SDN Contribution

BSP IN-DEPTH: CONFUSION BETWEEN STATELESS, STATEFUL AND AUTHENTICATION

Summary

Currently, we are processing a problem with the following text:

We have a stateless BSP application. After a predefined time, we would like all inactive sessions to be deleted from ICM, and thereafter the user must be forced to again supply a name and password.

The rest of the problem description is not listed here. If only we can succeed in clearing the confusion in this one paragraph, the rest will be easy. What is required is a clear understanding of the expressions stateless sessions, stateful sessions, and authentication.

Created on: 9 March 2006

Author Bio

Brian McKellar

Test Harness

As usual, as first step, let us build a small test program with which the different concepts can be explained. All that the program should do, is increment a counter, plus allow us to switch the session between stateless, and stateful.

Create a new BSP application, with one page. Define the following page attribute:

```
counter TYPE I
```

This counter is defined as page attribute. While we are stateless, the page will be created new each time, and the counter will always be initialised to zero. However, once the page is switched into stateful mode, the page (object) will be cached and reused. The counter will keep its previous value, allowing us to verify that we are always running in the same session and that this session is keeping its state.

For the layout page we have:

```
<%@page language="abap" %>

<html><body><form>

  <%-- increment counter, and write out --%>
  <% counter = counter + 1. %>
  <input type = "submit" value = "Counter=<%=counter%>">

  <%-- determine port from URL, for port get service timeout, sleep handling --%>
  <%
    DATA: port TYPE STRING.
    port = request->get_header_field( if_http_header_fields_sap=>server_port ).

    DATA: services TYPE TABLE OF ICM_SINFO.
    CALL FUNCTION 'ICM_GET_INFO' TABLES SERVLIST = services.

    FIELD-SYMBOLS: <service> TYPE ICM_SINFO.
    DATA: wait TYPE STRING.
    READ TABLE services ASSIGNING <service> WITH KEY service = port.
    wait = <service>-KEEPALIVE + 5.
    CONDENSE wait.

    IF request->get_form_field( 'wait' ) IS NOT INITIAL.
      WAIT UP TO wait SECONDS.
    ENDIF.
  %>
  <input type = "submit" name = "wait" value = "Wait<%=wait%>seconds">
  <BR>

  <%-- stateful or stateless? --%>
  <%
     DATA: stateful TYPE STRING.
     stateful = request->get_form_field( 'stateful' ).
     IF stateful IS NOT INITIAL. runtime->keep_context = 1.
     ELSE.                       runtime->keep_context = 0.
     ENDIF.
  %>
  <input type="checkbox" name="stateful" value="checked" <%=stateful%>>Stateful
  <BR>

  <%-- output similar to transaction SM04 --%>
  <%
    DATA: list TYPE TABLE OF UINFO,
```

```
          user LIKE LINE OF list.
    CALL FUNCTION 'TH_USER_LIST' TABLES LIST = list.
  %>
  <table border=1 cellspacing=2 cellpadding=0>
  <% LOOP AT list INTO user WHERE BNAME = sy-uname. %>
    <tr>
       <td> <%= user-MANDT %>     </td>
       <td> <%= user-BNAME %>     </td>
       <td> <%= user-TERM %>      </td>
       <td> <%= user-ZEIT %>      </td>
    </tr>
  <% ENDLOOP. %>
  </table>

</form></body></html>
```

The program can be split into a few parts.

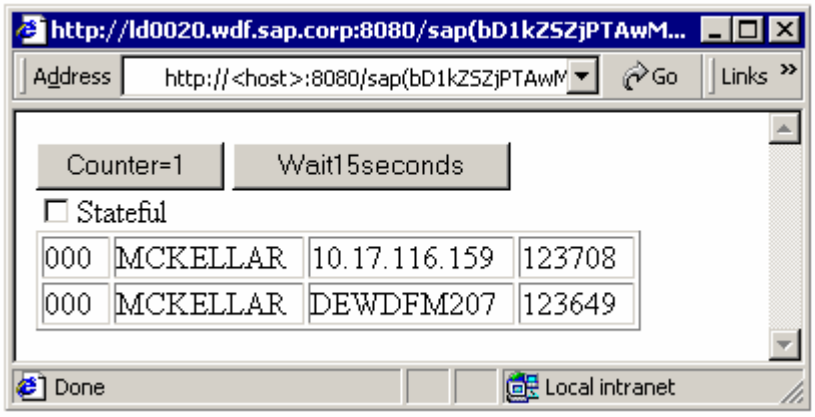
- The first part just increments a counter, and writes the counter out (on a button) to the browser. Important, the value of the counter is not stored anywhere in the page. We are only interested to see if the counter (an attribute of the page object) is incremented for each request/response cycle or not.

- The next coding sequence looks slightly complex, but does not actually do all that much. It first looks at the incoming request, and determines the HTTP port number from the request. Then all ICM defined services are retrieved and we determine the processing time that is configured for this specific port. A button is rendered to “sleep” this number (plus five) seconds. If this button is actually pressed, the ABAP “WAIT UP TO” statement is used to simulate a long running HTTP request. This will be used later in the test to show the effect of the service timer.

- The next section toggles stateful or not, and renders a checkbox to manage this.

- The last second outputs exactly the same information as transaction SM04. This lists all active sessions for the specific user.

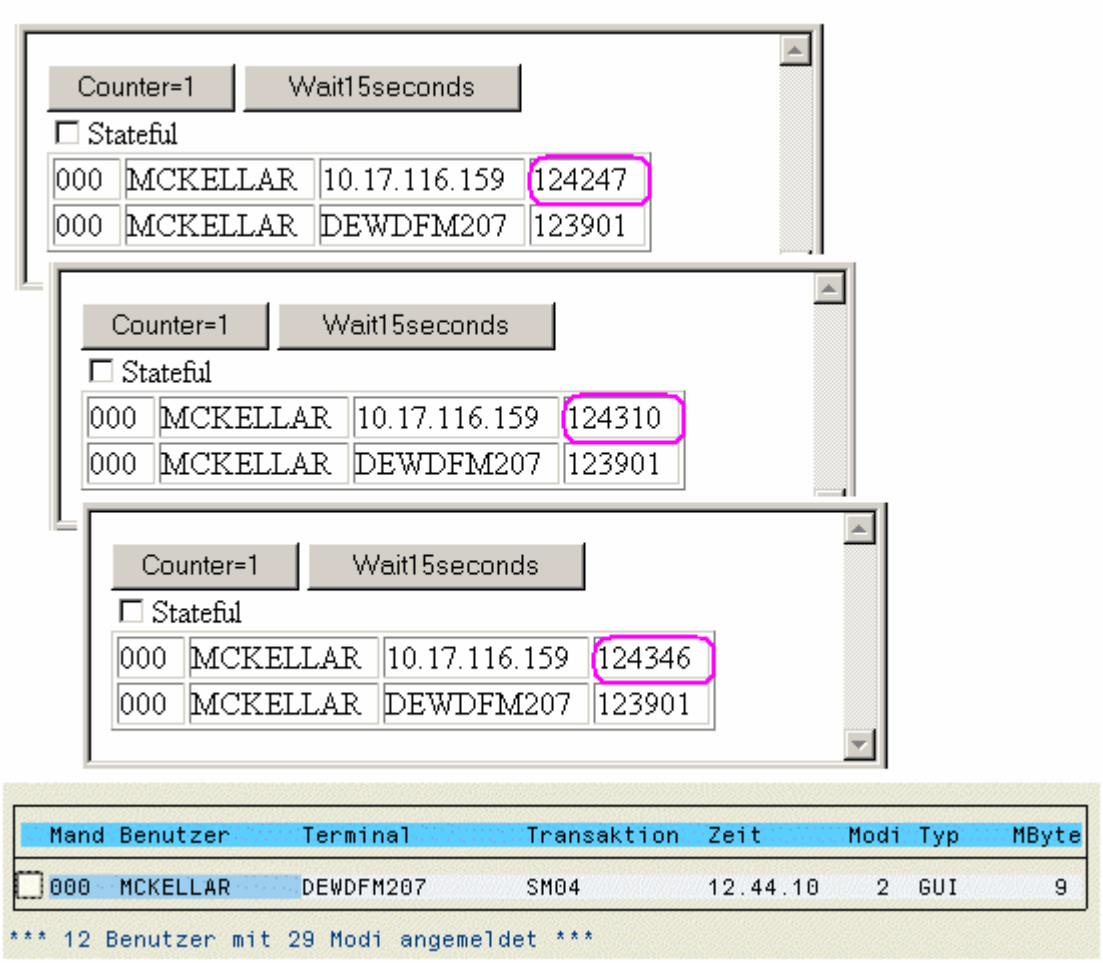
The output of the program after a first run is:



The table displays client, user, terminal, and last update time (shortly before lunch:).

Stateless

As first test, we leave the mode as stateless, and press the button three times. At the same time, we keep an eye on transaction SM04.



We see in the browser a number of very interesting aspects.

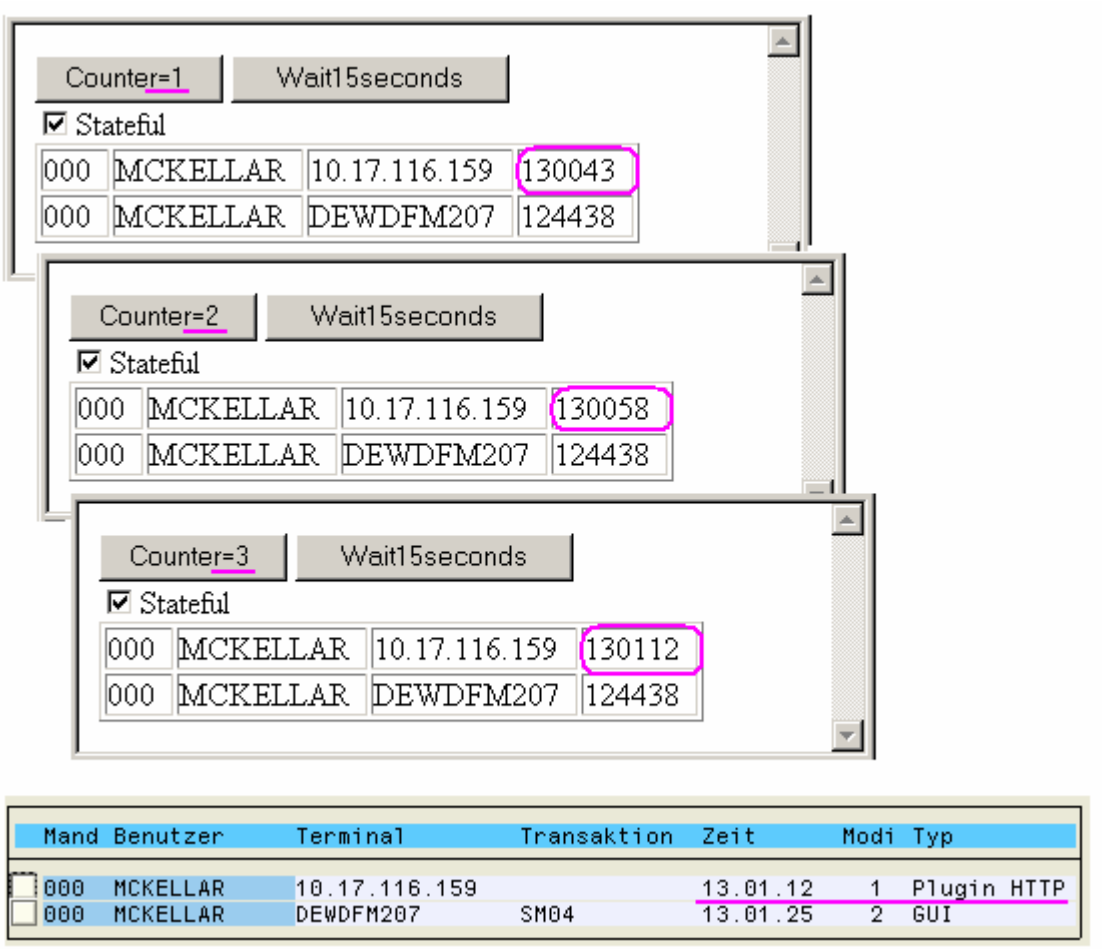
The first is that the counter is always one. Effectively, the session is stateless. Each time that the request hits the server, a complete new session (ABAP roll-area) is opened, and is used to execute this request. Thus, the BSP page is also created new each time, resulting in counter starting new. This is what would be expected from a stateless application.

The other interesting aspect is that the transaction SM04 shows only one session for us. However, the output of each request shows two sessions! Thus, during the time that the BSP page is been processed, there is actually an active session for this HTTP request. If we were to be very fast in refreshing the SM04 display (below 10 milliseconds!), we would actually see this second session for one moment. Directly after the HTTP request has been processed, the session is cleared and does NOT exist anymore. However, during the time that the HTTP request is been processed, there does actually exist a normal WebAS session for this HTTP request.

After the HTTP request has been processed, the session is closed, and the WebAS “forgets” completely about the user. No information is stored.

Stateful

As next step, we switch to stateful mode, and again press the button three times.



The first aspect we notice is that the counter is incremented, and now keeps state. This is the proof that we are running into the same session each time (so actually the session is kept even after the HTTP request has been processed).

Inside the request, again we see at all times only two sessions listed. One is the SAP GUI session that we have open, and the other is the session for the HTTP request. If we now look into transaction SM04, we will also both sessions there.

So even after the HTTP request has been processed, the session is kept alive on the server.

Sidebar: HTTP, the One-Way Communication Protocol

In most communications, both partners are allowed to say something, and keep the discussion alive. However, this is not true for HTTP. HTTP is a strictly request/response protocol. Only the browser is allowed to fire a request at the server. The server is only allowed to answer this specific request with a response.

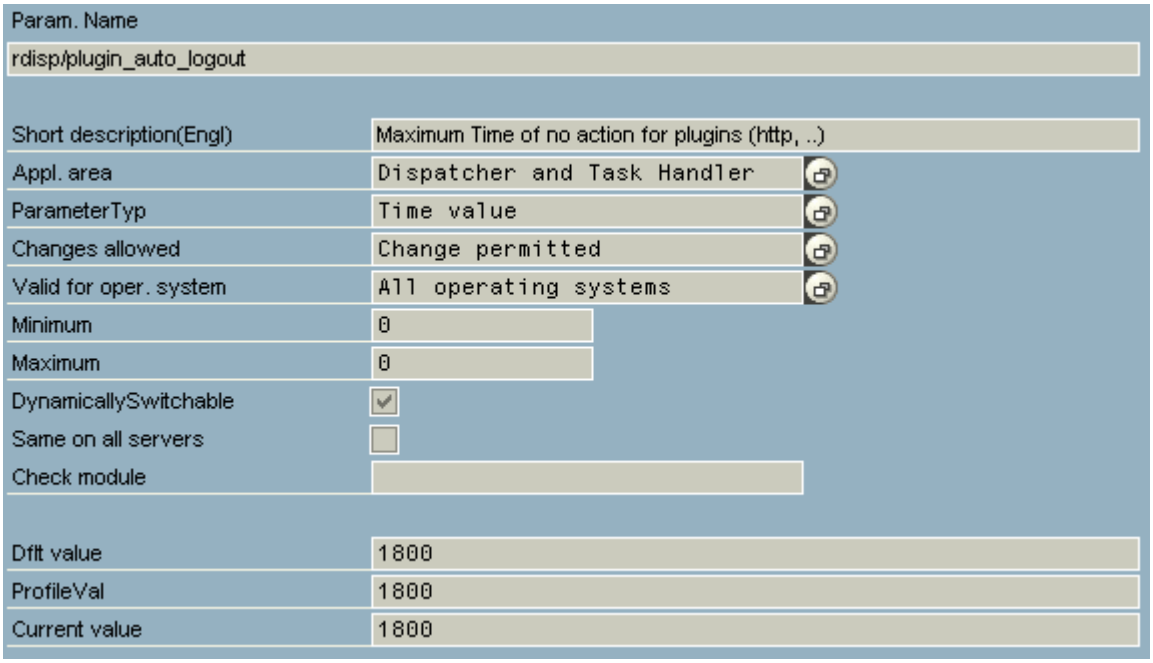
It is not possible for the server to start a request to the browser, and it is not possible for the server to send a response, without having first received a request.

It is true that the HTTP/1.1 protocol allows the browser and server to keep the TCP/IP connection open between them for a short time after use, as to reuse the same connection again. However, in terms of communication, the next HTTP request must still come from the browser. There is no way for the server to inform the browser of a specific even, such as a session termination.

Timer: Session Management

After about 35 minutes (after lunch!), we look into SM04, and see that only the SAP GUI session is listed. The server has completely "forgotten" about the HTTP session that was previously active. (In the browser, we see nothing of this event, due to the one side nature of the HTTP protocol.)

Session management is controlled by the profile parameter "rdisp/plugin_auto_logout". This parameter, with a default value of 30 minutes, controls the time that a session can be idle before it is cleared.

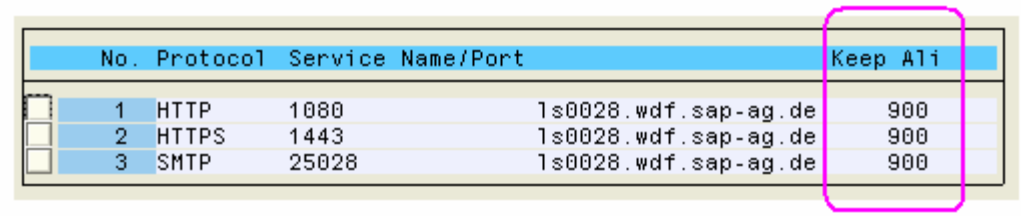


Use transaction RZ11 to see/change this value. Important, any change to this parameter affects all HTTP sessions for the application server!

After the idle timer has expired, the session is silently cleared. All active locks are released.

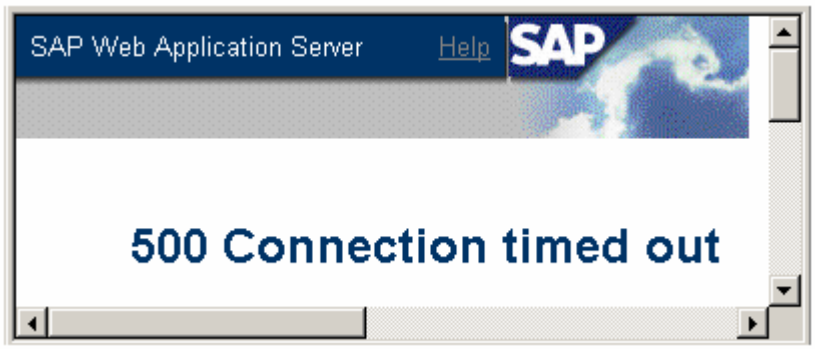
Timer: HTTP Request Processing Time

The other interesting time is the time that one HTTP request will be processed, before it is aborted. This time can be seen in transaction SMICM, under services. It is called "keep-alive" time (very unfortunate terminology), and is often confused with the session timeout value. This time is specified in the profile as part of the icm/server_port specification (which includes port number to use, and protocol for this port).



No.	Protocol	Service Name/Port		Keep Ali
1	HTTP	1080	ls0028.wdf.sap-ag.de	900
2	HTTPS	1443	ls0028.wdf.sap-ag.de	900
3	SMTP	25028	ls0028.wdf.sap-ag.de	900

For our example program, we have a second button that will cause a very long running HTTP request. We press this button, and after waiting for ages, receive the answer in the browser:



Effectively after the timer expired, the ABAP processing of the HTTP request was aborted by ICM. The error message is returned by ICM (reflecting the “connection” between ICM and ABAP).

After the HTTP request processing has been aborted, the session is still kept intact in WebAS 620. (To verify this, run the test program in stateful mode, increment the counter, and then press the wait button to cause the HTTP request processing error. Thereafter, delete the sequence “wait=Wait15seconds&” from the URL, and press ENTER again. The URL is submitted into the same session, and the counter will be incremented one more!)

(Basic) Authentication

The last part of the puzzle to explain is how basic authentication works. With this knowledge is will then be possible to answer the initial question that lead us down this road.

Below is a stripped down HTTP trace of the communications between the browser and the server.

```
GET /sap/bc/bsp/sap/bcm06/session_test.htm HTTP/1.1
Accept: */*

HTTP/1.1 401 Unauthorized
www-authenticate: Basic realm="SAP Web Application Server [B20]"
```

As a first step, we enter the URL in the browser, and it tries to access the page immediately. The server answers with unauthorized (HTTP return code 401). Now the browser pops up a window asking for a username and password. (We assume for this scenario simple basic authentication, and not the more interesting X.509 certificates, etc.)

```
GET /sap/bc/bsp/sap/bcm06/session_test.htm HTTP/1.1
Accept: */*
Authorization: Basic V2h5IG5vdCB3cml0ZSBhIFNETiBhcnRpY2xlPzpJdCBpcyBhIGxvdCBvZiBmdW4h

HTTP/1.1 302 Moved temporarily
location: /sap(bD1lbiZjPTAwMA==)/bc/bsp/sap/bcm06/session_test.htm
```

Now that the browser has the username and password, the information is base64 encoded, and send with the next request (in the “Authorization” header). The server has now receives a HTTP request, from an authenticated user, and passes the information to the BSP runtime. The BSP runtime does its URL mangling routine, and redirects the browser back to a new URL. (URL mangling is discussed in another BSP In-Depth article!).

```
GET /sap(bD1lbiZjPTAwMA==)/bc/bsp/sap/bcm06/session_test.htm HTTP/1.1
Accept: */*

HTTP/1.1 401 Unauthorized
www-authenticate: Basic realm="SAP Web Application Server [B20]"

GET /sap(bD1lbiZjPTAwMA==)/bc/bsp/sap/bcm06/session_test.htm HTTP/1.1
Accept: */*
Authorization: Basic V2h5IG5vdCB3cml0ZSBhIFNETiBhcnRpY2xlPzpJdCBpcyBhIGxvdCBvZiBmdW4h

HTTP/1.1 200 OK
content-length: 693
content-type: text/html; charset=iso-8859-1
<html><body><form>...</form></body></html>
```

Now comes the most interesting sequence of all! The browser has a new URL to load (because of the redirect before). As a first step, it tries to load this new URL without supplying any authentication information (notice no Authorization header!). The server does not like this request from an unknown user, and rejects it again. At this moment, the browser has already authorization information (keep the popup for username and password in mind that was done previously). Therefore, the browser immediately requests again the new URL, but now supplying the authorization information that it had stored previously. The server is happy, the BSP runtime is even happier, and then BSP application is executed to do its song and dance. An OK answer (HTTP return code 200) is returned to the browser, and the page is displayed.

```
GET /sap(bD1lbiZjPTAwMA==)/bc/bsp/sap/bcm06/session_test.htm HTTP/1.1
```



```
Accept: */*
Authorization: Basic V2h5IG5vdCB3cml0ZSBhIFNETiBhcnRpY2xlPzpJdCBpcyBhIGxvdCBvZiBmdW4h

HTTP/1.1 200 OK
content-length: 693
content-type: text/html; charset=iso-8859-1
<html><body><form>...</form></body></html>
```

From now onwards, the URL stays the same for the duration of the application, and the browser will in all cases always send the authorization information as well.

In summary: (basic) authentication is a feature of the browser. When requested from the server to supply some form of credentials, the browser wills popup a window asking for this information. Thereafter, this information is used sparingly. For all new URLs, the browser will first attempt to access the URL without the authorization information it has stored. Only on failure is a second attempt made with the authorization header.

Sidebar: Defining a New ICM Service/Port

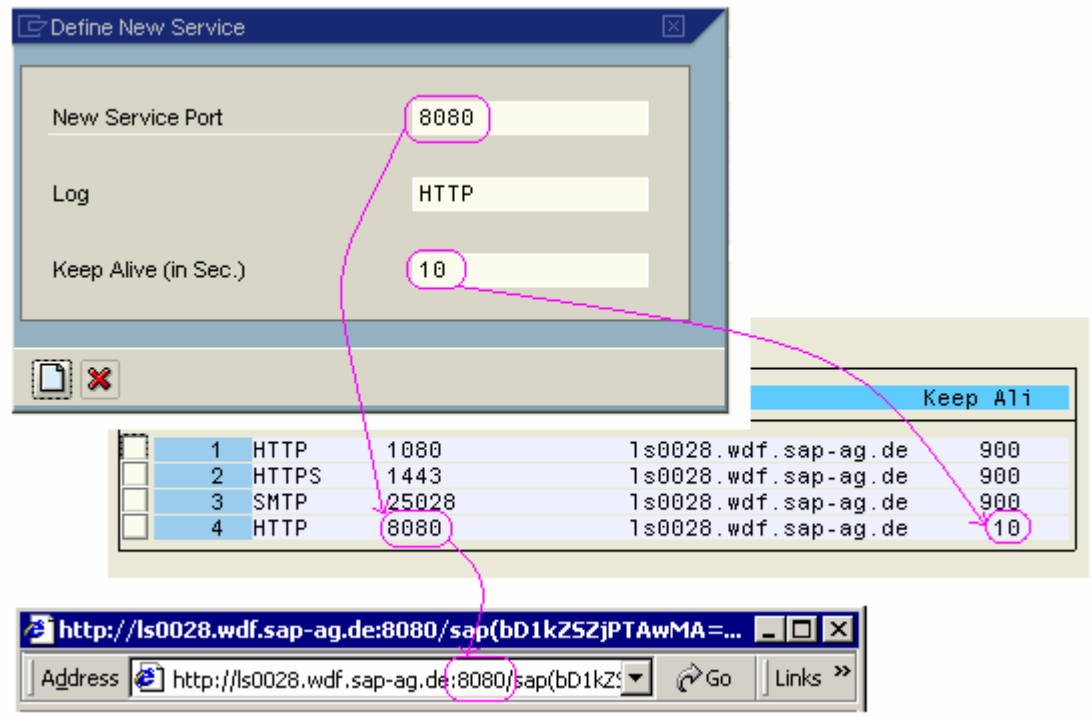
Each HTTP request is only allowed a limited time to be processed, before ICM will abort processing, and return a HTTP return code 500. For usual production systems, this time is typically set between 15 and 90 seconds. This allows each request to be processed completely, and at the same time guards that the HTTP requests do not run out of control. (Effectively one aspect of a denial of service attack that must be controlled.)

Once a colleague debugged very fast for about 60 seconds, and then he spoke a few words of perfect German, before starting again. (No wonder my wife keeps asking me where I learn German:). The reason for this is that after 60 seconds, the processing time was up, and ICM aborted the HTTP request, and thus the debugging session was stopped.

The solution is very simple: It is possible to define a new HTTP service (port) dynamically. For the new port, define a large processing time, say 900 seconds (= 15 minutes).

To do this, start transaction SMICM. Then Menu: Goto → Services → Menu: Go To → Service → Define. Specify a new port number, and a large “keep alive” time. When testing, just change the port number in the URL to use the newly defined one.

This is the same technique used in this test program to get a very short processing time, so as to not wait for ages.



The Final Answer

The original question started out with a stateless BSP application. It was shown that for stateless applications, no information is stored on the server. Once the HTTP request has been processed, the server completely "forgets" about this user and the session is destroyed. During the lifetime of this HTTP process, its session can be seen in SM04.

Furthermore, it was shown that for stateful sessions, the server actually keeps the session, and that this session can be seen in SM04. After the inactivity timer bites, the session is "forgetting".

As extra bon-bon we quickly looked at the two more most interesting timers that are active for session management and HTTP processing.

Lastly, we showed how basic authentication works, and that this is a feature of the browser. The authentication information is stored in the browser, and cannot be controlled from the server.

Now back to the original question: *We have a stateless BSP application. After a predefined time, we would like all inactive sessions to be deleted from ICM, and thereafter the user must be forced to again supply a name and password.*

The answer is now relatively simple. As the applications are stateless, there is no information on the server about this application (any of its sessions). It does not matter if the user waits a few seconds, or a few hours, on the next request the browser will again send the basic authentication information with the request. The server will see an incoming HTTP request, that contains authentication information, and process it. The session is then closed again.

The problem must be reformulated: "How to get rid of authentication?". As already shown, for basic authentication, this is not possible.

Some Parting Ideas

The question originally comes from a company that values their security, and their data.

The best technique to remove authentication information from the browser is to NOT use basic authentication, but form based authentication (see OSS/CSN note **517860**). With this technique, a SSO2 cookie is set in the browser that contains the authentication information. Cookies can again be deleted to remove the authentication information. Form based authentication is quite a long topic by itself, and can be considered for a future BSP In-Depth.

One very simple idea, which is just a fake, is to add the following coding at the bottom of the page:

```
<script>
  window.setTimeout("document.location.href='about:blank'", 300000 /*5min*60sec*/);
</script>
```

As long as the user is working, the HTML document will get loaded new each time, cancelling the previous timer, and setting a new timer. After 5 minutes of inactivity, the screen is blanked. **ULTRA IMPORTANT:** This only results in removing confidential information from screen. It does NOT remove the authentication information. Using back navigation, it is still possible to use the application.

If there is any interest in addressing the de-authentication process, please send me a short email, and we will schedule it for another In-Depth article in the near future.

Copyright