Do-It-Yourself: Part XIV—A Virtual Shopping Cart Using PHP Session Variables

By Scott Courtney

This month, we'll see how to use session variables in PHP to create an efficient virtual "shopping basket" for the Web site. Let's begin by understanding three things: why we need session variables, how they work in PHP, and why this is a good approach to implementing a Web shopping basket.

As of last month's installment, Ice Floe Housing is up and running, with an online catalog Web site and a user-friendly form to maintain the database. Penny Penguin is well on her way to having a Web-savvy business! This month, we will see how to use session variables in PHP to create an efficient virtual "shopping basket" for the Ice Floe Housing Web site. Let us begin by understanding three things: why we need session variables, how they work in PHP and why this is a good approach to implementing a Web shopping basket.

HTTP

Most Web pages are fetched from a server at the request of the client browser using HyperText Transfer Protocol, or HTTP, the current version of which is 1.1. HTTP is a simple, text-oriented protocol that actually has a lot in common with the old Telnet protocol. This is typical of many Internet application protocols and offers the advantage of easy debugging using the standard Telnet client. A simple transaction in HTTP looks something like FIGURE 1.

FIGURE 1: A SIMPLE HTTP REQUEST, RESPONSE HEADERS AND RESPONSE

```
GET /catalog.php HTTP/1.1
Host: www.icefloehousing.com

HTTP/1.1 200 OK
Date: Fri, 04 Apr 2003 22:10:09 GMT
Server: Apache/1.3.26 (Unix) PHP/4.2.3
X-Powered-By: PHP/4.2.3
Set-Cookie: PHPSESSID=4417051a204a24e289848c1ad8b15a17; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Transfer-Encoding: chunked
Content-Type: text/html

d4e
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta http-equiv="description" content="Ice Floe Housing offers residential
structures for cold climates.">
<meta http-equiv="keywords" content="igloos, Ice Floe Housing, ice, homes">
<title>Ice Floe Housing, Inc.</title>
</head>
<html><head><title>Ice Floe Housing, Inc.</title>

(rest of response removed for brevity)
```

The first two lines, the ones beginning with "GET" and "Host:," are the request sent from the browser to the server. The rest of the listing is the data coming back from the server. Everything from "HTTP/1.1 200 OK" (the status header) through the "Content-Type:" line comprises the response headers. The blank line after the headers is the delimiter between headers and actual content (only the first few lines of the content are shown here). "GET" identifies this as a "GET" request method (as opposed to the "POST" method used in last month's CGI forms) and specifies the desired absolute local path. Note the absence of "http://" and the hostname, because this is a document path, not a full URL.

The "Host:" header indicates to the server which Web server is being contacted. Modern Web servers often have multiple sites on a single physical computer, so "www.domain1.com" and "www.sample.com" might both connect to the same IP address. IP connections are

made to a numeric address, not to a hostname, so the "Host:" line is needed to allow the Web server software (in this case, Apache) to know which of the virtual servers is desired. The HTTP protocol has other headers that can be specified in the request and in the response, specifying the preferred language and character sets, the revision dates of documents and other "metadata" (that is, information about the document, which is not actually part of the document).

Regardless of the specifics, however, there is one major problem with using HTTP for transactional applications, such as a virtual storefront. HTTP treats each transaction completely independently from every other. That is, each time the user selects a new URL, or reloads the previous URL, the Web server sees a completely new connection. With HTTP 1.1, it is possible to fetch multiple documents (such as an HTML page and associated graphics) with a single socket connection, but this doesn't change the fundamentals of the process, because the connection is dropped after a short time (typically just a couple of seconds) of inactivity, or when the browser has all the files it needs to render the current page for the user.

So, if the user adds an item to a virtual shopping basket, then goes back to the catalog, the server would have no way to "remember" that item. Even storing the entry in a database will not help, because the browser's next request to the server has no association to its previous request. Does the server remember the IP address of the client? Nope! Some clients, such as Linux or Unix, are multiuser, and even on Windows, the same person may have more than one browser running at a time. From a Web server's perspective, behind a firewall, many computers can seem to have the same IP address due to masquerading.

One solution to this thorny problem is to use a pseudo-random identity number, generated each time a particular instance of the browser first connects to a particular Web server. This "session ID" is passed back and forth between the browser and the server with each transaction, establishing persistent identity for the browser for as long as neither it nor the server is restarted. This does not uniquely identify the human being who is using the browser, so it is still possible (and quite common) to browse a Web site anonymously (that is, without being logged in with a personal account) and still have an active session.

FIGURE 2: INITBASKET() MAKES SURE THE SHOPPING BASKET DATA ARRAY EXISTS, WHILE CLEARBASKET() EMPTIES IT.

```
function initBasket() {
        if (! is_array($_SESSION["BASKET"])) {
                clearBasket();
        }
}

function clearBasket() {
        $_SESSION["BASKET"] = array();
}
```

The HTTP protocol has no intrinsic support for sessions, but the session ID can be transferred in two ways. One is to encode it within every URL link on the site, as a GET parameter. This works, but makes the URLs really long and ugly (www.sample.com/catalog.php?session=24f7ca5f6ff1a5afb9032 for example). It also means that URLs saved in a bookmark (or favorites) list will contain a session identifier that will not be valid on the next visit to the site.

A better approach is the use of a "cookie," which is simply a very small piece of data stored by the server on the client's disk drive and automatically sent by the browser with each future request header in the same domain where the cookie was generated. There is a cookie sent with the response headers in the previous example. Look for the line with "Set-Cookie: PHPSESSID=". If the browser accepts the cookie, all subsequent requests during the current session will contain that session ID value. PHP is smart enough to automatically rewrite all of the URLs on the site, before sending the data to the browser, if the browser happens not to accept this cookie.

It is important to understand that the session ID is the only data sent back and forth with the cookie. PHP allows server-side programs to store large amounts of data in a built-in array called $_SESSION[], but the array itself is not sent back and forth. This has two advantages. First, it is impossible for the browser to directly read or write the data in this array; only the PHP scripts on the server can do that. Session variables are therefore secure, if the PHP code is written sensibly. Second, sending only the session ID saves a great deal of network bandwidth on each HTTP transaction.

Therefore, session variables are a great place to store the temporary data for an online shopping basket! Another advantage is that, if the user decides to go elsewhere on the Web, shuts down the browser, or loses the Internet connection altogether, the session will automatically expire and be deleted after a period of time. If the temporary data went into MySQL tables, there would need to be code in the application to manage that process.

CREATING THE SHOPPING BASKET

Penny Penguin can easily create her shopping basket, with PHP's session support doing most of the hard work. The support functions are in /home/penny/icefloehousing.org/include/basket.inc and the actual shopping basket page is in /home/penny/icefloehousing.org/include/basket.php (don't be confused by the similar filenames). Penny also has to edit the header.inc file so that it refers to basket.inc in a new include() statement. Also added to header.inc is the following very simple line:

```
session_start();
```

This line informs PHP that we intend to use the session features and that it should attempt to send the appropriate cookie to the browser or modify linked URLs as needed. It is possible to configure PHP to have this happen automatically with every page, but it does not hurt to add this line to your code just to be safe.

The heavy lifting for the shopping basket is done in basket.inc. The initBasket() and clearBasket() functions ensure that a subarray, with the name "BASKET," exists within the automatically defined $_SESSION[] array. Note that $_SESSION[], like PHP's other built-in arrays, is automatically declared global to all functions. clearBasket() can be called separately to empty the shopping basket. See FIGURE 2 (on page 31).

The setBasket() function (see FIGURE 3) modifies the quantity of an item in the shopping basket array. Each element of the basket array has a catalog number (cat_number column

from the database) as a key, and the current order quantity as the value. setBasket() can either add a new item, change the quantity of an existing item, or delete an item. The function deletes the entry for any item whose quantity is being set to zero.

These are simple functions. Things get more interesting with getBasketCatalogData(). See FIGURE 4. This function looks at the list of catalog numbers in the current shopping basket and builds an SQL query that will obtain detailed catalog information for each of these items. The statement is of the form "SELECT …. FROM …. WHERE cat_number IN (…)," and all of the applicable catalog numbers are placed within that "IN" clause. This is an efficient way to select a very small number of records with no common feature from a table containing thousands of rows. Note that the function is rather selective in what it returns—the "summary" and "detail" columns from the catalog table are omitted, because we do not need to show that information on the shopping basket page. In addition, this function's name begins with an ampersand (&) to indicate that it returns an array by reference rather than by value. This improves performance by eliminating a memory-to-memory copy.

The catalog data is returned from getBasketCatalogData() as an associative array, with the catalog number as the key. The function setQuantities() merges the quantity data from the session array $_SESSION["BASKET"] into the catalog detail array, so that later code will have a unified array with all of the needed information. See FIGURE 5.

Penny needs two more support functions, not shown here, to copy quantities between the shopping cart form (which allows the user to easily change how many of each item they want) and the session variable containing the actual shopping cart array. Similarly to what was done in last month's catalog editing form, the field names are mangled by a string replacement function so that catalog numbers with dashes don't produce invalid HTML code. The two functions, getFormQuantity() and setSessionQuantities(), are simple and are available in the download file for this article.

The most interesting function is getShoppingBasketListHTML(). It takes the detail array, containing all the catalog data plus the quantities from the shopping basket, and turns it into HTML. This HTML is mostly just display fields in a table, except that the quantities are actually text input fields so that the user can change them. As each line of HTML (a table row per line) is generated, the function examines the order quantity to see if the user could save money by ordering all or part of that quantity in lots rather than as individual units. These savings are reflected in the unit and total costs and are pointed out to the shopper so that they know Penny is giving them a good deal on their purchase! See FIGURE 6 (on page 33).

FIGURE 3: SETBASKET() PROVIDES AN EASY WAY TO MANAGE ENTRIES IN THE SHOPPING BASKET ARRAY.

```
function setBasket($cat_num,$qty) {
        # Be sure the basket array exists
        initBasket();
        if ($qty == 0) {
                unset($_SESSION["BASKET"][$cat_num]);
        } else {
                $_SESSION["BASKET"][$cat_num] = $qty;
        }
}
```

FIGURE 4: GETBASKETCATALOGDATA() RETRIEVES THE DETAILED CATALOG INFORMATION, MINUS THE LONG TEXT FIELDS, FOR THE ITEMS IN THE SHOPPING BASKET.

```
function &getBasketCatalogData($db) {
        initBasket();
        $cat_num_list = array_keys($_SESSION["BASKET"]);
        $detail = array();
        if (count($cat_num_list)) {
                $in_list = implode(',',$cat_num_list);
                $in_list = "('" . ereg_replace(",","','",$in_list) . "')";
                $sql1 = "select catalog.cat_number, catalog.name name,
catalog.price,";
                $sql1 .= " catalog.lot_size, catalog.lot_unit_price,";
                $sql1 .= " manufacturers.name mfg, product_types.name prod_type";
                $sql1 .= " from catalog left join manufacturers on
catalog.mfg_id=manufacturers.id";
                $sql1 .= " left join product_types on
catalog.product_type=product_types.id";
                $sql1 .= " having cat_number in " . $in_list;
                $sql1 .= " order by cat_number";
                $result = $db->getAll($sql1,DB_FETCHMODE_ASSOC);
                if (DB::isError($result)) {
                        print("<br>ERROR: Database query failure. SQL=" .
htmlspecialchars($sql1) . "<br>\n");
                        $detail = array();
                } else {
                        foreach($result as $row) {
                                $cat_num = $row["cat_number"];
                                $detail[$cat_num] = $row;
                        }
                }
        }
        return $detail;
}
```

FIGURE 5: SETQUANTITIES() MERGES THE SHOPPING CART INFORMATION INTO THE PRODUCT DETAIL ARRAY SO THAT OTHER CODE CAN USE THE COMBINED DATA MORE EASILY.

```
function setQuantities(&$detail_array) {
        initBasket();
        reset($_SESSION["BASKET"]);
        while (list($cat_num,$qty) = each($_SESSION["BASKET"])) {
                $detail_array[$cat_num]["qty"] = $qty;
        }
}
```

getShoppingBasketHTML() augments the results from getShoppingBasketListHTML() to create a complete HTML form with submit and reset buttons. Together, these two functions do almost all of the work for the basket.php page. See FIGURE 7.

Once all the library functions are done, only two tasks remain. First, the catalog.php page from last month has to be modified to add a very simple form to its item detail page. That form has a text input field for the desired quantity, a hidden field for the catalog number, and a submit button that, when clicked, takes the user to the shopping basket page and carries along the catalog number and quantity. The changes are trivial, and they are reflected in the new version of catalog.php that is included with this month's download file. See FIGURE 8. FIGURE 9 shows how the new catalog detail page, with the option to add items to the shopping basket, will look in the browser.

The second task is to add the code framework in basket.php. This code simply responds to the POST request, updating quantities as needed. For the sake of simplicity, we do not have a way for the user to actually place an order. To do that, all that is needed is to add a form with nothing but a submit-type button, taking the user to another script that would actually process the order. Because all of the data is in session variables, it is not necessary for the order data to be part of that form. The order processing script can actually use getBasketCatalogData() to query the database before placing the final order.

CONCLUSION

Penny has accomplished a lot, with relatively little code, by letting session variables do the hard part for the shopping basket. FIGURE 10 shows how the finished shopping basket page looks in the browser. While falling short of being a complete e-commerce or online shopping application, Penny's simple shopping basket demonstrates a number of useful techniques that can be applied to other applications.

As with the other articles in this series, a complete source code archive can be downloaded. The code archive contains some functions not discussed in the text, due to space constraints and has more detailed comments about the workings of many of the library functions. The code is released as freeware under the GNU General Public

FIGURE 6: GETSHOPPINGBASKETLISTHTML() TURNS THE DETAIL DATA ARRAY INTO MULTIPLE HTML TABLE ROWS, INCLUDING AN EXTRA SUMMARY ROW AT THE BOTTOM OF THE TABLE.

```
function getShoppingBasketListHTML($detail_array) {
        reset($detail_array);
        $savings = 0;
        $total_cost = 0;
        $lot_items = array();
        $html = "<table border=\"1\" cellpadding=\"3\" cellspacing=\"2\">\n";
        $html .= "<tr align=\"center\" valign=\"bottom\">\n";
        $html .= "\t<th width=\"10%\">Cat.
Num.</th><th>Description</th><th>Quantity</th><th>Price Each</th><th>Final
Cost</th>\n";
        $html .= "</tr>\n";
        while (list($cat_num,$row) = each($detail_array)) {
                $field_name = "FORM__QTY__" . $cat_num;
                $field_name = ereg_replace('-','___',$field_name);
                $lot_size = $row["lot_size"];
                if ($lot_size) {
                        $lots = intval($row["qty"] / $lot_size);
                        $non_lot_qty = $row["qty"] % $lot_size;
                } else {
                        $lots = 0;
                        $non_lot_qty = $row["qty"];
                }
                $extended_cost = $row["qty"] * $row["price"];
                if ($lots) {
                        $lot_qty = $lots * $lot_size;
                        $lot_cost = $lot_qty * $row["lot_unit_price"];
                        $non_lot_cost = $non_lot_qty * $row["price"];
                        $final_cost = $lot_cost + $non_lot_cost;
                        $savings += ($extended_cost - $final_cost);
                } else {
                        $final_cost = $extended_cost;
                }
                $total_cost += $final_cost;
                $unit_cost = $final_cost / $row["qty"];
                $html .= "<tr valign=\"top\" align=\"left\">\n";
                $html .= "\t<td>" . $row["cat_number"] . "</td>\n";
                $html .= "\t<td>" . htmlspecialchars($row["name"]);
                if ($lots) {
                        $html .= "<br><small><i>";
                        $html .= "Savings of $" .
sprintf("%9.2f",$extended_cost-$final_cost);
                        $html .= " on this item by grouping into lots of " .
$lot_size . ".";
                        $html .= "</i></small>";
                }
                $html .= "</td>\n";
                $html .= "\t<td align=\"right\">";
                $html .= getTextFieldHTML($field_name,$row["qty"],5,5);
                $html .= "</td>\n";
                $html .= "\t<td nowrap>";
                $price_html = '$' . sprintf("%8.2f",$unit_cost);
                $html .= ereg_replace(' ',' ',$price_html) . "</td>\n";
                $html .= "\t<td nowrap>";
                $price_html = '$' . sprintf("%10.2f",$final_cost);
                $html .= ereg_replace(' ',' ',$price_html) . "</td>\n";
                $html .= "</tr>\n";
        }
        # Now display some summary information at the bottom.
        $html .= "<tr align=\"left\" valign=\"middle\">\n";
        $html .= "\t<td colspan=\"3\">";
        $html .= "Change quantities as desired above, then press <b>Update</b> to
recalculate pricing.";
        $html .= " Enter a zero (0) quantity to delete any item.";
        if ($savings > 0) {
                $html .= "<p>Your order saves a total of $" .
sprintf("%10.2f",$savings);
                $html .= " by grouping items into lot quantities. Unit prices
shown above";
                $html .= " are adjusted downward to reflect the discount.";
        }
        $html .= "</td>\n";
        $html .= "\t<td align=\"right\"><b>TOTAL COST</b></td>\n";
        $html .= "\t<td nowrap align=\"right\"><b>$"
. sprintf("%11.2f",$total_cost) . "</b></td>\n";
        $html .= "</table>\n";
        return $html;
}
```

License (http://www.gnu.org/licenses/gpl.html). Have fun!

NaSPA member Scott Courtney is a senior engineer with Sine Nomine Associates, an engineering consulting company. His career has included fifteen years in engineering and IT at a large manufacturing company. He also worked as a technical journalist and editor for an online publisher for one year. Scott is an active open source developer in both PHP and Java languages and maintains a number of production Web sites using open source tools.

FIGURE 7: GETSHOPPINGBASKETHTML() FINISHES THE PROCESS OF CREATING THE SHOPPING BASKET DISPLAY AND EDIT FORM.

```
function getShoppingBasketHTML($detail_array) {
        $html = getFormTag();
        $html .= getShoppingBasketListHTML($detail_array);
        $html .= "<br clear=\"all\">\n";
        $html .= getFormTableTag();
        $html .= "<tr align=\"center\" valign=\"middle\">";
        $html .= "<td>" . getButtonHTML("SUBMIT","UPDATE","Update Quantities") .
"</td>\n";
        $html .= "<td>" . getButtonHTML("RESET","RESET","Reset") . "</td>\n";
        $html .= "</tr></table>\n";
        $html .= "</form>\n";
        $html .= getFormTag("/catalog.php","GET");
        $html .= getFormTableTag();
        $html .= "<tr align=\"center\" valign=\"middle\">";
        $html .= "<td><input type=\"SUBMIT\" value=\"Continue
Shopping\"</td>\n";
        $html .= "</tr></table>\n";
        $html .= "</form>\n";
        return $html;
}
```

FIGURE 8: SINCE MOST OF THE WORK IS DONE BY THE LIBRARY FUNCTIONS AND THE SESSION VARIABLE FEATURES OF PHP ITSELF, THE BASKET.PHP PAGE SCRIPT IS AMAZINGLY SHORT AND SIMPLE.

```
<?php
include($_SERVER["DOCUMENT_ROOT"] . "/../include/header.inc");

$db =& DB::connect("mysql://icefloe:tux@localhost/icefloe");
if (!is_object($db)) {
        die("<b>ERROR</b> Failed to create database object.\n");
}

initBasket();

# See if we are adding an item to the cart.
$add_cat_num = getSimpleFormValue("cat_number");
$add_qty = abs(intval(getSimpleFormValue("qty")));
if ($add_qty && !empty($add_cat_num)) {
        if (isset($_SESSION["BASKET"][$add_cat_num])) {
                $_SESSION["BASKET"][$add_cat_num] += $add_qty;
        } else {
                $_SESSION["BASKET"][$add_cat_num] = $add_qty;
        }
}

# Handle the update function, if any were posted.
setSessionQuantities();

if (!empty($_GET["cat_num"])) {
        setBasket($_GET["cat_num"],$_GET["qty"]);
}

$detail = getBasketCatalogData($db);
setQuantities($detail);


print(getShoppingBasketHTML($detail));

include($_SERVER["DOCUMENT_ROOT"] . "/../include/footer.inc");
?>
```

FIGURE 9: THE CATALOG ITEM DETAIL DISPLAY NOW CONTAINS A SIMPLE FORM TO ALLOW THE USER TO ADD THIS ITEM TO THE SHOPPING CART.

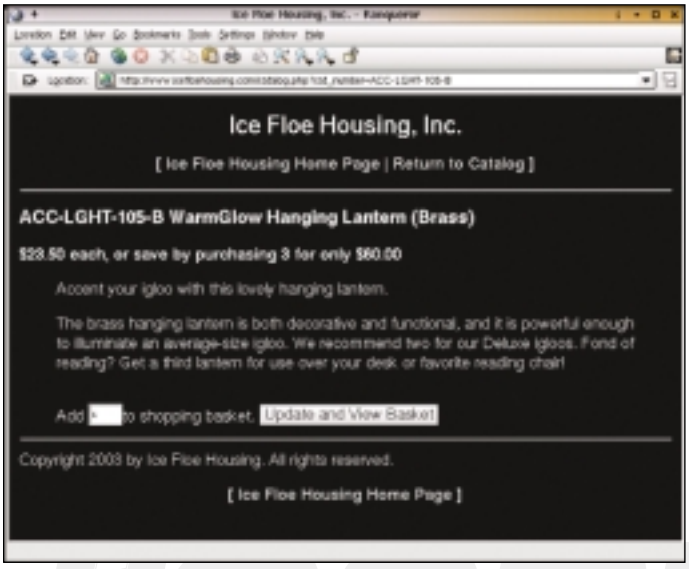

FIGURE 10: THE BASKET.PHP PAGE LETS THE USER REVIEW THE CONTENTS OF THEIR BASKET AND CHANGE QUANTITIES AS DESIRED. IT ALSO SHOWS DISCOUNTS FROM BUYING IN LOT QUANTITIES.