Bs_SimpleSession:
simple but not stupid

Sam Blume

Andrej Arn

blueshoes

FROM THE PHP-MANUAL:

"THE SESSION SUPPORT ALLOWS YOU TO REGISTER ARBITRARY NUMBERS OF VARIABLES TO BE PRESERVED ACROSS REQUESTS. WHEN A VISITOR ACCESSES YOUR SITE, PHP WILL CHECK AUTOMATICALLY (IF SESSION.AUTO_START IS SET TO 1) OR ON YOUR REQUEST (EXPLICITLY THROUGH START() OR IMPLICITLY THROUGH REGISTER()) WHETHER A SPECIFIC SESSION ID (SID) HAS BEEN SENT WITH THE REQUEST. IF THIS IS THE CASE, THE PRIOR SAVED ENVIRONMENT IS RECREATED."

Simple-Session extends PHP's standard session handling making it more usable.

Source Location:

Class:	core/net/http/session/Bs_SimpleSession.class.php
Example:	core/net/http/session/examples/simpleSession.php

Typical Use Cases:

- You are programming object oriented and also think it's very 'dirty' to have to define the session variables *global*. (See below "Reason for Writing this Class")
- It's possible to have more then one session at a time from the same user using only **one** session ID (we call this **Parallel Sessions**).
- Only the session vars are loaded that are used. (See "Futures".)

Features :

- Object vars can be passed directly and do **not have to be global**.
- Is only based on PHP's built in sessions handling, no additional SID is used. So we benefit from all features that PHP sessions handling uses to pass the SID (like url_rewriter.tags)
- Automatically stores session data to disk on PHP exit.
- Auto-change detection. If session-data is left unchanged, then no data is written to disk.
- Supports garbage collecting.
- Supports 'maxLifeTime' and 'maxStandbyTime'
- You may have **Parallel Sessions** .

What are Parallel Sessions?
The short answer is: “To have multiple session (from the same user) using *__one__* SID”. Let me give you some examples in where you would like to have paralle l sessions:

1) You have 2 (or more) independent login pages on the same site and want to keep a session for each. Also you must guarantee that the same session vars are not overwritten by accident.

2) You would like to have a different timeout for some session-vars. E.g. Some pages in a portal could be very conservative and would like to drop data while others would like to keep it until the browser is closed.

3) You have to store a high volume of session data, but most of the data is **NOT** needed in every session. Using a normal PHP-session would load *all* session-vars collected so far (if used or not). If the session data is complex (like serialized objects) this will result in a high overhead.

Todo:

- 2do Protection against Session hijacking

The Mail Methods of Bs_SimpleSession are:

```
bool register (string $key, string &$value, [string $sessName = 'default'])
bool isRegistered (string $key, [string $sessName = 'default'])
bool unRegister (string $key, [string $sessName = 'default'])
bool destroy ([string $sessName = 'default'])
```

The last paramter $sessName is optional and is used to prevent large chunks of data from being loaded or for **Parallel Sessions** . Ignore it for now.

Behaviour of register():

1. If the passed key is ***not*** registered, the passed value is taken as default.
2. If the passed key is registered, the session value will overwrite the passed value (this is possible because it's passed by reference)
3. From the moment you call register, that value is 'watched' (by holding a pointer to it). That means that any change to that value will be stored at the request end and available at the next request.

NOTE: The above behaviour leads to vary simple code for initialising and registering a var without the need for a “fetchRegisteredVar”-function. Following is fully sufficient:

```
$aVariable=0;
$session->register('my_var', $aVariable);
```

Don’t get irritated by the fact that `$aVariable` in initialised to 0 and then registered. This has only effect if the var wasn’t registered before otherwise `$aVariable` gets overwritten by the registered value.

Following is a fully functional sample of a counter that will add +1 to the session-var `$aVariable` on every reload and display it's value.

Example 1:

```php
<?php
  # include the configuration file.
  require_once($_SERVER['DOCUMENT_ROOT'] . '../global.conf.php');
  require_once($APP['path']['core'] . 'net/http/session/Bs_SimpleSession.class.php');

  ###########################################################################
  # Sample 1
  # --------
  # Simple Counter.
  $session =& $GLOBALS['Bs_SimpleSession']; // Use the reference operator '=&'

  $aVariable=0;
  $ok = $session->register('my_var', $aVariable);
  if (!$ok) echo $session->getLastError();
  echo "<hr>Sample 1 - Simple Counter: <B>" . $aVariable++ . "</B><hr>";
?>
```

Following sample demonstrates the use of **Parallel Sessions.** In this case we use it as optimisation in that we only load the `$bigDataRec` when needed (and not on every request).

Example 2:

```php
<?php
# include the configuration file.
require_once($_SERVER['DOCUMENT_ROOT'] . '../global.conf.php');
require_once($APP['path']['core'] . 'net/http/session/Bs_SimpleSession.class.php');

###########################################################################
# Sample 2
# --------
# TIP: If you have a lot of session data that is not needed on every request,
# then you may want to use following technique (called parallel sessions)
# In this manner the $bigDataRec will only be loaded when needed and not on
# every request.

$smallDataRec  = NULL;   // Data used in every session
$bigDataRec    = NULL;   // Data not used in every session
$session->register('my_sessionData', $smallDataRec);
if (@$needThatBigData) {
  $session->register('my_bigData', $bigDataRec, 'myBigDataSession');
  // do something with the $bigDataRec
}
// do something with the $smallData
?>
```

Reason for Writing this Class:

- In PHP you can only register variables that are global with PHP's session handling :-(. This is very unsatisfying when programming OO. So if I have a var $this->foo in an object that you would like to register with 'session_register()', you'd have to somehow globalize it. *Yack!*
  This Class makes it much easier with a method like $session->register($key, $value).

This class is called simple-session because:

a) One main feature is the practical approach when working with PHP-objects. Session vars don't have to be global (read below for more).
b) Based on PHP's Session handling functions. This means that all the session features PHP are inherited like the use of cookies or

Because this object is based on the built-in session handling of PHP, all PHP session properties set in php.ini also have an influence on this object. The most important ones are (Note: the defaults settings are usually OK):

session.auto_start:	Initialize session on request start-up.
session.use_cookies:	specifies whether the module will use cookies to store the session id on the client side. Defaults to 1 (enabled).
session.use_only_cookies:	specifies whether the module will only use cookies to store the session id on the client side.
session.cookie_lifetime:	(default 0 == "until restart of browser"). If a time >0 is given, all session data will be lost after that time is over. You can shorten this time in this class but NOT longer it.
session.gc_maxlifetime:	(default 1440 == 24min) defines how long after the last access the PHP-session should be considered as garbage. You can shorten this time in this class but NOT longer it.
variables order:	(default "EGPCS") Describes the order in which PHP registers GET, POST and Cookie vars and therefore the order how PHP looks for the SID.

For more info on PHP-session handling see http://www.zend.com/zend/tut/session.php and the PHP manual.

NOTES:
Check PHP-bug report: http://bugs.php.net/?id=14798 (PHP V4.1.0)
Sessions lifetime will not work with Win and FAT32!