

HOL P302 - Improving Page Performance with Caching

Table of Contents

Table of Contents	2
HOL Requirements	3
Summary.....	4
Lab Objective	4
1.1. Background	4
1.2. Expiration	5
1.3. Validation	6
1.4. A Special Note about Firefox.....	7
Exercise 1: Configuring a basic Padarn web resource	9
Exercise 2: The code-behind logic for CacheExample page	14
Exercise 3: Configuring Padarn to use caching.....	15
Exercise 4: Verify Caching Behavior.....	16
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 discover an important technique in building high-performance, resource friendly web applications under Padarn called caching. The lab will lead you through several exercises to first understand how caching is defined by the HTTP 1.1 definition then see caching in action to verify its impact on network traffic.

Lab Objective

Upon completion of this lab, you will be familiar with Padarn's caching feature support and have configured Padarn to enable caching to drive the point home.

In this HOL, you will perform the following exercises:

- ✓ Learn how to modify the `ocfhttpd.exe.config` file to enable global caching
- ✓ Create a web application that serves various image files, thus emulating file delivery
- ✓ Watch the impact of caching through a network packet analyzer tool

1.1. Background

HOL 112 focused a good bit of attention on the nature of HTTP headers and how Padarn's implementation matches with that of HTTP 1.1. We continue that investigation with our discussion of caching.

Caching pertains to user experience when browsing web applications or other web resources. Since a connection a network – even if a high speed broadband type – is nearly always slower to access than resources local to a client accessing those entities. Your hard disk can read well over 50 megabytes per second, whereas a standard cable modem only can pull down data at around 1 megabyte per second (on an 8 mbps channel). As such, it's prudent to allow content that was recently downloaded onto a client machine to redisplay that content if rules dictate it doesn't need to be refreshed. If a user accesses OpenNETCF.com at 6:00 AM, it's likely that many of the image data, cascading style sheets (CSS), and JavaScript (JS) files haven't changed if the user navigates back five minutes later. HTTP 1.1 specifies the rules for how and when content is cached. The W3 establishes the crux of caching in its RFC 2616:

Caching would be useless if it did not significantly improve performance. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. The former reduces the number of network round-trips required for many operations; we use an "expiration" mechanism for this purpose. The latter reduces network bandwidth requirements; we use a "validation" mechanism for this purpose.

Not only does caching offer the promise of reduced user wait time, it can also greatly decreased stress put on a web server. In the case of Padarn, where the server host is generally a low-powered machine that cannot simultaneously serve as many users as a full IIS 6.0/7.0 setup, this offers immediate benefits. However, since web applications tend to be data driven and have a blend of

dynamic and static content, it's necessary to leverage some of the cache-related headers in the official specification to ensure the user sees fresh content where appropriate. In this article, we will discuss caching and how it pertains to Padarn 1.2.X. For greater detail on the contracted agreement between HTTP 1.1-compliant server and client, please check out the RFC 2616

There are two basic models of dealing with caching in HTTP.

1.2. Expiration

Much as a product purchased at a supermarket has a specified date at which point its contents are useless, any object transferred over HTTP can be decorated with a point at which requester will be forced to re-download the content. If this is done over a web browser then this is handled without the user knowing (beyond seeing more network traffic than if it were a cached copy). The expiration model makes the most amount of sense when the content in question changes infrequently and if it does change, the cost of sending an out of date version is low. For things like non-animated images, JavaScript files, or cascading style sheets, it'd be advisable to set an expiration date of a day or greater and then when someone accesses this resource over the course of a day, the user agent (e.g. a web browser) will keep that content stored locally and not re-download it. If, on the other hand, the content in question is something that changes based upon the time of the day, such as a graphical clock, then the expiration date should be an appropriate amount in the future, such as every 30 minutes.

An HTTP Expires header just contains a full date time object indicating the precise moment of resource expiration. An example would be:

```
Expires: Wed, 18 Mar 2009 03:41:33 GMT
```

Although this is simple, this could lead to issues if the server clock gets out of sync. Mobile devices (with exception to those requiring real time functionality) often have inaccurate clocks. To compensate for this, the cache-control header was made available in HTTP 1.1. This method provides more granularity though is still easy to use. An example of cache-control would be:

```
Cache-Control: s-maxage=1400, proxy-revalidate
```

Here is a list of all the directives for the cache-control response:

- **public** – Makes the response (served content) always cacheable even if authentication wouldn't ordinarily allow it
- **private** – Indicates the response is intended for only one user and cannot be placed in a shared cache (a proxy is a shared cache). Parts of the response can be marked as shareable and others marked as non-shareable. If the client is storing the cache in a user-specific location (like a Firefox local profile), then the content can be stored for later usage.
- **no-cache** – If the no-cache directive does not specify a field-name, then a cache cannot use the response future requests without revalidation with the server. This aids in preventing caching even by caches that have been configured to return stale responses.

- no-store – This applies to the entire message and indicates the content is likely sensitive and therefore must not store the message in a non-volatile place in memory. The client should make attempts to clear this content out from volatile memory (RAM) as quickly as possible.
- no-transform – This prevents downstream proxy servers from changing any header values specified by the Content-Encoding, Content-Range, or Content-Type headers (this includes the entity body). For example, it prevents proxies from converting GIF images to PNG.
- s-maxage – For a shared cache (not a private cache), the maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header.
- max-age – Specifies the maximum age of content, in seconds, acceptable to the client unless a max-stale directive is also sent.
- min-fresh – Specifies in seconds up until what point the content expires that the content can be cached.
- max-stale – Specifies the client is willing to accept content that has been expired. A time value can be specified so that only content that has been in an expired state for a particular interval will be read from cache
- must-revalidate – Indicates that the cache must recheck its copy of content when the content has expired. This usually is implicit behavior but for good measure, this is used to better indicate to a network sniffer the intent of the server is to abide by expiration policy.
- proxy-cache – Same as must-revalidate except used in shared caches.

1.3. Validation

This caching model allows a client to query the server to determine if a particular resource is still fresh and therefore can be read from local cache. If the resource is not available in the local cache, it will be re-fetched from the server and stored into the local cache based upon some of the rules appearing above (no-cache trumps the desire to store the content locally). Because this method allows the server to not send the client anything if the client is up-to-date, it can drastically reduce bandwidth consumption as well as server processing pressure. Two headers are used in the validation cache modl:

- Last-Modified – Specifies, clearly, when the particular resource was last modified. File requests might return the timestamp of the local file store, whereas dynamically served pieces might relay the last time the most recent piece was sent to the client. An example of this header would be:

Last-Modified: Wed, 15 Apr 2009 22:13:21 GMT

If a client sends a “If-Modified-Since” request, the timestamp sent in the request can be compared against the resource’s actual last modified timestamp. If there is a match, a 304 “Not Modified HTTP” is returned.

- ETag (Entity Tag) – Etag has replaced Last-Modified in HTTP 1.1. This is a string that is uniquely linked to a particular resource and contains an MD5 hashed value of the resource

or resource URL with the timestamp indicating its last date of modification. An ETag can be generated at the time of initial resource request, then queried at a later point with the second ETag being compared to the first. If the ETags differ for the same resource, then clearly the resource in cache is out of date. An example of an ETag would be:

```
ETag: W/"abcd1234"
```

It is important to realize that a week ETag (as shown above) will likely get confused by minor differences between resource versions, such as when a newline is added and no additional text is inserted into a document file. If a client sends a "If-None-Match" request, the returned ETag is compared to the cache-contained resource and if it matches, a 304 "Not Modified HTTP" is returned.

1.4. A Special Note about Firefox

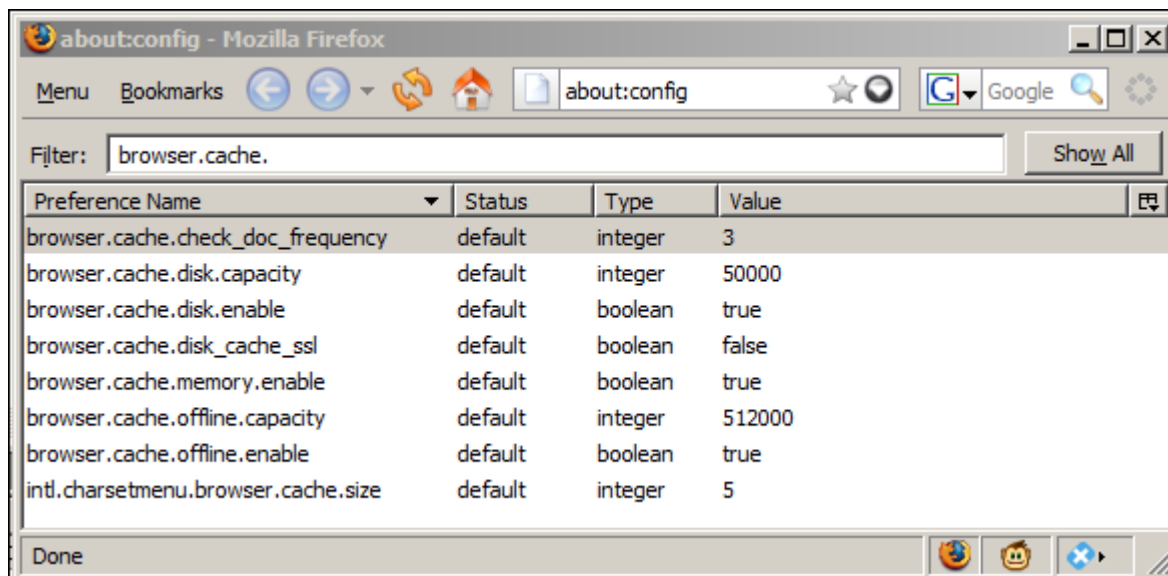
The Mozilla line of browsers has numerous features in place to minimize the effects of a slow network connection. Because Firefox and Gecko variants run on slim, embedded devices as well as powerful professional workstations, several design decisions have been made that surround HTTP caching rules to speed up navigation. All documents are cached by default in Mozilla. This is done so that back/forward, saving, viewing-as-source, etc. can all be done without revisiting the server. Even when a no-cache response header is sent, documents are still cached in the private, non-volatile profiles folder.

Expiration dates are strictly adhered to but in situations where content does not have one available, the following equation is used:

$$\text{expirationTime} = \text{responseTime} + \text{freshnessLifetime} - \text{currentAge}$$

`responseTime` is determined by the browser and is the timestamp at which point the particular resource was received by the browser.

Otherwise, Firefox behaves as expected given the rules set forth in RFC 2616 – pressing "Refresh" in the browser window forces a revalidation of all content displayed in the current page. Furthermore, it is possible to require revalidation on every page transaction by modifying for about:config settings.



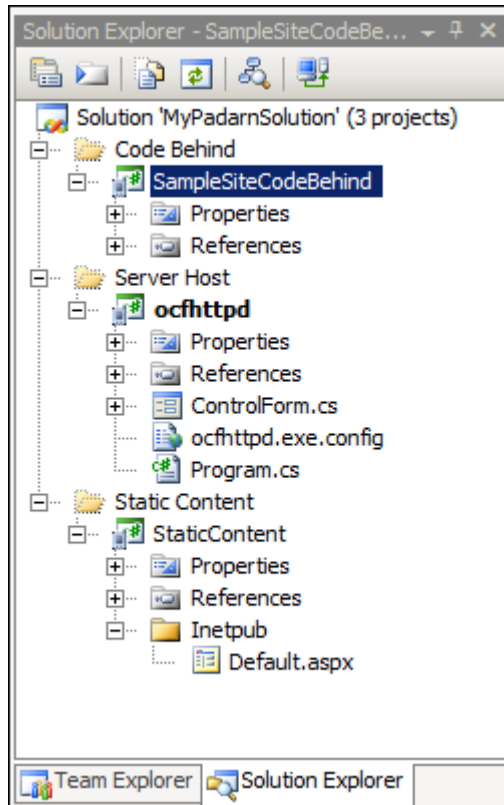
By modifying the `browser.cache.check_doc_frequency` value, you can alter the caching behavior of Firefox (from Mozilla's MozillaZine):

0. Check for a new version of a page once per session (a session starts when the first application window opens and ends when the last application window closes).
1. Check for a new version every time a page is loaded.
2. Never check for a new version - always load the page from cache.
3. Check for a new version when the page is out of date. (Default)

It is important to realize that cache settings established in Padarn might be ignored or unexpectedly acted upon by client browsers!

Exercise 1: Configuring a basic Padarn web resource

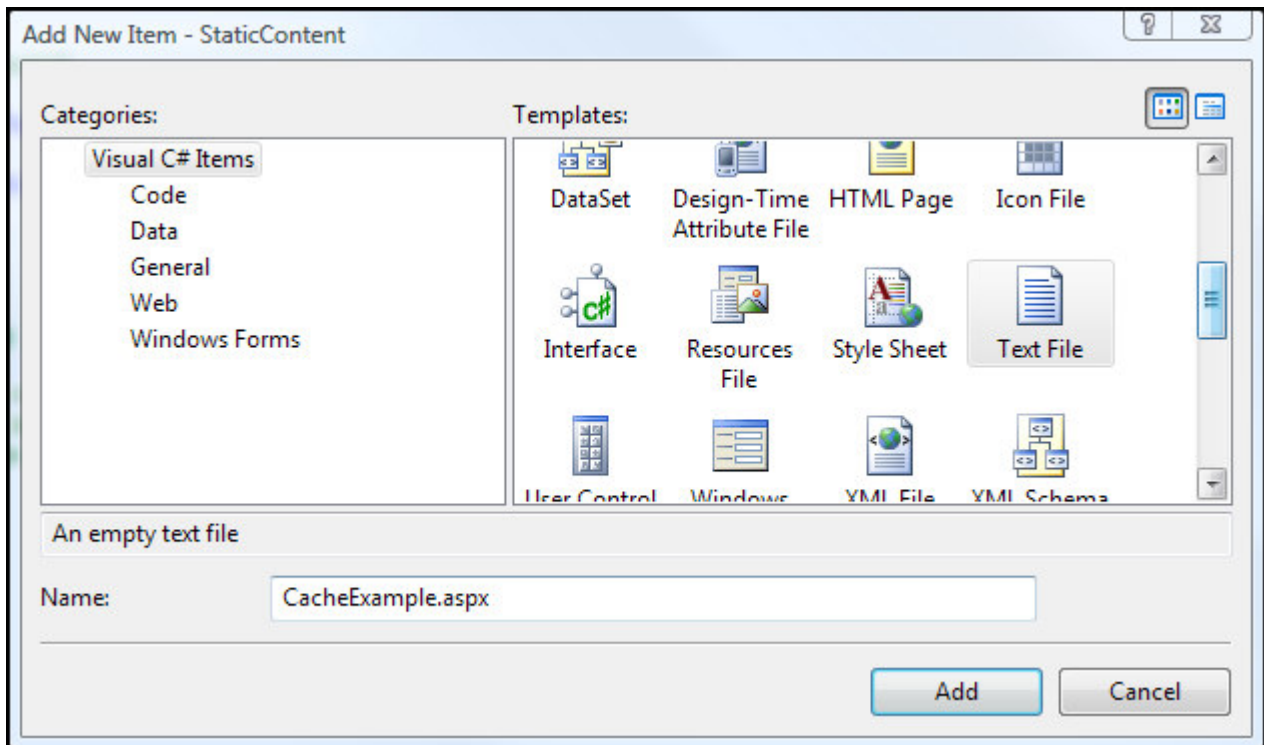
In this exercise, we will configure Padarn to serve a single Active Server Page (ASP) and we'll create some basic HTML as the test harness. To begin, please open the Visual Studio 2008 solution entitled "MyPadarnSolution.sln". Here is how the Solution Explorer displays the contents of this solution:



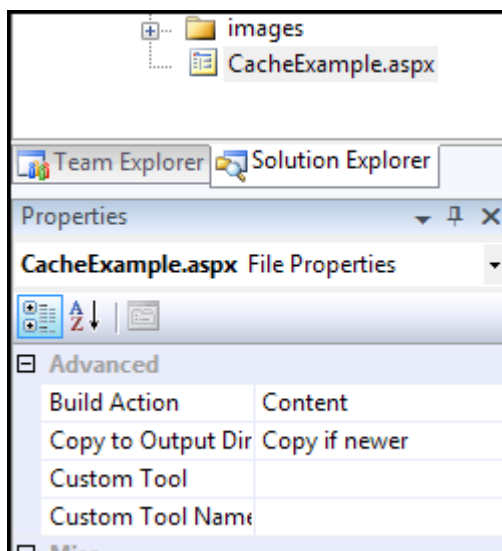
The two projects listed are a sample Padarn web server launch utility (Smart Device project) and an empty project to hold static content. We will now add a single Active Service Page (ASP) to the StaticContent project as our protected (and default) webpage for the Padarn instance.

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 "CacheExample.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 CacheExample.aspx file
6. In the Properties pane, set the Build Action for Default.aspx to "Content"
7. In the Properties pane, set the Copy to Output Directory property for Default.aspx to "Copy if Newer"



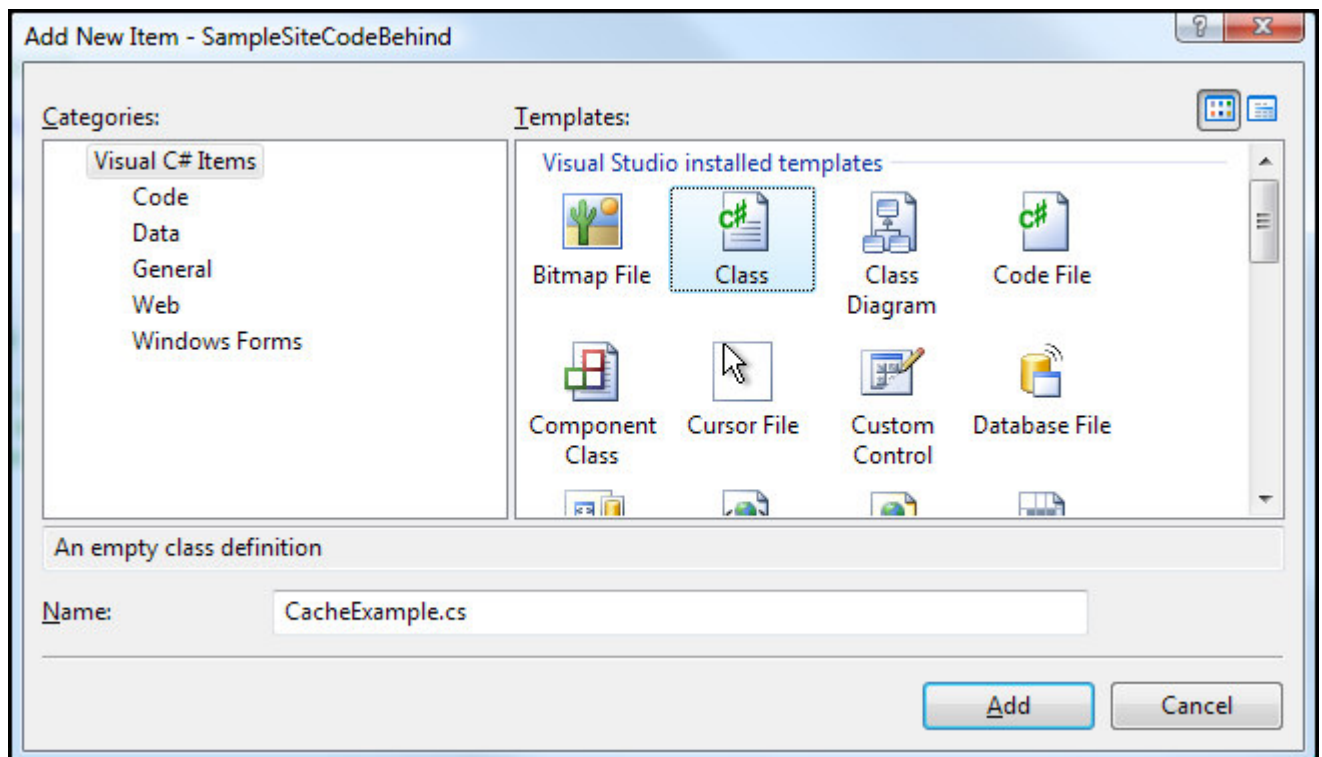
8. Paste the following code into the newly-created Default.aspx file:

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

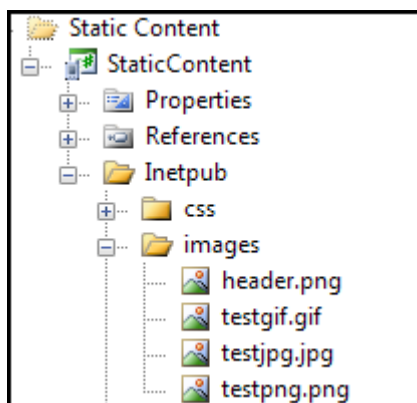
Next, you will use Visual Studio to create classes containing the code-behind logic for the sample startup webpage. The Padarn server will create an instance of the main class and run the Page_Load method of the class when a client browser (or Net client) accesses the Default.aspx page.

To create the main code-behind class:

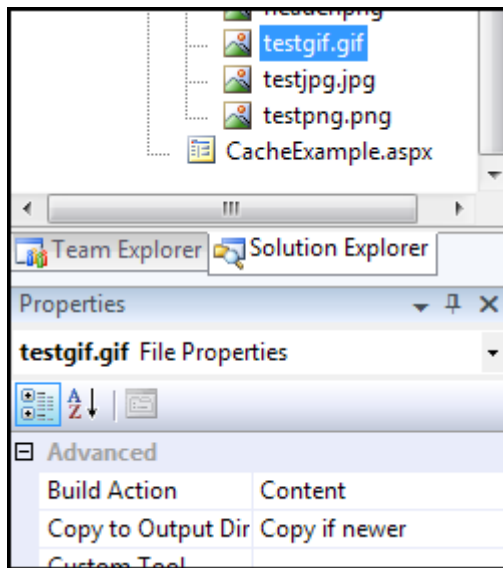
1. In Visual Studio's Solution Explorer pane, right-click on the SampleSiteCodeBehind project and select **Add -> New Item** to display the Add New Item dialog.
2. From the list of Visual Studio Installed Templates select **Class**.
3. In the Name textbox, enter "CacheExample.cs".



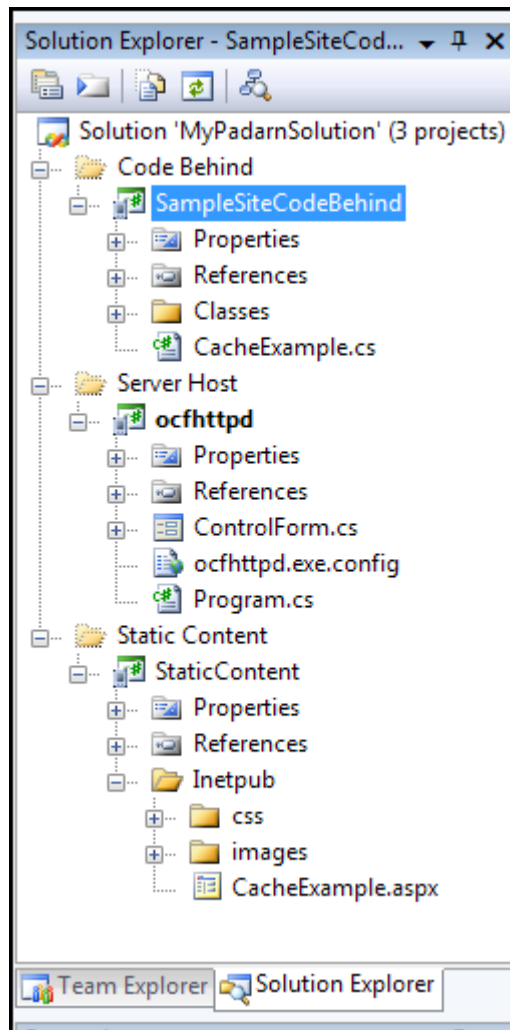
4. Click **Add** to add the new file to the project
5. We will be using several test images throughout this lab. All images need to be set as deployed content. Here is a snapshot of those images:



6. Each image should be set as deployable content:



7. When all is done, the solution should look like the following:



Exercise 2: The code-behind logic for CacheExample page

The Page_Load function that will be established for this lab does little more than display three images of type GIF, JPEG, and PNG. We will maintain the total execution time, as captured by Padarn, and display that along with the images to the user. By utilizing the caching support of Padarn, we'll be able to test whether the graphic files are being pulled freshly from Padarn or if they're being loaded from the client browser's cache.

1. Let's create the structure of the Page_Load function.

```
protected override void Page_Load(object sender, EventArgs e)
{
    int startTime = Environment.TickCount;

    Document page = new Document();

    int et = Environment.TickCount - startTime;

    Paragraph loadTimeParagraph = new Paragraph(string.Format("Page loaded in {0:0.000} sec",
        (float)((float)et / 1000f)));

    loadTimeParagraph.Styles.Add(new ElementStyle("font-size", "10px"));
    loadTimeParagraph.Styles.Add(new ElementStyle("text-align", "left"));
    page.Body.Elements.Add(loadTimeParagraph);

    Response.Write(page.OuterHtml);
    Response.Flush();
}
```

We'll use the Environment tick count to keep track of runtime. The typical OpenNETCF.Web.HTML classes are used to create an HTML document and add formatting instructions.

2. To display the three image files, we use the Image class in-line.

```
Div headerImageDiv = new Div("header");
headerImageDiv.Elements.Add(new Image("/images/testgif.gif", "Test GIF"));
headerImageDiv.Elements.Add(Generator.LineBreak);
headerImageDiv.Elements.Add(new Image("/images/testjpg.jpg", "Test JPG"));
headerImageDiv.Elements.Add(Generator.LineBreak);
headerImageDiv.Elements.Add(new Image("/images/testpng.png", "Test PNG"));

page.Body.Elements.Add(headerImageDiv);
page.Body.Elements.Add(Generator.LineBreak);
```

Exercise 3: Configuring Padarn to use caching

Padarn v1.2 and later versions support global caching profiles. This means that settings established apply to all documents loaded across the entire set of web applications hosted by Padarn. In future releases, support for individual domains or virtual directories might be supported. Cache profiles are set by document type, which is usually a document extension. For each profile, Padarn supports:

- Extension: The document type to be filtered
- Location: Clarifies where the document caching will take place.
 1. Downstream. The output cache can be stored in any HTTP 1.1 cache-capable devices other than the origin server. This includes proxy servers and the client that made the request.
 2. Client. The output cache is located on the browser client where the request originated.
 3. None. The output cache is disabled for the requested page.

The default location is Client.

- Duration: Specifies the time that the page or user control is cached, in seconds. The default is 00:00:30. Some thought should be put into the duration period, given the nature of the content being served and expected demands placed on the server.

Applying one or several profiles to a Padarn instance is done by modifying the `ocfhttpd.exe.config` file in the `MyPadarnSolution\ocfhttpd` project. These settings exist within the `WebServer` section:

```
<Caching>
  <Profiles>
    <add extension=".bmp" location="Downstream" /> <!-- defaults to 00:00:30 -->
    <add extension=".css" location="Client" duration="00:01:00" />
    <add extension=".gif" location="Client" duration="00:05:00" />
    <add extension=".png" location="Client" duration="00:05:00" />
    <add extension=".jpg" location="Client" duration="00:05:00" />
  </Profiles>
</Caching>
```

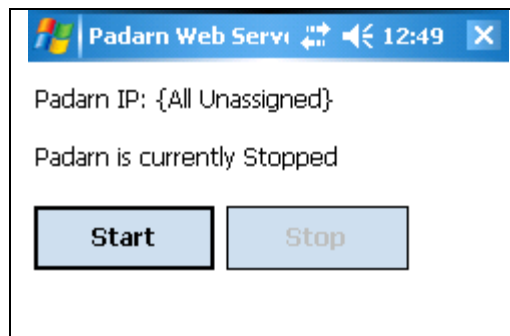
It is important that the `ocfhttpd.exe.config` is deployed as content and is pushed whenever changes to the file are made. Notice that we'll be requesting that our three image types are cached by the client browser (following the rules explained at the top of this guide).

Our settings establish client-side caching for Cascading Style Sheets (CSS), GIF images, PNG images, and JPG images. Bitmap (BMP) images will be cached at the server. The three image types that show up in our code-behind for this HOL will be marked as fresh for 5 minutes past download.

Exercise 4: Verify Caching 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; furthermore, you should validate that CacheExample.aspx has been deployed as static content to that Padarn's instance inetpub folder. Instructions for how to accomplish this are in HOL 100.

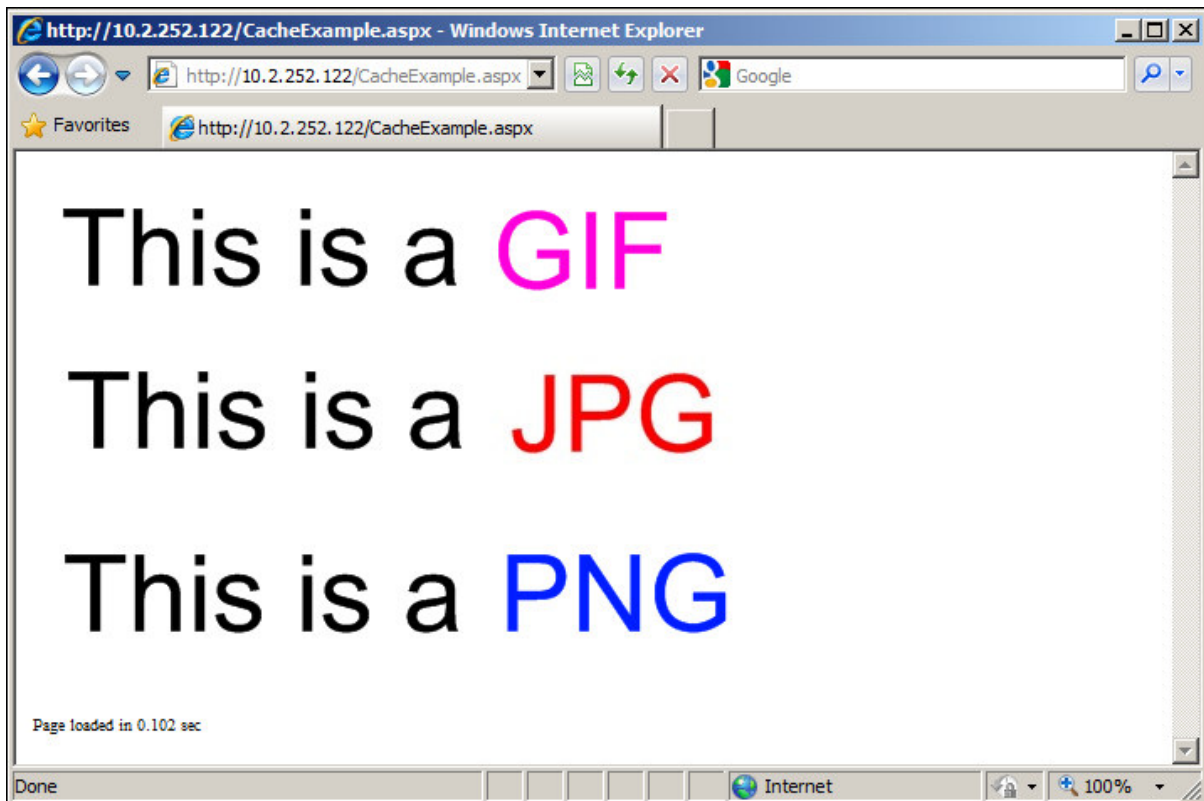
1. Initiate Padarn through the MyPadarnSolution Visual Studio 2008 solution. Right-click on the ocfshttpd project and select **Debug -> Start Without Debugging** (or press Ctrl + F5). For this demonstration, we will be using the Windows Mobile 5.0 Pocket PC R2 Emulator.



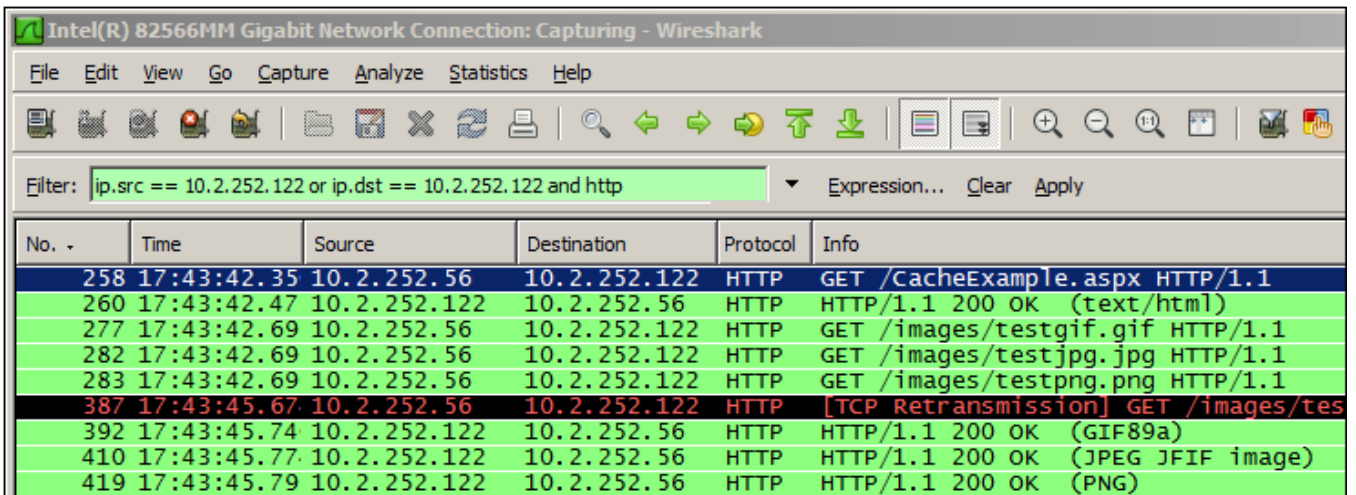
2. Press the **Start** button to initiate the Padarn instance.



3. 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. Since the CacheExample.aspx code behind does not parse any POST input, the page will always display the same content. To see the result of the web application we've created, go to `http://<Padarn IP Address>/CacheExample.aspx`. Here is what you will see:



5. The first time the page is visited, as would be expected, each image must be pulled down freshly into the browser cache.

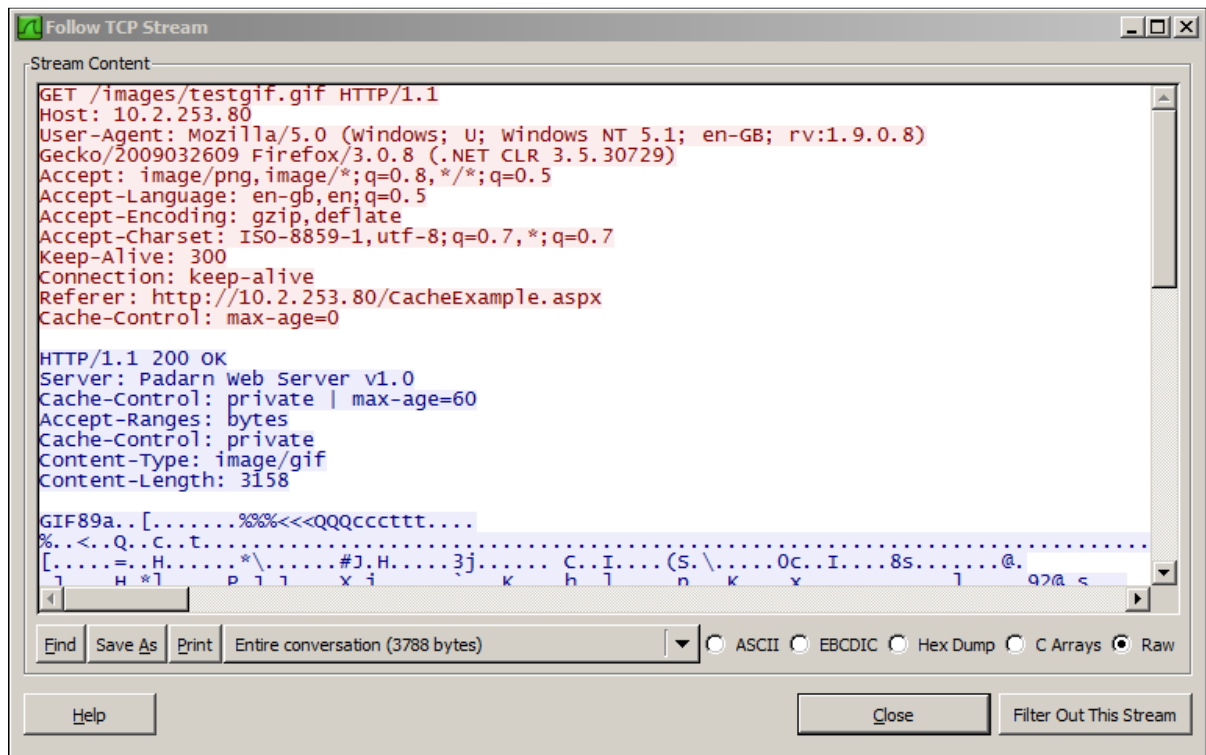


No.	Time	Source	Destination	Protocol	Info
258	17:43:42.35	10.2.252.56	10.2.252.122	HTTP	GET /CacheExample.aspx HTTP/1.1
260	17:43:42.47	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 200 OK (text/html)
277	17:43:42.69	10.2.252.56	10.2.252.122	HTTP	GET /images/testgif.gif HTTP/1.1
282	17:43:42.69	10.2.252.56	10.2.252.122	HTTP	GET /images/testjpg.jpg HTTP/1.1
283	17:43:42.69	10.2.252.56	10.2.252.122	HTTP	GET /images/testpng.png HTTP/1.1
387	17:43:45.67	10.2.252.56	10.2.252.122	HTTP	[TCP Retransmission] GET /images/tes
392	17:43:45.74	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 200 OK (GIF89a)
410	17:43:45.77	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
419	17:43:45.79	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 200 OK (PNG)

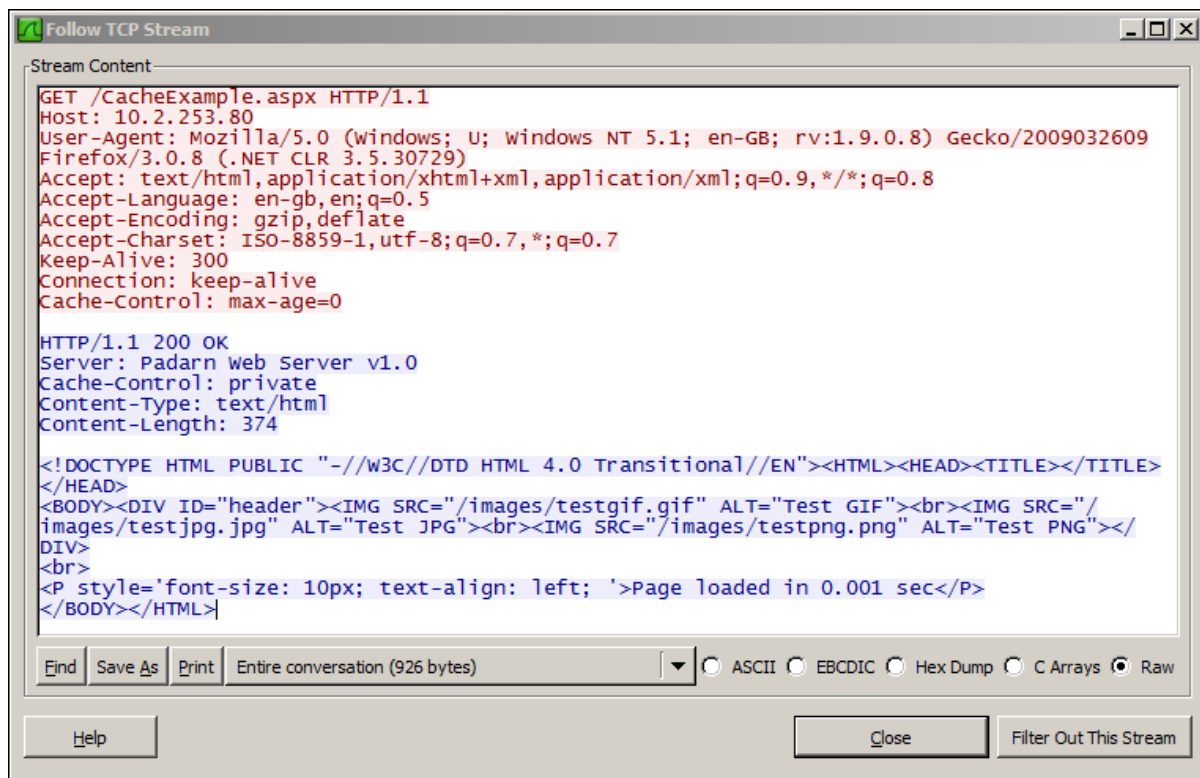
Once the page fully loads, we can use a network sniffer tool such as Wireshark ([information here](#)) to determine all headers exchanged between Padarn and the desktop instance. Above is a brief snapshot of what we would expect to see on the server. The following events have taken place:

1. The CacheExample.aspx page is requested from the server
2. The CacheExample.aspx is received and has a status of 200 (OK)
3. The first image, testgif.gif, is requested from the server (cache miss)

4. The second image, testjpg.jpg, is requested from the server (cache miss)
5. The third image, testpng.png, is requested from the server (cache miss)
6. Testgif.gif is received and has a status of 200 (OK). Content type is GIF89a
7. Testjpg.jpg is received and has a status of 200 (OK). Content type is JPEG
8. Testpng.png is received and has a status of 200 (OK). Content type is PNG.

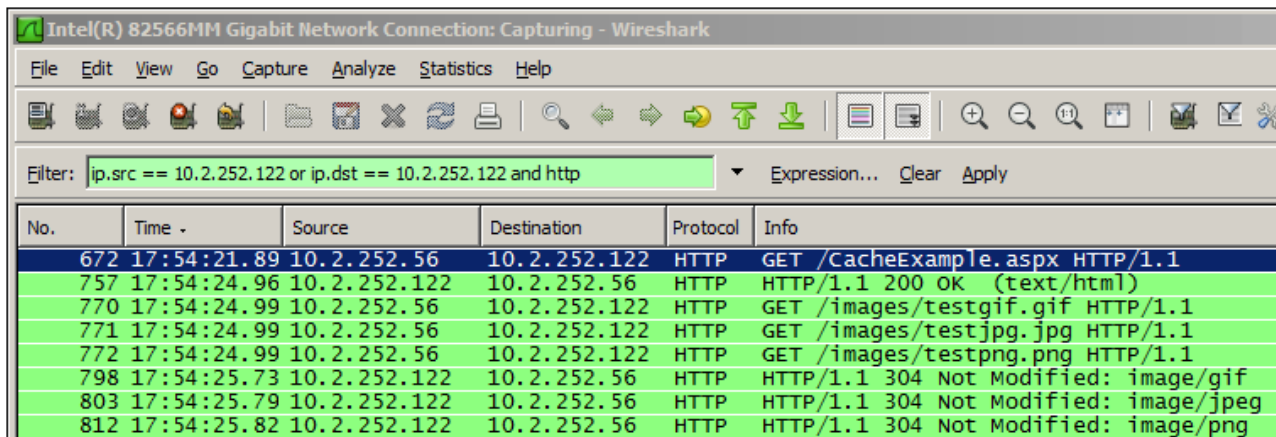


If we look at the entire TCP stream of an image capture, we see that cache-control setting is left at private, with a maximum age of 60. Since the image had not yet been placed in the local browser cache, this will be treated as a normal download.



Nothing special exists in the TCP Stream for the CacheExample.aspx document.

If we immediately refresh the page, all of the images will be requested.



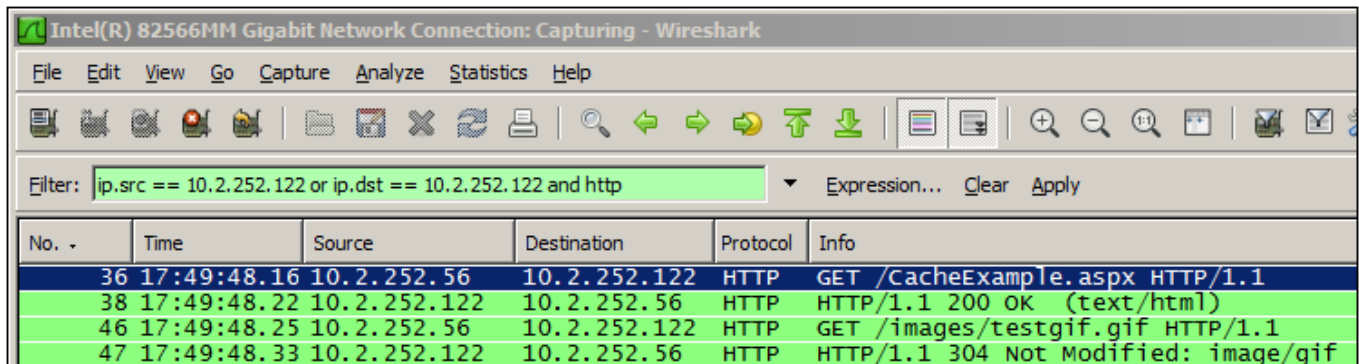
No.	Time	Source	Destination	Protocol	Info
672	17:54:21.89	10.2.252.56	10.2.252.122	HTTP	GET /CacheExample.aspx HTTP/1.1
757	17:54:24.96	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 200 OK (text/html)
770	17:54:24.99	10.2.252.56	10.2.252.122	HTTP	GET /images/testgif.gif HTTP/1.1
771	17:54:24.99	10.2.252.56	10.2.252.122	HTTP	GET /images/testjpg.jpg HTTP/1.1
772	17:54:24.99	10.2.252.56	10.2.252.122	HTTP	GET /images/testpng.png HTTP/1.1
798	17:54:25.73	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 304 Not Modified: image/gif
803	17:54:25.79	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 304 Not Modified: image/jpeg
812	17:54:25.82	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 304 Not Modified: image/png

Because all of the images requested exist in a fresh state in the local browser cache, they all report HTTP 304 and no download occurs.

If we decide that one of our document types, such as GIF, should not be cached, we will need to modify the Padarn configuration file:

```
<Caching>
  <Profiles>
    <add extension=".bmp" location="Downstream" /> <!-- defaults to 00:00:30 -->
    <add extension=".css" location="Client" duration="00:01:00" />
    <add extension=".gif" location="None" duration="00:05:00" />
    <add extension=".png" location="Client" duration="00:05:00" />
    <add extension=".jpg" location="Client" duration="00:05:00" />
  </Profiles>
</Caching>
```

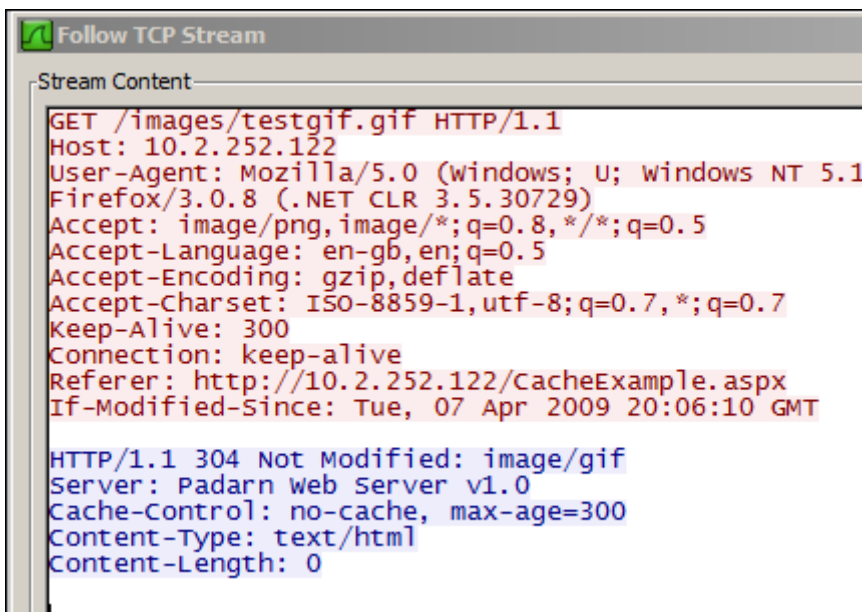
If we type in the URL `http://<Padarn IP Address>/CacheExample.aspx`, the PNG and JPG files will not be requested but the GIF file will be. This is because Padarn is being instructed to always pull down the GIF regardless of freshness.



No.	Time	Source	Destination	Protocol	Info
36	17:49:48.16	10.2.252.56	10.2.252.122	HTTP	GET /CacheExample.aspx HTTP/1.1
38	17:49:48.22	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 200 OK (text/html)
46	17:49:48.25	10.2.252.56	10.2.252.122	HTTP	GET /images/testgif.gif HTTP/1.1
47	17:49:48.33	10.2.252.122	10.2.252.56	HTTP	HTTP/1.1 304 Not Modified: image/gif

The following events have taken place:

1. The CacheExample.aspx page is requested from the server
2. The CacheExample.aspx is received and has a status of 200 (OK)
3. The first image, testgif.gif, is requested from the server (cache *miss*).
4. The second image, testjpg.jpg, is requested and is pulled from local browser cache (cache hit)
5. The third image, testpng.png, is requested and is pulled from local browser cache (cache hit)
6. Testgif.gif on the server matches what's in local browser cache so no download occurs. Result is an HTTP 304 code.

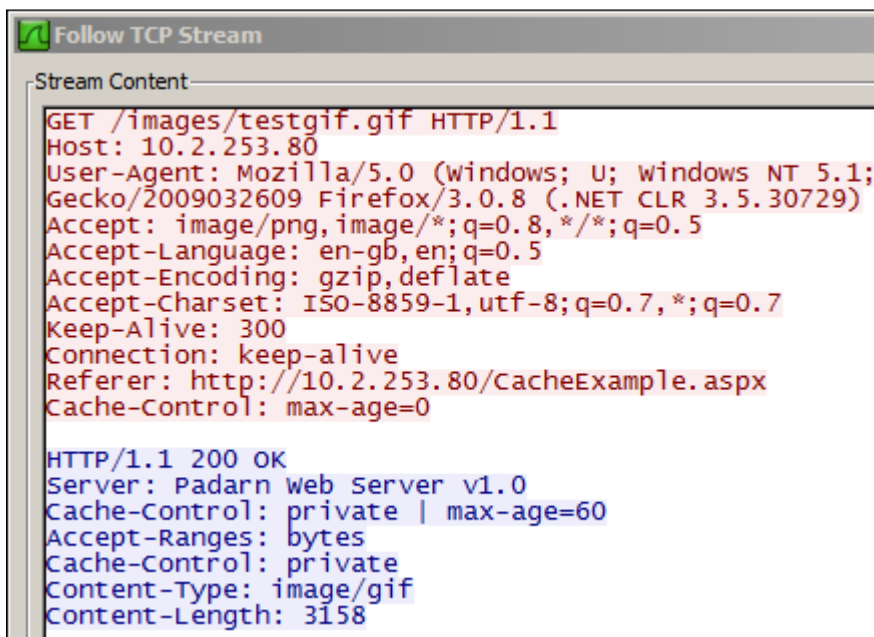


```
Follow TCP Stream
Stream Content
GET /images/testgif.gif HTTP/1.1
Host: 10.2.252.122
User-Agent: Mozilla/5.0 (windows; U; windows NT 5.1
Firefox/3.0.8 (.NET CLR 3.5.30729)
Accept: image/png, image/*; q=0.8, */*; q=0.5
Accept-Language: en-gb,en; q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8; q=0.7, *; q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://10.2.252.122/CacheExample.aspx
If-Modified-Since: Tue, 07 Apr 2009 20:06:10 GMT

HTTP/1.1 304 Not Modified: image/gif
Server: Padarn web Server v1.0
Cache-Control: no-cache, max-age=300
Content-Type: text/html
Content-Length: 0
```

HTTP 304 on GIF image

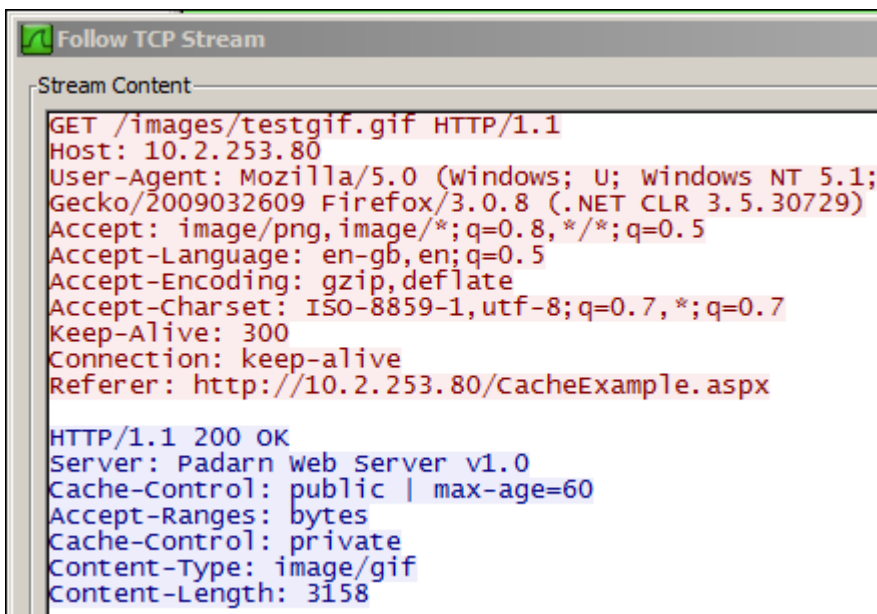
As mentioned above, three modes of cache location are supported: Client, Server, or None. As per the HTTP 1.1 specification, only the Cache-Control directive will change. See [Expiration](#). TCP streams for the three cache locations are shown below:



```
Follow TCP Stream
Stream Content
GET /images/testgif.gif HTTP/1.1
Host: 10.2.253.80
User-Agent: Mozilla/5.0 (windows; U; windows NT 5.1;
Gecko/2009032609 Firefox/3.0.8 (.NET CLR 3.5.30729)
Accept: image/png, image/*; q=0.8, */*; q=0.5
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://10.2.253.80/CacheExample.aspx
Cache-Control: max-age=0

HTTP/1.1 200 OK
Server: Padarn web Server v1.0
Cache-Control: private | max-age=60
Accept-Ranges: bytes
Cache-Control: private
Content-Type: image/gif
Content-Length: 3158
```

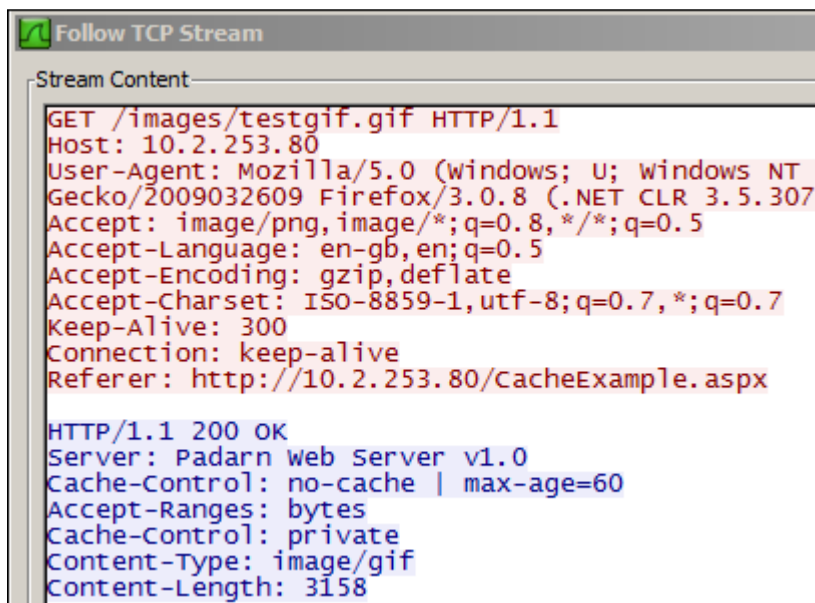
Client (Local browser profile-stored, hence private)



```
Follow TCP Stream
Stream Content
GET /images/testgif.gif HTTP/1.1
Host: 10.2.253.80
User-Agent: Mozilla/5.0 (windows; U; windows NT 5.1;
Gecko/2009032609 Firefox/3.0.8 (.NET CLR 3.5.30729)
Accept: image/png,image/*;q=0.8,*/*;q=0.5
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://10.2.253.80/CacheExample.aspx

HTTP/1.1 200 OK
Server: Padarn Web Server v1.0
Cache-Control: public | max-age=60
Accept-Ranges: bytes
Cache-Control: private
Content-Type: image/gif
Content-Length: 3158
```

Downstream (Proxy Server Cached, hence public)



```
Follow TCP Stream
Stream Content
GET /images/testgif.gif HTTP/1.1
Host: 10.2.253.80
User-Agent: Mozilla/5.0 (windows; U; windows NT
Gecko/2009032609 Firefox/3.0.8 (.NET CLR 3.5.307
Accept: image/png,image/*;q=0.8,*/*;q=0.5
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://10.2.253.80/CacheExample.aspx

HTTP/1.1 200 OK
Server: Padarn Web Server v1.0
Cache-Control: no-cache | max-age=60
Accept-Ranges: bytes
Cache-Control: private
Content-Type: image/gif
Content-Length: 3158
```

No Cache

Hands-on Lab Summary

Although the full ASP.NET supports page, partial page (fragment), and data caching, Padarn supports only page-level caching, configured at the global level, as of v1.2 of the product. Because of the limited resources available on the host hardware of a Padarn instance, it makes little sense to store a copy of user's data at level; instead, it's best to denote the client-side copy of a particular entity is sufficiently fresh to be reused during subsequent request until the freshness expires. For these reasons, only Client and Downstream caching modes are available.

It should be clear from reading through the [background](#) section above why caching data will save a Padarn server from being needlessly pegged over the course of its operation.

In this lab you:

- ✓ Re-used an existing your Padarn website solution
- ✓ Made a changes to the Padarn ocfhttpd.exe.config to enable caching
- ✓ Built a web application that serves several cacheable data entities (images) and displays page load time
- ✓ Verified through a network packet sniffer that content got result codes of 200 (OK) when cache misses occurred and 304 (Not Modified) when cache hits occurred
- ✓ Made a simple "Hello World" style test landing page for verification of a valid authentication

Please make use of Padarn's caching features where content is known to remain static over a period of time so as to free up resources to be dedicated to requests needing real-time processing.