpResponse](#) classes
- ✓ Wrote and tested code to access [HttpRequest](#) properties through a sample `Page_Load` routine
- ✓ Wrote and tested code to access browser capability detection as supported by Padarn through a sample `Page_Load` routine

In future HOLs, we will use the concepts explained in this HOL to build a Remote Procedure Call service and client, as well as take a detailed look into the specifics of cookie reading and writing from code-behind logic.