

APPLICATION EXPRESS – DELIVERING PAGES IN 3 SECONDS OR LESS (BECAUSE USERS DON'T LIKE WAITING!)

John Edward Scott, Shellprompt Hosting

INTRODUCTION

It took me a while to come up with the title for this Whitepaper (and associated Presentation) and I am still not entirely happy with it, mainly because it may actually be misleading. Yes that's right, I may have misled you from the start. This paper isn't going to guarantee that your pages will be delivered to your users in less than 3 seconds, since there are far too many varying factors to be able to guarantee that. There is also nothing magical about the figure of 3 seconds, you may find that the users of your application are quite happy to wait 5 seconds, or 7 seconds for certain pages, or conversely they might start to get mad if the page takes more than 2 seconds to render.

So, now that we've established that this whitepaper will not guarantee that your pages are delivered in 3 seconds, what is it *really* about? Well, quite simply it's going to show two, very different, techniques that can potentially have a great impact on the amount of time it takes for your pages to be delivered to the end user.

Why is it important to deliver pages quickly? Well, the obvious answer is that the quicker your can deliver the pages the more productive the end users can be. There is however another reason, a reason that I have seen a number of times in production systems and it is this:

If your users perceive that your application runs slowly, then they will become frustrated with it.

What do I mean by that? Well, they will begin to hate using it, they will find other faults with it (I don't mean finding bugs, because finding bugs is good right?) and they will generally grade their experience with the application as a bad experience. In the worst case, where the users have a choice, they may even abandon using the application altogether. If you are running a commercial website on the Internet, having a 'slow website' could result in users never returning to your website after their first visit. Users do not need to sit with a stopwatch to determine whether a website is slow or not, we (and we're all users, even if we're also developers) are able to gauge extremely quickly whether we believe a site is responding quickly or not.

A WORD ABOUT OPTIMIZING

As I mentioned in the opening introduction, this paper is going to present two different techniques you can use that will have an impact on how quickly your application pages are delivered to the end user. However it is important to stress that these techniques should only form part of your arsenal of tools that you use. There are a huge number of ways in which you can make your application more responsive from the users perspective, from tuning the SQL queries in your report, using AJAX to avoid pages having to be resubmitted, tuning the database itself, tuning the Apache webserver and also the Operating system itself.

I have generally found two schools of thought when it comes to optimization, those who pro-actively look for ways to optimize things and those who believe you can always optimize things 'at the end'. I firmly believe that it is better to confront and investigate potential areas of performance problems as early in your development cycle as possible, otherwise you may find that you end up designing a solution that will be incredibly difficult to modify and optimize later on.

There is, of course, a danger that you can try to over-optimize your solution for example I did once see someone spending a couple of days trying to reduce the time that a query took to run from 10 seconds to 3 seconds, which on the face of it is a good optimization. However to put it into context this was a query which was only ran during a month end batch run (i.e. it only ran 12 times a year). So even though the total saving in time as a percentage was good (a 70% reduction), in real terms the two days spent optimizing that query resulted in only an 84 second time saving in a year. In other words, knowing what to optimize is just as important as the optimizing itself.

THE TECHNIQUES

There were many different techniques and tips I could have discussed in this paper, some of which would only be useful to a very small minority of cases. I decided to present two very different techniques each of which can greatly affect the user perceived performance of your application. Both of these techniques are quite specific, however they can also be used to complement each other and I would definitely recommend investigating both of them for your application.

IMAGE CACHING

If your page contains images that you have stored in the database, then you may find that on every view of that page every individual image is downloaded to the user again regardless of whether the image has changed or not. By taking advantage of image caching we can remove the need for the images to be downloaded each time. This will result in three areas of improvement:

- Reduced bandwidth requirements (the images will not be downloaded multiple times)
- The page will load faster from the user perspective
- Reduced load on the webserver and database (we will avoid repeated requests for the same images).

PAGE COMPRESSION

By default the supplied Oracle HTTP server does not have any form of compression enabled. We can take advantage of using Apache modules, such as *mod_gzip* and *mod_deflate*, which allow us to compress the webserver response before sending it to the users browser. When the users browser receives the compressed response the browser then decompresses it before displaying the result, all of this happens transparently as far as the end user is concerned, in other words your end users do not need to configure anything differently for them to be able to take advantage of compression.

The main aim in compressing the webserver response is to reduce the size of the information that needs to be sent to the browser, certain items will be more compressible than others, for example raw HTML is usually very compressible, as are CSS files, however compressing images such as JPEG's will usually result in a poor compression ratio because the JPEG format has already optimized the size of the file to a high degree. Sometimes, in some rare cases, you can actually end up with a file that is bigger after compression than it was before, so it is important to only compress things that should result in good compression ratios rather than trying to compress everything for the sake of it. Also, you need to be aware that the compression process itself incurs an overhead, i.e. it will involve using processor and memory resources on your webserver, so you need to ensure that your webserver is going to be capable of the extra processing requirements.

By using page compression, we can achieve three areas of improvement:

- Reduced bandwidth requirements (the amount of information transferred to the user will be reduced)
- The pages will load faster from the user perspective (they don't have to wait so long for the information to be transferred).
- A potential increase in the number of requests your webserver is able to handle (it is transferring less information per request, therefore it can service more requests over the same period of time).

TAKE ADVANTAGE OF IMAGE CACHING

To demonstrate the benefits of image caching with Application Express, I'm going to build a simple application that displays a number of images in a report, as you might have with a webstore or inventory type application. Actually it will be very similar to the 'Sample Application' that you can install with Application Express, which I could have used, however to accentuate the effects of image caching I'm going to build my own application which will allow me to use slightly larger images.

BUILDING THE APPLICATION

Firstly we will create a new application, as shown in Figure 1, the application will have two pages called Home and Report (shown in Figures 1 and 2). The reason it will have two pages is so that we can navigate between the two pages to show the effect that image caching will have.

The screenshot shows the Oracle Application Express interface. At the top, there's a navigation bar with 'Home', 'Application Builder', 'SQL Workshop', and 'Utilities'. Below that, a breadcrumb trail reads 'Home > Create Application'. The main content area is titled 'Create Application' and contains the following fields and options:

- Method:** A dropdown menu with 'Name' selected.
- Name:** A text input field containing 'Imaging Caching'.
- Application:** A text input field containing '273'.
- Create Application:** Two radio buttons: 'From scratch' (selected) and 'Based on existing application design model'.
- Schema:** A dropdown menu showing 'JES' with a plus sign to expand it.

Navigation buttons include 'Cancel', '< Previous', and 'Next >'.

Figure 1 – Create a new application

Home > Create Application

Method
Name
Pages
Tabs
Shared Components
Attributes
User Interface
Confirm

Create Application Cancel < Previous Next >

Page	Page Name	Page Type	Source Type	Source	Delete
1	Home	Blank	-	-	✘
2	Report	Blank	-	-	✘

Add Page Add Page

Select Page Type:

Blank Report Form Tabular Form Report and Form

Action: Add blank page to application

Subordinate to Page: - Top Level Page -

Page Name: Page 3

Add Page

Select the type of page you wish to create:

- o **Blank** creates a page with no built-in functionality.
- o **Report** creates a page that contains the formatted result of a SQL query. You can choose to build a report based on a table you select, or based on a custom SQL SELECT statement or a PL/SQL function returning a SQL SELECT statement that you provide.
- o **Form** creates a form to update a single row in a database table.
- o **Tabular Form** creates a form to perform update, insert, and delete operations on multiple rows in a database table.
- o **Report and Form** builds a two page report and form combination. On the first page, users select a row to update. On the second page, users can update the selected table or view.

Figure 2 – Add two new pages to the application.

I've also chosen to use One Level of Tabs (Figure 3) and also to keep the example simple I've chosen to use No Authentication (Figure 4).

Home > Create Application

Method
Name
Pages
Tabs
Shared Components
Attributes
User Interface
Confirm

Create Application Cancel < Previous Next >

Application: 273
Name: Imaging Caching

Tabs:

No Tabs One Level of Tabs Two Levels of Tabs

Figure 3 – Using One Level of Tabs.

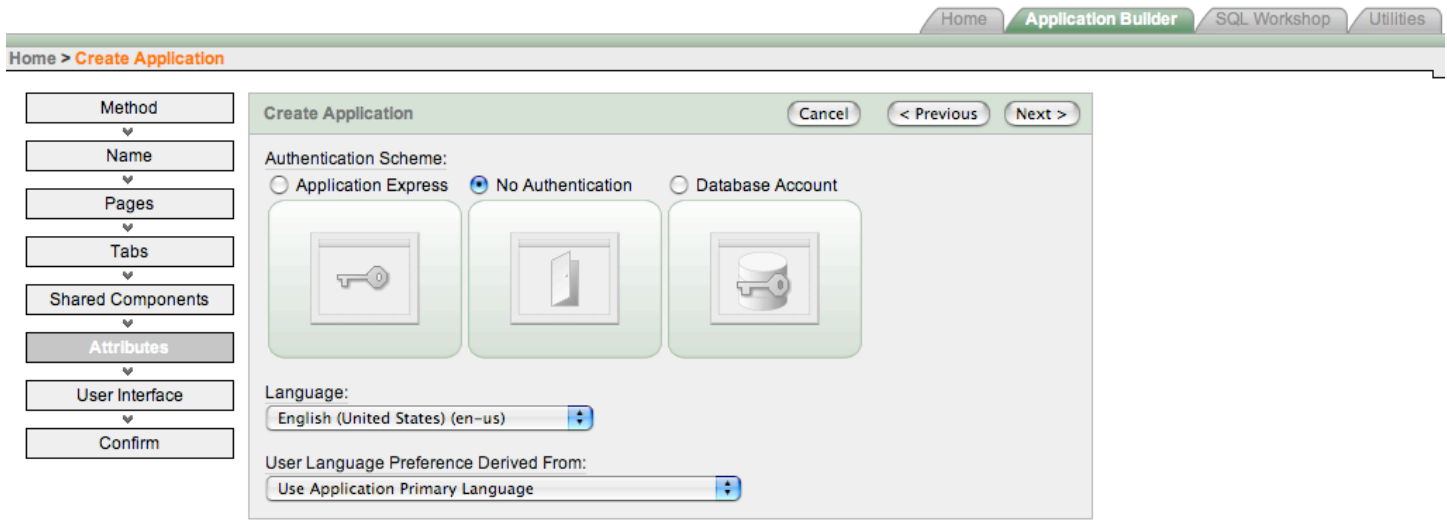


Figure 4 – Using No Authentication.

You can choose whichever theme you like, since it doesn't matter. I've chosen theme 15 since it's one of my favorites. So now we have our skeleton application, which doesn't do much at the moment, other than having a couple of blank pages as shown in Figure 5.

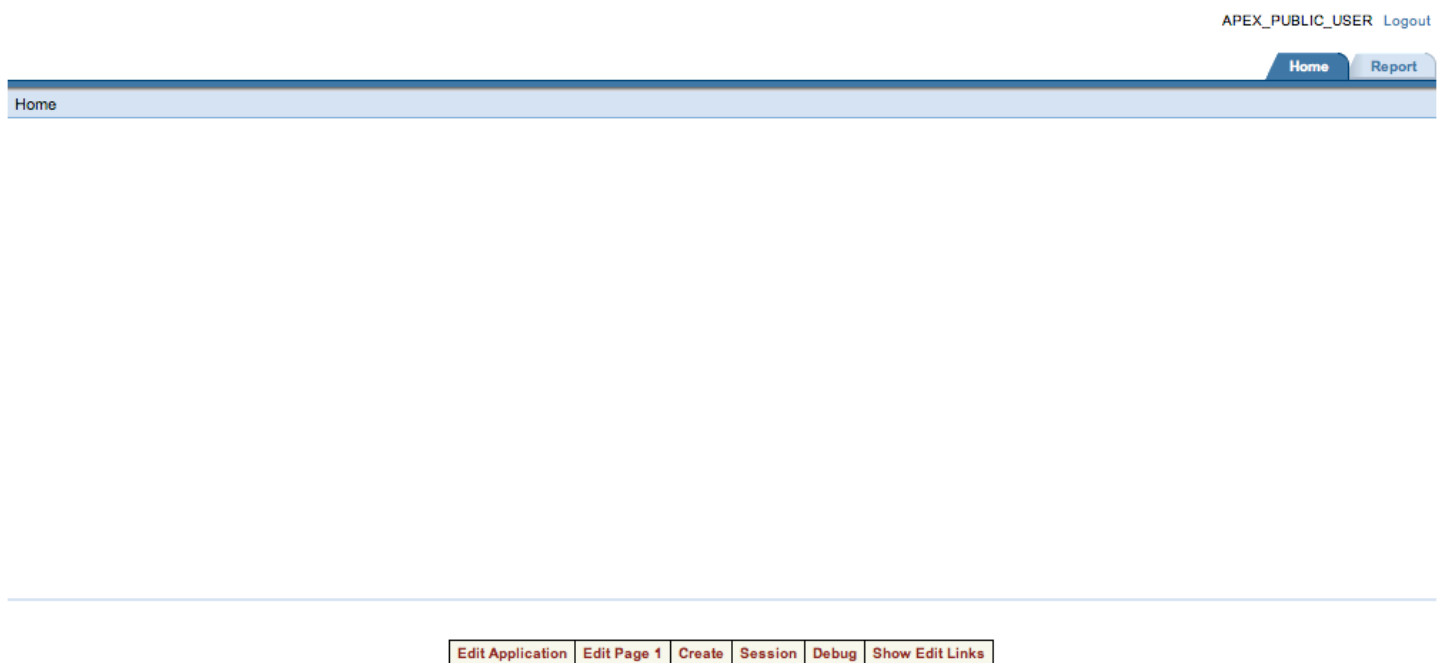


Figure 5 – The skeleton application.

So, to make it a more interesting application we need to do two things –

- Allow the user to upload some images
- Allow the user to view the pictures in the report.

BUILDING THE UPLOAD FORM

We will allow the user to upload images by adding a new region to the home page, as shown in Figure 6 and Figure 7.

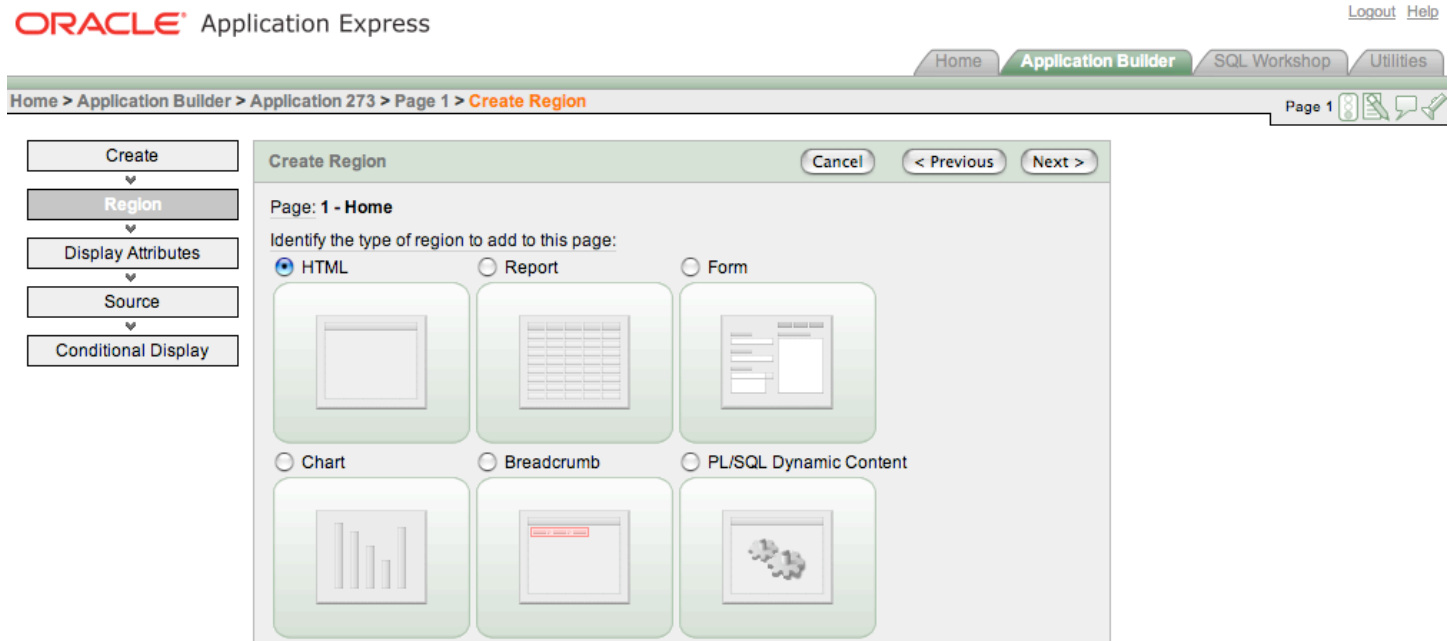


Figure 6 – Add a new HTML region to the Home page.

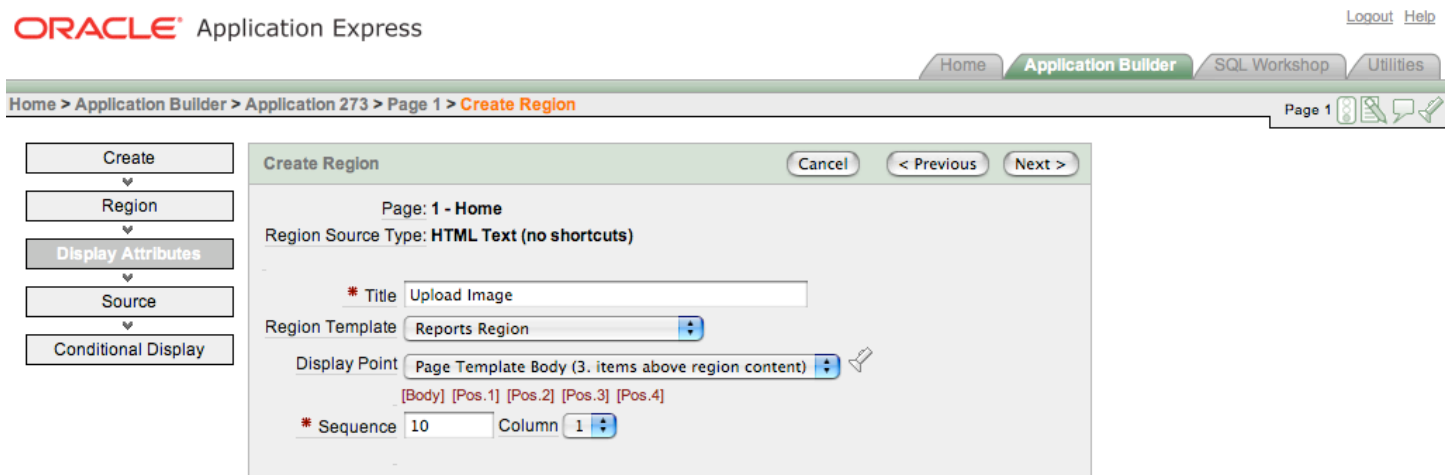


Figure 7 – Add a new HTML region to the Home page.

We will leave the region source empty, since all we are going to do with this region is to add a couple of page items to it. Firstly we add a File Browse item, as shown in Figure 8.

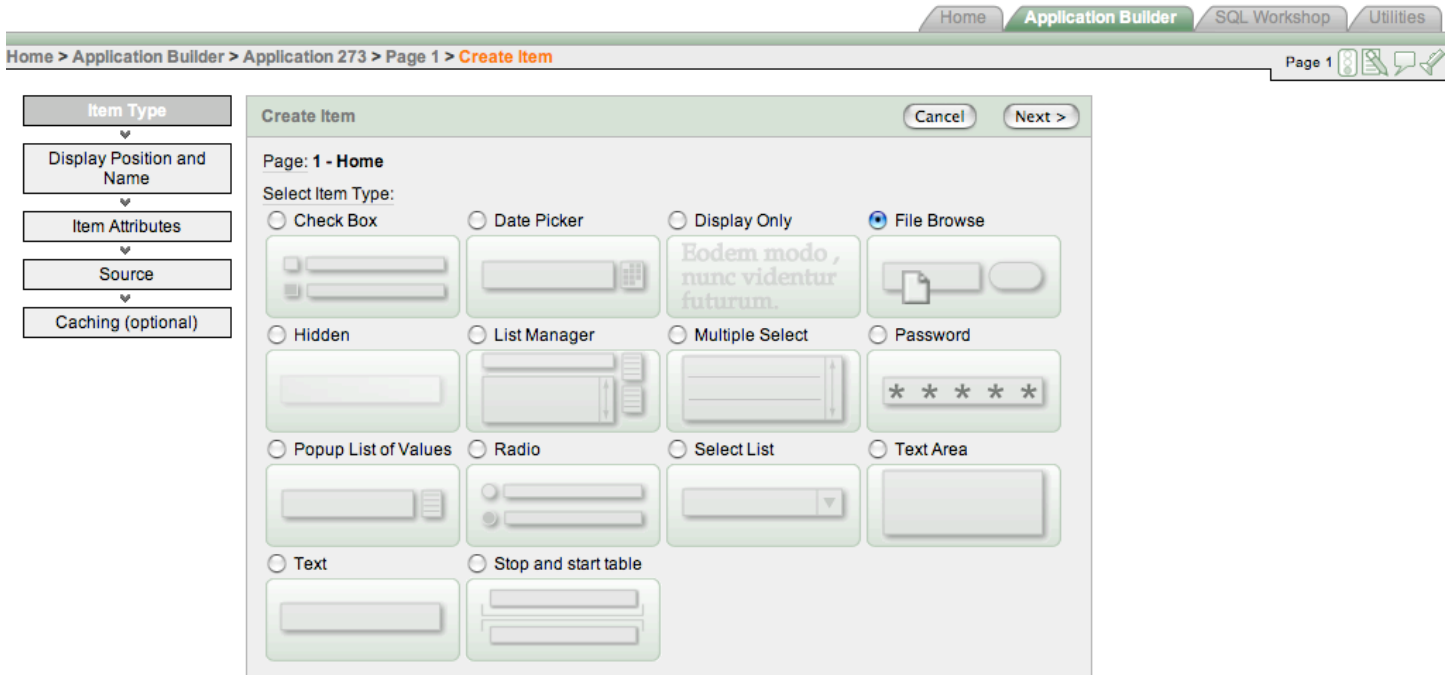


Figure 8 – Add a File Browse item to the Home page.

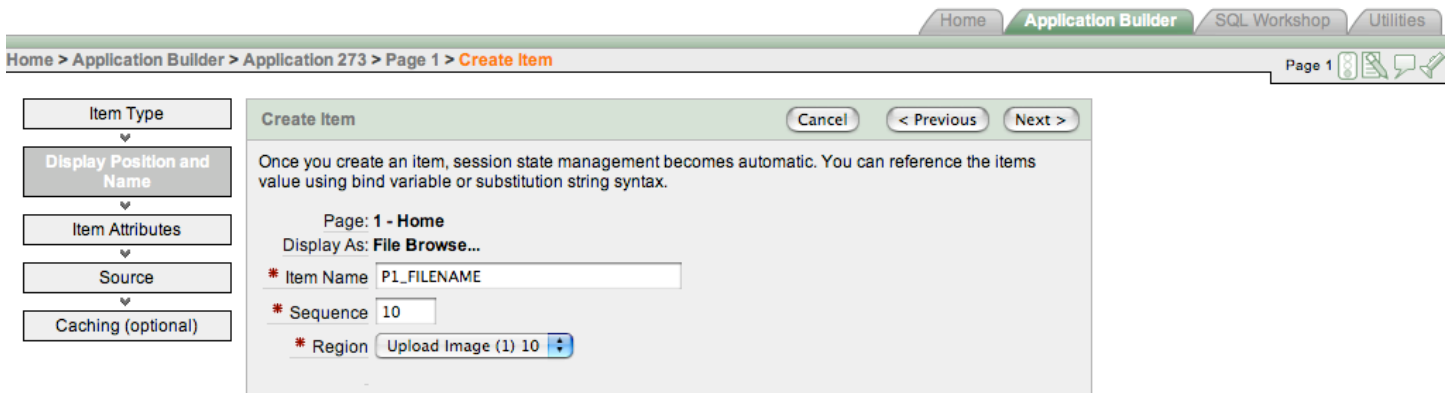


Figure 9 – Add a File Browse item to the Home page.

I'll also add a button to the form that will be used to submit the page, as shown in Figure 10. Note that I'm making the button submit back to the Home page, so after the user has uploaded an image they will be able to upload another one.

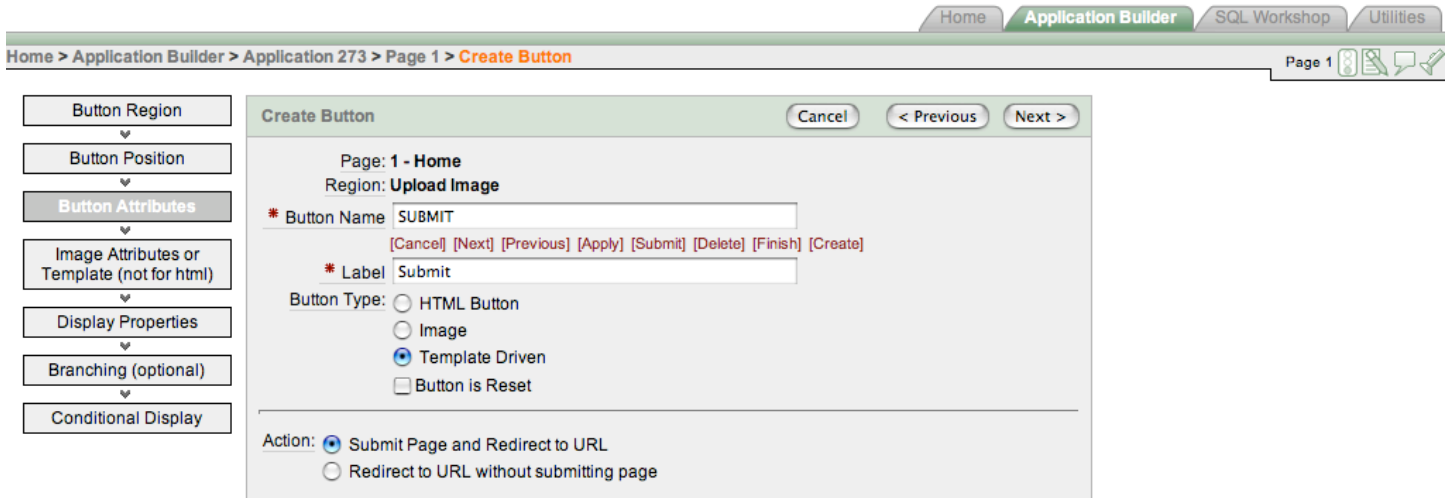


Figure 10 – Add a submit button to the Home page.

Now if we run the application, we should see something like Figure 11, note that I’m using Safari as my browser so the File Browse item may look slightly different than some other browsers.

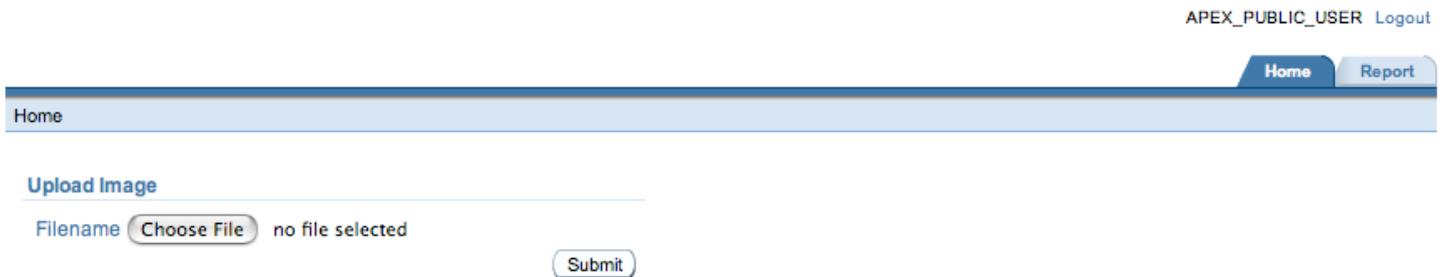


Figure 11 – Home page in Safari.

I have shown what the file browse item looks like in Firefox in Figure 12.

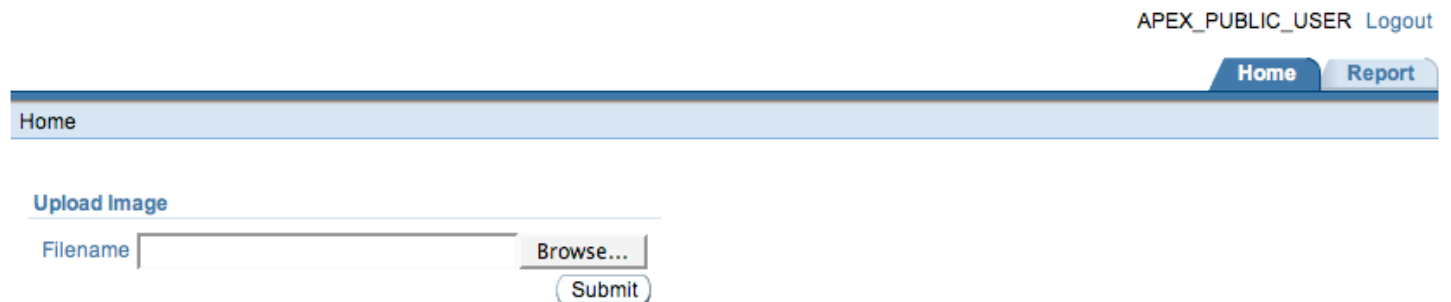


Figure 12 – Home page in Firefox.

CREATING THE REPORT

Now we can create the report that will display the uploaded files. Firstly we will create a new Report Region on the Report page (shown in Figure 13) and make it a SQL Report (shown in Figure 14).

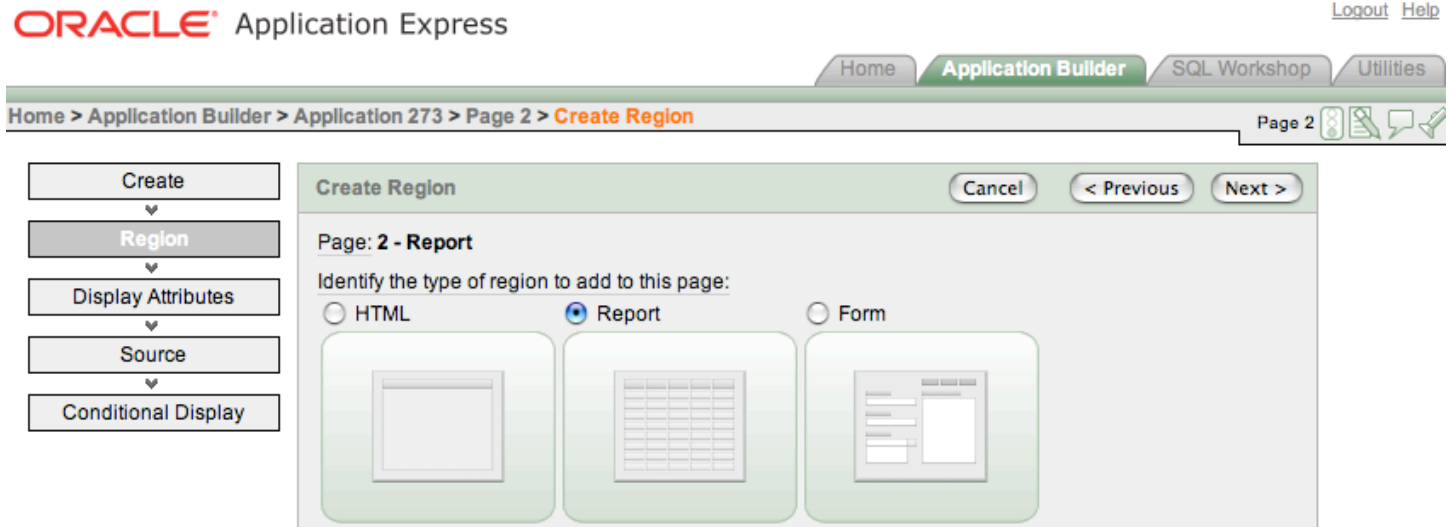


Figure 13 – Creating a new region on the Report page.

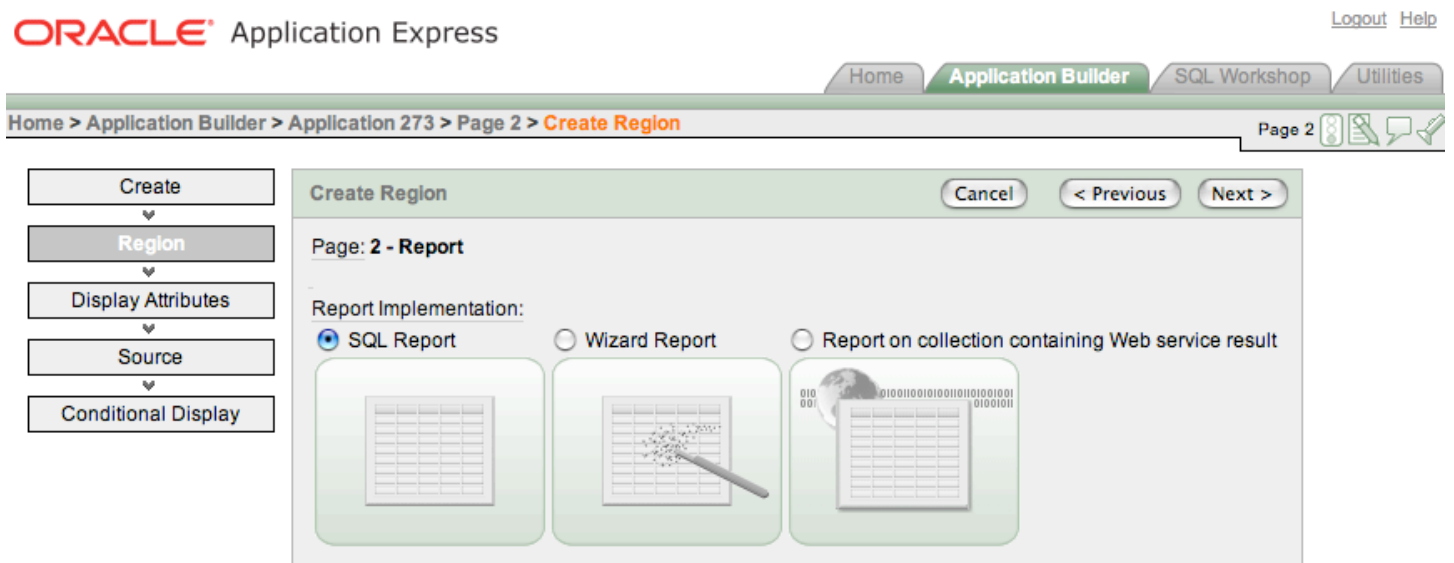


Figure 14 – Creating a SQL Report region.

Now it is important to understand at this stage where the files you have uploaded will go to. By using the File Browse item your files will automatically get uploaded to the `wwv_flow_file_objects$` table, the contents of this table are available to every workspace via the `apex_application_files` view (or `htmldb_application_files` if you're using an older version of Application Express). The definition of the `apex_application_files` view is shown below in Listing 1. Note that throughout this document I might refer to `apex_application_files` as a table to avoid having to reference the underlying table each time, however bear in mind that it is actually a view.

```

jes@DBTEST> desc apex_application_files;
Name                Null?    Type
-----
ID                  NOT NULL NUMBER
FLOW_ID            NOT NULL NUMBER
NAME                NOT NULL VARCHAR2(90)
FILENAME            VARCHAR2(400)
TITLE               VARCHAR2(255)
MIME_TYPE           VARCHAR2(48)
DOC_SIZE            NUMBER
DAD_CHARSET         VARCHAR2(128)
CREATED_BY          VARCHAR2(255)
CREATED_ON          DATE
UPDATED_BY          VARCHAR2(255)
UPDATED_ON          DATE
LAST_UPDATED        DATE
CONTENT_TYPE        VARCHAR2(128)
BLOB_CONTENT        BLOB
LANGUAGE            VARCHAR2(30)
DESCRIPTION         VARCHAR2(4000)
FILE_TYPE           VARCHAR2(255)
FILE_CHARSET        VARCHAR2(128)

```

Listing 1 – Definition of the apex_application_files view.

As you can see, quite a lot of information is recorded about the file, such as the mime_type, file_type and who uploaded the file, although that will be the user specified in your Database Access Description (DAD) file rather than the user logged into your application (the actual application username may be stored in the updated_by column).

So we can use the apex_application_files view in our report query, for now we will use a relatively simple query that is shown in Figure 15.

Home > Application Builder > Application 273 > Page 2 > Create Region

Page 2

Create Region

Cancel < Previous Next > Create Region

Page: 2 - Report
Region Title: **Uploaded Files**

Enter SQL Query or PL/SQL function returning a SQL Query:

```
select
id,
name,
filename
from
apex_application_files
```

Query Builder

Report Template: template: 15. Standard Report

Rows Per Page: 15

Break Columns: - No Break Control -

Columns Headings: Derived from query columns Generic columns

Max. Columns: 60

Create
Region
Display Attributes
Source
Conditional Display

Home Application Builder SQL Workshop Utilities

Figure 15 – Report query using `apex_application_files`.

We can now run the report and see the result, as shown in Figure 16. Note that if you have already uploaded some files in any of your other applications in the workspace then you may already see some records being returned.

Home Report

Report

Uploaded Files

no data found

Figure 16 – Empty report using `apex_application_files`.

Now we can try uploading some files, for this I have resized some images so that they are roughly around 200 pixels wide (I haven't rescaled the images, just resized them and maintained the same aspect ratio). The reason I have done this is so that the images are small enough to be displayed in the report, but are still large enough to be able to demonstrate the benefits that caching will bring.

You could argue that in a real world system you would probably use smaller thumbnails than the images I'm using so the effect would not be as pronounced as I'm going to demonstrate. However it's also worth bearing in mind that you can still benefit from the effects of caching even if you are displaying small thumbnails, or a single logo image at the top of your page, since you will still be able to reduce the impact that repeated queries for the same image will have on your database.

Figure 17 shows the result of running the report after I have uploaded ten different image files using the upload image page.

Report

Uploaded Files

ID	NAME	FILENAME
88674018596790888	F486824417/kiwi.jpg	kiwi.jpg
88675324137792474	F853992870/lemon and lime.jpg	lemon and lime.jpg
88675925868793046	F1336367788/lemons 2.jpg	lemons 2.jpg
88676528292793708	F986349903/lemons.jpg	lemons.jpg
88677130370794308	F83567605/onion.jpg	onion.jpg
88677532102794837	F1933010935/oranges.jpg	oranges.jpg
88678101412795456	F1027575747/pears.jpg	pears.jpg
88678403836796083	F857019029/pineapple.jpg	pineapple.jpg
88679005914796672	F347790212/strawberry.jpg	strawberry.jpg
88679307645797256	F1792609945/tomato.jpg	tomato.jpg

1 - 10

Figure 17 –Report after uploading some images.

You can also notice from Figure 17 what the difference between the name and filename columns are, the filename contains the original filename that we uploaded, whilst the name contains a unique reference for the filename. You would usually use the name column when trying to refer to an uploaded file, since filename may not be unique if two files with the same name are uploaded, or you might prefer to use the id column instead, which will be guaranteed to be unique for each uploaded file.

DISPLAYING THE IMAGES

What we would like to do now is to display the image in the report, rather than just displaying the name and filename, this is actually quite easy to do once you know that Application Express already provides a mechanism to enable you to download a file from the apex_application_images table. We can use the notation #WORKSPACE_IMAGES# to refer to any files uploaded to the table which have not been associated with a particular application which is what happens when we upload files via the standard file browse page item.

So, we can change the report query to display the image by referencing the #WORKSPACE_IMAGES# shortcut, firstly we'll modify the filename column in the report and use an HTML expression to display the image, as shown in Figure 18.

Column Definition

Column Name **FILENAME**
 Column Heading Filename Heading Alignment
 Show Column Sum Sort Column Alignment

Column Formatting

Number / Date Format
 CSS Class CSS Style
 Highlight Words
 HTML Expression [Insert column value]

Figure 18 –Using #WORKSPACE_IMAGES# in the report column.

There are many different ways that we could display the image, using an HTML Expression is just one of those ways, for example we could have generated the img tags as part of the SQL query itself. Next we will change the column headings since using FILENAME for this column is no longer really appropriate, so we'll call it Thumbnail instead, as shown in Figure 19. When the report is run the #WORKSPACE_IMAGES# shortcut will be expanded out to:

```
wwv_flow_file_mgr.get_file?p_security_group_id=WORKSPACE_ID&p_fname=
```

Where WORKSPACE_ID will be replaced by a numeric value representing the workspace id that the application is running in. The p_fname= parameter represents the name of the file that we wish to retrieve.

Region Name: **Uploaded Files**

Column Attributes

Headings Type: Column Names Column Names (InitCap) Custom PL/SQL None

Alias	Link	Edit	Heading	Column Alignment	Heading Alignment	Show	Sum	Sort	Sort Sequence
ID			<input type="text" value="Id"/>	<input type="text" value="left"/>	<input type="text"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="text" value="-"/>
NAME			<input type="text" value="Name"/>	<input type="text" value="left"/>	<input type="text"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="text" value="-"/>
FILENAME			<input type="text" value="Thumbnail"/>	<input type="text" value="left"/>	<input type="text" value="left"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="text" value="-"/>

When moving the last column further down, it will show up as the first column of your report.
 When moving the first column up, it will be moved to the end of your report.

Figure 19 –Changing the column name to Thumbnail.

If we run the report again, we should see that we now see the images displayed correctly, as shown in Figure 20, note that I'm only showing the first three images in the report, although the others are also displayed correctly.

APEX_PUBLIC_USER Logout

Home Report

Report

Uploaded Files




Id	Name	Thumbnail
88674018596790888	F486824417/kiwi.jpg	
88675324137792474	F853992870/lemon and lime.jpg	
88675925868793046	F1336367788/lemons 2.jpg	

Figure 20 –Report displaying the images.

We have now created our prototype application that allows us to display images that we have uploaded. Although it is very simple, this application is quite representative of the way that people commonly design applications that need to display images, whether they are uploaded via a form or whether they are loaded as application or workspace images via the application builder.

BROWSERS AND IMAGE CACHING

If we run the report in a browser and examine the source code for the webpage, we will find that the images are being displayed by the something similar to the following HTML:

```
</img>
```

Note that I have made the value of the `p_security_group_id` parameter shorter so that the line is easier to read. The reason why browsers may not cache the image correctly is due to the fact that many of the common browsers will see that the `src` part of the image tag contains dynamic parameters, namely `p_security_group_id` and `p_fname`. When the browser detects dynamic parameters it will quite often assume that the result returned from requesting the URL could also be dynamic, i.e. it could change from call to call, therefore rather than using a cached version of the file they will request it each time.

If we examine the webserver logs after running the report, we will see some entries similar to those in Listing 2.

```
[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=kiwi.jpg
HTTP/1.1" 200 14635

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=lemon%20an
d%20lime.jpg HTTP/1.1" 200 9273

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=lemons%20.
.jpg HTTP/1.1" 200 11494

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=lemons.jpg
HTTP/1.1" 200 13952

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=onion.jpg
HTTP/1.1" 200 16716

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=oranges.jp
g HTTP/1.1" 200 10649

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=pineapple.
jpg HTTP/1.1" 200 28665

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=strawberry
.jpg HTTP/1.1" 200 12592

[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=tomato.jpg
HTTP/1.1" 200 10731
```

Listing 2 – Webserver logfile entries for the requests.

I apologise for the wrapping in the webserver log output, however it is important to see that there is a separate request for each image, as we'd expect, on our first visit to the page. If we examine an individual entry:

```
[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=kiwi.jpg
HTTP/1.1" 200 14635
```

We can see that the request for the file resulted in a response code of 200, i.e. success, we can also see the size of the file that was requested which in this case was 14635 bytes which is around 14Kb for the kiwi fruit image. If we add up the size of all the images combined it adds up to around 125Kb.

Now, what happens if we navigate back to the home page and then back to the report page? Well by looking at the log files again, we can isolate the request for the kiwi fruit image and we find that there are now two entries for that file:

```
[18/Feb/2007:14:09:20 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=kiwi.jpg
HTTP/1.1" 200 14635
[18/Feb/2007:14:11:12 +0000] "GET
/pls/apex/wwv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=kiwi.jpg
HTTP/1.1" 200 14635
```

So we can see that each time that the image needs to be displayed, it is requested again and each time we request it we not only give the web server and database extra work to do, we also have to transfer the same 14Kb file each time, even though it hasn't changed at all.

HTTP HEADERS

I'm not going to go into a huge amount of detail about how web servers work, however it is important to know the part that using HTTP headers can play in making your applications respond better from your users perspective.

In order to be able to examine the HTTP headers easily, I'm going to use a tool that is part of the `libwww-perl` (LWP) collection, you can easily install this collection of tools if you have Perl installed. The tools allow us to easily construct web requests and examine the responses using Perl scripting and command line tools. The installation of LWP is outside of the scope of this document, however I'd encourage you to examine it and more details can be found in the references section.

One of the tools available in LWP is the `GET` command, which allows us to construct a URL request easily from the command line, as shown in Listing 3.

```
[jes@pb tmp]$ GET
Usage: GET [-options] <url>...
  -m <method> use method for the request (default is 'GET')
  -f          make request even if GET believes method is illegal
  -b <base>   Use the specified URL as base
  -t <timeout> Set timeout value
  -i <time>   Set the If-Modified-Since header on the request
  -c <conttype> use this content-type for POST, PUT, CHECKIN
  -a          Use text mode for content I/O
  -p <proxyurl> use this as a proxy
  -P          don't load proxy settings from environment
  -H <header> send this HTTP header (you can specify several)
  -u          Display method and URL before any response
  -U          Display request headers (implies -u)
  -s          Display response status code
  -S          Display response status chain
  -e          Display response headers
  -d          Do not display content
  -o <format> Process HTML content in various ways
  -v          Show program version
  -h          Print this message
  -x          Extra debugging output
```

Listing 3 –GET command usage.

For example we could perform a command line request to the Google website (note I've only shown the first 10 lines of output by piping the output of the GET command through the head command), as shown in Listing 4.

```
[jes@pb tmp]$ GET www.google.com | head -n10
<html><head><meta http-equiv="content-type" content="text/html; charset=ISO-8859-
1"><title>Google</title><style><!--
body,td,a,p,.h{font-family:arial,sans-serif}
.h{font-size:20px}
.h{color:#3366cc}
.q{color:#00c}
--></style>
<script>
<!--
function sf(){document.f.q.focus();}
// -->
```

Listing 4 –Request to the Google homepage.

So the output we get from the GET command is the actual content response from the remote web server. Clearly in our case it wouldn't make sense to actually display the graphic image from a command line (you would just see garbled parameters), fortunately we can use the `-d` parameter so that output is not displayed.

To make the following examples easier to read, I'm going to setup an environment variable that represents the URL of one of our uploaded files so that I can reference it in the GET command without having to type out the URL in full each time, as shown in Listing 5.

```
[jes@pb tmp]$ export
KIWI_URL="http://ws1/pls/apex/wv_flow_file_mgr.get_file?p_security_group_id=361191749875596
1&p_fname=kiwi.jpg"
[jes@pb tmp]$ echo $KIWI_URL
http://ws1/pls/apex/wv_flow_file_mgr.get_file?p_security_group_id=3611917498755961&p_fname=
kiwi.jpg
```

Listing 5 –Defining the URL environment variable.

Now we can request the image using the same URL our application will use, as shown in Listing 6, this time we will use the `-e` parameter so that we can see the response headers that are returned by the webserver.

```
[jes@pb tmp]$ GET -d -e $KIWI_URL
Connection: close
Date: Sun, 18 Feb 2007 14:13:17 GMT
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Length: 14635
Content-Type: image/jpeg
Client-Response-Num: 1
Content-Disposition: inline
```

Listing 6 –Requesting the image.

Depending on your own infrastructure you might see slightly different headers returned. However we can see from Listing 6 that we get headers that give us some details about the content, such as the size of the content (14635 bytes), the content type and various other pieces of information. Now let's compare those headers with the headers returned when we request the logo image from Application Builder, as shown in Listing 7.

```
[jes@pb tmp]$ GET -d -e "http://ws1/i/html/db/apex_logo.gif"
Cache-Control: max-age=1296000
Connection: close
Date: Sun, 18 Feb 2007 14:13:41 GMT
Accept-Ranges: bytes
ETag: "1b0e73-cbf-44f7bb16"
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Length: 3263
Content-Type: image/gif
Expires: Fri, 16 Mar 2007 08:10:10 GMT
Last-Modified: Fri, 01 Sep 2006 04:46:14 GMT
Client-Response-Num: 1
```

Listing 7 – Requesting the APEX logo image.

As before, we get information about the content size and type, however this time we get an additional header called **Expires**. The **Expires** header tells the browser how long it is able to consider the image is valid, in other words the next time a request is made for the same file (i.e. the same URL is requested) then the browser will check whether the image in the cache can be used by examining the date in the expiry header or whether it needs to make a request to the remote web server instead. In this example the expiry date is set in the future (as I write this), therefore the browser is able to use the version it caches until the expiry date is reached.

So, is there a way we can make the standard Oracle download procedure add the Expiry header to our files? Well, unfortunately, no not easily. However there is another option, if we write our own download procedure then we can make it do pretty much whatever we want to do, fortunately as we will see this is not quite as difficult as it might sound at first.

CUSTOM UPLOAD & DOWNLOAD PROCEDURES

To add the expiry headers we only really need to write our own download procedure that uses the existing data in the `apex_application_files` table. However, just to show how easy it is I'm also going to show how you can adapt the upload procedure to store the uploaded file in your own custom table. Why would you want to do this? Well, you might want to do this for any number of reasons, for example you wish to store your own custom attributes against the uploaded file (such as the user who uploaded it), or you might want to store them in a custom table for backup/restore purposes, or perhaps because you want to create an Oracle Text index on your data.

CREATING THE CUSTOM UPLOAD PROCEDURE.

Firstly we need to create our own table to store the uploaded files in, for the purposes of this example, we'll use quite a simple table, however you are limited only by your own creativity since you could store almost any information you like in this table. Our simple table is shown in Listing 8.

```
jes@DBTEST> create table uploaded_images as
2 select
3   id, name, filename, mime_type, blob_content
4 from
5   apex_application_files;
```

Table created.

Listing 8 – Creating the custom upload table.

Here we have used a CTAS (Create Table As Select) statement to create our custom table based on the existing apex_application_files definition. Since I ran the CTAS statement from SQLPlus, it will not have copied any data over since to view the data in apex_application_files our session would have needed to have been associated with a particular workspace, for example if we had executed the command from SQL Workshop rather than SQLPlus.

Unfortunately there is no way to directly modify the File Browse page item to use our table by default, so we need to create a page process on our page which, after an upload, copies the new record from the apex_application_files table into our uploaded_images table and then removes the record from the apex_application_files table (this is of course optional, you could leave the record there too if you wished to).

We will create a PL/SQL page process that will be executed after the page has been submitted, as shown in Figure 21.

The screenshot shows the Oracle Application Express interface for creating a page process. The main window is titled "Create Page Process" and is for "Page: 1 - Home". The configuration is as follows:

- Name:** Insert Record
- Sequence:** 10
- Point:** On Submit - After Computations and Validations
- Type:** PL/SQL anonymous block

The left sidebar contains a navigation menu with the following items:

- Process Type
- Process Attributes
- Process
- Messages
- Process Conditions

The top navigation bar includes "Home", "Application Builder", "SQL Workshop", and "Utilities". The breadcrumb trail is "Home > Application Builder > Application 273 > Page 1 > Create Page Process".

Figure 21 –Creating the upload page process.

The source of the PL/SQL page process is pretty straightforward, first we check if we have tried to upload a file (by making sure that P1_FILENAME is not null), then we use the value of the P1_FILENAME page item to retrieve the record from the apex_application_files table and to insert it into the uploaded_images table, then we delete the record from the apex_application_files table. I have set the page process to only execute when the Submit button is pressed and also defined a success message of “Image Uploaded” and a failure message of “Image could not be Uploaded”.

Home > Application Builder > Application 273 > Page 1 > Create Page Process

Page 1

Process Type
Process Attributes
Process
Messages
Process Conditions

Create Page Process

Cancel < Previous Next > Create Process

Page: 1 - Home
Point: On Submit - After Computations and Validations

* Enter PL/SQL Page Process

```
if (:P1_FILENAME is not null) then
  insert into uploaded_images(id, name, filename, mime_type, blob_content)
  select id, name, filename, mime_type, blob_content
  from apex_application_files
  where name = :P1_FILENAME;

  delete from apex_application_files where name = :P1_FILENAME;
end if;
```

Do not validate PL/SQL code (parse PL/SQL code at runtime only).

> Page Items

Figure 22 – Storing the record in the custom table after upload.

Now we can test the upload procedure by trying to upload an image (Figure 23) and then querying the `uploaded_images` table afterwards, as shown in Listing 9.

APEX_PUBLIC_USER Logout

Home Report

Home

Record Uploaded

Upload Image

Filename Choose File no file selected

Submit

Figure 23 – Uploading to the `uploaded_images` table.

```
jes@DBTEST> select id, name, filename, mime_type
2 from uploaded_images;

ID NAME          FILENAME          MIME_TYPE
-----
9.6274E+16 F278231389/kiwi.jpg kiwi.jpg image/jpeg
```

Listing 9 – Record is stored in the `uploaded_images` table.

So we can now see that the upload procedure works just fine. The next step is to be able to retrieve our images by creating a corresponding download procedure.

CUSTOM DOWNLOAD PROCEDURE

To download our images we will write a procedure and grant execute access on the procedure to user specified in our database access descriptor (DAD). Please note that there are actually different ways we could write the procedure, for example as an application process running within our application, however for the purposes of this example I will use a standalone procedure. Also please note that there are many different ways in which you can protect your procedure with security and authorization checks, however that is outside the scope of this document. Listing 10 shows our custom download procedure.

```

1 create or replace
2 procedure download_image(p_id IN NUMBER) AS
3   v_mime_type VARCHAR2(48);
4   v_length NUMBER;
5   v_name VARCHAR2(2000);
6   v_image BLOB;
7 BEGIN
8   SELECT name,
9     mime_type,
10    dbms_lob.getlength(blob_content),
11    blob_content
12 INTO v_name,
13    v_mime_type,
14    v_length,
15    v_image
16 FROM uploaded_images
17 WHERE id = p_id;
18 -- set up the HTTP header
19 owa_util.mime_header(
20   nvl(v_mime_type, 'application/octet'),
21   FALSE);
22 -- set the size so the browser knows how much to download
23 http.p('Content-length: ' || v_length);
24 -- the filename will be used by the browser if the users does a save as
25 http.p('Content-Disposition: attachment; filename="' ||
26   SUBSTR(v_name, instr(v_name, '/') + 1) || '"');
27 -- close the headers
28 owa_util.http_header_close;
29 -- download the BLOB
30 wpg_docload.download_file(v_image);
31 END download_image;

```

Procedure created.

Listing 10 – Custom download procedure

The procedure might look complicated at first, however all it does is accept the numeric id of the record you wish to retrieve, it then selects the relevant record (using the id) from the `uploaded_images` table. The next section of the procedure uses the `http.p` procedure to create the HTTP response header and writes out some text, finally it closes the HTTP header and outputs the image using the standard `wpg_docload.down_file` procedure.

The next step is to grant execute on our procedure to the user specified in our DAD, this is so the procedure can be called directly via a URL:

```
jes@DBTEST> grant execute on download_image to apex_public_user;
```

We should now test that the procedure works by testing the URL from our browser, the format of the URL we need to use is:

`http://servername/DAD/schema_name.procedure_name`

So in our example, our URL is –

`http://ws1/pls/apex/jes.download_image?p_id=96273915222065433`

The `p_id` parameter was obtained by querying the `uploaded_images` table to get the value of the record we uploaded earlier. You should find if you enter that URL (substituting values appropriate for your own configuration and data) that the image we uploaded is downloaded to your browser. We have now successfully created our own download procedure.

TYING IT ALL TOGETHER

We now need to modify our report so that it references our own custom download procedure rather than using the `#WORKSPACE_IMAGES#` one. Listing 11 shows the report query modified to reference our custom table.

```
select
  id,
  name,
  filename
from
  uploaded_images
```

Listing 11 – New report query.

The next step is to modify the HTML Expression we are using in the column used to display the image, as shown in Figure 24.

The screenshot shows the 'Column Formatting' dialog box with the following fields:

- Number / Date Format:
- CSS Class:
- CSS Style:
- Highlight Words:
- HTML Expression [Insert column value]:

Figure 24 – Modified HTML Expression.

This new HTML Expression replaces the `src` attribute for the image tag, to now reference our custom download procedure:

```
</img>
```

At run time the `#ID#` will be substituted for the value of the `id` for the current record in the report.

If we now run the report page again, as shown in Figure 25, we should now just see one record returned (remember we only uploaded on image to our custom table).

Report

Uploaded Files

Id	Name	Thumbnail
96273915222065433	F278231389/kiwi.jpg	
1 - 1		

Figure 25 – Report on our custom upload table.

Great, so now we're now able to upload files to our own custom table and then retrieve them via our own custom download procedure. If we look again at our webserver logfiles we should see the request to our custom download procedure:

```
[18/Feb/2007:14:11:12 +0000] "GET /pls/apex/jes.download_image?p_id=96273915222065433 HTTP/1.1" 200 14635
```

However, as before, if we navigate away from that page and then back to it, we will see a new request for the image each time. In other words we still haven't managed to get the browser to cache the image. However referring back to Listing 7, we add another HTTP header, named `Expires`, with a string that represents a date in the future then we should be able to get the browser to use the image from the cache. Listing 12 shows the modified `download_image` procedure.

```

1 create or replace
2 procedure download_image(p_id IN NUMBER) AS
3   v_mime_type VARCHAR2(48);
4   v_length NUMBER;
5   v_name VARCHAR2(2000);
6   v_image BLOB;
7 BEGIN
8   SELECT name,
9     mime_type,
10    dbms_lob.getlength(blob_content),
11    blob_content
12 INTO v_name,
13    v_mime_type,
14    v_length,
15    v_image
16 FROM uploaded_images
17 WHERE id = p_id;
18 -- set up the HTTP header
19 owa_util.mime_header(
20   nvl(v_mime_type, 'application/octet'),
21   FALSE);
22 -- set the size so the browser knows how much to download
23 http.p('Content-length: ' || v_length);
24 http.p('Expires: ' || to_char(sysdate + 1/24, 'FMDy, DD Month YYYY HH24:MI:SS') ||
25   ' GMT');

26 -- the following line is used incase the user performs a save as
27 http.p('Content-Disposition: attachment; filename="' ||
28   SUBSTR(v_name, instr(v_name, '/') + 1) || '"');
29 -- close the headers
30 owa_util.http_header_close;
31 -- download the BLOB
32 wpg_docload.download_file(v_image);
33 END download_image;

```

Procedure created.

Listing 12 – Custom download procedure with Expires header.

The only modification to the procedure is the inclusion of the line:

```
http.p('Expires: ' || to_char(sysdate + 1/24, 'FMDy, DD Month YYYY HH24:MI:SS') || ' GMT');
```

This should create an HTTP Expires header with a date one hour in the future (based on the server time). Note that I have hardcoded the GMT in this example, however you could retrieve the timezone information from your own server if you preferred. We can test this by using checking the headers as we did earlier using the GET tool, as shown in Listing 13.

```
[jes@pb tmp]$ GET -d -e "http://ws1/pls/apex/jes.download_image?p_id=96273915222065433"
Connection: close
Date: Sun, 18 Feb 2007 16:32:08 GMT
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Length: 14635
Content-Type: image/jpeg
Expires: Sun, 18 Feb 2007 17:32:08 GMT
Client-Response-Num: 1
Content-Disposition: attachment; filename="kiwi.jpg"
```

Listing 13 – Expiry header added to our custom download procedure

Now we have added the Expiry header to our custom download procedure, the question is will the browser use the header? Once again, this is easy to test, first I'll count the number of times the URL has been requested in the logfile so far, using some Unix commands (which you don't need to worry about).

```
[jes@pb tmp] grep "download_image" cache_log | wc -l
5
```

So there are currently five entries in the logfile where the image has been requested, we now use the browser and completely refresh the report page so that it picks up the image with the Expiry header, we now check the logfile to check that the additional request has been logged:

```
[jes@pb tmp] grep "download_image" cache_log | wc -l
6
```

As we'd expect, the number of requests has increased by one. However if we now navigate away from the report page to the home page and back again, a number of times (as many times as you like within the hour in fact!), we see from the log file:

```
[jes@pb tmp] grep "download_image" cache_log | wc -l
6
```

Success! The browser is no longer requesting the image each time the page loads, it is using the copy of the image that is in the cache. The image will not be requested again within the same session until either the Expiry time has passed, we request a full refresh or the cache is emptied.

BENCHMARKING

Ok, so far we have managed to write our own custom upload and download procedure and the only benefit we have seen is that we have reduced the number of URL request entries that get logged to our logfile. Surely there is more to it than that? Well, recall at the beginning I said that if you take advantage of caching then you can reduce the amount of work that your webserver and database are having to perform, also your end users will benefit from not having to download the image each time, therefore the pages should load faster for your users.

However we really need to quantify the benefits that caching can bring and to do that we need to benchmark the effect of using caching and compare it against not using caching.

There are many different tools that will help you to benchmark the performance of your webserver, and remember when we benchmark the performance of the webserver we are also indirectly benchmarking the performance of our application since we will be measuring how quickly we can view the pages of our applications.

The tool I have chosen to use is called `HTTPPERF` and is quite a complex tool in itself, so I'm not going to try and explain it in too much detail here, however I do encourage you to investigate using `HTTPPERF` or some other benchmarking tool such as `ApacheBench` on your own system. Note that the installation and configuration of `HTTPPERF` is outside the scope of this document.

Note that `HTTPPERF` does not attempt to parse webpages at all, therefore if we simply requested the report page it would not also know to request the resources (i.e. the linked images) that are contained within that page, however as I mentioned `HTTPPERF` is a complex tool in its own right and fortunately we can script the URLs that it needs to retrieve. So, we will compare two cases –

- The images are not cached, therefore the page along with all the images need to be fetched every time.
- The images are cached, therefore after the first request only the page needs to be re-requested.

Also note that the URL we use for the report page needs to have all the correct parameters, such as the session id, otherwise the response we would get back would simply be a redirect to a page with a new session established.

So, the two example tests we will perform will both use different `HTTPPERF` script files, Listing 14 shows the non-cached images version where we will request the main report page, followed by each of the images, Listing 15 shows the cached version where we simply request the report page and use the images cached in the browser cache.

```
[jes@pb bench]$ cat nocache.log
/pls/apex/f?p=273:2:2807160253709943::NO
/pls/apex/jes.download_image?p_id=97210515069355199
/pls/apex/jes.download_image?p_id=97210717147355777
/pls/apex/jes.download_image?p_id=97210920264356703
/pls/apex/jes.download_image?p_id=97211526497358511
/pls/apex/jes.download_image?p_id=97211728575359107
/pls/apex/jes.download_image?p_id=97212201348360634
/pls/apex/jes.download_image?p_id=97212608274362719
/pls/apex/jes.download_image?p_id=97212810352363236
/pls/apex/jes.download_image?p_id=96273915222065433
```

Listing 14 – HTTPPerf script for non-cached images.

```
[jes@pb bench]$ cat cache.log
/pls/apex/f?p=273:2:2807160253709943::NO
```

Listing 15 – HTTPPerf script for cached images.

If we now run HTTPPerf to similar a single request to the non-cached page, we get the results shown in Listing 16.

```
[jes@pb bench]$ httpperf --server ws1 --wssesslog 1,1,nocache.log
Total: connections 10 requests 55 replies 10 test-duration 1.212 s

Connection rate: 8.2 conn/s (121.2 ms/conn, <=2 concurrent connections)
Connection time [ms]: min 102.5 avg 121.2 max 161.7 median 114.5 stddev 19.0
Connection time [ms]: connect 29.0
Connection length [replies/conn]: 1.000

Request rate: 45.4 req/s (22.0 ms/req)
Request size [B]: 519.0

Reply rate [replies/s]: min 0.0 avg 0.0 max 0.0 stddev 0.0 (0 samples)
Reply time [ms]: response 47.8 transfer 44.3
Reply size [B]: header 282.0 content 137248.0 footer 0.0 (total 137530.0)
Reply status: 1xx=0 2xx=10 3xx=0 4xx=0 5xx=0

Session rate [sess/s]: min 0.00 avg 0.82 max 0.00 stddev 0.00 (1/1)
Session: avg 10.00 connections/session
Session lifetime [s]: 1.2
Session failtime [s]: 0.0
Session length histogram: 0 0 0 0 0 0 0 0 0 1
```

Listing 16– HTTPPerf results for a single request to the non-cached page.

As you can see there is a huge amount of detail and statistics returned, I have even removed some of the statistics to make it more readable. However some of the statistics are incredibly useful, such as the time it took to perform the test (1.212 second) and the size of the content returned (137248 bytes), also we can see that we statistics about the average session rate (i.e. how many sessions per second we achieved).

To avoid posting copious amount of output, I'm going to summarize the benchmark runs so that we can easily compare them. Please bear in mind, this is not really a benchmark of how fast the web server is, it is a comparison of the relative difference that caching makes.

TEST 1 - SINGLE CONNECTION, NO MULTI-THREADING

	Without Caching	With Caching	Factor
Connection Rate (conn/s)	8.2	7.9	~ 1.04x faster
Connection Rate (ms/conn)	121.2	127.1	~ 1.04x faster
Session lifetime (Seconds)	1.212	0.127	~ 10.5x faster
Total Content Size Returned (Kb)	137	8.3	~ 16x smaller
Average Session Rate (sessions/sec)	0.82	7.87	~ 9.6x faster

In the first test, we compare a single connection with no multi-threading, i.e. in the without caching example the page was requested first then each image is requested in term in a sequential manner. Many modern browsers today allow sub-requests to be handled in parallel, although this is sometimes throttled to only 1 or 2 simultaneous sub-requests at a time.

We can see that since this is a single connection, the difference in connection rate is not that substantial, in other words since both tests made a single connection they each took around the same time to connect. However when we look at the session lifetime there is a huge difference, the non-cached version took around 1.2 seconds to download the page and images, whilst the cached version returned in around 0.12 seconds since it did not need to request all the images again. This means the cached version can run in around a tenth of the time it takes the non-cached version, in other words it will take a tenth of the time for the page to be delivered to the users browser.

Similarly when we look at the content size returned, unsurprisingly the non-cached version is substantially bigger, having transferred around 137Kb rather than the 8.3Kb of the cached version, a saving in bandwidth of around 94%. When we look at the average session rate, in other words how many sessions per second the webserver was able to handle, there is again a massive difference. Since the cached version is not re-requesting the images each time, the webserver is able to deal with the request much faster, thereby enabling the webserver to deal with more requests per second overall.

TEST 2 - 10 CONNECTIONS, 10 PARALLEL SUBREQUESTS.

This time we will try a more realistic case, where the browser is able to make multiple simultaneous sub-requests for the images. This test essentially simulates 10 different users requesting the report page at more or less the same time.

	Without Caching	With Caching	Factor
Connection Rate (conn/s)	16.7	17.4	~ 1.04x faster
Connection Rate (ms/conn)	59.9	57.5	~ 1.04x faster
Session lifetime (Seconds)	5.989	0.575	~ 10x faster
Total Content Size Returned (Kb)	1370	83	~ 16x smaller
Average Session Rate (sessions/sec)	1.67	17.40	~ 10x faster

As with the previous test, we can see big gains by using caching, whilst the non-caching version does improve by using parallel subrequests, the cached version is still an order of magnitude faster when it comes to how long it takes to service a page request.

TEST 3 - 50 CONNECTIONS, 10 PARALLEL SUBREQUESTS.

This time we will simulate 50 connections, each request is capable of being handled by 10 parallel subrequests.

	Without Caching	With Caching	Factor
Connection Rate (conn/s)	19.4	21.2	~ 1.09x faster
Connection Rate (ms/conn)	51.5	47.3	~ 1.09x faster
Session lifetime (Seconds)	17.1	0.14	~ 130x faster
Content Size Returned (Mb)	6.7	0.4	~ 16x smaller
Average Session Rate (sessions/sec)	2.12	19.43	~ 9x faster

The major thing to note here was the dramatic increase in the session lifetime factor, basically what has happened is because of the increased number of connections, the non-cached version took much longer for each session because the webserver was having to deal with so many other simultaneous requests.

TEST 4 - 200 CONNECTIONS, 10 PARALLEL SUBREQUESTS.

Just to illustrate the point even more, we'll now test 200 connections, which will approximate the effect of 200 users all accessing the report page simultaneously.

	Without Caching	With Caching	Factor
Connection Rate (conn/s)	3.0	19.8	~ 6.6x faster
Connection Rate (ms/conn)	336.9	50.6	~ 6.6x faster
Session lifetime (Seconds)	56.6	0.2	~ 130x faster
Total Content Size Returned (Mb)	26.75	1.6	~ 16x smaller
Average Session Rate (sessions/sec)	0.27	19.78	~ 73x faster

Here we are really starting to push the web server, with the non-cached version the webserver needs to process around 2000 requests (i.e. 200 connections each requesting the webpage and images), whilst the cached version is only causing 200 requests to be issued.

We see a dramatic drop in the connection rate of the non-cached version as the webserver is busy dealing with all the requests, there is a huge increase in the average session lifetime (taking almost a minute), the session rate drops right down to the point where the webserver is servicing 0.27 sessions per second.

Comparing this with the cached version, we can really start to see the advantage of caching the images, even with 200 clients requesting the page, the webserver is able to deal with each session within 0.2 seconds, and even manages to slightly increase the average session rate, i.e. it is handling more requests per second than before.

One other thing that is worth noting here, simulating 200 people requesting the non-cached version resulted in over 26Mb of content being generated by the webserver, that is 26Mb of bandwidth that needs to be transferred over the network (in itself 26Mb might not sound much, however this is a single webpage, so scale it up to a daily or monthly quota). The cached version however has resulted in only 1.6Mb of network traffic, a substantial saving. Even if you're not in the situation where you have to pay for bandwidth, being able to reduce the amount of unnecessary traffic sent over your network will allow other

applications to take advantage of it and will also mean you are not forced into upgrading your infrastructure until it is strictly necessary.

IMAGE CACHING WRAP-UP

I hope I have managed to demonstrate in this section the substantial benefits that using image caching can add to the performance of your application. If your APEX application uses images that you have uploaded through the Application Builder interface, or you are using the default file download procedures then you may not be taking advantage of image caching and you will be penalized every time that your users request a page that contains one of these images.

There are four main areas that you can benefit from when you use cached images:

- Reduced load on the webserver
- Reduced load on the database
- Users experience faster page generation
- Substantial savings in network bandwidth

Any one of those reasons should be enough to convince you to investigate using cached images. The reduced load and network savings will also help substantially with the scalability of your application, whilst of course your users will also appreciate the fact that the page takes less time to load.

PAGE COMPRESSION

In this section we are going to look at how you can take advantage of page compression to reduce the amount of content that needs to be sent from the webserver to the browser.

You may be wondering why it's necessary to compress the content, after all the speed of internet connections is getting faster and faster, we no longer need to use slow dial-up modems and can get broadband speeds at home that would never have been affordable by a home user years ago.

However, whilst connection speeds have increased, so has the complexity of some of the websites we access today. They can quite often contain references to images, flash movies, CSS files, Javascript files etc, so before we can view the webpage we need to wait for our browser to request the original page, along with any linked files.

In the last section we looked at how we can use image caching to avoid the need to repeatedly transfer the same image, so now we can begin to look at how we can reduce the size of files such as the HTML output, the CSS files and any Javascript files so that downloading them will take less time and then we can view the webpage quicker.

APACHE MODULES

This paper is not intended to be a tutorial on how Apache modules work, there is enough information about that already on the internet. It is sufficient for you to know that Apache modules are essentially plug-ins that you can use to modify the way that Apache works. There are many different Apache modules available, for all sorts of different purposes. We will just discuss a couple of the modules here:

- `mod_gzip` – this apache module works with Apache 1
- `mod_deflate` – this module works with Apache 2

The choice of whether to use `mod_gzip` or `mod_deflate` comes down to your infrastructure, the Oracle supplied HTTP server (and IAS) is based on Apache 1, therefore to enable compression on that webserver you would need to use `mod_gzip`. If however you use an Apache 2 server to proxy requests to your OHS/IAS then you may wish to use `mod_deflate` on that Apache 2 server to enable compression to be used.

HOW DOES IT WORK?

Whenever a browser makes a request to a webserver, the browser will send a number of HTTP request headers that tell the webserver what information the browser is requesting, along with information and the capabilities of the browser.

As an example, if we request the Google homepage using the (hopefully familiar by now) GET tool, we see the output in Listing

```
[jes@pb tmp]$ GET -d -U -e www.google.com
GET http://www.google.co.uk/
User-Agent: lwp-request/2.07

Cache-Control: private
Date: Sun, 18 Feb 2007 21:12:03 GMT
Server: GWS/2.1
Content-Type: text/html
Content-Type: text/html; charset=ISO-8859-1
Client-Response-Num: 1
Client-Transfer-Encoding: chunked
Set-Cookie: PREF=ID=506820471aef2070:TM=1172783523:LM=1172783523:S=iVgPJnGJwayh0Ngz;
expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.co.uk
Title: Google
```

Listing 17– Request header sent in a URL request.

Here I have used the `-U` parameter to display the request headers (highlighted in bold).

The command line GET tool sends very few headers by default, it is nowhere near the amount of information that the average browser will send, however it is sufficient information for the webserver to process the request. The `User-Agent` header can be used by the webserver to identify what sort of web browser is making the request.

If you have got one of the compression modules loaded in your webserver, then when the webserver receives a request it will examine the request headers to see whether the browser supports handling compressed content. If the browser does support compressed content, then the webserver can use the compression module to compress the content and then send the response to the browser. If the browser does not advertise that it supports compressed content, then the response will always be delivered in an uncompressed format.

So, what are the headers that tell the webserver whether the browser supports compressed content? Well, there's only one header and it's called `Accept-Encoding`. This header can support a few different values though, for example:

```
Accept-Encoding: gzip
Accept-Encoding: compress
Accept-Encoding: gzip, compress
```

Where more than one value is listed it means that the browser supports more than one compression format. Most browsers these days will actually announce multiple values, so as long as you use a compression module that is in the list then they should be able to work together.

Therefore if we can enable our webserver to support compression, then any browsers that also support compression should benefit, whilst those browsers that do not support compression will also continue to work.

CONFIGURING MOD_GZIP

One important thing I need to point out here, `mod_gzip` is not officially supported by Oracle, so if you are the least bit wary of changing the configuration on your OHS/IAS, or you are worried that you may be left in an unsupported position then perhaps you should consider using an Apache 2 to proxy requests to the OHS/IAS and load the `mod_deflate` module on the Apache 2 server instead.

Having said that I (and others) have successfully run `mod_gzip` for a long time now without any ill effects, however you are well advised to try this on a test system before trying it on your production setup. In other words, I won't take responsibility if you break your production system!

The installation of `mod_gzip` is fairly straightforward and well documented, you simply need to place the module into the directory containing all your other Apache modules (for example, usually in the `libexec` directory) and it's also recommended that you use the separate configuration file (`mod_gzip.conf`) for all the `mod_gzip` related configuration and include this new configuration file from your main Apache configuration file (`httpd.conf`), rather than placing the `mod_gzip` configuration directly in the main file.

```
[jes@pb OraHome]$ ls -al Apache/Apache/libexec/mod_gzip.so
-rwxr-xr-x 1 oracle oinstall 90998 Dec 9 2004 Apache/Apache/libexec/mod_gzip.so*
```

So here we can see that the module is located in the `ORACLE_HOME/Apache/Apache/libexec/mod_gzip.so` directory. Remember that `ORACLE_HOME` refers to where the Apache server is installed rather than the database.

I have also copied the sample `mod_gzip.conf` to the Apache configuration file directory.

```
[jes@pb OraHome]$ ls -al Apache/Apache/conf/mod_gzip.conf
-rw-r--r-- 1 oracle oinstall 14837 Jan 6 2006 Apache/Apache/conf/mod_gzip.conf
```

Whilst the example `mod_gzip.conf` should work fine in most cases, I have made a few changes, one of which is adding the following line:

```
mod_gzip_item_include    handler    ^pls_handler$
```

The purpose of this line is to include compression on anything that is being handled by the `pls_handler`, in other words the `mod_plsql` handler which is responsible for handling requests for our database access descriptor (DAD) which is how our APEX sessions are handled. I have done this because I've found in certain cases where the `mime_type` is not detected properly then certain items will not be compressed, even though they may be highly compressible items such as CSS or JavaScript files. You may want to check whether this line is suitable for your own configuration or not (I can't guide you there, it's something you need to determine yourself through testing).

Next we need to include the `mod_gzip` configuration by adding the following line to the main Apache configuration file (`httpd.conf`):

```
# Include the mod_gzip settings
include "/u1/app/oracle/OraHome/Apache/Apache/conf/mod_gzip.conf"
```

Obviously make sure you use the correct path to the `mod_gzip.conf` file for your own installation.

We can now restart Apache. Note that if you get a warning along the lines of "this module might crash under EAPI!" you don't need to worry, the module does seem to work fine even though this warning is issued. If you want to get rid of the error you can try recompiling the module yourself.

We should now have a working installation of `mod_gzip`, however before we test it I'll also cover configuring `mod_deflate`.

CONFIGURING MOD_DEFLATE

As I mentioned earlier you can use `mod_deflate` on an Apache 2 server if you don't want to modify your existing OHS/IAS installation. You can then proxy requests from the Apache 2 server to the existing OHS/IAS server. I won't cover how to proxy here (that's a separate article in its own right).

I'm not going to cover how to compile or install `mod_deflate`, since compiling your Apache installation is very site-specific depending on which custom modules and configuration you need. However if you have downloaded the binary distribution of Apache then you should already have the precompiled module, otherwise you can quite easily compile the module yourself, you can compile it either into the main binary or as a separate loadable module. The instructions for compiling are included in the Apache distribution.

In my example I have compiled `mod_deflate` so that it is part of the Apache binary, as shown in Listing 18, where I use the `-l` (lowercase letter L) parameter to list the modules that are compiled into Apache.

```
[jes@ap Apache] bin/httpd -l
Compiled in modules:
  core.c
  mod_access.c
  mod_auth.c
  util_ldap.c
  mod_auth_ldap.c
  mod_include.c
  mod_deflate.c
  mod_log_config.c
  mod_env.c
  ... extra output removed
```

Listing 18– Listing the modules compiled into Apache.

This means that I do not need to explicitly load the module, since it is already compiled into Apache, if you have compiled it as a loadable module then you would need to add the following line to your `httpd.conf` file:

```
LoadModule deflate_module libexec/mod_deflate.so
```

The configuration for the `mod_deflate` module can seem a bit simpler than that of `mod_gzip`, we can add the following line to our main Apache configuration file:

```
AddOutputFilterByType DEFLATE text/html text/plain text/xml text/css
```

This line tells Apache that `mod_deflate` should be used for the content using the `text/html`, `text/plain`, `text/xml` or `text/css` mime types.

Once you restart the webserver you should have a working `mod_deflate` module.

TESTING COMPRESSION

For the rest of this article I'm going to discuss `mod_gzip` specifically, however you should find that the results with `mod_deflate` are very similar in terms of the compression ratio and the benefits you receive.

First we need to test that the compression module is working correctly, so we will use the `GET` tool to request the default page of our OHS installation (you can use any static page you like for this test it doesn't really matter).

```
[jes@pb bench]$ GET -d -e http://ws1:7777
Connection: close
Date: Sun, 18 Feb 2007 23:47:07 GMT
Accept-Ranges: bytes
ETag: "46cbac-37a3-4159b828"
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Length: 14243
Content-Type: text/html
Content-Type: text/html; charset=windows-1252
Last-Modified: Tue, 28 Sep 2004 19:14:48 GMT
Client-Response-Num: 1
Link: </ohs_images/portals.css>; rel="stylesheet"
Title: Oracle Application Server - Welcome
```

Listing 19– Retrieving an uncompressed static page.

Here we can see that the size of the returned HTML is 14243 bytes, or around 14Kb. Now by making the `GET` command use the `Accept-Encoding` header we should be able to get the webserver to compress the response content for us, as shown in Listing 20.

```
[jes@pb bench]$ GET -d -e -H "Accept-Encoding: gzip,compress" http://ws1:7777
Connection: close
Date: Sun, 18 Feb 2007 23:50:50 GMT
Accept-Ranges: bytes
ETag: "46cbac-37a3-4159b828"
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Encoding: gzip
Content-Length: 2838
Content-Type: text/html
Last-Modified: Tue, 28 Sep 2004 19:14:48 GMT
Client-Response-Num: 1
```

Listing 20 – Retrieving a compressed static page.

This time the `Content-Length` is only 2838 bytes, which is around a fifth of the size of the uncompressed version.

If you enabled logging in the `mod_gzip.conf` configuration file, then you should be able to see an entry in the logfile similar to the following line:

```
pb - - [18/Feb/2007:23:55:01 +0000] "ws1 GET / HTTP/1.1" 200 3150 mod_gzip: OK In:14243 -< Out:2838 = 81 pct.
```

The log entry tells you the original size of the document (14243 bytes), what it was compressed to (2838 bytes) and the resulting compression ratio (81%), this log can be extremely helpful in showing you the benefit of using compression on particular files.

Ok, so we know we can compress static HTML files, how about some of our APEX pages? Well, if we try the URL of our report page, we should results similar to those shown in Figure 21 and Figure 22.

```
[jes@pb bench]$ GET -d -e "http://ws1/pls/apex/f?p=273:2:2807160253709943::NO"
Connection: close
Date: Sun, 18 Feb 2007 23:58:53 GMT
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Length: 8541
Content-Type: text/html; charset=UTF-8
Content-Type: text/html; charset=utf-8
Client-Response-Num: 1
Link: </i/css/core_V22.css>; /="/"; rel="stylesheet"; type="text/css"
Link: </i/themes/theme_15/theme_V2.css>; /="/"; rel="stylesheet"; type="text/css"
Title: Report
```

Listing 21 – Retrieving the uncompressed report page.

```
[jes@pb bench]$ GET -d -e -H "Accept-Encoding: gzip,compress"
"http://ws1/pls/apex/f?p=273:2:2807160253709943::NO"
Connection: close
Date: Sun, 18 Feb 2007 23:59:22 GMT
Server: Oracle-Application-Server-10g/9.0.4.0.0 Oracle-HTTP-Server
Content-Encoding: gzip
Content-Length: 2136
Content-Type: text/html; charset=UTF-8
Client-Response-Num: 1
```

Listing 22 – Retrieving the compressed report page.

Success! Once again the compression ratio is quite high, as we can see from the logfile:

```
pb - APEX_PUBLIC_USER [18/Feb/2007:00:08:40 +0000] "ws1 GET
/pls/apex/f?p=273:2:2807160253709943::NO HTTP/1.1" 200 2365 mod_gzip: OK In:8541 -< Out:2136 = 75
pct.
```

The report page compresses to around a quarter of the uncompressed size. It is also worth noticing that the document contains other linked items, `core_V22.css` and `theme_V2.css`, which should also be compressible, as shown here:

```
pb - - [18/Feb/2007:00:22:02 +0000] "ws1 GET /i/themes/theme_15/theme_V2.css HTTP/1.1" 200 4626
mod_gzip: OK In:23836 -< Out:4315 = 82 pct.
pb - - [18/Feb/2007:00:22:02 +0000] "ws1 GET /i/css/core_V22.css HTTP/1.1" 200 2037 mod_gzip: OK
In:6509 -< Out:1726 = 74 pct.
```

As you can see we have achieved quite a significant compression ratio on these files too.

BENCHMARKING

Ok, so far we have managed to compress the size of some of the files being downloaded to the users browser, but what does that mean in terms of performance and resources?

Well, as before, we will use the HTTPERF tool to query the report both with the `mod_gzip` module enabled and disabled. I'll also create a custom script that will do a fetch of the main report page, plus the linked CSS files and JavaScript files so that the test is representative of a real user browsing the page.

TEST 1 - SINGLE CONNECTION, NO MULTI-THREADING

	<code>mod_gzip</code> Off	<code>mod_gzip</code> On	Factor
Connection Rate (conn/s)	8.6	12.1	~ 1.4x faster
Connection Rate (ms/conn)	116.8	82.8	~ 1.4x faster
Session lifetime (Seconds)	0.4	0.2	~ 2x faster
Total Content Size Returned (Kb)	38	8	~ 5x smaller
Average Session Rate (sessions/sec)	0.47	4.03	~ 8.5x faster

As we can, using compression improves performance in all areas, the response content is downloaded in roughly half the time it takes without compression. Also, the average session rate when using compression is roughly 8.5 times higher, this is due to the compressed sessions being processed by the webserver much quicker (since there is less data to transfer).

TEST 2 - 10 CONNECTIONS, 10 PARALLEL SUBREQUESTS.

We will now perform a more realistic case, where the browser is able to make multiple simultaneous sub-requests for the linked files. This test essentially simulates 10 different users requesting the report page more or less at the same time.

	<code>mod_gzip</code> Off	<code>mod_gzip</code> On	Factor
Connection Rate (conn/s)	8.7	23.1	~ 2.6x faster
Connection Rate (ms/conn)	115.2	43.4	~ 2.6x faster
Session lifetime (Seconds)	1.1	0.5	~ 2.2x faster
Total Content Size Returned (Kb)	380	80	~ 5x smaller
Average Session Rate (sessions/sec)	2.89	7.69	~ 2.6x faster

This time we see that the connection rate for the non zipped version remained more or less uniform, whereas because the compressed response can be delivered to the browser quicker it frees up the Apache processes to deal with other requests faster, therefore the connection rate increases. The average session rate factor drops a little however, this is probably mainly due to the initially bad session rate returned in the previous test.

TEST 3 - 50 CONNECTIONS, 10 PARALLEL SUBREQUESTS.

This time we will simulate 50 connections, each request is capable of being handled by 10 parallel subrequests.

	mod_gzip Off	mod_gzip On	Factor
Connection Rate (conn/s)	23.2	53.6	~ 2.3x faster
Connection Rate (ms/conn)	43.1	18.7	~ 2.3x faster
Session lifetime (Seconds)	1.1	0.4	~ 2.7x faster
Total Content Size Returned (Mb)	1.8	0.4	~ 4.5x smaller
Average Session Rate (sessions/sec)	7.74	17.85	~ 2.3x faster

Once again the test with the compression is still performing better and we are able to achieve double the connection rate if we use compression.

TEST 4 - 200 CONNECTIONS, 10 PARALLEL SUBREQUESTS.

We'll again test 200 connections, which approximate the effect of 200 users all accessing the report page simultaneously.

	mod_gzip Off	mod_gzip On	Factor
Connection Rate (conn/s)	3.2	34.6	~ 11x faster
Connection Rate (ms/conn)	312.7	28.9	~ 11x faster
Session lifetime (Seconds)	4.9	3.8	~ 1.2x faster
Total Content Size Returned (Mb)	7.4	1.5	~ 5x smaller
Average Session Rate (sessions/sec)	1.06	11.55	~ 11x faster

As before, once we start really hitting the webserver with lots of simultaneous requests, the non-compressed test starts to show signs of stress, whereas the compressed test shows that the webserver is able to deal with an order of magnitude more requests per second.

COMPRESSION WRAP-UP

Hopefully in this section I have shown just want a huge impact configuring your webserver to support compression can have. The benefits of compression are threefold:

- Less network traffic is created, i.e lower bandwidth requirements.
- Faster page loading for the user, since less data needs to be downloaded.
- The webserver is able to service a great number of users since it is transferring less data per request, it can service more requests in the same period of time.

You may be wondering at this point whether there is a risk that the processing overhead of compressing the response output will actually make using compression slower rather than faster. In my experience (to date) that has not been the case and as I have shown the results are demonstrably in favour of using compression. The only time when the processing overhead would be substantial when compared to overall result would be when you're compressing extremely small files (say 2Kb or less), the effort expended in trying to compress files of this size probably does not outweigh the benefits of reducing the size.

Fortunately the `mod_gzip` configuration file lets us specify a minimum file size so that we don't try and compress files smaller than this size (see the `mod_gzip_minimum_file_size` parameter for more information).

FINAL WORDS

I really would like to stress that the two techniques I've discussed in this paper can be of great benefit to many projects, even if you don't have to pay for your bandwidth or you don't think you use that many images in your application so you don't think it would be worthwhile.

The good thing about both techniques is that their benefits also scale with your project. By that I mean as the number of users of your application grows then so will the savings in terms of bandwidth and resource usage. Also by using the image caching technique your application will be able to scale to support a larger number of users than if you didn't use it.

However please don't think you need lots of users to implement these techniques, I really would advise using them from day 1 of your project. If you setup the `mod_gzip` module, then not only will the users of your application benefit, but so will you as you use the Application Builder during your project development (yes, you'll also benefit from compression of the webpages).

It's very tempting to look at these examples and think "well it's only a few kilobytes here and there, who care. It may not look a lot in terms of a single page view, however if you are saving 80-90% bandwidth on a weekly/monthly or yearly basis then that could equate to a considerable saving. Likewise, if you use image caching and manage to substantially cut down on the number of unnecessary requests that are made to your webserver and database, then that will also equate to tangible savings in terms of investment, scalability and performance.

REFERENCES

- LWP - <http://search.cpan.org/~gaas/libwww-perl-5.805/lib/LWP.pm>
- HTTPPerf - <http://www.hpl.hp.com/research/linux/httpperf/>
- mod_gzip - <http://sourceforge.net/projects/mod-gzip/>
- Apache - <http://www.apache.org/>
- Shellprompt Hosting – <http://www.shellprompt.net>