Linux Do-It-Yourself: Part XIII—Processing Web Forms with PHP and MySQL

By Scott Courtney

This month, we'll learn how to accept data entry and selections from an end user HTML form, process the form in PHP, and return the results to the browser as a pure HTML Web page that is compatible with any browser.

PENNY Penguin accomplished a lot last month, and now has an online catalog of the products offered by Ice Floe Housing, Inc. She still has a lot of work to do, and this month's project will give her a way to update her catalog online without using cryptic SQL database commands. The techniques in this article are applicable to a wide variety of database tables. We'll start with the basic site as it stood at the end of the last article, and will be adding quite a bit of code. The full PHP code is available for you to download and use, but not every line of code will be discussed in this article due to space constraints.

Recall that Ice Floe Housing's database has three tables so far: catalog, manufacturers and product_types. In many web applications, the forms that allow editing of databases have a separate page for each database table. This makes sense, because database tables typically contain one type of information each, and so this maps well to a separate type of form. Penny will follow that same logic for Ice Floe Housing's site. The catalog table is the most complex, because two of its fields (product_type and mfg_id) are actually integer ID numbers that point to rows in the other two tables. Penny has decided to tackle the catalog table first because once she has PHP code to manage this table, it will be easy to strip down and simplify that code to handle each of the other two tables. Going the other way, from code of a simple design toward a more complex design, is a much trickier project.

In the web environment, forms are processed using a protocol called Common Gateway Interface, or CGI. CGI works very much like an old-fashioned "green screen" terminal. The server presents a form for data entry or data editing. The user types the new data and tells the browser to submit the form back to the server. The server then processes the updates and informs the user of the results. It's really a very old way of doing things, but it's still commonplace after all these years because it works well, and because it minimizes the load on the server when many users are working simultaneously.

In CGI, there are two methods for the browser to make a request of the server. A "GET" request is used when the browser is retrieving information from the server, while "POST" is used when the browser is sending data to the server. It is possible to do send data with GET or to retrieve it with POST, but this is generally inadvisable. One of the ground rules for GET is that any GET request must be "idempotent." That means that a GET request may be repeated unlimited times without ill effect. If a GET request inserts or deletes rows from a database, attempting to repeat it (by refreshing the browser) may have unfortunate consequences, such as a customer order being placed multiple times.

Penny is smart enough to know that following the standards for GET and POST will save her a lot of trouble in the long run. So, her database editing forms will use the GET method only for initially selecting a record to edit, and will use POST for actually making database changes.

PHP, being a language specifically designed for Web programming, provides two predefined variables (each an array, actually) that handle CGI forms. These global variables are $_GET[] and $_POST[]. Each contains all of the data from the browser according to its respective CGI method.

How does the data go from browser to server? In the case of POST, it is always from an HTML form. For example, the following form would send data to the browser using the POST method:

```
<form action="/mypage.php" method="POST">
<p>What is your name?
<input type="text" size="30" name="whoareyou" value="Your name here">
<br><input type="submit" value="OK">
</form>
```

This form has the question "What is your name?" and a simple blank to enter the text in response. A button below the form, labeled "OK", sends the form to the browser. There are many different types of input fields, including text areas for long descriptions, drop-down boxes for selecting items from a predefined list, radio buttons, checkboxes, image icons and several kinds of buttons. Any good book on HTML will list these, or you can find the official reference at http://www.w3c.org/. The details of HTML design are beyond the scope of this article, so from here onward we will assume that you know the HTML tags or can look them up.

Now, what about using the GET method? That's just as easy as POST. If you change the word "POST" in the preceding example to "GET", you have just changed the form. The difference is that GET parameters are actually appended to the URL, while POST parameters are sent invisibly by the browser as part of the HTTP request stream. The above form, with a GET method, would request a URL like this from the server:

```
http://icefloehousing.com/mypage.php?whoa
  reyou=Jane+Doe
```

Note that the space in "Jane Doe" was turned into a plus sign. Other special characters are turned into their hexadecimal equivalents, preceded by a percent sign (a space can also be coded as %20, because 0x20 is the ASCII code for a blank). This process is called "URL encoding." While PHP provides built-in functions to encode and decode URLs, the language is smart enough to realize that this is always needed for GET and POST variables, so if you use the predefined arrays, the decoding is already done for you.

The interesting thing about GET requests is that, because they are just URLs, you can encode them into your own HTML as regular <A> tag links. We saw that last month in the product catalog, where the catalog numbers on the main page provide a link to detail pages for each item. You were actually doing a primitive CGI request. Now it's time to get much more sophisticated, though.

Before getting down to the business of editing the database, Penny needs to define some HTML generating functions to make life easier. Forms processing is full of repetitive sequences, and in programming terms that means reusable functions! A new file, forms.inc, is created in the include directory, and two functions from the html.inc file are moved into forms.inc because that's where they really belong. Notice also that forms.inc has to be added into header.inc so that it will be part of every page.

There are two very simple functions called getFormTag() and getFormTableTag(). These functions, the ones migrated from html.inc, create the basic HTML tags to start a form. A table is typically, but not always, needed with a form because you will want to line up the prompt text with each appropriate input field or button, and a table is the best way to do that in HTML. It's important to watch the nesting of the table and form tags, so that one is completely inside the other, or else you have invalid HTML. Most of the time, it will work

FIGURE 1: GETFIELDCELLS() RETURNS A TABLE CELL FOR A PROMPT AND ANOTHER FOR THE DATA ENTRY FIELD.

```
function getFieldCells($prompt,$field_html,$prompt_attribs="",
    $field_attribs="",$begin="",$end="") {
        $html = "<td align=\"right\" valign=\"top\"";
        $html .= makeAttributeString($prompt_attribs) . ">";
        $html .= $begin;
        $html .= htmlspecialchars($prompt);
        $html .= $end . "</td><td align=\"left\" valign=\"top\"";
        $html .= makeAttributeString($field_attribs) . ">";
        $html .= $begin;
        $html .= $field_html . $end . "</td>";
        return $html;
}
```

FIGURE 2: GETSELECTHTML() PROVIDES AN EASY WAY TO CREATE A MULTI-ITEM SELECTION FIELD IN A WEB FORM.

```
function getSelectHTML($name,$values,$selected="",$multi_allowed=FALSE,$displayed=1)
{
        $name = makeAttribute("NAME",nameToForm($name));
        $size = makeAttribute("SIZE",$displayed);
        # First, we need to ensure that $selected is an array, for
        # simplicity's sake.
        if (! is_array($selected)) {
                $selected = array($selected);
        }
        # This array will contain TRUE for each value that was
        # preselected. We use this to speed up processing of long
        # value lists by avoiding an inner loop of array walking.
        $select_boolean = array();
        foreach ($selected as $selected_value) {
                $select_boolean[$selected_value] = TRUE;
        }
        $html = "<SELECT" . $name . $size;
        if ($multi_allowed) {
                $html .= " MULTIPLE";
        }
        $html .= ">\n";
        reset($values);
        while (list($value,$string) = each($values)) {
                $html .= "\t<OPTION VALUE=\"" . $value . "\"";
                if ($select_boolean[$value]) {
                        $html .= " SELECTED";
                }
                $html .= ">" . htmlspecialchars($string[0]) . "</OPTION>\n";
        }
        $html .= "</SELECT>\n";
        return $html;
}
```

FIGURE 3: GETDBSELECTHTML() POPULATES A MULTI-ITEM SELECTION FIELD FROM A DATABASE QUERY.

```
function getDBSelectHTML($db,$name,$sql,$selected="",$multi_allowed=FALSE,$displayed=1) {
        # Fetch an associative array. Force an array, even if only one
        # row happens to be returned. The keys will be the values of the
        # first column.
        $result = $db->getAssoc($sql,TRUE);
        if (DB::isError($result)) {
                # Take a reasonable default.
                $html = getSelectHTML($name,"",$selected,$multi_allowed,$displayed);
                # ... but append an error
                $html .= "<br>ERROR: Database error retrieving field values.\n";
        } else {
                $html =
getSelectHTML($name,$result,$selected,$multi_allowed,$displayed);
        }
        return $html;
}
}
```

anyway, but such things are often the cause of problems in specific browsers.

The following function automates a very common HTML generation task, specifically, putting a prompt immediately to the left of the appropriate input field. It returns a string containing two table cells, side by side, but without surrounding tags. See FIGURE 1 (on page 25).

This function, in turn, is called by a higher-level function, getFieldRow(), to generate an entire table row. This higher function allows the application to specify predefined HTML code to be inserted into the generated table row, useful for specifying boldface or italics for the prompt, and so forth. getFieldRow(), along with other functions not detailed in the article itself, can be downloaded from the Internet. The downloaded file also contains more detailed comments on each function than the listings in this article.

Now things start getting more interesting. There are some types of input field, such as dropdown lists, that potentially can return multiple values. For example, if you have a field called "hobbies" that is a dropdown list, you may wish to allow the user to select more than one. That is no problem; PHP can handle it just fine. The catch is that you have to make the field name an array, so it becomes "hobbies[]" instead of "hobbies". PHP will then set $_POST["hobbies"][0] to the first selection, …[1] to the second, and so on, for as many items as the user selected. Penny wants to have her utility functions be as general as possible, and special cases are the enemies of code reusability. So all of the utility functions will use field names that are arrays, even if the field type returns only one value. It's very easy to write code to handle an arbitrary number of values, and have it behave properly if that "arbitrary number" just happens to be one!

Furthermore, the field names are all prepended with "FORM__" to avoid possible collisions with names for buttons and possibly for GET parameters on the URL, something we might need to do later as the site gets more complex. The functions nameToForm() and nameFromForm() take care of translating a nice, intuitive name like "hobbies" to and from a canonical form like "FORM__hobbies[]" and back again. These functions are included with the code file for this article, which you can download.

There are functions for generating the HTML for simple text input fields, for hidden fields, and so on. These are fairly simple and won't be covered in detail here, but you will find them in the download files. Generating a button isn't particularly complex, either, but there is one important thing to note: In PHP, it is generally necessary to use the "name" attribute on buttons if you are going to have more than one. This is because functions like "Save Changes", "OK", "Cancel", "Delete Item", and so forth, in spite of their buttons having different visible text, are all of the SUBMIT button type. Giving each button a unique name lets your PHP code determine which button was pressed. The function getButtonHTML() provides what is needed.

The only complex field function is getSelectHTML(), which is the one that generates HTML code for the <SELECT>

FIGURE 4: GETEDITFORMFIELDS() CREATES MOST OF THE HTML FORM, EXCEPT FOR CONTROL BUTTONS.

```
function getEditFormFields($db,$row) {
        if (empty($row["cat_number"])) {
                $html = getFieldRow("Catalog Number",
                getTextFieldHTML("cat_number",$row["cat_number"],16,16));
        } else {
                $html = getFieldRow("Catalog Number",
                        getHiddenFieldHTML("cat_number",$row["cat_number"])
                        . "<b>" . htmlspecialchars($row["cat_number"]) .
"</b>");
        }
        $html .= getFieldRow("Product Name",
                getTextFieldHTML("name",$row["name"],40,40));
        $html .= getFieldRow("Product Type",
                getDBSelectHTML($db,"product_type",
                "select id,name from product_types order by name",
                $row["product_type"]));
        $html .= getFieldRow("Manufacturer",
                getDBSelectHTML($db,"mfg_id",
                "select id,name from manufacturers order by name",
                $row["mfg_id"]));
        $html .= getFieldRow("Price Each (Singly)",
                getTextFieldHTML("price",$row["price"],10,10));

(…rest removed for brevity…)
```

FIGURE 5: GETEDITFORM() CREATES THE COMPLETE HTML FORM FOR ADDING OR EDITING RECORDS.

```
function getEditForm($db,$row) {
        $html = getFormTag();
        $html .= getFormTableTag();
        $html .= getEditFormFields($db,$row);
        # Now make a three-column subtable spanning the two columns
        # of the parent table, to hold the buttons.
        $html .= "<TR ALIGN=\"CENTER\" VALIGN=\"BOTTOM\"><TD COLSPAN=\"2\">\n";
        $html .= getFormTableTag();
        $html .= "<TR ALIGN=\"CENTER\" VALIGN=\"MIDDLE\">";
        if (is_array($row) && count($row)) {
                $html .= "<TD>" . getButtonHTML("SUBMIT","BUTTON__SAVE","Save
Changes") . "</TD>\n";
                $html .= "<TD>" . getButtonHTML("SUBMIT","BUTTON__DELETE","Delete
Item") . "</TD>\n";
                $html .= "<TD>" . getButtonHTML("RESET","BUTTON__RESET","Reset
Form") . "</TD>\n";
        } else {
                $html .= "<TD>" . getButtonHTML("SUBMIT","BUTTON__INSERT","Add New
Item") . "</TD>\n";
        }
        $html .= "</TR></TABLE>\n";
        $html .= "</TD></TR>\n";
        # End of subtable.
        $html .= "</TABLE>\n";
        $html .= "</FORM>\n";
        # Now do the Cancel button, which is a special case because it has to be
outside
        # of the rest of the form.
        $html .= getFormTableTag() . "<TR ALIGN=\"CENTER\">";
        $html .= "<TD>" . getButtonHTML("CANCEL","BUTTON__CANCEL","Cancel") .
"</TD>\n";
        $html .= "</TR></TABLE>\n";
        return $html;
}
```

and <OPTION> tags. The tricky part is correctly marking the item(s) that is/are already selected in an existing record. The code to accomplish this is in FIGURE 2 (on page 25).

Note the use of a Boolean array to speed up the code execution. Without it, there would need to be two nested loops, an outer loop to step through the list of possible values and an inner loop to examine each preselected value to see if it matches the current option from the outer loop. If M is the number of possible items and N is the number of preselected ones, the execution time is proportional to M*N, whereas with the Boolean array, the time is proportional to M+N because the would-be inner loop actually executes only once. When Penny's company gets larger, and more people start accessing the site, she will be very glad she took the time to optimize the PHP code.

Now, how does one obtain that "list of possible values" for the <OPTION> tags within a <SELECT> block? In Penny's case, they come from the database. The Manufacturer field and Product Type field each allow the user to select from those items listed in another table, manufacturers and product_types, respectively. So Penny needs a function that will obtain the rows from a database query and put them into a <SELECT> HTML tag. See FIGURE 3 (on page 25).

Give this function a valid SQL query, such as "select id,name from manufacturers order by name", and it will return the ready-to-use HTML for the field tag. Pretty handy, eh?

At last, we've worked our way through most of the utility functions. Penny Penguin has a very nice forms processing library that will serve her well on future projects. Now it's time to put these functions to work. In the icefloehousing.com/html directory on her server, Penny creates a brand-new file called catalog_edit.php. This file will use some of the code from the catalog.php file discussed last month, so the discussion that follows will focus on the new code only.

After the common lines that begin every page on her site, Penny adds the following code, which will execute before the main body of the page:

```
if (!empty($_POST["FORM__cat_num-
  ber"][0])) {
$_GET["cat_number"] =
  $_POST["FORM__cat_number"][0];
}
```

The forms processing code is all set up to work with the POST method, but when the user first selects an existing catalog number from the main page, that number is part of the URL and therefore is a GET parameter. This little snippet of code simply copies that value from GET to POST, so that the remainder of Penny's PHP code can forget about this special case and always assume that all of the data is in the POST variables.

The getRowHTML() function, copied from the catalog page, is stripped down to remove the "detail mode." Now getRowHTML() will

FIGURE 6: USE BUILDUPDATESQL() TO CREATE AN SQL STATEMENT FOR DATABASE CHANGES.

```
function buildUpdateSQL($row) {
        $fields = array("product_type","name","mfg_id",
                  "price","lot_size","lot_unit_price","summary","detail");
        $sql = "UPDATE catalog SET ";
        foreach ($fields as $field) {
                  $sql .= $field . "='" . $row[$field] . "',";
        }
        $sql = ereg_replace(',$','',$sql);
        $sql .= " WHERE cat_number='" . $row["cat_number"] . "'";
        return $sql;
}
```

FIGURE 7: UPDATEDB() IS RESPONSIBLE FOR ACTUALLY MODIFYING DATA RECORDS IN THE MYSQL TABLE.

```
function updateDB($db,$row) {
        $errors = array();
        if (!empty($_POST["FORM__BUTTON__INSERT"][0])) {
                  # Attempt to insert a new row.
                  $row = setFields();
                  if (!empty($row["cat_number"])) {
                            $sql = buildInsertSQL($row);
                            $result = $db->query($sql);
                            if (DB::isError($result)) {
                                $errors[] = "SQL was: " . $sql;
                            }
                  } else {
                            $errors[] = "Missing Catalog Number";
                  }
        } elseif (!empty($_POST["FORM__BUTTON__DELETE"][0])) {
                  $row = setFields();
                  if (!empty($row["cat_number"])) {
                            $sql = buildDeleteSQL($row);
                            $result = $db->query($sql);
                            if (DB::isError($result)) {
                                $errors[] = "SQL was: " . $sql;
                            }
                  } else {
                            $errors[] = "Missing Catalog Number";
                  }
        } elseif (!empty($_POST["FORM__BUTTON__SAVE"][0])) {
                  $row = setFields();
                  if (!empty($row["cat_number"])) {
                            $sql = buildUpdateSQL($row);
                            $result = $db->query($sql);
                            if (DB::isError($result)) {
                                $errors[] = "SQL was: " . $sql;
                            }
                  } else {
                            $errors[] = "Missing Catalog Number";
                  }
        } else {
                  # No operation required - return FALSE so we
                  # don't think we did an update.
                  return FALSE;
        }
        # Print out the error(s), if any
        foreach ($errors as $error) {
                  print("<br><b>" . htmlspecialchars($error) . "</b>\n");
        }
        # Return TRUE if successful (zero error count)
        return (count($errors) < 1);
}
```

FIGURE 8: DATA ENTRY SCREEN FOR A NEW CATALOG ITEM

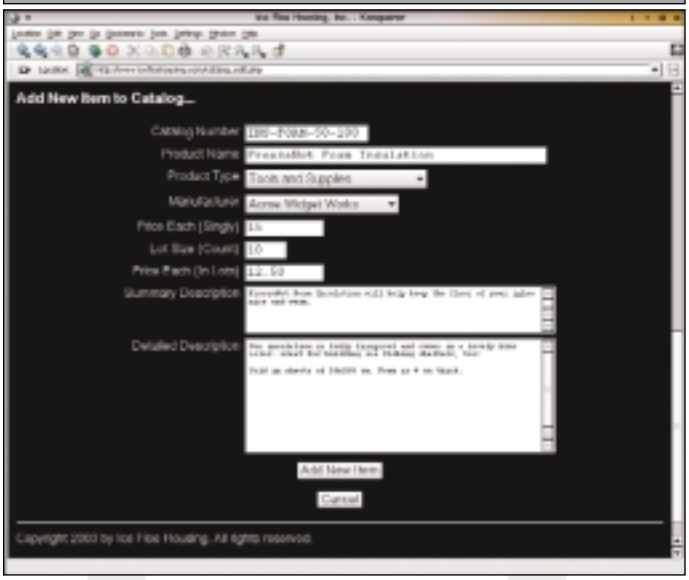
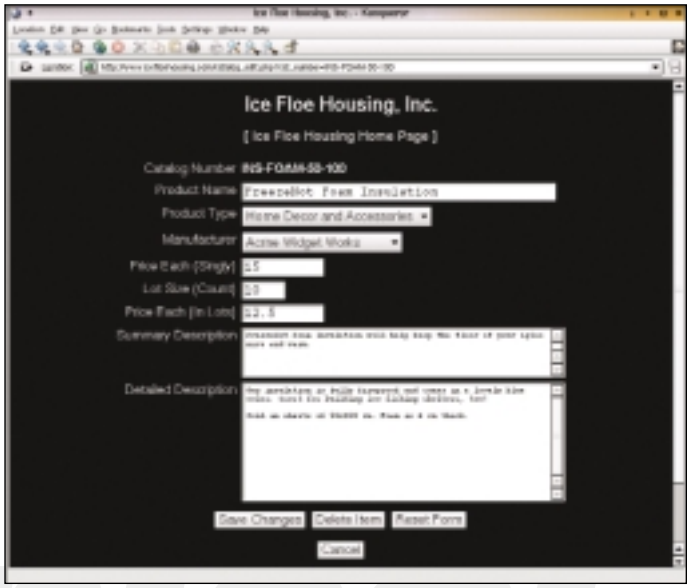


FIGURE 9: DATA EDIT SCREEN FOR EXISTING ITEMS. NOTE THAT THE CATALOG NUMBER IS DISPLAYED BUT NOT EDITABLE



be used in a selection display that looks just like the regular catalog, except that if the user selects an item, they will see an editing screen rather than just a detailed display. getRowHTML() isn't responsible for the edit form, so Penny removes that part of the code from this function. In the main display page, a blank editing form will be added at the bottom of the page to allow a new item to be inserted. FIGURE 8 shows the insert form, with a new item being entered into the database.

A new function, getEditFormFields(), does most of the work of creating the visible edit form for the user. See FIGURE 4 (on page 26).

Notice the repetitive part at the bottom, which generates field after field of pretty HTML using the functions in the forms.inc library. Think of all the repeated code that would be needed if not for that library! The beginning of getEditFormFields() examines the all-important cat_number variable, to see if there is a pre-existing row that is to be edited. This is to distinguish between an empty form in which Penny can create a new catalog item, and a pre-filled form in which she can change an existing item's information. The cat_number field is the primary key in the database, and allowing that to be changed introduces some SQL complications that are beyond the scope of this article. So for now, the cat_number of an item is displayed in the edit form as static text. A HIDDEN-type HTML field captures that number for the POST of the form, but the user cannot change it in the browser. FIGURE 9 shows the edit form, with the catalog number as static text.

getEditFormFields() generates the HTML for the entry fields, but not for the complete form. getEditForm() takes care of the rest of the work, calling getEditFormFields() as needed. See FIGURE 5 (on page 26).

Once again, note the distinction made between inserting a new item and editing an existing one. Among other things, the buttons get different names, and there is no such thing as deleting a record that does not yet exist! The Cancel button simply takes the user back to this same page, but with no catalog number selected (causing the list of items to again display). There are actually better ways to implement this feature, but this one is simple and will suffice for this example.

The function setFields(), which calls getSimpleFormValue() from the library for each field, copies the data from the $_POST[] array into an array called $row[], which has fields corresponding directly to the columns in the MySQL database table. This is used for both insert and update operations, and also for delete, though the only field that matters for delete is the cat_number, which is the table's primary key.

Given a $row[] array containing the information, the functions buildInsertSQL(), buildUpdateSQL() and buildDeleteSQL() create SQL statements that can be passed to the PEAR database functions to make actual changes. These are similar to one another in structure, but of course create different SQL syntax. See FIGURE 6 (on page 27) for the code for buildUpdateSQL().

When generating a long list of comma-separated items, such as the update sequence, it's sometimes easier to let every iteration, including the last one, append the comma, then remove the extra final comma from the result string, rather than testing each iteration of the loop to see if it's the last time through. This is what the ereg_replace() function does in the preceding code.

The most interesting part of the cata-log_edit.php page is the code to do the actual database updates. See FIGURE 7 (on page 27).

Notice how each button is tested by looking to see if its form field is empty. The actual contents of a button's field in $_POST[] will be the text displayed on the button, but Penny knows not to test for equality with that text! If she did, and she later decided to change the button text to make it more intuitive for the user (or to translate to another language), then the database update code would break. The code shown here simply checks whether each button is the one that was pressed, and doesn't care what the button says.

Penny has used an array called $errors[], and fills messages into it with an assignment like this:

```
$errors[] = "Missing Catalog Number";
```

Then a loop at the end of the function iterates through the array, printing any messages that were queued. A simple count() detects if the array is empty, indicating no errors were found, and the updateDB() function returns TRUE if it was successful and FALSE if it failed or if there was no need of an update at all.

The main body of the code (not shown here) calls all of these functions in the correct order, as needed. If a database change is successful,

FIGURE 10: THIS SCREEN NOTIFIES THE USER OF A SUCCESSFUL DATABASE UPDATE.

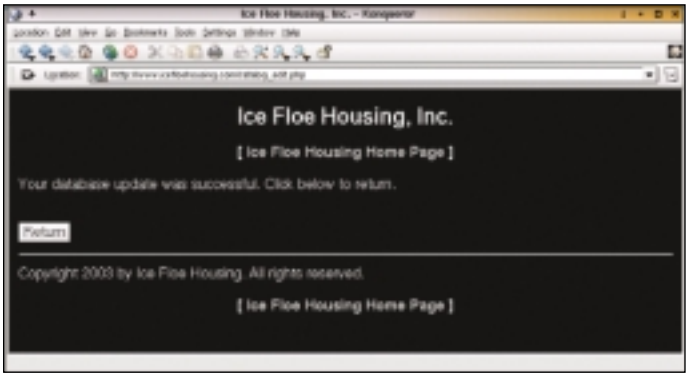
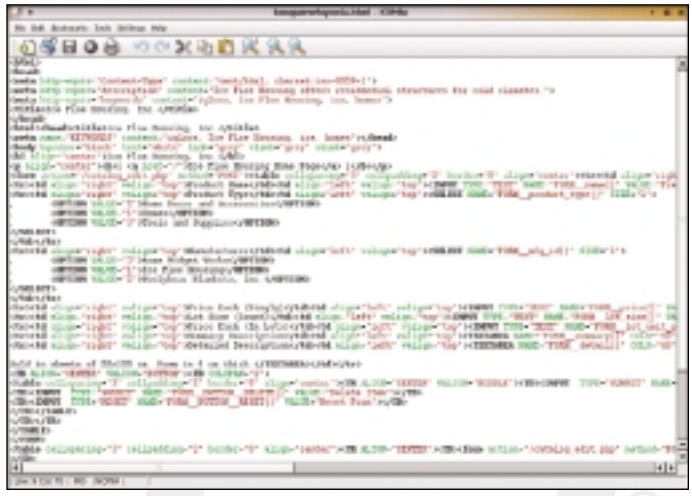


FIGURE 11: GENERATED HTML CODE AS THE EDIT FORM IS SENT TO THE BROWSER. NO PHP IS EVER SENT TO THE BROWSER, SO COMPATIBILITY IS ASSURED.



a simple message page is displayed, and a button allows the user to easily return to the main page to select another item for editing. See FIGURE 10 (on page 30). FIGURE 11 shows the actual source code for the editing form. As with our previous examples, notice that there is no PHP code whatsoever sent to the browser, an important point for compatibility with any client computer. Everything is just standard HTML, as far as the browser is concerned.

Penny has created a page to allow catalog edits, but one piece is missing: security! In this example, there is really nothing wrong with letting anyone edit the catalog, because Ice Floe Housing is a make-believe company. If you use this code, or a derivative of it, on your web site, you will definitely want to use Apache's security features (documented on the Apache web site; look for "htpasswd") to lock down this page so that only trusted staff can access it.

Almost all of the complex code for this application is contained within reusable functions that do not make specific reference to any particular database table. That means Penny can recycle much of her work to make other pages for editing the Manufacturers and Product Types in her database, the tables that feed a list of choices to the form she just created. Separating generic data handling and HTML functions from those things that are specific to a particular application is a good idea, especially as a site becomes more complex.

Ice Floe Housing has a new web page, and Penny no longer has to type SQL commands just to change the price of an item in her catalog. More importantly, Penny can hire an assistant, who need not be a technical guru, to help keep her database up to date. Best of all, Penny's web toolkit is growing, and she now has code that will help a great deal when she puts her order form online for customers. Next month, Penny will put this code to good use with a simple virtual shopping cart.

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.