Bortel...

Tuesday, July 18, 2006

Authentication in mod_plsql

Ever wondered what all these possibilites in your DAD configuration meant? I did, and I have used 'Per package' since.

Introduction

Let me assume you have your database (9i Release 2, or 10g, Release 1 or 2) installed and configured, and you have the Oracle Webserver by Apache installed, too. After you started the http server, you should be able to fire up your favourite browser, and browse to
localhost:7778.
If that does not work, look in the httpd.conf file (ORACLE_HOME\Apache\Apache\conf) for the Listen directive. If it reads something else that 7778 (e.g. 7777), use that - mine uses 7778, so I will continue to use that in this entry.
Now that we have the standard Oracle HTTP server home page, look at the entries. Fourth from the bottom, there is this 'Mod plsql configuration menu'. Click it, and click the second option on the following page, the link reading 'Gateway Database Access Descriptor Settings'. By the way - that is what the acronym DAD stands for: Database Access Descriptor.

New DAD entry.

Let's create a new entry, and leave the defaults for what they are. From top to bottom, the entries are:
DAD Name: the name you want to give this configuration. It will be visible to the world; when in use, you will access your application via a browser bij pointing to

```
[your_host:7778]/pls/[your_dad]/[package.procedure]
```
, unless we do some fancy things, which I will show later. For this demonstration, let's use TEST for a name.
Oracle user name, password and connectstring: you have a choice here. You can fill in the username, and leave the password blank, or leave username and password blank, or fill in both. The connectstring must always be filled in.
What are the consequences? Well, that depends somewhat on the next entry, Authentication Mode.
Assuming the default Authentication mode of 'Basic', just filling in the username, and leaving the password blank will present you a grey login popup, requesting username and password. Just filling in the password will not help; you must provide a valid database user *and* it's password.
The beauty of this appraoch is the fact user and password are cached, and sent every time by mod_plsql, so once a session is established, it remains "active". Once disconnected, and recalled, mod_plsql sends user and password, so the session is reestablished. But how to really log off, you ask? Simple - call the logmeoff page of mod_plsql:
localhost:7778/pls/test/logmeoff.

It's built in, and hidden, but documented.

Leaves the option to fill in username and password. Should I explain? Credentials are stored in the DAD configuration, and thus always sent. Logging off doesn't help much, it does clear the cache, but as user and password are stored in the configuration...
By the way, here's a simple procedure to test your DAD:

```
CREATE OR REPLACE PROCEDURE home IS
begin
htp.p('This is a test page to verify login:');
htp.p('Current user: '||user);
END home;
/
grant execute on home to public;
create public synonym home for home;
```

Test it, using
localhost:7778/pls/test/home

Session cookie: as the explanation shows, leave it blank. It's only used in clustered environments.
Package/State management: Stateless (Fast Reset). We are using 9i, or 10g, better than 8.1.7.2, anyway.
Connection pooling: Leave it on - it means multiple requests can be handled through one database session.
Default page: now it is getting interesting again. If you get fed up by typing `localhost:7778/pls/test/home`, you can enter home here. Instead of `localhost:7778/pls/test/home`, you can now enter `localhost:7778/pls/test` - big deal!
Document Access and Path aliasing: out of scope for this entry. Maybe in a later one.
For now, let's see how you can get rid of the `localhost:7778/pls/test`...
One step can be easy, but will only work if you plan to use just one DAD. That involves changing the Default DAD. It's the first entry on the 'Mod pl/sql configuration settings' menu page: Gateway Global Settings. Change the Default DAD descriptor to TEST, and you can now reach the
localhost:7778/pls/test/home
page by just entering
localhost:7778/pls.
This works, because:
You changed the default DAD to test, so all references to mod_plsql are routed to this DAD.
You changed the default page for the TEST DAD to HOME.

So far so good.

Now, let's see what the other Authorization Modes do.
Single Sign On: If you don't have portal (more correctly: LDAP), forget about this option.
Global OWA, Custom OWA and Per package have so much in common, I'll treat them as one entry. All three are grouped into "Custom Authentication" in the manual as opposed to Oracle standard authentication, either by the database (Basic) or by Portal LDAP (Single Sign On). the differences are:
Global OWA: Access control applies to all packages, governed by the

owa_custom.authorize function in the OWA package schema.
Custom OWA: Access control applies to all packages, governed by the owa_custom.authorize function in the user's package schema.
Per Package: Access control applies to the specified package, governed by the authorize function in that package.
Let's see what happens if we change the TEST DAD Authentication mode from Basic to Per Package. It results in a 404: Not Found error. A glance at the Apache log file reveals:

```
[Mon Jul 17 14:52:25 2006] [warn] mod_plsql:
/pls/test/home HTTP-404 Custom Authentication Failure.
[authorize] oerr = 6550
ORA-06550: line 7, column 6:
PLS-00201: identifier 'AUTHORIZE' must be declared
ORA-06550: line 7, column 2: PL/SQL: Statement ignored
```

Now what was it about Custom Authorization? It needs a package (oops... we still have a procedure), and that package needs to have a function, called authorize. That function must return a boolean; true means, OK - you're authorized, false means no go.
So, here are the changes:

```
create or replace package test as
function authorize return boolean;
procedure home;
end test;
/

create or replace package body test is
function authorize return boolean is
begin
return true;
end authorize;
procedure home is
begin
htp.p('This is a test page to verify login:');
htp.p('Current user: '||user);
END home;
end test;
/
```

Change the default page on the TEST DAD to read
test.home ,
and retry
http://localhost:7778:/pls

Change the authorize function to return false, and you will receive the a 403: Forbidden page.

So far, so good...

Where does this lead to? Well, you can now have an application, that has a public part and a private part. You can design your own login screens (which should be public!). In fact, there are two ways of doing that:

1. Use the owa_sec package
2. use your own package

Authentication, using owa_sec.

Basically, you can use the `owa_sec.set_protection_realm` procedure. In takes a character string as input, which is displayed on this ugly grey popup. Let's give it a try:

```
create or replace package body test is
function authorize return boolean is
begin
owa_sec.set_protection_realm('TestSec');
if (owa_sec.get_user_id = 'guest') then
 return true;
else
 return false;
end if;
end authorize;

procedure home is
begin
htp.p('This is a test page to verify login:');
htp.p('Current user: '||user);
END home;
end test;
/
```

You must make a call to the owa_sec package in order to activate the logon popup.
The nice thing about this package is that the username and password are cached, and you do not need to worry about cookies - it's all taken care of.
However, I'm not really thrilled by the look and feel of all this, and would like a custom login page, which I can format to my own look and feel. Also, I would like some self provisioning in my applications, so there would be a link to a "I forgot my password" page, to name one.
Obviously, you cannot reach that page if you would have to login.

Custom Authentication.

So, let's do something else, and split the packages into a public part, and a private part.
The public part would display:

- self provisioning pages, like "I want access, too", "I forgot my password"
- a customized login page
- a basic menu with the above options

The private part would hold all, well, shielded off screens of the application.
Of course, the whole lot of session state, cookies and session id's needs to be taken care of, so it is getting more complicated. But hey, you choose to read this far, so let's continue!

HTML Wrapper.

I think, it's about time to introduce a wrapper tool. The original was Sten Vesterli's work and appeared in his book Web Apps 101. Sadly, his page is no longer maintained.
Update jan-2009: checking links, I did find a working link, where you supposedly can

download his original code. Didn't work for me... I made some changes (I'd like to call it improvements), so that the wrapper now produces XHTML. The basic idea is to have a wrapper package that does a lot of htp.p in a single call, e.g. format.pagestart will send the stylesheet, open html, set the header, close the header and open the body, or htmlform.selecttable, taht will take a table name as input and produce a drop down box with all values from that table. Drop me a line (comment) and I'll email the packages to you.
That said, let's do some inventory; what we need is:

- session state
- password encryption
- logon and logoff procedures
- private and public authorization

Session State.

I have tried several things to maintain state, but as HTTP is essentially stateless, I just have to use a table here. This is my session table definition (as well as Tom Kyte's...)

```
create table session_table
( session_id    varchar2(32) primary key,
timestamp     date,
username      varchar2(30),
userprofile     varchar2(16)
userid    number
)
/
```

And the basic user table:

```
create table webapp_users
(  id number not null,
identification varchar2(16) not null,
email varchar2(80) not null,
passwd varchar2(20),
userprofile varchar2(16) not null ,
valid_until date not null,
status varchar2(3)  default 'REQ' not null,
remarks varchar2(4000)
)
/
```

Authorize does not change, as this is the public part. What I do is:

```
procedure showlogin
is
begin
begin
g_cookie := owa_cookie.get ('demoapp'); -- try to get the cookie
exception
when others
then
null;  -- no problem if we could not get the cookie
end;

if g_cookie.num_vals > 0 -- if we did get the cookie, remove it
```

```
then
begin
owa_util.mime_header ('text/html', false);
owa_cookie.remove (name => 'demoapp', val => g_cookie.vals (1));
owa_util.http_header_close;
exception
when others
then
   null;
end;
end if;

delete from session_table
where timestamp < p_title      =""> 'Demo application - Login',
           p_text       => 'Welcome
Authorised access only!</span>' ));
HTP.p ('</div>');
HTP.p (   '<div class="menu">'
          || format.hyperlink (p_text             => 'Login page',
                               p_destination      => 'test.showLogin',
                               p_title            => 'Back to login
page'
                              )
          || '</div>' );
HTP.p ('<div class="content">');
HTP.p (htmlform.formstart (p_action => 'test.Validate_User'));
HTP.p (htmlform.tablestart (p_summary => 'Login'));
HTP.p (htmlform.textfield (p_label    => 'User name',
                         p_name     => 'p_username',
                          p_maxlength => 16));
HTP.p (htmlform.passwordfield (p_label => 'Password',
                             p_name  => 'p_password',
                             p_maxlength => 20));
HTP.p (htmlform.buttonsstart);
HTP.p (htmlform.submitbutton (p_label => 'Login'));
HTP.p (htmlform.buttonsend);
HTP.p (htmlform.formend);
HTP.p ('</div>');
HTP.p (format.pageend);
END showlogin;

PROCEDURE validate_user (p_username IN VARCHAR2, p_password IN
VARCHAR2)
is
BEGIN
IF p_username IS NOT NULL AND p_password IS NOT NULL
   THEN
      IF validate_usr (p_username, p_password)
      THEN
         OWA_UTIL.mime_header ('text/html', FALSE);
         owa_cookie.send ('demoapp', g_sid, NULL);
         IF g_profile IN ('ADMIN', 'MODERATOR')
         THEN
            OWA_UTIL.redirect_url ('testpr.apage1', bclose_header =>
TRUE);                             ELSE
            OWA_UTIL.redirect_url ('testpr.upage1', bclose_header =>
TRUE);
         END IF;
      ELSE
         HTP.p (format.infopage ('Invalid login'));
      END IF;
   ELSE
      OWA_UTIL.redirect_url ('test.showLogin', bclose_header =>
FALSE);
   END IF;
END validate_user;
```

Based on the profiles, I switch to the user pages in the private area (testpr.upage1), or the administrative pages (testpr.apage1). If there is an invalid attempt, I display an error page; if something else is wrong (e.g. no password given), I simply redisplay the login page. The validate_usr function is:

```
function validate_usr (p_uid in varchar2, p_pswd in varchar2) return
boolean
is
l_crypted_psw   raw (32);
l_id            gebruikers.id%type;
begin
l_crypted_psw := crypt (p_pswd);
begin
 select w.userprofile, w.id
 into g_profiel, l_id
 from webapp_users g
 where w.identification = upper (p_uid)
 and w.passwd = l_crypted_psw
 and w.status not in ('REQ', 'EXP', 'DEL');
exception
when no_data_found
then
 return false;
end;

select cast (sys_guid () as varchar2 (32))
into g_sid
from dual;

insert into session_table
  (session_id, timestamp, username, userprofile, userid)
values
  (g_sid, sysdate + 1 / 48, p_uid, g_profiel, l_id);

commit;
return true;
exception
when others
then
htp.p (format.errorpage (sqlerrm));
return false;
end validate_usr;
```

The function crypt is derived from examples by Tom Kyte:

```
FUNCTION crypt (p_str IN VARCHAR2)
RETURN RAW
AS
l_data   VARCHAR2 (80);
BEGIN
l_data := RPAD (p_str, (TRUNC (LENGTH (p_str) / 8) + 1) * 8, CHR (0));
DBMS_OBFUSCATION_TOOLKIT.desencrypt
                                 (input_string          => l_data,
                                  key_string            =>
'SomeStrAgeCharATersH3re',
                                  encrypted_string      => l_data
                                 );
RETURN UTL_RAW.cast_to_raw (l_data);
END crypt;
```

What remains, is the private code to authorize, get and set a cookie and maintain session state. Remember, this code all comes from the private package (TESTPR):

```
function get_cookie (p_str in varchar2)return varchar2 is
begin
g_cookie := owa_cookie.get (p_str);
if g_cookie.num_vals > 0
then
 return g_cookie.vals (1);
else
 return null;
end if;
end get_cookie;

function authorize return boolean
is
l_sid   varchar2 (32);
begin
-- this part is private, so, validate..
l_sid := get_cookie ('demoapp');
debug.log (1, 'cookie=> ' || l_sid);

update    session_table
set timestamp = sysdate + 1 / 48
where session_id = l_sid and timestamp > sysdate - 1 / 48
returning username, userprofile
into g_session_rec.username, g_session_rec.userprofile;

if (sql%rowcount = 0)
then
 return false;
else
 return true;
end if;
end authorize;

procedure set_cookie (p_str in varchar2, p_val in varchar2)
is
begin
dbms_application_info.set_module ('DemoApp', 'set_cookie');
owa_util.mime_header ('text/html', false);
owa_cookie.send (name => p_str, value => p_val, expires => null);
owa_util.http_header_close;
end set_cookie;
```