

OtterMart: Admin Site

In this tutorial we will create a set of web pages that will allow an OtterMart Administrator to add, update, and delete products. These pages will be password-protected, so administrators will have to log in using a username and password.

Here is a completed version of the project: <http://mmensinger-cst336.herokuapp.com/labs/Lab7/> (the username is “admin” and the password is “secret”, without the quotes) Please DO NOT remove products!

As we work in this project, we will practice using:

- HTML Sessions
- SQL INSERT
- SQL UPDATE
- SQL DELETE

Planning!

Take a moment to think about our goals for the project and how we would go about achieving them.



1. How will administrators authenticate? Where to store their username and password?
2. How to prevent access to the admin pages to anyone who hasn't authenticated yet?
3. How to allow administrators to logout?

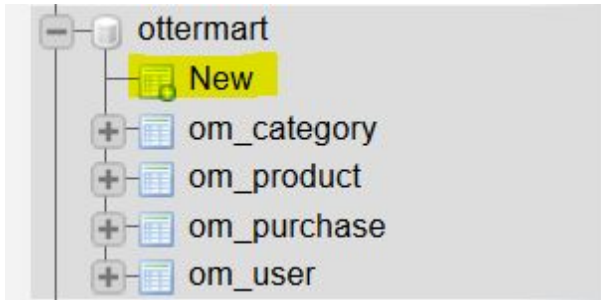
Optionally, you can watch the [accompanying video tutorial](#) that Matt Mensinger has graciously created for all of you!

Lesson 1: Adding an Admin Table

1.1 Create the “Admin” table

We're getting ready to authenticate the website for an admin. The admin of the site will be able to update, insert, and delete items. Therefore, a login needs to be implemented.

Go to phpMyAdmin. Select the ottermart database. In the dropdown, click on “New.”



Create a new table called: **om_admin**.

Table name: Add column(s)

Structure								
Name	Type	Length/Values	Default	Collation	Attributes	Null	Index	A.I.
adminId	TINYINT	4	None			<input type="checkbox"/>	PRIMARY	<input checked="" type="checkbox"/>
firstName	VARCHAR	25	None			<input type="checkbox"/>	---	<input type="checkbox"/>
lastName	VARCHAR	25	None			<input type="checkbox"/>	---	<input type="checkbox"/>
username	VARCHAR	8	None			<input type="checkbox"/>	---	<input type="checkbox"/>
password	VARCHAR	40	None			<input type="checkbox"/>	---	<input type="checkbox"/>

Make sure that there are 5 columns with the data as shown above.

1.2 Review the table structure

Review the field names, data types, and length. The fields include:

the admin's ID (**adminId**) (This should be the Primary Key, it is Auto-Incremented).

the admin's first name (**firstName**), the admin's last name (**lastName**),

the admin's username (**username**), and

the password (**password**). Make sure to use a length of 40. The password will never be stored as plain-text. It will be encrypted using SHA1, which requires 40 bytes.

1.3 Add a record to the "om_admin" table

Let's go ahead and create a new record using the following info:

Column	Type	Function	Null	Value
adminId	tinyint(4)	<input type="text"/>	<input type="checkbox"/>	<input type="text"/>
firstName	varchar(25)	<input type="text"/>	<input type="checkbox"/>	Admin <input type="button" value="🔑"/>
lastName	varchar(25)	<input type="text"/>	<input type="checkbox"/>	Admin <input type="text"/>
username	varchar(8)	<input type="text"/>	<input type="checkbox"/>	admin <input type="text"/>
password	varchar(40)	SHA1 <input type="text"/>	<input type="checkbox"/>	s3cr3t <input type="text"/>

The value for **adminId** is left blank on purpose. Remember, it's auto incrementing and it will assign a value on its own.

Passwords need to be encrypted. In order to do this, the function **SHA1** is selected. This function will take our password value, s3cr3t, and encrypt it before storing. The encrypted value is what will be stored in the table.

adminId	firstName	lastName	username	password
1	Admin	Admin	admin	fa3216628b5474393960f325d09a47f1815214e4

adminId was assigned the number 1 (one) since it was the first record. The next record should be a two since it will increment on its own.

Notice that the password value is not s3cr3t. As mentioned above, **SHA1** took care of the encryption. This is also the reason why the varchar was set to 40.

Note: Once you complete the lab, **DO NOT** forget to export the new "om_admin" table to the Heroku ClearDB database, otherwise, your login process won't work!

Lesson 2: Creating the Authentication Process

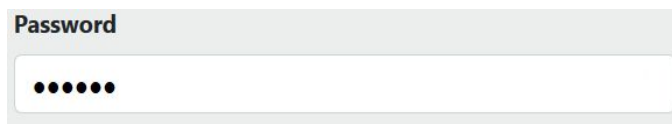
2.1 Add the login form in the “login.php” file

```
24 <form method="POST" action="loginProcess.php">
25     Username: <input type="text" name="username"/> <br />
26     Password: <input type="password" name="password"/> <br />
27
28     <input type="submit" name="submitForm" value="Login!" />
29 </form>
```

The login form should display text fields for the username and password.

Notice that the POST method is being used (line 24). The account information should not be revealed in any way. The GET method would do just that by showing the values in the URL once the form gets submitted.

The input tag for password has the type value of **password** (line 26). This prevents the password from being shown. Instead, bullets are used to replace characters.



2.2 Writing the login Process in the “loginProcess.php” file

Create a new blank file called “**loginProcess.php**”

Include the **dbConnection.php** adjusting the path accordingly, based off how your project is setup (for instance, in the screenshot below, you’ll notice that in line 7 the path to the “dbConnection.php” file is “../../dbConnection.php”. This means that the file is located two folders above the loginProcess.php file).

We need to encrypt the password submitted by the user using the “sha1()” function. This will allow us to compare the value submitted by the user to the encrypted password stored in the database (see line 12 in the screenshot below).

The **WRONG approach** to write the SQL statement to check for the username and password is to surround the values with single quotes, as it’s been done in lines 20 and 21 in the screenshot below.

```

7      include '../../dbConnection.php';
8
9      $conn = getDatabaseConnection("ottermart");
10
11     $username = $_POST['username'];
12     $password = sha1($_POST['password']);
13
14     //echo $password;
15
16
17     //following sql does not prevent SQL injection
18     $sql = "SELECT *
19           FROM om_admin
20           WHERE username = '$username'
21           AND password = '$password'";

```

Bad Practice!

Using single quotes as part of the SQL statement allows for SQL injection! Anyone could inject malicious code and gain authorized access to the database.



You should never use single quotes as part of the SQL statement.

An example of SQL injection is the following value as the username: ' or 1 = 1 or ' (notice the starting and ending single quotes). The "or" logical operator makes the condition always true, regardless of the value of the password. So, users could bypass the authentication process.

Username

Password

To avoid using single quotes as part of the SQL statement, it is always better to bind named parameters. Replace lines 17 - 21 with the following code shown in the next screenshot.

```
17 // following sql prevents sql injection by avoiding using single quotes
18 $sql = "SELECT *
19     FROM om_admin
20     WHERE username = :username
21     AND password = :password";
22
23 $np = array();
24 $np["username"] = $username;
25 $np["password"] = $password;
26
27
28 $stmt = $conn->prepare($sql);
29 $stmt->execute($np);
30 $record = $stmt->fetch(PDO::FETCH_ASSOC); //expecting one single record
```

Notice that in lines 23 to 25 we are creating an associative array (\$np) and assigning the actual values to each named parameter.

In line 29, we are passing the associative array when calling the “execute” method. MySQL needs these values in order to process the query correctly. So, don’t forget to always pass the associative array as part of the “execute” method when using named parameters.

If the admin’s records are found then the user would be redirected to the admin page, else the user will get an error.

```
34 if (empty($record)) {
35     $_SESSION['incorrect'] = true;
36     header("Location:login.php");
37 } else {
38     //echo $record['firstName'] . " " . $record['lastName'];
39     $_SESSION['incorrect'] = false;
40     $_SESSION['adminName'] = $record['firstName'] . " " . $record['lastName'];
41     header("Location:admin.php");
42 }
43
```

We are using **session variables** to store the administrator’s name and being able to display it in other pages. Another session variable is used to keep track of whether the authentication process was successful or not. Since we are using session variables, go ahead and start the session at the beginning of the file using `session_start()`:

```
3 session_start();
```

Lesson 3: Displaying “Wrong Credentials” Error

The user may not have entered the correct data to login. Therefore, they should be directed to the login form again and get a warning.

3.1 Updating the “login.php” file

Insert the following condition (lines 35 to 40) in the **login.php** file.

```
33 <input type="submit" class = 'btn btn-primary' name="submitForm" value="Login!" />
34 <br /><br />
35 <?php
36     if($_SESSION['incorrect']){
37         echo "<p class = 'lead' id = 'error' style='color:red'>";
38         echo "<strong>Incorrect Username or Password!</strong></p>";
39     }
40     ?>
41 </form>
```

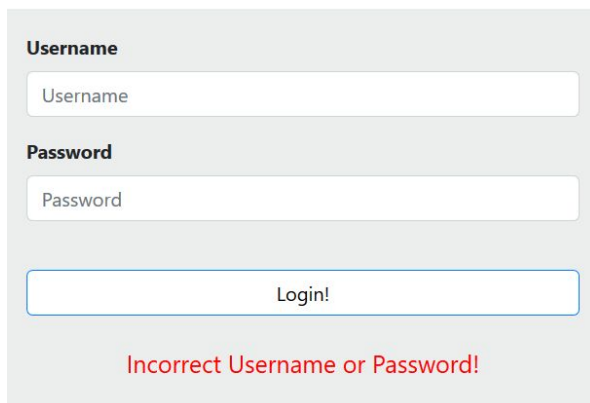
3.2 Starting the session

Do not forget to start the session in login.php.

```
1 <?php
2     session_start();
3     ?>
```

3.3 Testing the error message.

Try logging in using a wrong username or password. You should be able to see the error in red indicating that either their username or password was wrong.



The screenshot shows a login form with a light gray background. It contains two input fields: 'Username' and 'Password', both with placeholder text. Below the password field is a 'Login!' button. At the bottom of the form, a red error message 'Incorrect Username or Password!' is displayed.

Lesson 4: Displaying the “Admin” Main Page

4.1 Verifying that the user has authenticated

Create a new file called “**admin.php**” and add the following condition at the top (within a PHP block)

```
3 session_start();
4 if(!isset( $_SESSION['adminName']))
5 {
6     header("Location:login.php");
7 }
```

Without this condition, any user would be able to see the content of the “admin.php” file, even without having gone through the login process! This condition checks whether the session variable `$_SESSION['adminName']` exists (is set). It should only exist if the user entered the right admin credentials. If it does not exist, it redirects the user to the “login.php” file (the login screen).

Note: All the admin pages should verify that an admin has logged in. You don’t want users to access these pages just by typing directly the URL.

4.2 Displaying all products from the database

Once an admin has been authenticated, they will be able to add, update, delete, and logout in “admin.php”.

First let’s show the available items in the store. The following SELECT query retrieves the data for all products in the database. Put this code in the “admin.php” file:

```
11 function displayAllProducts() {
12     global $conn;
13     $sql="SELECT * FROM om_product ORDER BY productId";
14     $statement = $conn->prepare($sql);
15     $statement->execute();
16     $records = $statement->fetchAll(PDO::FETCH_ASSOC);
17
18     return $records;
19 }
```


Call the “displayAllProducts()” function and iterate through each product to display it:

```
62 <?php $records=displayAllProducts();
63 echo "<table class='table table-hover'>";
64 echo "<thead>
65     <tr>
66         <th scope='col'>ID</th>
67         <th scope='col'>Name</th>
68         <th scope='col'>Description</th>
69         <th scope='col'>Price</th>
70         <th scope='col'>Update</th>
71         <th scope='col'>Remove</th>
72     </tr>
73 </thead>";
74 echo"<tbody>";
75 foreach($records as $record) {
76     echo "<tr>";
77     echo "<td>" . $record['productId'] . "</td>";
78     echo "<td>" . $record["productName"] . "</td>";
79     echo "<td>" . $record["productDescription"] . "</td>";
80     echo "<td>" . $record["price"] . "</td>";
81     echo "<td><a class='btn btn-primary' href='updateProduct.php?productId=" . $record['productId'] . "'>Update</a></td>";
82
83     echo "<form action='deleteProduct.php' onsubmit='return confirmDelete()'>";
84     echo "<input type='hidden' name='productId' value= " . $record['productId'] . " />";
85     echo "<td><input type='submit' class = 'btn btn-danger' value='Remove'></td>";
86     echo "</form>";
87
88 }
89 echo"</tbody>";
90 echo"</table> ";
91 ?>
```

The setup is straightforward. Each record will show the product ID, name, description, price, and update and delete button. The buttons will not work at the moment since their corresponding files have yet to be created.

ID	Name	Description	Price	Update	Remove
1	Sceptre 32	"Sceptre 32"" Class HD (720P) LED TV (X322BV-SR)"	\$89.99	<button>Update</button>	<button>Remove</button>
2	Happy Feet Two	Happy Feet Two: The Videogame - Nintendo DS	\$12.69	<button>Update</button>	<button>Remove</button>

Lesson 5: Adding a new Product

5.1 Displaying “Add Product” button

Add the following form in the “admin.php” file. The button should be located above the list of all products.

```
50 <form action="addProduct.php">
51   <input type="submit" class = 'btn btn-secondary' id = "beginning" name="addproduct" value="Add Product"/>
52 </form>
```

Notice that the “action” of the form (line 50) is “addProduct.php”. In other words, this file will be opened when clicking on the button.

5.2 Creating the Form to add products

Create and open the “addProduct.php” file. This file will display a form like this:

Product name

Description

Price

Category

Select One

Set Image Url

Add Product

The form must have fields for product name, description, price, category, image URL, and a submit button. Here is the code:

```
60 <form>
61     <strong>Product name</strong> <input type="text" class="form-control" name="productName"><br>
62     <strong>Description</strong> <textarea name="description" class="form-control" cols = 50 rows = 4></textarea><br>
63     <strong>Price</strong> <input type="text" class="form-control" name="price"><br>
64     <strong>Category</strong> <select name="catId" class="form-control">
65         <option value="">Select One</option>
66         <?php getCategories(); ?>
67     </select> <br />
68     <strong>Set Image Url</strong> <input type = "text" name = "productImage" class="form-control"><br>
69     <input type="submit" name="submitProduct" class='btn btn-primary' value="Add Product">
70 </form>
```

5.3 Displaying the Category List within the Form

As past of the form to add a new product, the administrator needs to select the product category. Instead of hardcoding the list, we'll get it from the database itself. To do that, we need to implement the "getCategories()" function that we are calling in line 66 (within the "Select" tag)

A SELECT query from the table om_category will grab all the categories to be displayed under the "Select" tag as options. Here is the code:

```
10 function getCategories() {
11     global $conn;
12
13     $sql = "SELECT catId, catName FROM om_category ORDER BY catName";
14
15     $statement = $conn->prepare($sql);
16     $statement->execute();
17     $records = $statement->fetchAll(PDO::FETCH_ASSOC);
18     foreach ($records as $record) {
19         echo "<option value='". $record["catId"] .' '>". $record['catName'] ." </option>";
20     }
21 }
```

5.4 Adding a new product in the database

A new product should be stored in the database ONLY after the user has clicked on the "Add Product" button. This line of code created the button:

```
<input type="submit" name="submitProduct" value="Add Product" >
```

We can then check whether `$_GET['submitProduct']` is set. It'd be set only if the form was submitted. If it is set, then we execute an INSERT SQL statement using named parameters.

```
23 if (isset($_GET['submitProduct'])) {
24     $productName = $_GET['productName'];
25     $productDescription = $_GET['description'];
26     $productImage = $_GET['productImage'];
27     $productPrice = $_GET['price'];
28     $catId = $_GET['catId'];
29
30     $sql = "INSERT INTO om_product
31         ( productName, productDescription, productImage, price, catId)
32         VALUES ( :productName, :productDescription, :productImage, :price, :catId)";
33
34     $namedParameters = array();
35     $namedParameters[':productName'] = $productName;
36     $namedParameters[':productDescription'] = $productDescription;
37     $namedParameters[':productImage'] = $productImage;
38     $namedParameters[':price'] = $productPrice;
39     $namedParameters[':catId'] = $catId;
40     $statement = $conn->prepare($sql);
41     $statement->execute($namedParameters);
42 }
```

After adding the new product, you could display a message letting the user know that the product was added successfully.

Notes:

Notice that we shouldn't use "fetch" or "fetchAll" when executing an INSERT statement. They are only used when executing a SELECT statement.

5.5 Checking that new products are added to the database

The new products should be listed in the main admin page, along with the other products. You should also be able to see them directly in the database using phpMyAdmin.

Lesson 6: Updating a Product

6.1 Reviewing the link to Update a product

Let's look back at the link to update products. The code is located in the **"admin.php"** file, specifically, in line 81 of the screenshot below:

```
62 <?php $records=displayAllProducts();
63 echo "<table class='table table-hover'>";
64 echo "<thead>
65     <tr>
66         <th scope='col'>ID</th>
67         <th scope='col'>Name</th>
68         <th scope='col'>Description</th>
69         <th scope='col'>Price</th>
70         <th scope='col'>Update</th>
71         <th scope='col'>Remove</th>
72     </tr>
73 </thead>";
74 echo"<tbody>";
75 foreach($records as $record) {
76     echo "<tr>";
77     echo "<td>" . $record['productId'] . "</td>";
78     echo "<td>" . $record["productName"] . "</td>";
79     echo "<td>" . $record["productDescription"] . "</td>";
80     echo "<td>$" . $record["price"] . "</td>";
81     echo "<td><a class='btn btn-primary' href='updateProduct.php?productId=" . $record['productId'] . "'>Update</a></td>";
82 }
```

We are using an “anchor tag” to create the link:

```
<a href='updateProduct.php?productId=".$record['productId']."> Update </a>
```

Notice that the **productId** is being sent as part of the URL. The product ID will allow the **updateProduct.php** program to know which item the admin wants to update. Since it's being passed in the URL, it can be retrieved using the **\$_GET** array.

6.3 Displaying the form for the product to update

Upon clicking the update link, a form will appear on a new page (see screenshot below). Notice that the form is very similar to the “Insert a New Product” form, but it has the actual product values prefilled.

Product name

Sceptre 32

Description

"Sceptre 32"" Class HD (720P) LED TV (X322BV-SR)"

Price

89.99

Category

Electronics

Set Image Url

https://i5.walmartimages.com/asr/cd51992c-b3d6-4aad-be53-07e9fee0d01a_1.f2b4

Update Product

Here is the code to create the form. It's very similar to the code used to Add a New product. The only difference is adding the **"value"** attribute on each form element. The **"value"** attribute will allow us to prefill the form with the product's information. The variable **\$product** hasn't been initialized yet, it will contain all the information about the product.

```

87 form>
88 <input type="hidden" name="productId" value="<?=$product['productId']?>" />
89 <strong>Product name</strong> <input type="text" class="form-control" value = "<?=$product['productName']?>" name="productName"><br>
90 <strong>Description</strong> <textarea name="description" class="form-control" cols = 50 rows = 4><?=$product['productDescription']?></textarea><br>
91 <strong>Price</strong> <input type="text" class="form-control" name="price" value = "<?=$product['price']?>"><br>
92
93 <strong>Category</strong> <select name="catId" class="form-control">
94   <option>Select One</option>
95   <?php getCategories( $product['catId'] ); ?>
96 </select> <br />
97 <strong>Set Image Url</strong> <input type = "text" class="form-control" name = "productImage" value = "<?=$product['productImage']?>"><br>
98 <input type="submit" class='btn btn-primary' name="updateProduct" value="Update Product">
99 /form>

```

It is best to verify that the product ID (productId) was successfully sent over from **admin.php** (in other words, that it's part of the URL). A simple **isset()** function can do the verification. If so, then we can set the value of **\$product** to all the information about the product selected.

```
62 if (isset ($_GET['productId'])) {  
63     $product = getProductInfo();  
64 }
```

A SELECT query will be used to implement the getProductInfo() function and retrieve all info about the product:

```
21 function getProductInfo()  
22 {  
23     global $connection;  
24     $sql = "SELECT * FROM om_product WHERE productId = " . $_GET['productId'];  
25  
26     $statement = $connection->prepare($sql);  
27     $statement->execute();  
28     $record = $statement->fetch(PDO::FETCH_ASSOC);  
29  
30     return $record;  
31 }
```

Notice that the code is not using named parameters because there are no single quotes as part of the SQL statement, thus, there is no risk for SQL injection. No single quotes are needed because productId is an integer.

6.4 Displaying the Category List

Similar to the “Add a New Product” form, we want to display the list of categories from the database instead of hardcoding the category names. However, this time we also want to display as selected the corresponding category of the product. For that reason, we need to make a few changes to the “getCategories()” function

Here is the code:


```

6 function getCategory($catId) {
7     global $connection;
8
9     $sql = "SELECT catId, catName from om_category ORDER BY catName";
10
11     $statement = $connection->prepare($sql);
12     $statement->execute();
13     $records = $statement->fetchAll(PDO::FETCH_ASSOC);
14     foreach ($records as $record) {
15         echo "<option ";
16         echo ($record["catId"] == $catId)? "selected": "";
17         echo " value='". $record["catId"] .'>". $record['catName'] ." </option>";
18     }
19 }

```

Under the *foreach* loop, there is a ternary operator in line 16 that checks whether each category id matches the product's category (passed as an argument to the function). If both of them match, then it displays the attribute **selected**. This attribute is used as part of the `<select>` tag to display a particular option as selected.

The correct category should be shown now in the dropdown as selected.

Test your code by clicking on the “Update” link of any product in the “admin.php” page. You should be taken to the `updateProduct.php` file and you should be able to see the form with most of the data prefilled.

6.5 Updating the product info

Based off the form for **updateProduct.php**, the form's submission will go straight to **updateProduct.php** again since there was no “*action*” attribute.

We need to check whether the GET variable **updateProduct** has been set, in other words, we'll check whether the administrator has clicked on the “Update” button. If so, we'll need to execute an UPDATE SQL statement using named parameters.

The code to Update the database must be placed **above** the condition that checks whether “productId” is set and assigns the product information to the **\$product** variable.

```

62 if (isset ($_GET['productId'])) {
63     $product = getProductInfo();
64 }

```

```

34     if (isset($_GET['updateProduct'])) {
35
36         $sql = "UPDATE om_product
37             SET productName = :productName,
38                 productDescription = :productDescription,
39                 productImage = :productImage,
40                 price = :price,
41                 catId = :catId
42             WHERE productId = :productId";
43         $np = array();
44         $np[":productName"] = $_GET['productName'];
45         $np[":productDescription"] = $_GET['description'];
46         $np[":productImage"] = $_GET['productImage'];
47         $np[":price"] = $_GET['price'];
48         $np[":catId"] = $_GET['catId'];
49         $np[":productId"] = $_GET['productId'];
50
51         $statement = $connection->prepare($sql);
52         $statement->execute($np);
53         echo "Product has been updated!";
54
55     }

```

Test the Update feature. You should be able to update any product and the updated information should be displayed in the form.

Lesson 7: Deleting a Product

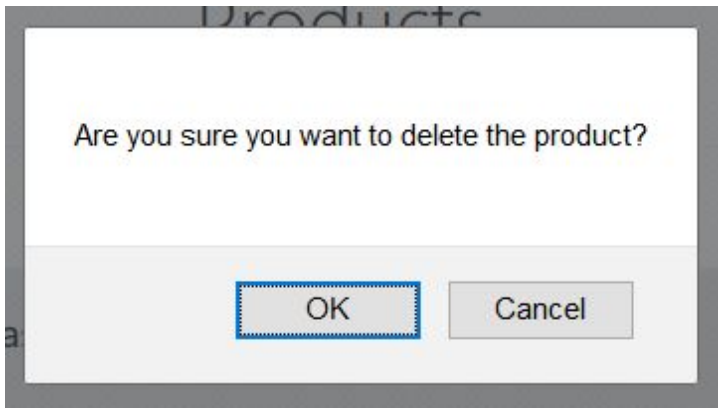
7.1 Confirming product deletion

The deletion is based off a form instead of a link. Here is the form code from the **admin.php** file:

```
83     echo "<form action='deleteProduct.php' onsubmit='return confirmDelete()>";
84     echo "<input type='hidden' name='productId' value= \" . $record['productId'] . \" />";
85     echo "<td><input type='submit' class = 'btn btn-danger' value='Remove'></td>";
86     echo "</form>";
```

Notice that similar to the “Update” link, we also need to pass the productId for the program to know which product we are clicking on. However, instead of passing the productId in the URL, we are passing it using a “hidden” form element (line 84).

We are also using an “onsubmit” JavaScript event. We want to display a confirmation box that asks users to confirm the deletion of the product:



The confirmation box is triggered by calling a JavaScript function that we are calling “confirmDelete()”, but it could be called anything. All JavaScript code must be included within the <script></script> tags. Add this JavaScript code:

```
32 <script>
33     function confirmDelete() {
34
35         return confirm("Are you sure you want to delete the product?");
36     }
37 }
38 </script>
```

Notes:

The **confirm()** JavaScript function displays a confirmation dialog box with two buttons: OK and Cancel. It returns true upon selecting OK and false upon selecting Cancel.

If the admin decides to confirm the deletion, then the form will send its data to **deleteProduct.php**. In this case, the `productId` is the only data sent over that is needed to delete the product from the database.

7.2 Deleting a product

The product can be deleted using its ID and the DELETE query. The admin should be sent over to **admin.php** to continue maintaining the website upon a successful deletion. Here is the PHP code to be added in the “deleteProduct.php” file (don’t forget to add at the top the code to verify whether the administrator has already authenticated)

```
7  $sql = "DELETE FROM om_product WHERE productId = " . $_GET['productId'];
8  $statement = $connection->prepare($sql);
9  $statement->execute();
10
11 header("Location: admin.php");
```

Lesson 8: Logging out

8.1 Adding a Logout button

Admins should have the option to logout. Otherwise, non-admins might be able to use their session. Here is a button to be added in the “admin.php” file:

```
54 <form action="logout.php">
55     <input type="submit" class = 'btn btn-secondary' id = "beginning" value="Logout"/>
56 </form>
```

8.2 Destroying the session (logging out)

Here is the code to destroy the session. It should be added in the “logout.php” file

```
1 <?php
2
3 session_start();
4
5 session_destroy();
6
7 header("Location: index.php");
8
9 ?>
```

Note: Once again, after completing this lab in C9, DO NOT forget to export the new “om_admin” table to the Heroku ClearDB database!

