PHP / MYSQL – PART 2

Before starting the other tasks in this tutorial ensure that you have a connection file above the route of your site. We'll add this file to all the pages that query the MySQL database.

In this lab we're going to build a simple Content Management System (CMS).

DRILLING DEEPER USING QUERY STRINGS

Open the file *searchPage.php*. This file uses a prepared statement to query the 'movies' table based on film name. We now want to extend this example such that the list of results, appear as hyperlinks to a page saved as *details.php*. The *details.php* page will display all the details about a chosen film.

In order to do so we will create a hyperlinks around the search results that have querystrings taking name/value pairs to the *details.php* page. The best value to send to the *details.php* page would be that of the primary key. As such you need to amend the prepared statement such that the primary key 'filmID' is extracted as well as the 'filmName' which is currently the only field retrieved.

Amend the PHP at the top of the file to extract the 'filmID' field.

```
// prepare SQL
$stmt = $mysqli->prepare("SELECT filmID, filmName FROM movies
WHERE filmName LIKE ?");
$stmt->bind_param('s', $searchString);
$stmt->execute();
$stmt->bind_result($filmID, $filmName);      $stmt-
>store_result();
$myNumRows = $stmt->num_rows;?>
```

Amend the code in the PHP while loop that produces the output to add a hyperlink to the details page as follows:

```
echo "<li><a
href=\"details.php?filmID={$filmID}\">{$filmName}</a></li>";
```

Open the *details.php* page. Notice how this uses a `mysqli` prepared statement to query the database based on the primary key of the 'movies' table.

There is an error in *details.php*. It is using the wrong super global. Change the `bind_param` to use the value from the querystring.

IMPROVING SECURITY

To fix the issue in *details.php* you should have changed the `$_POST` to a `$_GET`.

This will work, however receiving a variable via a URL can be used by hackers to try and inject Javascript into your database. As such we should add some security.

Open the page *cms/includes/functions.inc.php*. This file includes a number of functions that use PHPs `filter_var()` function to cleanse the data. We could extend what we have here to also filter floats by adding:

```
function safeFloat($float){
     return filter_var($float,
╚FILTER_SANITIZE_NUMBER_FLOAT,FILTER_FLAG_ALLOW_FRACTION);
      }
```

Notice there is already a function build for integers called `safeInt()`. As the SQL query in *details.php* expects an integer representing a possible value of 'filmID' the primary key in 'movies' then we should check that the value that comes from the user in the `$_GET` is indeed a integer.

Amend the query in *details.php* by adding a reference to safeInt(). We could create a 'safe' variable, prefixed with a 's' as follows:

```
safeInt($_GET['filmID']);
```

Then make sure that this new variable is used in the bind_param as opposed to a direct reference to `$_GET`.

Tip: By placing these validation functions in an include we can easily add it to other pages and perhaps extend its offerings.

DISPLAYING IMAGES DYNAMICALLY

In order to display the image in the *details.php* page add a line of PHP that makes use of the 'filmImage' field in the database. You will need to extract 'filmImage' via the SQL and bind it to a results variable.

Amend your code as follows:

```
echo "<img src=\"images/{$filmImage}\" class=\"rightImg\">";
```

Extend the code above to automatically add an ALT attribute that has the value of the film name.

Our database also has a review field 'filmReview' that is an integer value between 0 and 5. There is a sub-directory of images, one for each of the possible star ratings.

Extract 'filmReview' via the SQL and bind it to a suitably named variable. Then add the following to the PHP controlling the output of HTML.

```
echo "<p><img src=\"images/stars/star-rating-
↳{$filmReview}.png\"</p>";
```

When testing your page in the browser try changing the numeric value of filmID which is exposed in the URL. You should be able to retrieve different records by just changing this value.

BUILDING A SIMPLE CONTENT MANAGEMENT SYSTEM

Notice there is a link to *cms/index.php* in both *searchPage.php* and *details.php*. This is where we'll build our CMS. For now this is not protected, but you could use techniques such as PHP sessions to restrict access to this page.

Open the file *index.php*. This file lists all the films in the 'movies' database.

Amend the SQL so that they are listed alphabetically.

This page will form the basis of a simple Content Management System.

In the table amend the links to the 'details.php', the 'edit.php' and the 'delete.php' PHP pages to pass the primary value of 'filmID' to them via a querystring. Your amended code should look as follows:

```
echo "<tr>";
echo "<td>{$rowFilms['filmName']}</td>";
echo "<td><a
href=\"details.php?filmID={$rowFilms['filmID']}\">View</a></t
d>";
echo "<td><a
href=\"edit.php?filmID={$rowFilms['filmID']}\">Edit</a></td>"
;
echo "<td><a
href=\"delete.php?filmID={$rowFilms['filmID']}\">Delete</a></
td>";
echo "</tr>";
```

UPDATING A RECORD

Open the PHP page *edit.php*. Notice that this contains an SQL query based on the primary key value of 'filmID'. The results from this query are then used to populate the form fields with the current values for that film. The HTML form's action attribute points the form at a file called *process/editRecord.php*. You need to create this file.

```
<?php
require('../../../../includes/conn.inc.php');
 require('../includes/functions.inc.php');
// sanitize user variables
$sFilmName = safeString($_POST['filmName']);
$sFilmDescription = safeString($_POST['filmDescription']);
$sFilmImage = safeString($_POST['filmImage']);
$sFilmPrice = safeFloat($_POST['filmPrice']);
$sFilmReview = safeInt($_POST['filmReview']);
$sFilmID = safeInt($_POST['filmID']);
// prepare SQL
$stmt = $mysqli->prepare("UPDATE movies SET
                                    filmName = ?,
                                    filmDescription = ?,
                                    filmImage = ?,
                                    filmPrice = ?,
                                    filmReview = ?
                                    WHERE filmID = ?");
 $stmt->bind_param('sssdii', $sFilmName,
     $sFilmDescription,
     $sFilmImage,
     $sFilmPrice,
     $sFilmReview,
     $sFilmID);
$stmt->execute();
$stmt->close();
header("Location: ../index.php");
// redirect browser
exit; // make sure no other code executed
?>
```

This file uses the connection include and then uses the mysqli prepare method to build a SQL query to send to MySQL. Notice the use of 'sssdii' indicating that the values to be sent to the query are string, string, string, decimal, integer, integer.

This page is not designed to be viewed but is a 'process' file that will edit the record and then 'bounce' the user back to the *index.php* page through the use of a the PHP header function.

INSERTING A RECORD

Open the file *insert.php* file. This contains a HTML form with all the required fields for the 'movies' table. Notice that the form does not include a field for 'filmID'. This is because the 'filmID' field in the 'movies' table is set up as auto-increment. Therefore a unique primary key value will be allocated to any new entry automatically.

Notice that the action of the form is to post the values to a file called *process/insertRecord.php*. The *insertRecord.php* file will be in the same vein as the *editRecord.php* example above. That is it is a file that will process the necessary SQL but will not produce any HTML to be displayed and will return the user to the index.php page.

Create the file *insertRecord.php*. You will find it useful to review *editRecord.php* and the way the prepare statement is created. You will also need to know the syntax of a SQL INSERT statement.

```
INSERT INTO tablename(field1, field2, field3)
VALUES (value1, value2, value3)
```

Remember you don't need to insert a value for 'filmID' the primary key.

All the data coming from the form in *insert.php* will also need to be filtered. Ensure you sent all these values through the functions in *functions.inc.php*.

DELETING A RECORD

Review the file *delete.php*. This file will be linked to from the *index.php* file. The link you added previously will pass the primary key of the chosen film to the *delete.php* file.

The *delete.php* file needs to query the database to retrieve the chosen film's name and display it to the user so they can decide whether to delete the record.

Inside *delete.php* file there are two HTML forms. The first form is designed to send the film's 'filmID' primary key to the *process/deleteRecord.php* file. This form uses the more secure POST method.nnThe second form will just return the user to the *index.php* file without further action.

Edit the first form to add a hidden field for the 'filmID" value to ensure the current 'filmID' is sent to the *deleteRecord.php* file.

The code required is as follows:

```
<input name="filmID" type="hidden" value="<?php echo
╚$filmID; ?>">
```

You also need to create the file *process/deleteRecord.php*. This is in the same vein as *process/editRecord.php* and *process/insertRecord.php* but will use the SQL `DELETE` statement to remove a record. You code should appear as follows:

```
<?php
require('../../../../includes/conn.inc.php');
require('../includes/functions.inc.php');
// sanitize user variables
$sFilmID = safeInt($_POST['filmID']);
// prepare SQL
$stmt = $mysqli->prepare("DELETE FROM movies WHERE filmID
╚= ?");
$stmt->bind_param('i', $sFilmID);
$stmt->execute();
$stmt->close();
header("Location: ../index.php");
// redirect browser
exit;
// make sure no other code executed
?>
```