OpenNETCF Consulting

HOL P203 – Setting Client Browser Cookies from a Padarn Page

Table of Contents

Table of Contents 2

HOL Requirements 3

Summary........ 4

Lab Objective 4

Pre-Requisites........ 4

Introduction to Cookies 4

Exercise 1: Adding the CookieWork ASPX 7

Exercise 2: Stubbing out the Code-Behind Classes........ 9

Exercise 3: Writing to a Cookie File in Padarn 11

Exercise 4: Reading from a Cookie File in Padarn........ 14

Exercise 5: Deleting a Cookie File in Padarn 17

Exercise 6: Inspecting the Http Header of Cookie Usage 18

Exercise 6: Creating the Code-Behind Structure 19

Exercise 7: Configuring Padarn for Cookies........ 21

Hands-on Lab Summary 22

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 apply the basics of cookie retrieval and storage facilitation through a Padarn website. Though many concepts discussed here apply to ASP.NET applications, the full cookie support provided by Padarn establishes a similar level of feature support.

Lab Objective

Upon completion of this lab, you will be familiar with the process of reading and writing cookie values by way of the `HttpRequest` and `HttpResponse` classes.

In this HOL, you will perform the following exercises:

- ✓ Review the hosting of a Padarn web server within a CF.NET manager application
- ✓ Create a Padarn webpage that handles basic authentication requests
- ✓ Within that same Padarn webpage, understand how to parse a user cookie file
- ✓ Within that same Padarn webpage, display cookie properties for authenticated users

Pre-Requisites

This lab builds on the code created through the HOL 100 Series. Although the necessary components from those labs are included in this lab's example solution, it's advisable to go through the explanations in the HOL 100 Series to better understand how Padarn webpages interact with the Padarn web server and to understand how the `HttpRequest`.Page object behaves.

Introduction to Cookies

Cookies are a popular and cross-platform mechanism by which website authors can store user-specific content for later use. For example, a user's display settings for a stock tracker might be stored in a cookie –due to the way `HttpRequest` objects are brought into an ASP.NET (or Padarn) web application, the cookie information can be read easily and parsed like a dictionary data structure. Cookies can be optionally persisted to the user's hard drive; generally cookies are stored in the user's browser-defined location. For Internet Explorer users, this is generally in the Temporary Internet Files folder. For Firefox users, this is stored in the cache folder under the profile main folder.

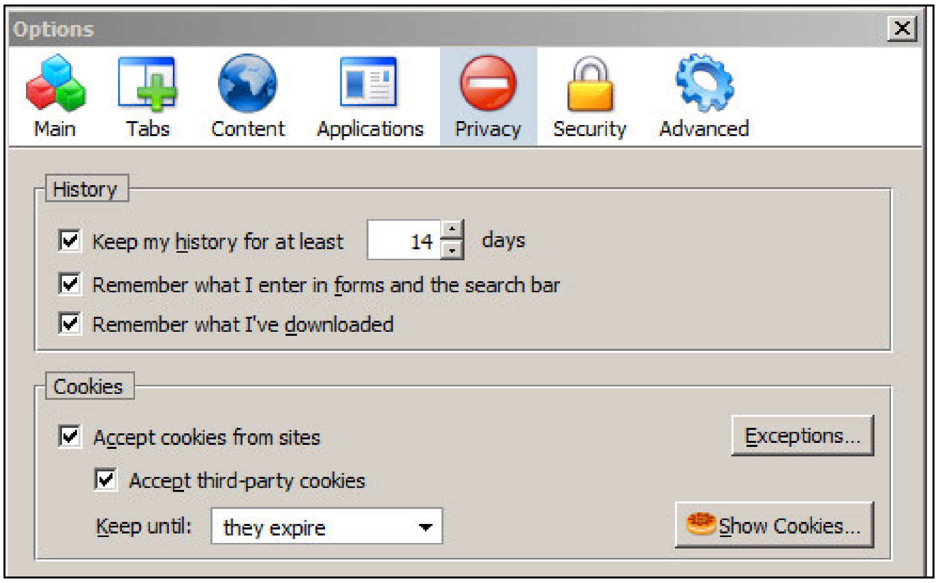
The cookie itself is a small text file that gets attached to any `HttpRequest` posted to a web server, so long as the cookie's domain has been established appropriate for the web server. If your website is configured as <www.hostname.com>, the domain of the cookie would have to be established as <www.hostname.com>. Cookies also support application path specificity. So if you have a cookie that should only be used for applications stored in <www.hostname.com/application>, your cookie will need to have a Path property value set.

In addition to user-specific preferences, many web developers use cookies to determine if an authentication session remains valid. Instead of forcing the user to login every time a request to a particular page occurs, a cookie can be used to store the last visit date and time. If the last visit happened recently, and the cookie contains a valid authentication token (Guid, for example), then the user won't be asked to login. The demonstration code in this HOL illustrates how this can be used in a Padarn web application.

Another well-advertised feature of cookies is they can be disabled in situations where only SSL connections should be supported. Unfortunately, this feature doesn't help all that much in minimizing security breaches since information is stored in an unencrypted (by default) format so if sensitive data is exchanged over a secured connection and that data is persisted to disk, if the user's machine is hacked, that information can be read from the cookie.

As powerful as cookies can be in developing compelling web applications, they do have some other weaknesses. Cookies only support up to 4096 bytes, so it's best to think carefully about streamlining all cookie needs for an entire site into one file as opposed to having different cookies for different user particulars. Often, cookie information is keyed off of a particular user ID. Cookie support is dictated by the web browser used to access the site. Some browsers only support 20 or so cookies per website. Some browsers have a hard limit on total cookies across all sites. That number is usually around 300. Some users opt to disable cookies completely and so cookie exchanges will fail. Our code ensures browser cookie support is enabled. Of course, the user can always delete the cookie associated with your website, so it's critical to check for its existence before attempting to read from it.

In Firefox, the Privacy tab houses the settings for cookies across your profile:



☑ Keep my history for at least 14 days

☑ Remember what I enter in forms and the search bar

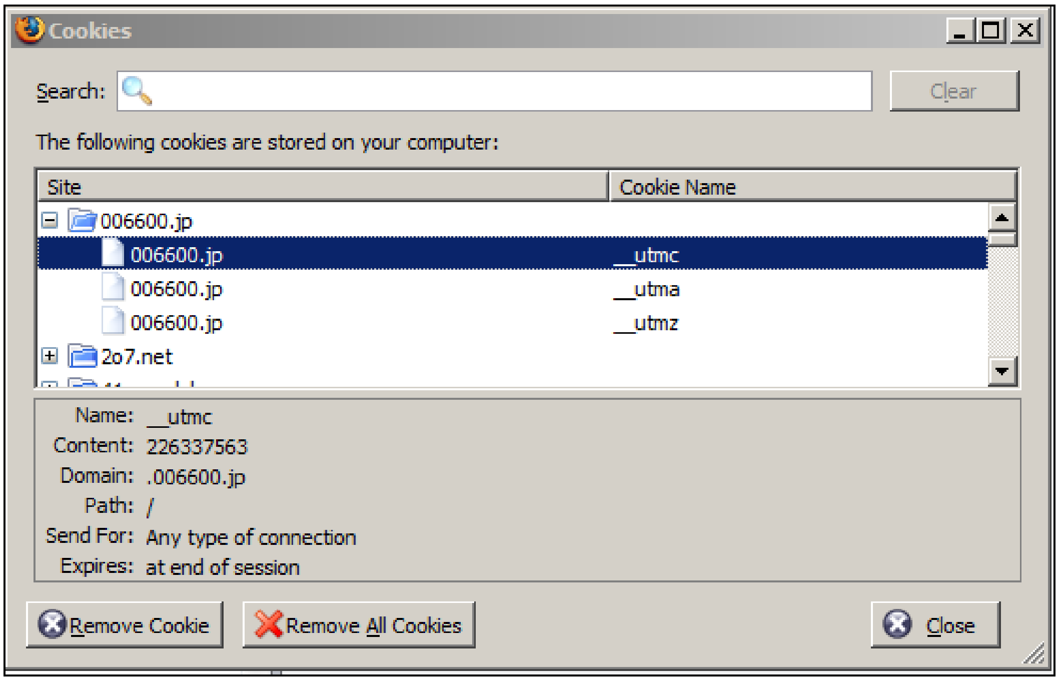
☑ Remember what I've downloaded

☑ Accept cookies from sites

☑ Accept third-party cookies

Keep until: they expire

Within this tab, it's possible to view individual cookie contents.



The following cookies are stored on your computer:

Site	Cookie Name
006600.jp	
006600.jp	__utmc
006600.jp	__utma
006600.jp	__utmz
2o7.net	

Name:	__utmc
Content:	226337563
Domain:	.006600.jp
Path:	/
Send For:	Any type of connection
Expires:	at end of session

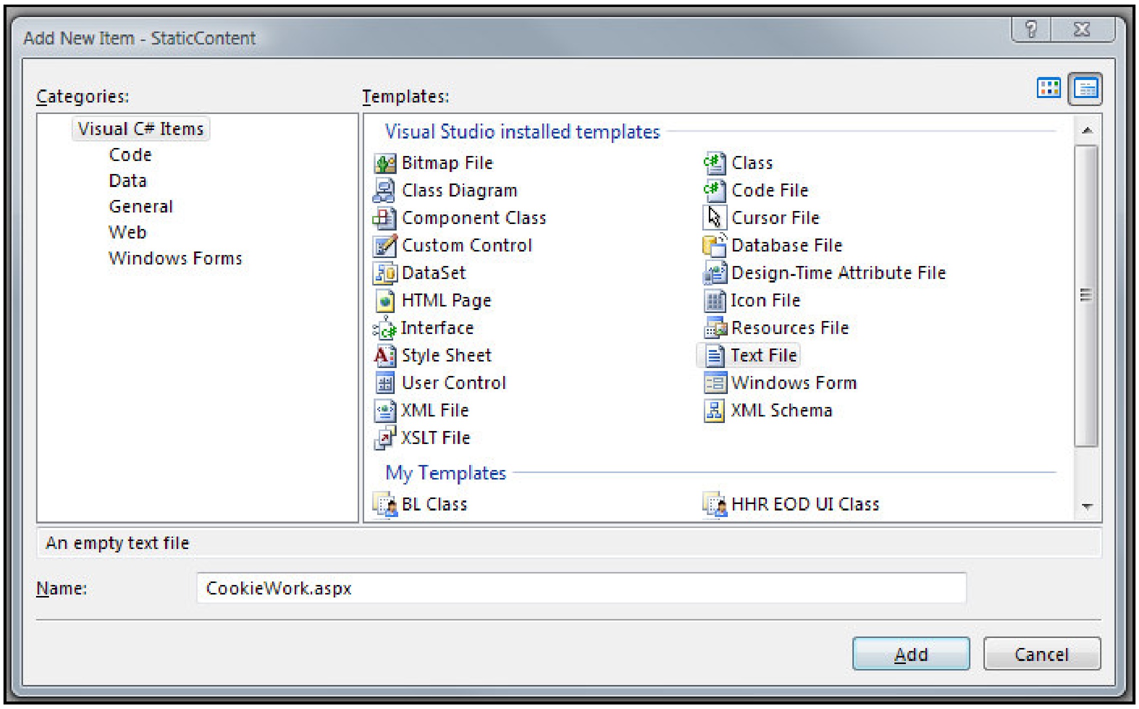
In the "Show Cookies" dialog, you can inspect the contents of individual cookies for a site. For the website <http://www.006600.jp>, you can see there are three cookies created. The first one doesn't ever get persisted to disk and will be deleted after the web response is returned to the browser. Cookies are never deleted explicitly by a website using a cookie; instead, websites can alter the expiration date of a cookie to alter the persistence behavior. A cookie can be set to expire in the past so that the cookie will be deleted once the `HttpResponse` is sent to the browser. Otherwise, a cookie can be set far into the future so that only when the user manually deletes the cookie or invokes other state management behavior (like logging out of a website) will the cookie be deleted.

Exercise 1: Adding the CookieWork ASPX

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

To create the ASPX file:

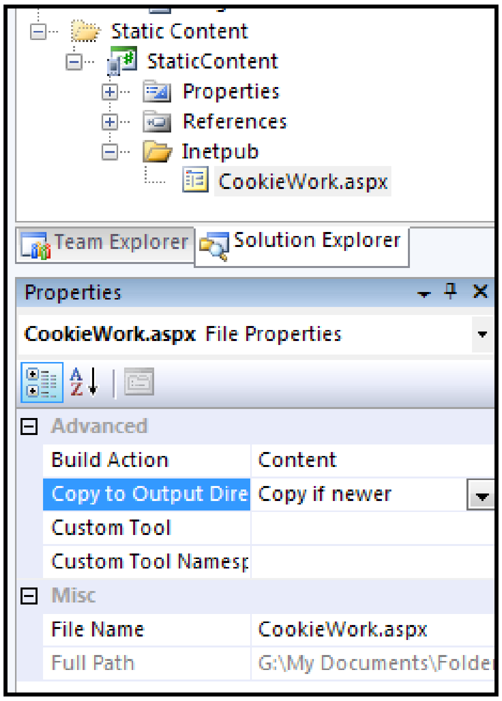
1. In Visual Studio’s **Solution Explorer** Pane, right-click on the **Inetpub** folder in the **StaticContent** project and select **Add -> New Item** to display the **Add New Item** dialog.
2. From the list of **Visual Studio Templates**, select **Text File**
3. In the **Name** textbox enter “CookieWork.aspx”



An empty text file

Name: CookieWork.aspx

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


Static Content
StaticContent
Properties
References
Inetpub
CookieWork.aspx

Team Explorer | Solution Explorer

Properties

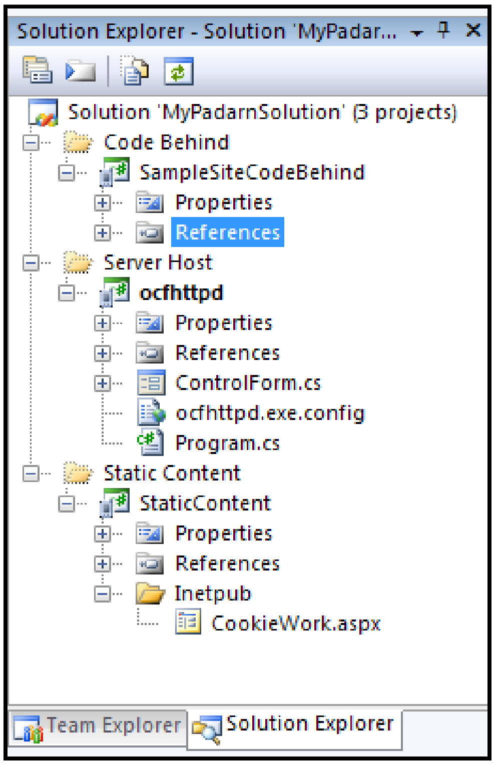
CookieWork.aspx File Properties

Advanced	
Build Action	Content
Copy to Output Dire	Copy if newer
Custom Tool	
Custom Tool Namesp	
Misc	
File Name	CookieWork.aspx
Full Path	G:\My Documents\Folde

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

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

9. The solution will now look like this:


Solution Explorer - Solution 'MyPadar...

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
CookieWork.aspx

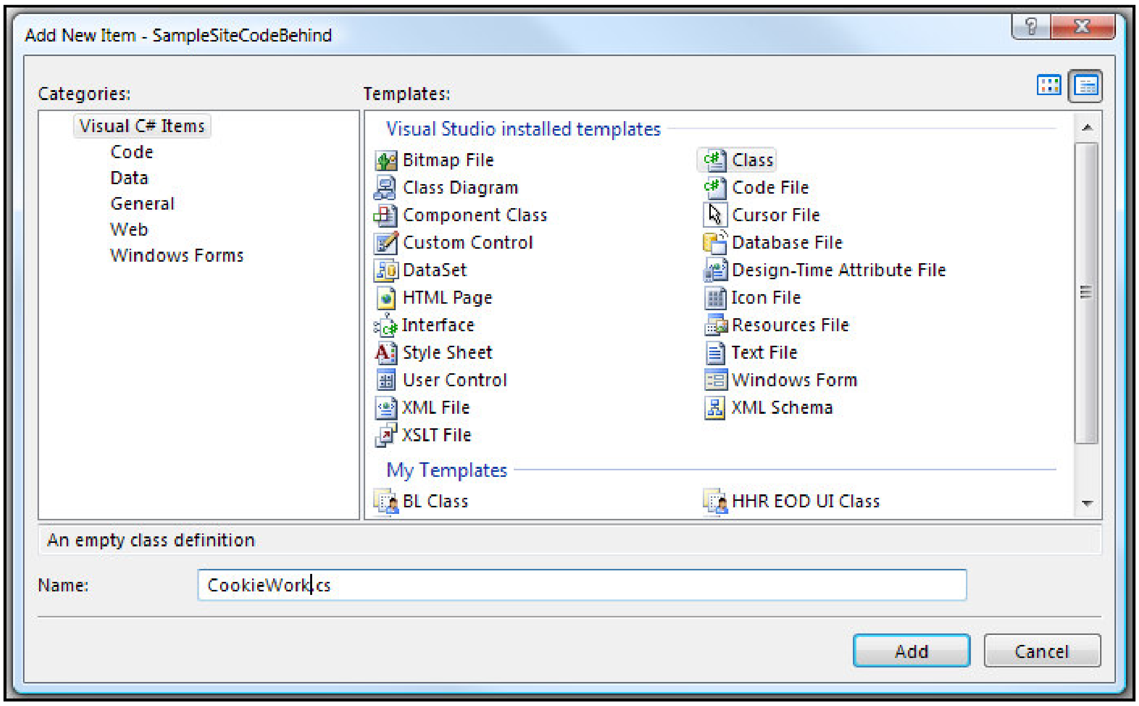
Team Explorer | Solution Explorer

Exercise 2: Stubbing out the Code-Behind Classes

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

To create the main code-behind class:

1. In Visual Studio's Solution Explorer pane, right-click on the SampleSiteCodeBehind project and select **Add -> New Item** to display the Add New Item dialog.
2. From the list of Visual Studio Installed Templates select **Class**.
3. In the Name textbox, enter "CookieWork.cs".



An empty class definition

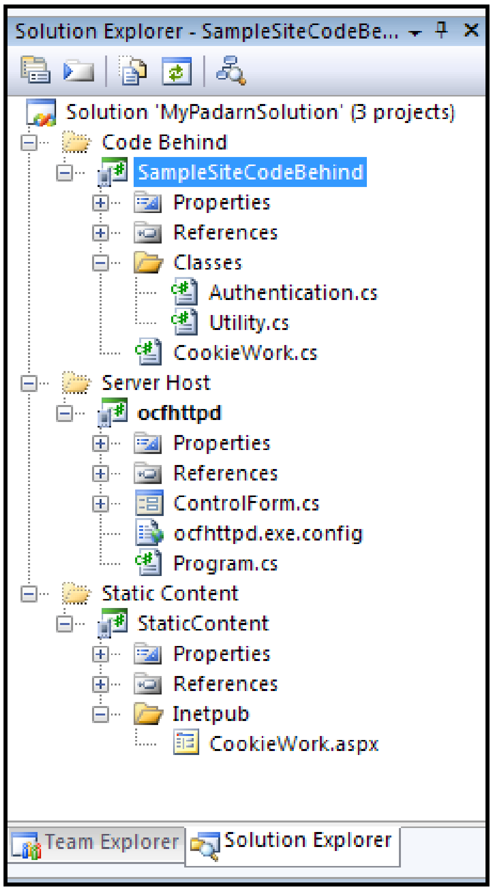
Name: CookieWork.cs

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

To create the auxiliary code-behind classes:

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

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



Solution Explorer - SampleSiteCodeBe...

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

Exercise 3: Writing to a Cookie File in Padarn

The web browser handles all of the plumbing required to persist cookies to a user's profile folder. When building a Padarn website, we use the `HttpResponse` object to access the Cookies collection (implemented as a `NameObjectCollectionBase`) to modify the contents of a cookie file. Remember: the cookie that we have access to is applicable only to the domain and path specified by the cookie. Since the Cookies collection is treated as a name/value pair, it's important to use unique names for anything stored in the cookie, as this will be set as the key. If uniqueness is not enforced, then the last name/value pair entered will overwrite any prior entries.

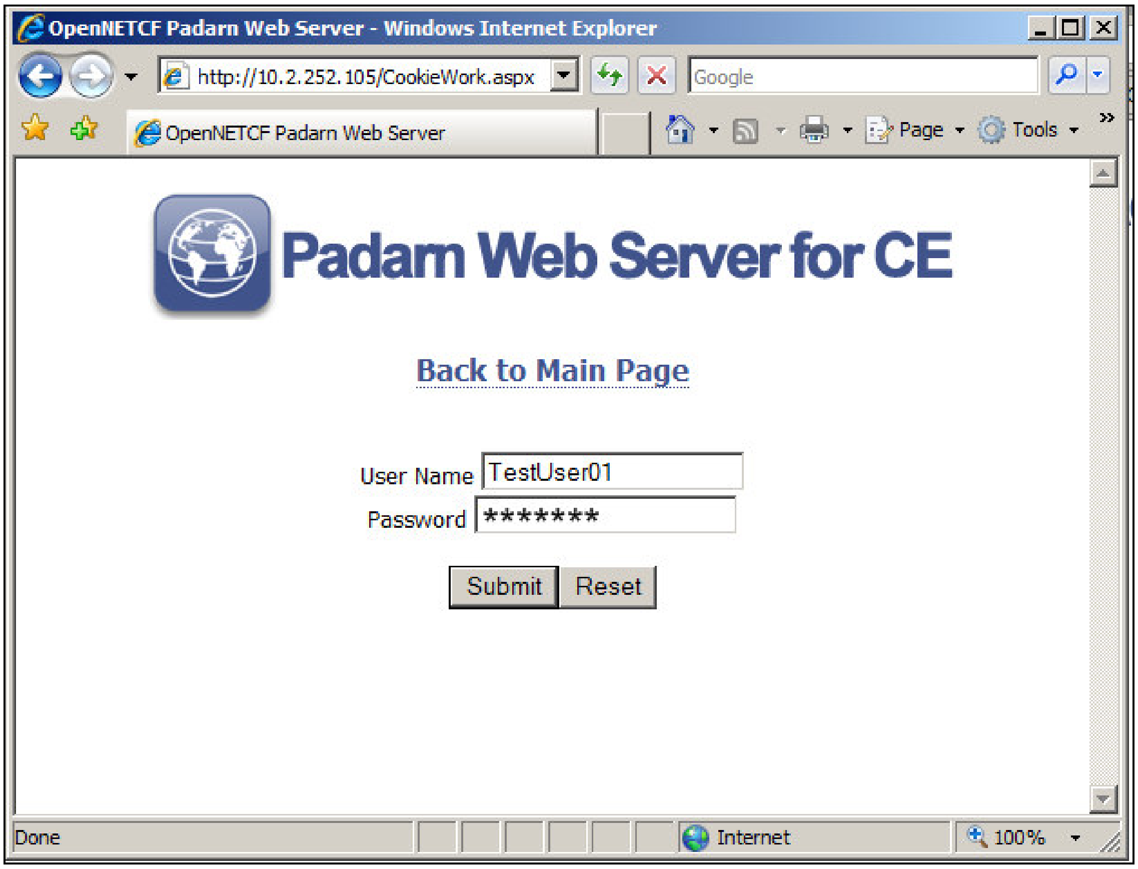
If you fail to specify the expiration date (using the DateTime `Expires` property), then the content stored will be treated as session information and the cookie will not be persisted to the user's hard disk. When the browser window is closed (or tab), the content is erased permanently. Often, websites ask the user if they are using a publicly accessible workstation to determine if cookies should be stored.

In previous HOLs, we have shown how to create a case statement with different behavior based upon the action defined by the Page request. In the case of the CookieWork example, we will be handling only two request actions: Submit and LogOut. Submit will take place when the user enters credentials into our simple login UI. LogOut will take place when the user has already logged in but wishes to clear his credentials. This not only invalidates the Guid that we use to track the user's session but it will also clear the cookie from his computer (if his browser supports cookies).

The entire login form is created using the OpenNETCF.Web.HTML namespace and the code is included in the `DisplayLoginForm` function. Please review the code to see how this is done. The only unique piece of code is how we create the password input box:

```
//Create a standard password input box
form.Add(new RawText(string.Format("<label>{0}</label>", "Password ")));
form.Add(new RawText("<input name=\"PW\" type=\"password\">"));
```

We insert HTML code so that the password is masked. Here is what the `DisplayLoginForm` function will show the user in a web browser:


OpenNETCF Padarn Web Server - Windows Internet Explorer
http://10.2.252.105/CookieWork.aspx
Google
OpenNETCF Padarn Web Server
Page
Tools
Padarn Web Server for CE
Back to Main Page
User Name
TestUser01
Password
****
Submit
Reset
Done
Internet
100%

Once the user presses the **Submit** button, a request is sent to CookieWork.aspx with the user name and password passed as Request values. Within the Submit action, we read those values:

```
string UID = Request.Form["UserName"];
string PW = Request.Form["PW"];
```

Next, we call our `Authentication.ValidateUser` function, which we'll discuss further down in the article. If the authentication succeeds, we prepare to create a cookie for the user.

```
OpenNETCF.Web.HttpCookie myCookie = new OpenNETCF.Web.HttpCookie("userInfo");
myCookie.Values["userName"] = UID;
myCookie.Values["GUID"] = uoGuid.ToString();
myCookie.Values["lastVisit"] = DateTime.Now.ToString();
myCookie.Expires = DateTime.Now.AddDays(1);
```

Although there are several ways to create a new cookie, we're taking the approach of creating a new `HttpCookie` object and then pass it into the Response.CookieCollection explicitly. We will be creating a cookie file called "userInfo" with several sub-values: userName, GUID, and lastVisit. We will set the expiration date of the cookie to be one day past current day. Because the expiration date is set to the future, this cookie will be persisted to the user's hard disk.

Next, we invoke the code to return the cookie back to the user and redirect back to the CookieWork.aspx page.

```
if (Request.Browser.Cookies)
{
        Response.SetCookie(myCookie);
}

Response.Redirect("CookieWork.aspx");
```

Notice the browser cookie check. This only checks whether the browser supports cookies. It does not indicate whether the user is blocking cookies for this particular domain. It is beyond the scope of this article to provide a more accurate check for cookie acceptance though the logic is straightforward.

Once the redirect has taken place, we will be able to begin reading from the cookie.

One feature of the HttpCookie that hasn't been discussed up until now is specificity over the path or virtual folder where a particular webpage is located on the web server. For example, if you wish to maintain a different cookie for a web application located at <www.domain.com/path1> compared to <www.domain.com/path2>, you could set the Path property of an `HttpCookie` object. In the code below, adding the following line would ensure that the cookie would be used only for webpages located in <www.domain.com/path1>:

```
myCookie.Path = "\path1";
Response.SetCookie(myCookie);
```

Exercise 4: Reading from a Cookie File in Padarn

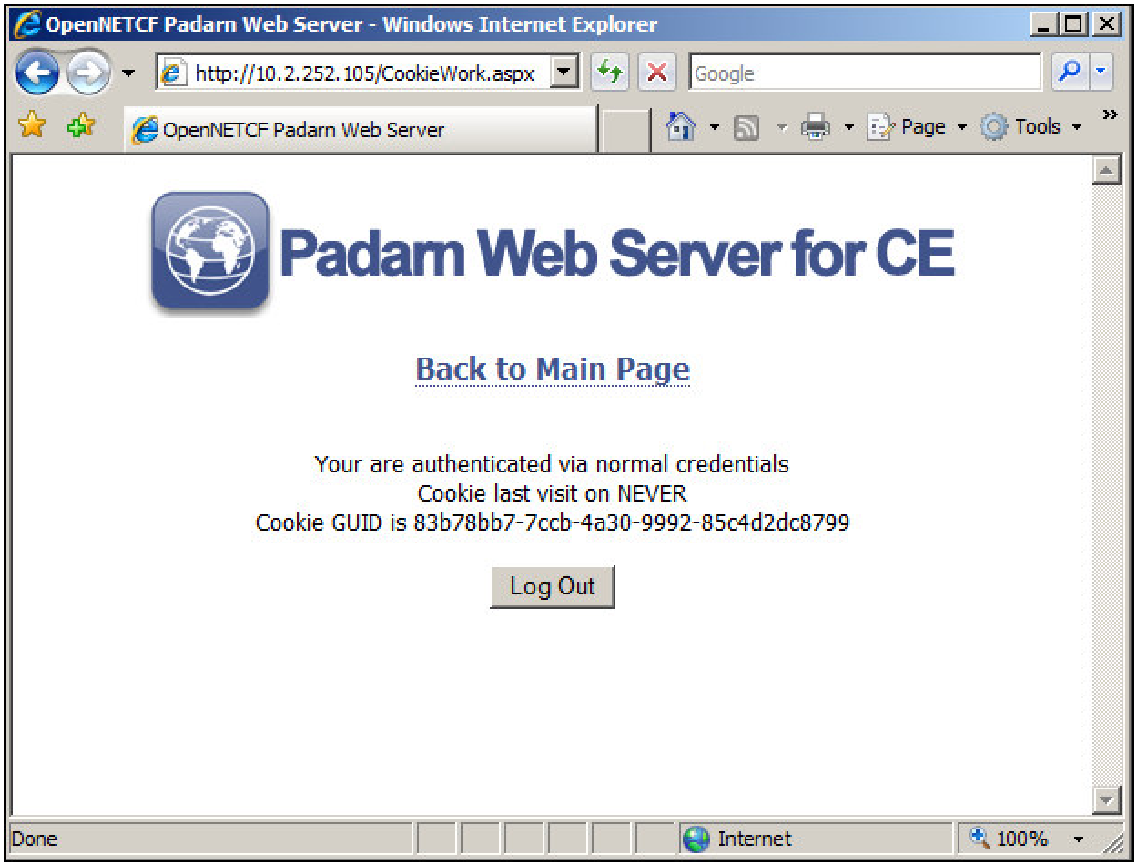
Clearly, once we've stored several values into a cookie during the proceeding steps, we'll want to read the contents of the cookie we've created. In our example, this is done because we want to redirect the user to an "authenticated" page, which could consist of pretty much any content. For the sake of demonstration, we'll merely display the contents of the cookie used to get to the authenticated page as well as whether the user had just come from the login page or if the user was authenticated because the cookie contained a Guid that pointed to a valid authentication session. Instead of writing to the cookie in the `HttpResponse` object, we read from the `HttpRequest` object.

```
//Check GUID in cookie; redirect to authenticated page if so
if (Request.Cookies["userInfo"] != null && Request.Cookies["userInfo"].HasKeys)
{
        string guidCookie = Request.Cookies["userInfo"].Values["GUID"];
        Guid g = new Guid(guidCookie);
        Authentication.UserObject uo;

        if (!(uo = Authentication.ValidateUser(g)).Null && !uo.IsGuidExpired())
        {
                if (Request.Cookies["userInfo"][ "lastVisit"] == null)
                {
                        //We got here upon manual passage of credentials
                        DisplayAuthenticated(false, uo.UserName);
                }
                else
                {
                        //We got here by reading cookie
                        DisplayAuthenticated(true, uo.UserName);
                }
        }
        else
        {
                //We need to re-authenticate because GUID is invalid or expired
                DisplayLoginForm(false);
        }
}
else
{
        //We need to re-authenticate; no cookie was found
        DisplayLoginForm(false);
}
```

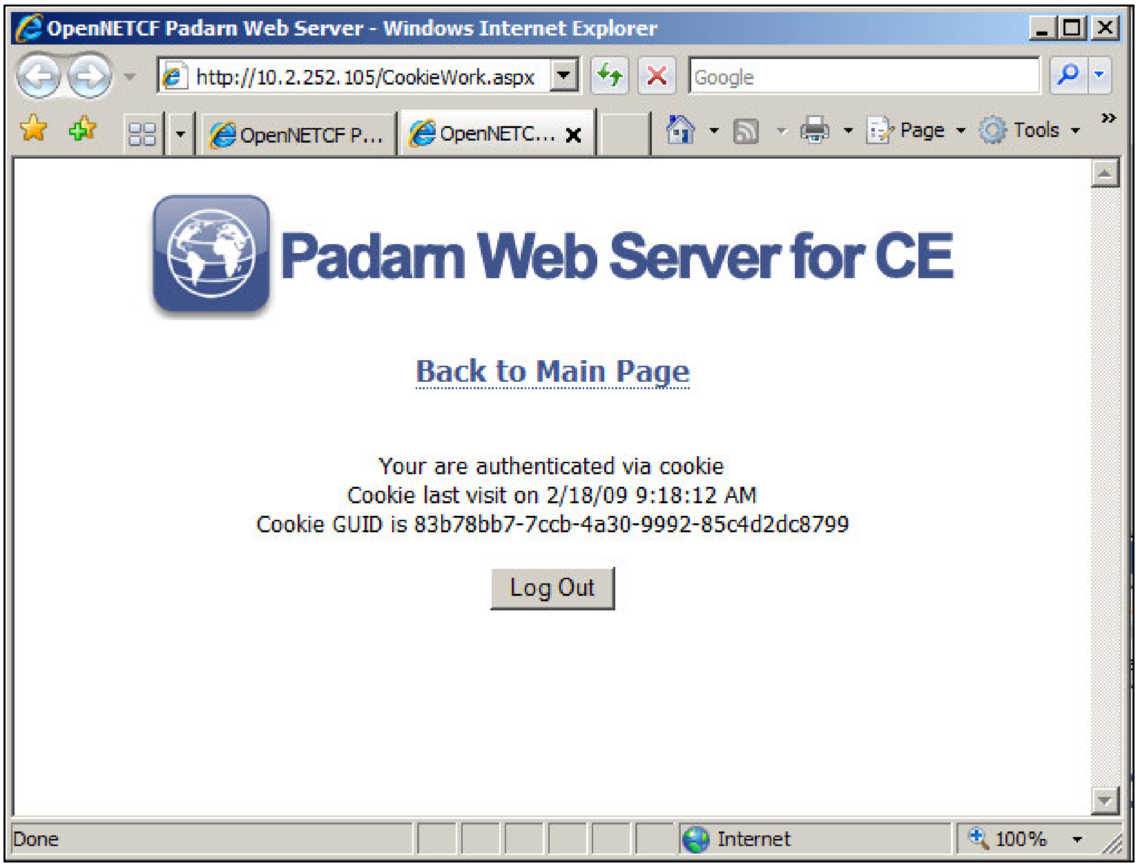
Once again, we're using the cookie named userInfo to read out the values set by us in a previous access to CookieWork.aspx. The Guid is what we use to uniquely identify a login session instead of storing a password in the user's cookie file. A Guid, by itself, provides no sensitive information about a user. The CookieWork application could go even further to make sure the incoming client request with a particular Guid maintains the same IP address or other particulars.

First, we check to ensure the userInfo cookie exists and isn't empty. Next, we try to read the the Guid as a string, parse that string to a Guid object, then again leverage our Authentication class to see if the user's authentication session is valid and hasn't yet expired. If this is the first time the user has gotten to CookieWork's authenticated page, we display a page indicating the user got here due to using the login UI shown above. Otherwise, we know it was the user's cookie Guid that brought us here. `DisplayAuthenticated` works to display the cookie contents and a suitable output page to the user. In this function, we read various sub-values by accessing the `Request.Cookies["userInfo"]` of type `HttpCookieCollection`. If the Guid is invalid or if the Guid has expired, we bring the user to the login UI.


OpenNETCF Padarn Web Server - Windows Internet Explorer
http://10.2.252.105/CookieWork.aspx
Google
OpenNETCF Padarn Web Server
Page
Tools
Padarn Web Server for CE
Back to Main Page
Log Out
Done
Internet
100%

Your are authenticated via normal credentials
Cookie last visit on NEVER
Cookie GUID is 83b78bb7-7ccb-4a30-9992-85c4d2dc8799

Screen shot of authenticated page after coming from login UI



Your are authenticated via cookie
Cookie last visit on 2/18/09 9:18:12 AM
Cookie GUID is 83b78bb7-7ccb-4a30-9992-85c4d2dc8799

Screen shot of authenticated page coming from cookie-based validation

Exercise 5: Deleting a Cookie File in Padarn

It's not possible to directly modify or delete a cookie file from an ASP.NET or Padarn web application. Instead, the contents of a cookie must be read into a suitable structure, such as HttpCookie, then added back to the HttpCookieCollection in the Response. Since cookie names must be unique, the act of adding the second instance of the same-named cookie file will remove the contents of the first cookie.

In our second action, LogOut, we will invoke the LogOff method of our Authentication class and we will delete the cookie file from the user's machine. Once the cookie has been deleted from the HttpResponse, we redirect back to the CookieWork.aspx webpage. Here is the code to handle that:

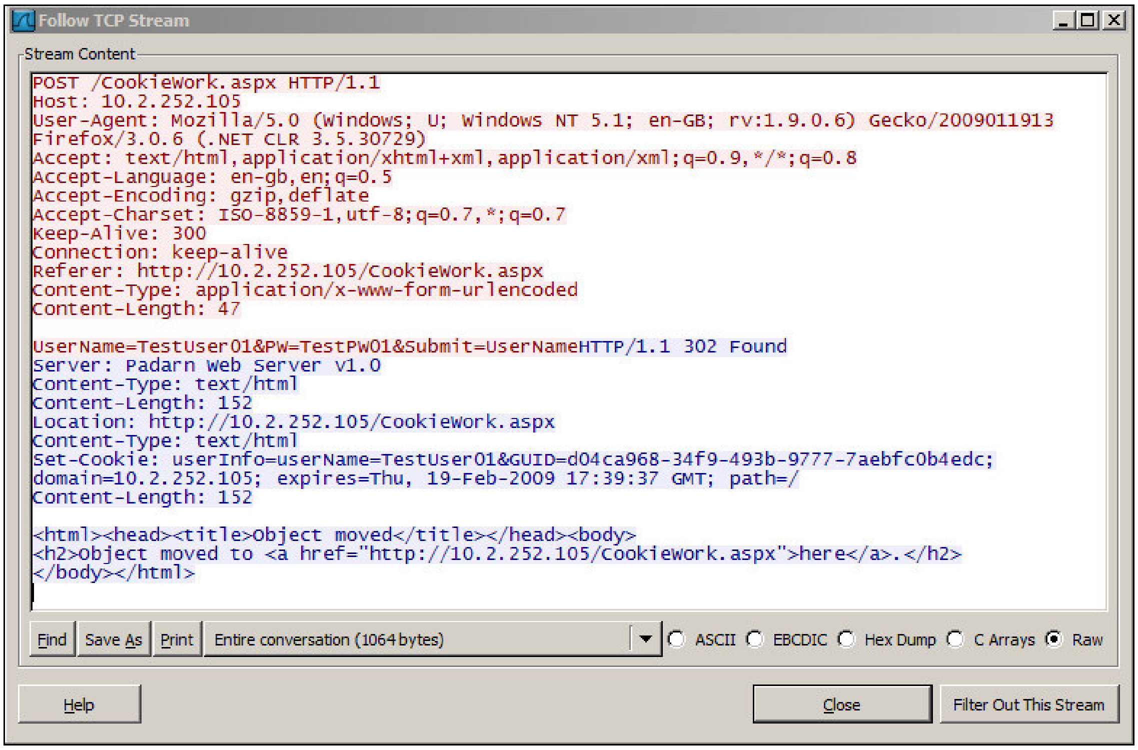
```
if (Request.Cookies["userInfo"] != null)
{
        OpenNETCF.Web.HttpCookie myCookie = Request.Cookies["userInfo"];
        myCookie.Expires = DateTime.Now.AddDays(-1);
        if (Request.Browser.Cookies)
        {
                //Verify browser can accept cookies
                Response.SetCookie(myCookie);
        }
}

//Refresh page
Response.Redirect("CookieWork.aspx");
```

Notice that to delete a cookie, we're setting the expiration date to the past. After the HttpResponse is parsed by the web browser, the cookie will be deleted from the user's machine. The same general approach to the above can be used for modifying a cookie. The existing cookie is fetched, desired properties are modified, then the cookie is re-added to the HttpCookieCollection in the HttpResponse.

Exercise 6: Inspecting the Http Header of Cookie Usage

Let's take a brief look at the HTTP header during both the webpage request and response from CookieWork, as interpreted by a web browser. When a web browser calls on a webpage where a cookie file applies to the domain being accessed, the cookie file is automatically submitted in the `HttpRequest` without the user performing any further action. If the cookie has been deleted or if a privacy setting has been enabled then obviously no cookie will be submitted.



```
POST /CookieWork.aspx HTTP/1.1
Host: 10.2.252.105
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.9.0.6) Gecko/2009011913
Firefox/3.0.6 (.NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://10.2.252.105/CookieWork.aspx
Content-Type: application/x-www-form-urlencoded
Content-Length: 47

UserName=TestUser01&PW=TestPW01&Submit=UserNameHTTP/1.1 302 Found
Server: Padarn Web Server v1.0
Content-Type: text/html
Content-Length: 152
Location: http://10.2.252.105/CookieWork.aspx
Content-Type: text/html
Set-Cookie: userInfo=userName=TestUser01&GUID=d04ca968-34f9-493b-9777-7aebfc0b4edc;
domain=10.2.252.105; expires=Thu, 19-Feb-2009 17:39:37 GMT; path=/
Content-Length: 152

<html><head><title>Object moved</title></head><body>
<h2>Object moved to <a href="http://10.2.252.105/CookieWork.aspx">here</a>.</h2>
</body></html>
```

Entire conversation (1064 bytes) ASCII EBCDIC Hex Dump C Arrays [✓ Raw]

Using Wireshark, we can inspect the transaction invoked by the web browser and sent to Padarn. Notice we are sending the user name and password in clear text since this is not an SSL transaction. This would never be acceptable in a production environment! The Padarn server then sends back a cookie that has the authentication Guid and last access time set. We are also being redirected back to CookieWork.aspx now that the user has been authenticated.

Exercise 6: Creating the Code-Behind Structure

Throughout this article, we've made numerous references to the Authentication class. Like in the other HOLs, we have simplified greatly the backend security story. Generally, user credentials will be stored in an Active Directory account or other credential management system. For sake of illustration, we will reference a cached copy of several users' credentials and then all authentication attempts will be made against this copy. In the Authentication class, we have a quick representation of a user:

```
        public struct UserObject
        {
                string _userName;
                string _userPW;
                Guid _GUID;
                DateTime _GUIDExpire;
        }
```

Our credentials will be stored in a dictionary:

```
private static Dictionary<string, UserObject> userCredentialStore = new Dictionary<string,
UserObject>();
```

We create a static constructor to pre-fill the `Dictionary` with user credentials.

```
        userCredentialStore.Add("TestUser01", new UserObject("TestUser01", "TestPW01"));
        userCredentialStore.Add("TestUser02", new UserObject("TestUser02", "TestPW02"));
        userCredentialStore.Add("TestUser03", new UserObject("TestUser03", "TestPW03"));
        userCredentialStore.Add("TestUser04", new UserObject("TestUser04", "TestPW04"));
```

We create a method to validate the user based upon a user name and password entered by the user (this comes from the login UI defined by `DisplayLoginForm`

```
Guid retVal = Guid.Empty;

if (!String.IsNullOrEmpty(userName) && !String.IsNullOrEmpty(userPW))
{
        UserObject uoTarget = userCredentialStore.ContainsKey(userName) ?
        userCredentialStore[userName] : new UserObject();
        if (!uoTarget.Null && userCredentialStore[userName].UserPW.Equals(userPW))
        {
                //If a GUID already exists and hasn't expired, return it
                if (uoTarget.UserGuid != Guid.Empty && !uoTarget.IsGuidExpired())
                        retVal = userCredentialStore[userName].UserGuid;
                else
                {
                        uoTarget.SetGuid();
                        userCredentialStore.Remove(userName);
                        userCredentialStore.Add(userName, uoTarget);
                        retVal = userCredentialStore[userName].UserGuid;
                }
        }
}

return retVal;
```

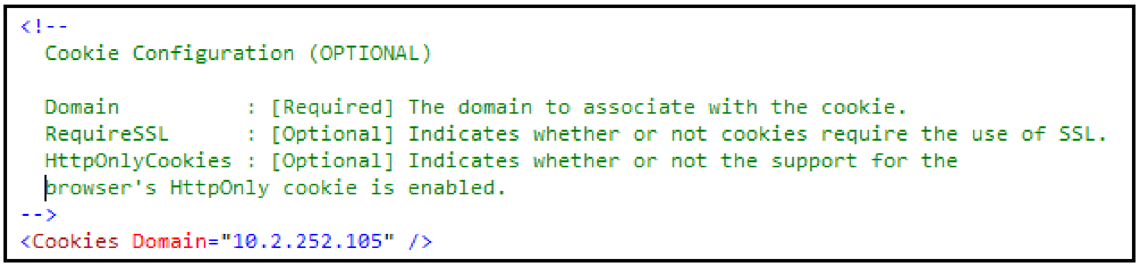
If the user is authenticated and the Guid hasn't expired, then we return the active Guid. Otherwise, we create a new authentication Guid so long as the user name and password validate against our credentials dictionary.

Our last method, `LogOff`, just clears out the Guid from the specified user's data.

We've also included a Utility class, which is included as part of the Padarn Evaluation Website available at http://www.opennetcf.com/padarn. This class creates a header and a footer and applies a basic Cascading Style Sheet (CSS) for our CookieWork webpage.

Exercise 7: Configuring Padarn for Cookies

A Padarn web server's global settings live within the octhttpd.exe.config file as part of the ocfhttpd project in Visual Studio. There are three settings for using cookies: domain, SSL enforcement, and HTTP-only cookie enablement. Here is the chunk of the XML pertaining to this:



```
<!--
  Cookie Configuration (OPTIONAL)

  Domain          : [Required] The domain to associate with the cookie.
  RequireSSL      : [Optional] Indicates whether or not cookies require the use of SSL.
  HttpOnlyCookies : [Optional] Indicates whether or not the support for the
  browser's HttpOnly cookie is enabled.
-->
<Cookies Domain="10.2.252.105" />
```

The most important setting is the Domain. Without this setting properly configured, no cookie will ever be sent in the `HttpRequest`. Depending on how the Padarn web server will be accessed, this value should either be an IP (as shown above) or a hostname (servername.domain.com). The second two settings provide more granularity in how cookies interact with Padarn. RequireSSL allows you to only enable cookies for SSL transactions. For a webpage exchanging sensitive information to and from the client, it's sometimes best to disable cookies. Setting this element to "true" ensures cookies are exchanged only over an SSL connection. See HOL 111 for details on how to enable Padarn to run over SSL.

In contrast to RequireSSL, HttpOnlyCookies is used to ensure cookies are only used over non-SSL connections. Its main purpose is to avoid cross-site scripting hacks. This is a common server-side vulnerability where information associated with one particular website is passed onto another website. If the HttpOnlyCookie attribute is enabled, the cookie is still sent when the user browses to a website in the valid domain. The cookie cannot be accessed through a script in the web browser (IE 6.0 SP1+ or FireFox 3.0+), even by the Web site that set the cookie in the first place. Setting the value of HttpOnlyCookies to true enables this option. Note that not all browsers support this feature and enabling it could prevent the cookie functionality above from working.

For more information on cross-scripting vulnerabilities, please visit http://msdn.microsoft.com/en-us/library/ms533046.aspx.

Hands-on Lab Summary

The purpose of this article was to show how straightforward it is to interact with Cookie data through the `HttpRequest` and `HttpResponse` classes within Padarn-hosted web applications. Because the programming model closely mirrors that of ASP.NET, you'll find that this is yet another area of feature support that you'd expect from a desktop-class web server.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Walked through a simple illustration of an authentication routine backed by Padarn
- ✓ Learned how to read from and write to cookies
- ✓ Watched cookie data appear in a packet sniffer application
- ✓ Saw how to modify and delete cookie data from a user's machine

Cookies have been around for many years and remain a basic mechanism by which to store and retrieve user preferences. While cookies have various technical limitations, in general they serve to improve the usability of common websites.