Build a Subfile with PHP

Module 2: Formatting Customer Records in an HTML Table, and Adding a Search Form

Contents

Formatting Customer Records in an HTML Table, and Adding a Search Form

In this exercise you will make two major changes to the functionality of our customer listing application:

1) Format the results in an HTML table, with a top row of column headings, with rows of data lining up in columns below the headings. This will make it look more like a subfile.
2) Add an input form above the table to allow search filters to be entered by the user.

We will also make some structural changes to the code base, as follows:

1) Separate all of the HTML and PHP into different files. This is similar to an RPG program and a DDS display file, and it makes management of the code base much simpler to have these two languages as separate as possible. However, due to limitations of HTML, we will need to insert small snippets of PHP into the HTML file to echo the data variables on the screen.
2) Create user-defined PHP functions to better organize the code base and make it easier to understand, test, and maintain.
3) Use an external CSS style sheet file, which makes it easy to reuse the styles in different pages and applications. In fact, you will use the same .css file for the remaining exercises. This css file is already created, you will just need to link to it from the html <head> section.

In this exercise you will be working with two files: one for the PHP (`sfl_02.php`) and one for the HTML (`sfl_02.phtml`). Notice the extension for the HTML file is .phtml – this indicates a file which is primarily html, but has some PHP in it as well. Both the .php and .phtml file extensions tell the Apache web server that these files need to be processed by the php interpreter before sending back to the client.

Another big difference from the first exercise, is that instead of fetching a record and then echoing it immediately, before the next record is fetched, you will store each record fetched in a PHP multi-dimensional array (similar to a multi-occurrence data structure). The goal is to separate the PHP processing steps and database access in one file, then pass all the prepared and formatted information to the screen at once (as a big array). Then in the .phtml file all we need to do is retrieve data elements from the array and echo them with small php snippets. We do not want to have any database access going on in the screen file (phtml).

All of the files for this module are found in the **`sfl_02`** subfolder. These files contain the structure of the application code, with several key pieces missing. Your goal will be to add all of the missing pieces to create the complete, working application. Let's start with the PHP code, then we'll tackle the HTML.

Open the PHP Template

⇨ In Zend Studio, go to the `sfl_02` folder and open file `sfl_02.php`

The beginning of it should look like this:

```
<?php
// Include the screen definition file

// connect to db2
$options = array('i5_naming'=> DB2_I5_NAMING_ON, 'i5_libl' => 'ZENDSVR6');
$conn = db2_connect("*LOCAL","jvlabsXX","jvlabsXXpw", $options)
          or die("Connection failed! ". db2_conn_errormsg());

// Initialize screen data array from inputs

// Construct SQL query. Call getSqlWhere() function to add any filters that were specified

// Prepare and execute the SQL query

// Retrieve all data rows into an array we can pass to the screen

// Show the screen

//-----------------------------------------------------------------------
// End of mainline
//-----------------------------------------------------------------------
```

The db2_connect() is already provided, since you did that in the last exercise. However, you must change the user/pswd to match your team#

⇨ Change **jvlabsXX** in both places such that **XX** is replaced with your team number

The rest of the comments indicate steps for which you must add php code. Let's add these steps one at a time…

Fleshing Out the Mainline

⇨ Find the comment below, and add the following statement to capture all inputs received from the client on the http request:

```
// Initialize screen data array from form inputs
$screenData = $_REQUEST;
```

The variable $_REQUEST is a pre-defined array in PHP. It is created automatically by PHP and contains all parameters passed from the client (browser) when it requests this script on the server. Whatever inputs (i.e., name/value pairs) were sent, are parsed by php and stored in an array format for easy access in php code. We may need to redisplay some of this data on the screen when we send back the response document. So we will store them in an array called $screenData, which will contain not only all the inputs received, but any additional output values that we want to show on the screen (i.e., the list of database records to populate the table). We cannot modify the $_REQUEST array, so we store everything in our own array. When we're ready to display the screen, we can pass this single array variable, $screenData, to the screen, and it will serve as a very handy container for everything we need to share on the screen.

⇨ Construct the SQL string to retrieve customer data

```
// Construct SQL query. Call getSqlWhere() function to add any filters that were specified
$sql = "SELECT * FROM SP_CUST " . getSqlWhere( $screenData );
```

The basic SQL is the same as our first exercise (select * from sp_cust). But in the second half of this exercise you will add filter fields on the screen to limit the customers to those matching the search values. In order to accomplish this, we will need to add SQL WHERE clause conditions, for each filter value entered. This will be done in the function named getSqlWhere(), to which we will pass the filter inputs via the $screenData array, and it will return the "AND" conditions for the WHERE clause, which is then concatenated onto our SQL string. In the second part of this exercise, you will flesh out the logic for this function.

⇨ Call the function which prepares and executes the SQL, and capture the returned prepared statement in variable $stmt

```
// Prepare and execute the SQL query
$stmt = getPreparedStatement($conn, $sql);
```

Function `getPreparedStatement()` will run both the `db2_prepare()` and `db2_execute()` functions, and will check for any errors with these actions. It requires as input both the db2 connection ($conn) and the SQL string to prepare/execute ($sql). It will return the prepared statement, which we need in order to fetch records. Note: `getPreparedStatement()` has already been written for you.

⇨ Call the function retrieveDataRows, which will fetch the data rows and return them as a multi-dimensional array.

```
// Retrieve all data rows into an array we can pass to the screen
$screenData['dataRows'] = retrieveDataRows($stmt);
```

The function `retrieveDataRows()` will perform the fetch loop you created in the first module, including formating name and address on each row. The main difference here is that the function will store all the data rows in an array, and then it returns this array, which we store in the $screenData array.

The last comment in the mainline refers to showing the screen:

```
// Show the screen
```

You will come back and add this statement in a later part of this exercise, once we starting building the HTML file.

With the above additions, you should now have an (almost) complete mainline for this PHP application. Let's continue by completing the missing pieces of the local functions in this file.

Fleshing Out the Functions

This file contains 4 functions:

1. `getSqlWhere` – This will build the SQL where clause based on the filters specified by the user. We will skip this function for now; in the 2nd part of this exercise you will add the filtering logic.
2. `getPreparedStatement` – This function is already written, with no changes needed.
3. `retrieveDataRows` – This has the fetch loop to get the customer records matching our search, and return all of the matches in an array list. This has a few pieces missing which you will need to supply.
4. `pre_dump` – This is a handy utility function for debugging purposes. It is already complete, and you will use it later.

As explained above, the only function you will need to modify at this point is `retrieveDataRows()`.

⇨ Look at the definition for function `retrieveDataRows()`:

```
/**
 * Fetches all rows returned by the SQL query. Rows are returned in a
 * multi-dimensional array, containing one row per 1st-level array element.
 * @param resource $stmt The prepared statement to fetch results from.
 * @return array - A multi-dimensional array containing all of our matching rows.
 */
function retrieveDataRows($stmt) {
      // Define $dataRows as an empty array

      // Initialize row counter variable

      while ( $row = db2_fetch_assoc( $stmt )) {
            // Increment the row counter and add it to the $row array

            // Format customer information for display
            $row['name'] = $row['FIRSTNAME'] . ' ' . $row['LASTNAME'];
            $row['address'] = "{$row['ADDRESS']}  {$row['ADDR2']}<br>
                              {$row['CITY']}, {$row['STATE']} {$row['ZIP']} {$row['COUNTRY']}";
```

```
            // add DB row to array

      }

      // return the array containing the set of matching rows

}
```

There are several comments in here with pieces of code missing that you will add.

- Add the following statement to define variable $dataRows as an empty array. In the while loop you will populate this array with customer records.

```
      // Define $dataRows as an empty array
      $dataRows = array();
```

- Initialize the row counter to zero

```
      // Initialize row counter variable
      $rowNumber = 0;
```

- At the top of the loop body, increment the $rowNumber (using ++), and add this value as a new element in the $row array.

```
      while ( $row = db2_fetch_assoc( $stmt )) {
            // Increment the row counter and add it to the $row array
            $row['rowNumber'] = ++$rowNumber;
```

The ++ operator is the numeric increment operator. Writing: `++$rowCounter;` is the same as writing `$rowCounter = $rowCounter + 1;` We will show the row number in the first column of the customer listing table.

- ⇨ Add the statement to add the current DB row to the returned $dataRows array. Note the empty square brackets [ ]
- ⇨ Also add the return statement, after the loop body's closing brace.

```
            // add DB row to array
            $dataRows[] = $row;
        }
        // return the array containing the set of matching rows
        return $dataRows;
```

In the first statement above, we're using a special syntax in PHP for adding a new entry to an array. Notice we did not specify an index (aka 'key') for this entry (i.e., nothing between [and]), just an empty pair of brackets. When you do this, PHP will add a new element to the array and automatically assign an index value for it. The index will be an integer value (unlike the character indices you've seen already). The value of this index is computed by scanning the array for any existing numeric indices: if there are numeric indices, the highest value is retrieved, and 1 is added to this to obtain the new index value. If no numeric indices are found, then the new entry will have index 0 (zero).

Since we started with an empty array, by using this logic we will wind up with an array, starting with index 0, ending with index = (number of rows retrieved) -1. The returned multi-dimensional array will contain an ordered list of customer record arrays. If you ran a pre_dump($dataRows) before the end of this function, the structure of the array would look like this (with some elements omitted for brevity):

```
array(54) {
  [0]=>
        array(16) {
          ["CUST_ID"]=>
          string(4) "1221"
          ["COMPANY"]=>
          string(30) "Kauai Dive Shoppe             "
          ["FIRSTNAME"]=>
          string(20) "LINA                "
          ["LASTNAME"]=>
          string(20) "Norman              "
          etc...
        }
  [1]=>
        array(16) {
          ["CUST_ID"]=>
          string(4) "1231"
```

```
          ["COMPANY"]=>
          string(30) "Unisco                         "
          ["FIRSTNAME"]=>
          string(20) "George              "
          ["LASTNAME"]=>
          string(20) "Weathers            "
          etc...
        }
[2]=>
       array(16) {
         ["CUST_ID"]=>
         string(4) "1351"
         ["COMPANY"]=>
         string(30) "Sight Diver                    "
         ["FIRSTNAME"]=>
         string(20) "Phyllis             "
         ["LASTNAME"]=>
         string(20) "Spooner             "
         etc...
 etc...
[53]=>
       array(16) {
         ["CUST_ID"]=>
         etc...
```

You can try this yourself:

⇨ Add a temporary statement to dump the `$dataRows` array, just before returning from this function:

```
            // add DB row to array
            $dataRows[] = $row;
        }
        pre_dump($dataRows);
        // return the array containing the set of matching rows
        return $dataRows;
```

Recall that earlier in this exercise, you added this statement in the mainline:

```
// Retrieve all data rows into an array we can pass to the screen
$screenData['dataRows'] = retrieveDataRows($stmt);
```

So the value returned by `retrieveDataRows()` will be an array of arrays, and we are assigning that as an element of the $screenData array (`$screenData['dataRows']`). So that makes $screenData a 3-dimensional array. Remember that PHP arrays can have any number of dimensions, and can easily mix data types – i.e. any element can contain any data type: string, integer boolean, array, object, etc..

After saving your changes, you can run this script now. We haven't yet connected this with the HTML to present a table, but the call to pre_dump( ) will show a variable dump of the $dataRows array, which should contain 54 customer data arrays.

Running the Script

- ⇨ Save your changes by pressing Ctrl/S.
- ⇨ Open your browser of choice, and type in or copy/paste the following URL:
  http://common1.frankeni.com:10080/jvlabs/teamXX/phpsf/sfl_02/sfl_02.php
  where the XX should be ***replaced with your team number***

You should see something similar to the array dump shown on the previous page (but much longer).

Adding the HTML

- ⇨ After you successfully get the dump output in the last step, remove the statement you added to dump the array (`pre_dump($dataRows);`). If you leave this in, it will show up above the actual listing table, and you'll have to scroll way down to see your results.
- ⇨ Open the file sfl_02.phtml

Initially, this file contains only HTML. This is a template for the HTML document that will be returned to the browser. But it is not complete: In this exercise, you will be adding additional HTML, as well as short snippets of PHP, necessary for it to work with the PHP controller script, and output database values.

This will include the following:

- Wrap all of the HTML in a PHP function body, so we can output the HTML after we're prepared all of the variable data contents in the controller script.

- Add a parameter on the function to receive the screenData array, with all the customer records to display.
- Add a foreach loop around the table row, to loop through all the customer records and output each as an HTML table row (<tr> tag)
- Add php echo statements inside each of the <td> tags within the foreach loop, to output the customer data values into HTML table columns (<td> tag).

⇨ Scroll through the file and study the HTML. Pay attention to the HTML comments that tell you where to add PHP & HTML code.

Throughout the file you'll see HTML comments like this:

```
<!-- ADD(1) => PHP foreach loop to retrieve dataRows array -->
```

The prefix `ADD(1) => PHP` means that you will be adding PHP code in the 1st exercise. If it says `ADD(2)` then you'll be doing it in the 2nd exercise. If it says HTML, then you'll be adding HTML there. If it doesn't say ADD, then it's just a comment explaining the existing HTML.

⇨ Go to the top of the file (by pressing Ctrl/Home).
⇨ Find the comment shown below, and add the PHP code under it

```
<!-- ADD(1) => PHP to define this entire file's content as an executable function -->
<?php
function showCustomerList( $screenData ) {
?>
```

⇨ Go to the end of the file (by pressing Ctrl/End)
⇨ Find the comment shown below, and add the PHP code to close the function body

```
<!-- ADD(1) => PHP to close the function definition -->
<?php
// end of function body
}
```

⇨ Save your changes

Now that we've wrapped all the HTML in a PHP function definition, the only way this HTML will be written to the browser is by including this file in another PHP file, and then calling the `showCustomerList()` function. Those are your next steps.

⇨ Open the editor to the PHP controller script (sfl_02.php)

⇨ At the top of the file, add the following statement to include the HTML/PHP file (sfl_02.phtml)

```
<?php
// Include the screen definition file
require_once('sfl_02.phtml');
```

The require_once() function is similar to using /copy in RPG. It will insert the contents of file sfl_02.phtml at this point in the code. But because we wrapped all the HTML in a function, the HTML will not be output to the browser at this point. You must add a statement to call the `showCustomerList()` function. You will add that at the end of the mainline, after all the data has been gathered into $screenData:

⇨ Find the comments below, and add the call to `showCustomerList( )`

```
// Show the screen
showCustomerList($screenData);

//-----------------------------------------------------------------------
// End of mainline
//-----------------------------------------------------------------------
```

⇨ Save your changes

Adding the foreach Loop

Next you need to add a foreach loop in the phtml file to populate the listing table, and then you can take this script for a test run.

- ⇨ Open the editor for the `sfl_02.phtml`
- ⇨ Find the comment below and add the PHP foreach loop:

```
            <!-- ADD(1) => PHP foreach loop to retrieve dataRows array -->
            <?php
            foreach($screenData['dataRows'] as $row) {
            ?>
```

- ⇨ Now find the comment below, just after the closing table row tag (</tr>)
- ⇨ Add the closing brace (}) to end the foreach loop body:

```
                </tr>
            <!-- ADD(1) => PHP to close the foreach loop -->
            <?php
            }
            ?>
        </table>
```

Now the HTML table row that sits inside the foreach loop will be output repeatedly, once for each row of data in the database. But you still need to echo the data from each customer row. First let's review what the foreach loop does.

How the PHP foreach Loop Works

The foreach loop is a special type of loop in PHP that allows you to easily iterate over the elements of an array. The foreach loop has the general format:

```
foreach (<array-expression>  as <$current_element>)  {
     <statements using $current_element>...
}
```

where:

- `<array-expression>` is any expression that evaluates to an array, such as an array variable, a function call returning an array, an array element containing an array, etc.
- `<$current_element>` is a variable name of your choosing, which will be populated with the next element of the array on each iteration through the loop. When all elements have been iterated over, the loop will end.
- `<statements using $current_element>...` is where you add statements that process each element of the array, according to your needs.

Once again, recall that earlier in this exercise, you added this statement in the PHP mainline:

```
// Retrieve all data rows into an array we can pass to the screen
$screenData['dataRows'] = retrieveDataRows($stmt);
```

So the value stored in `$screenData['dataRows']` is an array of customer records that we fetched from the database. So in the foreach you just added:

```
foreach($screenData['dataRows'] as $row) {
```

For each iteration of the loop, the variable $row will contain one complete customer record, represented as an associative array, just like the array returned by db2_fetch_assoc( ) that we used in `sfl_01` exercises. So now we can retrieve customer data values from the $row variable, using the db2 field names as keys for the $row array.

⇨ In the highlighted lines below, remove the HTML comments within the <td> </td> pairs, and replace them with the PHP snippets below, to echo data from the customer record:

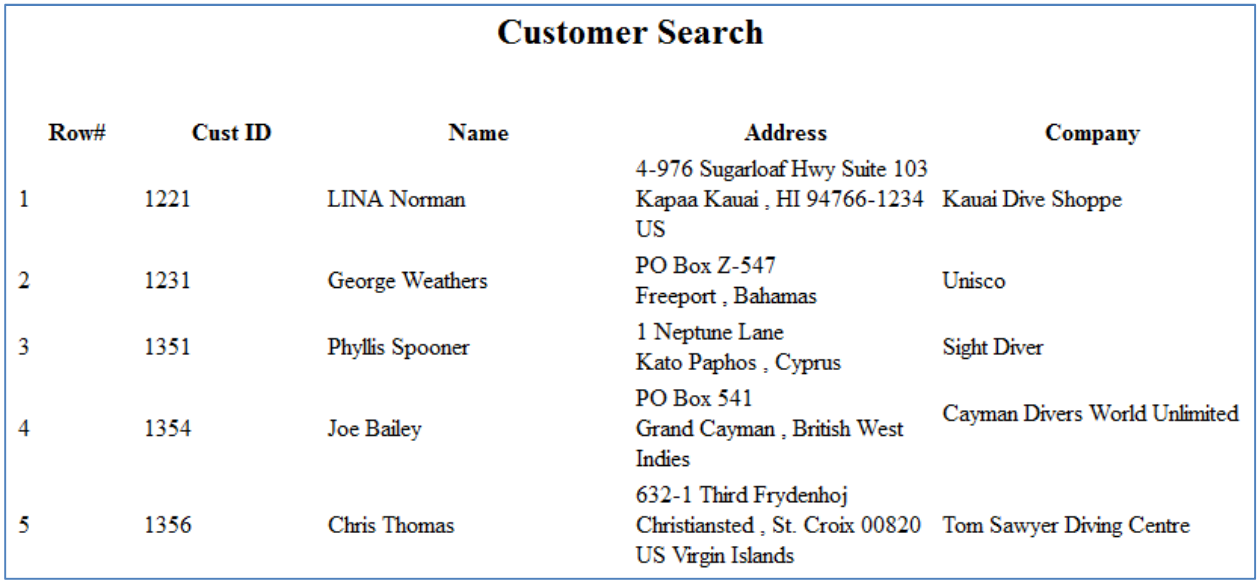
```
<?php
foreach($screenData['dataRows'] as $row) {
?>
      <!-- The following table row will be output for each customer record -->
      <tr>
            <td><?php echo $row['rowNumber'] ?>  </td>
            <td><?php echo $row['CUST_ID']; ?>  </td>
            <td><?php echo $row['name']; ?>  </td>
            <td><?php echo $row['address']; ?>  </td>
            <td><?php echo $row['COMPANY'] ?>  </td>
      </tr>
<!-- ADD(1) => PHP to close the foreach loop -->
<?php
}
?>
```

Notice that some of the field names are in uppercase (CUST_ID, COMPANY), and some are lower/mixed case (rowNumber, name, address). The ones in upper case came directly from the database, without any formatting or changes. The others are derived array elements, holding the computed row number, and the formatted name and address, which were added to the db row array by your PHP code.

Once these changes are made and saved, you can now run this script to see the customer records in a table.

⇨ Save all changed files
⇨ Go to the browser and refresh the sfl_02.php page.

You should see output in the browser similar to this:



Customer Search

Row#	Cust ID	Name	Address	Company
1	1221	LINA Norman	4-976 Sugarloaf Hwy Suite 103 Kapaa Kauai , HI 94766-1234 US	Kauai Dive Shoppe
2	1231	George Weathers	PO Box Z-547 Freeport , Bahamas	Unisco
3	1351	Phyllis Spooner	1 Neptune Lane Kato Paphos , Cyprus	Sight Diver
4	1354	Joe Bailey	PO Box 541 Grand Cayman , British West Indies	Cayman Divers World Unlimited
5	1356	Chris Thomas	632-1 Third Frydenhoj Christiansted , St. Croix 00820 US Virgin Islands	Tom Sawyer Diving Centre

This is a success worth celebrating, but it's not very stylish. Let's celebrate in style. In fact, we have one more thing to celebrate, which is that the style sheet for this screen has already been provided by your humble author. So you only need to add one more line of html and behold while something wonderful happens.

- ⇨ Switch to the editor for the HTML (sfl_02.phtml)
- ⇨ In the <head> section, find the comment below and add the <link> tag exactly as below, to import the style sheet.

```
<html>
      <head>
            <!-- ADD(2) => HTML link to external style sheet here -->
            <link type="text/css" href="phpsfl.css"  rel="stylesheet">
      </head>
```

- ⇨ Save the changes
- ⇨ Refresh the browser

Behold, the **Modern Age . . .**

Customer Search

Row#	Cust ID	Name	Address	Company
1	1221	LINA Norman	4-976 Sugarloaf Hwy Suite 103 Kapaa Kauai , HI 94766-1234 US	Kauai Dive Shoppe
2	1231	George Weathers	PO Box Z-547 Freeport , Bahamas	Unisco
3	1351	Phyllis Spooner	1 Neptune Lane Kato Paphos , Cyprus	Sight Diver
4	1354	Joe Bailey	PO Box 541 Grand Cayman , British West Indies	Cayman Divers World Unlimited
5	1356	Chris Thomas	632-1 Third Frydenhoj Christiansted , St. Croix	Tom Sawyer Diving Centre

It's actually a fairly simple style sheet, specifying rules for a few tag names and classes. You can see for yourself:

⇨ Open the file `phpsfl.css` in the `sfl_02` subfolder

It looks like this:

```
/* Default style for <body> content */
body {
      font-family: arial;
      color: navy;
      background-color: #ccc;
}
```

```
h2 {
      border: 3px ridge silver;
      background-color: #EEF;
      font-family: sans-serif;
      width: 50%;
      color: darkred;
      font-size: 1.75em;
      margin: 4px;
      padding: 3px
}

table.lists caption {
      background-color: #E0E0E0;
      border: 1px solid navy;
}
etc…
```

⇨ Browse around this file. It contains all the styling you will need for the rest of the exercises.

If you have any questions about the css, ask the instructor.

Adding Search Filters to the List

In this part of the exercise, you will add some input fields to allow the user to filter the list by customer ID, name, address and company. This will involve the following changes to both the .php and the .phtml files:

- Add a <form> tag within the table's <caption> tag in the phtml. The form tag defines a group of input fields, and what action to perform with them when the form is submitted for processing.
- Add a group of <label> and <input> tags within the form, which will provide a place for users to enter the search values
- Add a submit button to process the search filters
- Add logic in the PHP controller to test for the presence of filter values on the input (i.e. in the $_REQUEST array), and construct the proper SQL WHERE conditions before running the query.

Let's start with the HTML.

⇨ Switch to the editor for sfl_02.phtml

⇨ Find the comment below, and add the open <form> tag, exactly as below.

```
      <!-- ADD(2) => HTML form tag to define processing options -->
      <form name="customerList" action="<?= $_SERVER['SCRIPT_NAME']; ?>" method="post">
```

The <form> tag by itself does not render anything visible on the page. It simply defines a group of input fields that will all be processed by the same script (i.e. PHP file, in our case). In the next step you will add a closing </form> tag, and then add a series of <input>s between the <form> and </form> tags. Note that we are specifying three attributes on the open <form> tag: name, action, and method. The PHP script that will process these inputs is specified in the action="…" attribute of the <form> tag. Here the value of the action attribute is supplied by a snippet of PHP code which echos the name of the current PHP script that is controlling this user request. This bit of code:

```
<?= $_SERVER['SCRIPT_NAME']; ?>
```

which is embedded in the action attribute of the <form> tag, is short-hand for:

```
<?php echo $_SERVER['SCRIPT_NAME']; ?>
```

The short version, which replaces `"php echo"` with `"=",` is often used in phtml files, because we often have a need to simply echo a variable value within HTML tags and content. The value being echoed is `$_SERVER['SCRIPT_NAME'];`. The $_SERVER[ ] array is another pre-defined array in PHP, populated automatically on each request by the PHP runtime engine. It contains information about the server environment, including which server-side script was requested by the client. In our case this value will be sfl_02.php. Sometimes when you create forms, you will want a different script to process the inputs than the one that created the form originally. But in this case, we want the inputs to be reprocessed by the same PHP script, each time applying the new filter values in the SQL to retrieve the list. We could simply hard-code the attribute value, i.e. action="sfl_02.php". But by coding it using the $_SERVER array, you can copy this template to a new listing application without ever having to change this one line of code.

The <form> tag must be closed with a </form> tag. You will put this toward the end of this file, so that the form encloses most of the visible portion of the page.

⇨ Find the comment below, and add the end-form (</form>) tag

```
            </table>

            <!-- ADD(2) => HTML close the form tag -->
            </form>

            </center>
      </body>
</html>
```

Now we need to add the input tags.

Adding the Submit Button

Find the comment below and add the highlighted <input> tag, exactly as below

```
      <!-- caption tag provides a banner across top of the table -->
      <caption>
            <!-- ADD(2) => HTML form inputs for the search filters here -->
            <input type="submit" value="Search On =>" />
```

The <input> tag comes in a variety of types, each specified by the type=" " attribute. The default type is type="text", which provides an input field into which a user can type text. There's also a special input, with type="submit". This will produce a button on the screen, which, when clicked will submit the form fields to the server script, specified by the action attribute on the <form> tag. The value attribute specifies the text that should appear on the button, in this case we've created a button with a label that says "Search On =>".

Adding the Input Tags

Now you will add the text input fields, along with labels that tell the user what to enter.

⇨ Below the <input type="submit">, add the following <label> and <input> fields, for the customer ID, name, address and company fiters. Note that the <label> .. </label> pair completely surrounds the corresponding <input> tag.

```
      <caption>
            <!-- ADD(2) => HTML form inputs for the search filters here -->
            <input type="submit" value="Search On =>"></input>
            <label>Cust ID:
                  <input type="text" name="filt_id" value="<?= $_REQUEST['filt_id'] ?>" />
            </label>
            <label>Name:
                  <input type="text" name="filt_name" value="<?= $_REQUEST['filt_name'] ?>" />
            </label>
            <label>Address:
                  <input type="text" name="filt_addr" value="<?= $_REQUEST['filt_addr'] ?>" />
            </label>
            <label>Company:
                  <input type="text" name="filt_comp" value="<?= $_REQUEST['filt_comp'] ?>" />
            </label>
            <br />
      </caption>
```

Note that each <input> specifies three attributes: type, name, and value.

- All of these are type="text" inputs.
- The name attribute is very important: the values for each input are automatically parsed by PHP and stored in the $_REQUEST array. The key of the $_REQUEST array will be the name attribute specified on the <input>. So in PHP, to retrieve the value entered in the field with `name="filt_id"`, you would use `$_REQUEST['filt_id']`.

- The value attribute is used to preload a value into the input when the page is loaded. Remember that each request/response cycle loads a brand new HTML document, and there is no memory of the previous request, or what was typed into any form on the previously loaded page. So our input values would be lost, if we did not echo the values sent on the request. So within each value attribute we added a PHP snippet to echo the value received and stored in the $_REQUEST array: `name="filt_id" value="<?= $_REQUEST['filt_id'] ?>"`

What Does "Submit the Form" Really Mean?

When we say that clicking the submit button will submit the inputs to the server, this means that the browser will create a request string on our behalf (a URL, which we don't have to type into the address bar), send this request to the server, wait for a response document, and render the response appropriately in the browser window. The browser will take the action attribute of the <form> tag as the file to request (in our case, sfl_02.php). It will add to the request all the input fields and their values as a list of name/value pairs. This list is what PHP parses to populate the $_REQUEST array. The field names/values are sent with the request in one of two formats, depending on the value specified for the method attribute on the <form> tag. In our example, we specified `method="post"`, which means the inputs are sent with the HTTP headers, and we can send a large amount of data to the server. If we had specified method="get", the inputs would be appended to the URL in format like this:

`http://<server-name>:10080/phpsf/phpsfxx/sfl_02/`**`sfl_02.php`**`?filt_id=1351&filt_comp=diver`

A '?' follows the script name, followed by a list of name=value pairs, each pair separated by an '&'. The string following the '?' is called the ***query string***, since it is often supplying filter values for an SQL query. Using method="get" limits the amount of data that can be sent to what fits on a URL, which depends upon the client and server involved, but is usually about 1 or 2 KB of data. This method should only be used for inquiries where a short list of query parameters might be specified. For forms which are updating the server with a large amount of data (for example a blog entry or file upload), using method="post" will ensure complete transfer of all data.

Viewing the Search Form

⇨ Save all your changes
⇨ Go to the browser and refresh (F5) the customer table listing (sfl_02.php)

You should see something like this:

Row#	Cust ID	Name	Address	Company
1	1221	LINA Norman	4-976 Sugarloaf Hwy Suite 103 Kapaa Kauai , HI 94766-1234 US	Kauai Dive Shoppe
2	1231	George Weathers	PO Box Z-547 Freeport , Bahamas	Unisco
3	1351	Phyllis Spooner	1 Neptune Lane Kato Paphos , Cyprus	Sight Diver
4	1354	Joe Bailey	PO Box 541	Cayman Divers World Unlimited

If the format looks like above, with the search fields and button, then you've done everything correctly so far. But the search fields will not have any effect on the records displayed, until you add some PHP code to use them in the SQL query.

Adding PHP Logic to Process the Search Values

- ⇨ Switch to the editor for the .php file
- ⇨ Find the definition for function `getSqlWhere`. It is comprised of a skeleton, with comments indicating where you will add logic for each of the filter values.

```
function getSqlWhere( $screenData ) {

      // Start with a "dummy" where clause. Any additional WHERE conditions can now
      // be appended starting with "AND ..."
      $where = ' WHERE (1=1) ';

      // Filter on customer ID
```

```
        // Filter on customer name

        // Filter on address

        // Filter on company

        return $where;
}
```

⇨ Under the comment "Filter on customer ID", add the follwiong PHP code:

```
        // Filter on customer ID
        if (isset($screenData['filt_id']) && trim($screenData['filt_id']) != '') {
                $where .= " AND CUST_ID = {$screenData['filt_id']} ";
        }
```

Recall that in the mainline you added code to assign the entire $_REQUEST array to $screenData, which we use as a container for both screen inputs and outputs. So we can refer to the $screenData array to find all the form field input values.

We add an if statement to test if the filt_id value was received as input, and we also trim() it (i.e., remove all leading and trailing blanks) to see if a blank value was passed. If a non-blank value was received, we'll append this to the SQL WHERE clause string, specifying the database field name (CUST_ID) = the value received. Here we're using the combined concatenation/assignment operator (.=) to concatenate this where condition on the end of the current string. Writing:

```
        $where .= " AND CUST_ID = {$screenData['filt_id']} ";
```

is the same as writing

```
        $where = $where . " AND CUST_ID = {$screenData['filt_id']} ";
```

If a value of 1351 was passed for filt_id, then our where clause will now be:

```
WHERE (1=1) AND CUST_ID = 1351
```

- ⇨ Save your changes
- ⇨ Go to the browser, and refresh the listing display (F5)
- ⇨ Enter the value 1351 in the customer ID search field.
- ⇨ Click the "Search On =>" button. You should see only one record, for Phyllis Spooner/Site Diver

Now let's add similar logic for the name, address, and company filters. Note that for customer ID we wanted an exact match, so we used the '=' operator in the SQL. For name, address and company, we want to test for the search string anywhere within the database field value, so we will use the sql LIKE operator, together with the wildcard symbol '%'. We will also convert both the database field value and the search value to upper case (using the PHP core function, strtoupper, and the db2 sql function, upper) so that the search will be performed case-insensitive. For the address search, we'll concatenate all of the address related fields into a single string and then perform the LIKE search on the complete address string.

- ⇨ Add the PHP code below, under the appropriate comments

```
      // Filter on customer name
      if (isset($screenData['filt_name']) && trim($screenData['filt_name']) != '') {
            $upperFiltName = strtoupper(trim($screenData['filt_name']));
            $where .= " AND upper(FIRSTNAME || ' ' || LASTNAME) LIKE '%{$upperFiltName}%' ";
      }

      // Filter on address
      if (isset($screenData['filt_addr']) && trim($screenData['filt_addr']) != '') {
            $upperFiltAddr = strtoupper(trim($screenData['filt_addr']));
            $where .= " AND upper(ADDRESS  || ' ' || ADDR2   || ' ' || CITY   || ' ' ||
                                          STATE  || ' ' || ZIP  || ' ' || COUNTRY)
                                    LIKE '%{$upperFiltAddr}%' ";
      }

      // Filter on company
      if (isset($screenData['filt_comp']) && trim($screenData['filt_comp']) != '') {
```

```
            $upperFiltComp = strtoupper(trim($screenData['filt_comp']));
            $where .= " AND upper(COMPANY) LIKE '%{$upperFiltComp}%' ";
        }
```

Running the Finished Search Listing

- ⇨ Save all your changes
- ⇨ Go to the browser, and refresh the listing display (F5)
- ⇨ Enter various values for the search fields and click the "Search On =>" button.
- ⇨ Verify that the searches are working properly.