JOURNAL OF ENGINEERING annals OF FACULTY OF ENGINEERING HUNEDOARA
TOME VI (year 2008), FASCICULE 3, (ISSN 1584 – 2673)

HOW INFORMATION PASSES BETWEEN ONE SITES PAGES

SAV Sorin Mihail

UNIVERSITY OF PETROŞANI, FACULTY OF SCIENCES,
DEPARTMENT OF MATHEMATIC AND INFORMATICS

Abstract
The paper will illustrate some methods used for passing information, variable values between one site's pages. The common method used in PHP is: Get method, Post method, Cookies, Sessions.
Keywords HTTP, PHP, GET, POST, Cookies, Session

1. THE STATELESS HTTP. WHAT IS PHP?

The Web works with the HTTP protocol which is stateless. That means that each HTTP request is independent of all the others, knows nothing substantive about the identity of the client, and has no memory.

Even if you design your site with very strict one-way navigation (Page 1 leads only to Page 2, which leads only to Page 3, and so on), the HTTP protocol will never know or care that someone browsing Page 2 must have come from Page 1. You cannot set the value of a variable on Page 1 and expect it to be imported to Page 2 by the exigencies of HTML itself. You can use HTML to display a form, and someone can enter some information using it - but unless you employ some extra means to pass the information to another page or program, the variable will simply vanish into the ether as soon as you move to another page.

This is where a form-handling technology like PHP comes in. PHP stands for PHP: Hypertext Preprocessor. The product was originally named Personal Home Page Tools, and many people still think that's what the acronym stands for. But as it expanded in scope, a new and more appropriate name was selected by community vote. PHP is currently in its fifth major rewrite, called PHP5 or just plain PHP.

PHP is a server-side scripting language, which can be embedded in HTML or used as a standalone binary. Proprietary products in this niche are Microsoft's Active Server Pages, Macromedia's ColdFusion, and Sun's Java Server Pages. Some tech journalists used to call PHP "the open source ASP" because its functionality is similar.

There are so many reasons for us to use PHP alone and in combination with MySQL such as:

- ✓ Cost - PHP costs us nothing.
- ✓ The PHP license - the freeness of open source and free software is guaranteed by a gaggle of licensing schemes, most famously the GPL (Gnu General Public License). PHP used to be released under both the GPL and its own license, with each user free to choose between them.
- ✓ Easy to use - PHP is easy to learn, compared to the other ways to achieve similar functionality. Unlike Java Server Pages or C-based CGI, PHP doesn't require you to gain a deep understanding of a major programming language before you can make a trivial database or remote-server call.

Web developers it has come to connote a graphical, drag-and-drop, What You See Is What You Get development environment. To become truly proficient at PHP, you need to be

comfortable editing HTML by hand. You can use WYSIWYG editors to design sites, format pages, and insert client-side features before you add PHP functionality to the source code.

That said, let us reiterate that PHP really is easy to learn and write, especially for those with a little bit of experience in a C-syntaxed programming language. It's just a little more involved than HTML but probably simpler than JavaScript and definitely less conceptually complex than JSP or ASP.NET.

Open source software: don't fear the cheaper - But as the bard so pithily observed, we are living in a material world—where we've internalized maxims such as, "You get what you pay for," "There's no such thing as a free lunch," and "Things that sound too good to be true usually are." You may, therefore, have some lingering doubts about the quality and viability of no-cost software. It probably doesn't help that until recently software that didn't cost money—formerly called freeware, shareware, or free software—was generally thought to fall into one of three categories:

- ✓ Programs filling small, uncommercial niches
- ✓ Programs performing grungy, low-level jobs
- ✓ Programs for people with bizarre socio-political issues

It's time to update some stereotypes once and for all. We are clearly in the middle of a sea change in the business of software. Much (if not most) major consumer software is distributed without cost today; e-mail clients, Web browsers, games, and even full-service office suites are all being given away as fast as their makers can whip up Web versions or set up FTP servers. Consumer software is increasingly seen as a loss-leader, the flower that attracts the pollinating honeybee—in other words, a way to sell more server hardware, operating systems, connectivity, advertising, optional widgets, or stock shares. The full retail price of a piece of software, therefore, is no longer a reliable gauge of its quality or the eccentricity-level of its user.

On the server side, open source products have come on even stronger. Not only do they compete with the best commercial stuff; in many cases there's a feeling that they far exceed the competition. Don't take our word for it! Ask IBM, any hardware manufacturer, NASA, Amazon.com, the Queen of England.

HTML forms are mostly useful for passing a few values from a given page to one single other page of a Web site. There are more persistent ways to maintain state over many page views, such as cookies and sessions. The most basic techniques of information-passing between Web pages, which utilize the GET and POST methods in HTTP to create dynamically generated pages and to handle form data.

2. THE „GET" METHOD

The GET method passes arguments from one page to the next as part of the Uniform Resource Indicator query string. When used for form handling, GET appends the indicated variable name(s) and value(s) to the URL designated in the ACTION attribute with a question mark separator and submits the whole thing to the processing agent (in this case a Web server).

This is an example HTML form using the GET method (save the file under the name adauga.html):

```
<HTML>
<HEAD>
<TITLE>One small GET method example</TITLE>
</HEAD>
<BODY>
<FORM ACTION="http://localhost/adaugare.php" METHOD="GET">
<P>Selectati grupa:<BR>
<SELECT NAME="Grupa" SIZE="1">
 <OPTION VALUE="Inf1">Inf1</OPTION>
 <OPTION VALUE="Inf2">Inf2</OPTION>
 <OPTION VALUE="Inf3">Inf3</OPTION>
 <OPTION VALUE="Inf4">Inf4</OPTION>
</SELECT>
<P><INPUT TYPE="Submit" NAME="Select" VALUE="Selectare"></P>
```

```
</FORM>
</BODY>
</HTML>
```

When the user makes a selection and clicks the Selectare button, the browser agglutinates these elements in this order, with no spaces between the elements:

The URL in quotes after the word ACTION (http://localhost/adaugare.php)

A question mark (?) denoting that the following characters constitute a GET string.

A variable NAME, an equal sign, and the matching VALUE (Grupa=Inf1)

An ampersand (&) and the next NAME-VALUE pair (Submit=Select); further name-value pairs separated by ampersands can be added as many times as the server query string - length limit allows.

The browser thus constructs the URL string:

http://localhost/adaugare.php?Grupa=Inf1&Submit=Select

It then forwards this URL into its own address space as a new request. The PHP script to which the preceding form is submitted (adaugare.php) will grab the GET variables from the end of the request string, stuff them into the $_GET superglobal array, and do something useful with them-in this case, plug one of two values into a text string.

Strictly speaking, the name-value pairs in a GET query are not part of the HTTP or addressing standards.

The following code sample shows the PHP form handler for the preceding HTML form:

```
<HTML>
<HEAD>
<TITLE>Exemplu de transmitere al informatiilor folosind metoda GET</TITLE>
<STYLE TYPE="text/css"></STYLE>
</HEAD>
<BODY>
<P>Ati selectat grupa,<?php echo $_GET['Grupa']; ?>!</P>
</BODY>
</HTML>
```

Note that the value inputted into the previous page's HTML form field named "Grupa" is now available in a PHP variable called *$_GET['Grupa']*. Finally, you should see a page that says *Ati selectat grupa, Inf3!* in big type.

At this point, it makes some sense to explain just how to access values submitted from page to page. Each method has an associated superglobal array, which can be distinguished from the other arrays by the underscore that begins its name. Each item submitted via the GET method is accessed in the handler via the *$_GET array*; each item submitted via the POST method is accessed in the handler via the *$_POST array*.

The GET method of form handling had to be reinstated, largely because of the bookmark ability factor. Despite that it's still implemented as the default choice for form handling in all browsers, GET now comes with a strong recommendation to deploy it in idempotent usages only—in other words, those that have no permanent side effects. Putting two and two together, the single most appropriate form-handling use of GET is the search box. Unless you have a compelling reason to use GET for non-search-box form handling, use POST instead.

3. THE „POST" METHOD

POST method is the preferred method of form submission today, particularly in non idempotent usages (those that will result in permanent changes), such as adding information to a database. The form data set is included in the body of the form when it is forwarded to the processing agent (in this case, PHP). No visible change to the URL will result according to the different data submitted.

The POST method has these advantages:

It is more secure than GET because user-entered information is never visible in the URL query string, in the server logs, or (if precautions, such as always using the password HTML input type for passwords, are taken) onscreen.

There is a much larger limit on the amount of data that can be passed (a couple of kilobytes rather than a couple of hundred characters).

POST has these disadvantages:

The results at a given moment cannot be bookmarked.

The results should be expired by the browser, so that an error will result if the user employs the Back button to revisit the page.

Did you know that with PHP you can use both GET and POST variables on the same page? You might want to do this for a dynamically generated form, for example.

But what if we use the same variable name in both the GET and the POST variable sets? PHP keeps all ENVIRONMENT, GET, POST, COOKIE, and SERVER variables in the $GLOBALS array if we have set the register_globals configuration directive to "on" in your php.ini file. If there is a conflict, it is resolved by overwriting the variable values in the order you set, using the variables_order option in php.ini. Later trumps earlier, so if you use the default "EGPCS" value, cookies will triumph over POSTs that will themselves obliterate Gets. You can control the order of overwriting by simply changing the order of the letters on the appropriate line of this file, or even better, turning register_globals off and using the new PHP superglobal arrays instead. See the section on superglobals later in this chapter.

4. SESSIONS.COOKIES

Sessions and cookies are closely allied concepts in PHP and in Web programming more generally, largely because the best way to actually implement sessions is by using cookies. Sessions are a higher level concept than cookies.

What is a session? Informally, a session of Web browsing is a period of time during which a particular person, while sitting at a particular machine, views a number of different Web pages in his or her browser program and then it quits. If you run a Web site that this person visits during that time, for your purposes the session runs from that person's first download of a page from your site through the last.

If our web site's only mission is to offer various pages to various users, we may, in fact, not care at all where sessions begin and end. On the other hand, there are a number of reasons why we might in fact care. For example:

We want to customize our users' experiences as they move through the site, in a way that depends on which (or how many) pages they have already seen.

We want to display advertisements to the user, but we do not want to display a given add more than once per session.

We want the session to accumulate information about users' actions as they progress - as in an adventure game's tracking of points and weapons accumulated or an e-commerce site's shopping cart.

We are interested in tracking how people navigate through our site in general - when they visit that interior page, is it because they bookmarked it, or did they get there all the way from the front page?

For all of these purposes, we need to be able to match up page requests with the sessions they are part of, and for some purposes it would be nice to store some information in association with the session as it progresses. PHP sessions solve both of these problems for us.

Every HTTP request is dealt with independently, but each time your user moves from page to page within your site, it is usually via either a link or a form submission. If the very first page a user visits can somehow generate a unique label for that visit, every subsequent "handoff" of one page to another can pass that unique identifier along.

For example, here is a hypothetical code fragment that you might include near the top of every page on your site:

```
if (!IsSet($my_s_id))
$my_session_id = generate_session_id();
```

This fragment checks to see if the *$my_s_id* variable is bound - if it is, we assume that it has been passed in, and we are in the middle of a session. If it is not, we assume that we are the first page of a new session, and we call a hypothetical function called generate_session_id() to create a unique identifier.

After we have included the preceding code, we assume that we have a unique identifier for the session, and our only remaining responsibility is to pass it along to any page

we link or submit to. Every link from our page should include the $my_s_id as a GET argument, as in:

```
<A HREF="tabel.php
 ? my_s_id=<?php echo $my_s_id;?>"
 >Next</A>
```

And every form submission should have a hidden POST argument embedded in it, like this:

```
<FORM ACTION=tabel.php METHOD=POST>
body of form
<INPUT TYPE=HIDDEN NAME=my_s_id
VALUE="<?php echo $my_s_id;?>" >
</FORM>
```

What's wrong with this technique? Nothing. It works just fine as a way to keep different sessions straight (as long as you can generate unique identifiers). And once we have unique labels for the sessions, we can use a variety of techniques to associate other kinds of information with each session, such as using the session ID as a key for database storage.

If you want PHP to transparently handle passing session variables for you when cookies are not available, you need to have configured PHP with both the --enable-trans-sid and --enable-track-vars options. If PHP is not handling this for you, you must arrange to pass a GET or POST argument, of the form session_name=session_id, in all your links and forms. When a session is active, PHP provides a special constant, SID, which contains the right argument/value pair. Following is an example of including this constant in a link:

```
<A HREF="pag_urm.php?<?php echo(SID);?>">Pagina urmatoare </A>
```

The first step in a script that uses the session feature is to let PHP know that a session may already be in progress so that it can hook up to it and recover any associated information.

This is done by calling the function session_start(), which takes no arguments. (If you want every script invocation to look for a session without having to call this function, set the variable session.auto_start to 1 in your php.ini file, rather than the usual default of 0.)

Also, any call to session_register() causes an implicit initial call to session_start().

The effect of session_start() depends on whether PHP can locate a previous session identifier, as supplied either by HTTP arguments or in a cookie. If one is found, the values of any previously registered session variables are recovered. If one is not found, then PHP assumes that we are in the first page of a new session, and generates a new session ID.

Changes in PHP's treatment of global and external variables starting with version 4.1 have made certain things more inconvenient. In our view, though, these changes will also remove a lot of potential confusion about sessions. Accordingly, we'll list two approaches to propagating variables in sessions: one, which is simple and works in PHP version 4.1 or later, and another which is more complicated and works only in PHP version 4.1 or earlier.

The simple approach is that you've made a call to session_start() (as early in your script as possible), use the $_SESSION superglobal array as your suitcase for storing anything that you want to retrieve again from a later page in the same session. Assume that any other variables will be left behind when you leave this page and that everything in that suitcase will be there when you arrive at the next page. So, session code to propagate a single numerical variable can be as simple as this:

```
<?php
  session_start();
  $temp = 45;
  $salvare = 19;
  $temp1 = 33;
  $_SESSION['salveaza'] = $salvare;
?>
```

The receiving code can be as simple as the following example:

```
<?php
 session_start();
 $salvat_anterior = $_SESSION['salveaza'];
  [..]
  $temp = 45;
```

```
$temp1 = 33;
[..]
?>
```

A cookie is a small piece of information that is retained on the client machine, either in the browser's application memory or as a small file written to the user's hard disk. It contains a name/value pair-setting a cookie means associating a value with a name and storing that pairing on the client side.

Getting or reading a cookie means using the name to retrieve the value.

In PHP, cookies are set using the setcookie() function, and cookies are read nearly automatically. In PHP4.1 and later, names and values of cookie variables show up in the superglobal array $_COOKIES, with the cookie name as an index, and the value as the value it indexes.

Cookies have always been controversial from a privacy point of view, and that controversy heats up again periodically. The worry was that, after a consumer reveals his or her identity on a site by filling out a form and accepting a cookie, any other site that compares notes with the original site could conceivably know the true identity of the user (and lots of other information as well). If this practice became widespread, every e-commerce site you visit might be able to figure out not only your name, address, and buying habits, but also a list of other pages you have visited on the web. So, cookies worry some people, but at the same time they are also a reasonable and benign workaround to the statelessness of the HTTP protocol. There are plenty of good reasons to want a web client/server interaction to coherently span a few page requests in a row, rather than covering just a single request. As a web developer, you might well decide to use cookies for such a purpose, comfortable in the knowledge that there is no substantive invasion of privacy occurring.

It is hard to do much wrong with cookies purely at the PHP level. After all, setting a cookie involves only one function (set_cookie()), and reading cookies involves no functions at all. What could be easier than that? The problems that typically arise are those imposed by the HTTP protocol itself.

5. CONCLUSIONS

Sessions are useful for tracking a user's behaviour over interactions that last longer than one script execution or page download. If what you present to the user depends on which previous pages he or she has seen or interacted with, your code must store or propagate that information in a way that distinguishes one user from another. Because the HTTP protocol is stateless, this inevitably entails some kind of workaround technique—usually either hidden variables (which impose maintenance headaches) or cookies (which are not universally supported by client browsers).

The PHP implementation of sessions encapsulates these messy issues and presents a clean layer of abstraction to the scripter. Unique session identifiers are automatically created and propagated, and a variables can be passed from page to page by storing them in the superglobal $_SESSION array. Aside from one's having to connect to a session initially and store (or register) the variables that should persist beyond the current page, session use is virtually transparent to the programmer.

BIBLIOGRAPHY

[1.] Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, s.a – PHP Manual
[2.] Tim Converse, Joyce Park - PHP 4 Bible. August 2000, PHP through version 4.0, Wiley Publishing, Inc.
[3.] Tim Converse, Joyce Park - PHP Bible, Second Edition, September 2002, Wiley Publishing, Inc.
[4.] * * * * * - http://www.php.net/docs.php
[5.] * * * * * - http://www.oriceon.com/category/tutorial
[6.] * * * * * - http://www.tutoriale.far-php.ro/index.php