The Product Viewer application

Please try to run the code (`product_viewer` folder) on your browser (in localhost) and study the code along with this supporting material.

The user interface

The primary page of **the Product Viewer application** is divided into two parts. The sidebar on the left displays a list of category links that let the user select a category. The main portion of the page lists the products for that category in an HTML table.

When the application starts, it displays the products in the category named *Guitars*. Then, if the user wants to view the products in another category, the user can click on the link for that category. If, for example, the user clicks on the *Basses* link, this application displays the products in that category. To facilitate that, the `category_id` field is sent with the HTTP request for the same page. This is illustrated by the URL when the user clicks on the *Basses*, which requests the default page in the product_viewer directory.

To make it easy to identify columns and rows for the product data, this application uses CSS to format the HTML table. This formatting adds a solid border around the outside of the table and a dashed border between the columns and rows.

The code

The ***database.php*** file contains the code that creates the PDO object and stores it in the variable named `$db`. This file is executed by the ***index.php*** file.

However, if an error occurs when this file is executed, the catch block stores the error message in a variable named `$error_message`. Then, the catch block uses the include function to include the ***database_error.php*** file, which displays a page that contains the error message. Last, the catch block exits the script.

Note in the ***database.php*** file that the username for the connection is "`mgs_user`" and the password is "`pa55word`". For this connection to work, then, a user with this username and password must be stored with the MySQL database. In fact, this user is created by the SQL script that creates the databases for this book. Also, this user is given the privileges that are needed for running `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements against any table in the `my_guitar_shop1` database. As a result, this connection will work for this application.

The ***index.php*** file contains the code that displays the product data. Its first statement is a `require()` function that executes the code that's stored in the ***database.php*** file on the previous page. This sets up the `$db` variable so it's available for the other statements in this file.

After that, the first block of code gets the category ID for the category link that the user has clicked on. To do that, the first statement gets the category ID from the global `$_GET` array using the `filter_input()` function.

Note that the `filter_input()` function uses the `FILTER_VALIDATE_INT` constant. As a result, the `filter_input()` function returns `NULL` if the URL variable doesn't exist or `FALSE` if it contains anything other than an integer. This helps protect against malicious input, such as the user manually changing the URL.

When the application starts, the user hasn't yet clicked on a category ID. As a result, there is no URL variable and the category ID will be NULL. In this case, or in the case of bad or missing data as described above, the if statement sets the category ID to a default value of 1.

The second block of code gets the name for the current category. To do that, this code defines a `SELECT` statement that gets the row for the category with the specified ID. Next, it prepares and executes the `SELECT` statement, which creates a `PDOStatement` object. Then, it uses the `fetch()` method of that object to create an array for the only row in the result set that's returned by the `SELECT` statement. Next, it gets the name of the category from that array. Last, it closes the connection to the server.

The third block of code gets all categories and stores them in the `$categories` variable. Here, the first statement defines a `SELECT` statement that gets all columns and rows from the categories table. Then, the next four statements prepare and execute the `SELECT` statement, store the array for the result set in the `$categories` variable, and close the connection to the server. The fourth block of code works like the third block of code. However, the first statement in this block defines a `SELECT` statement that gets all columns and rows from the products table for the specified category ID. Then, the next five statements prepare and execute the `SELECT` statement, store an array for the result set in the `$products` variable, and close the connection to the server.

In these blocks, the code uses variables named `$statement1`, `$statement2`, and `$statement3` to refer to the `PDOStatement` objects you need for this page. Sometimes developers re-use a single variable name for each SQL statement. For example, they might use `$statement` for all three `PDOStatement` objects. However, this is generally regarded as a poor practice for two reasons. First, it can make your code harder to read. Second, it can make your code harder to debug.

After the initial PHP script, the rest of the ***index.php*** file contains mostly HTML tags. If you're familiar with HTML, you shouldn't have any trouble understanding these tags

The HTML for this file contains embedded PHP tags that display the variables that are defined by the PHP script at the beginning of the file. To start, the first foreach loop displays a list of links for each category. Here, the href attribute specifies a URL that includes the category ID, and the link displays the name for the category. As a result, if the user clicks on one of these links, this application uses the `GET` method to pass the category ID for the link to the ***index.php*** file. This returns a web page that shows the products for the new category.

These category links are stored within the `<li>` tags of an unordered list. However, the CSS for this application turns the bullets for this list off so the bullets aren't displayed. Although the CSS isn't shown for any of the applications in this book, it is available in the downloaded applications so you can review it to see how it works.

After the first foreach loop, this page displays the name of the current category above the table of products. Then, it uses a second foreach loop to display the products for the current category. This displays three of the five columns in the result set in an HTML table. Specifically, it displays the product's code in the first column, the name in the second column, and the price in the third column. However, it doesn't display the product ID or the category ID in this table.

The Product Manager application

Please try to run the code (product_manager folder) on your browser (in localhost) and study the code along with this supporting material.

This application enhances the Product Viewer application in several ways. Most importantly, it lets the user add and delete products.

The user interface

The primary pages of the user interface for the Product Manager application are the Product List page and the Add Product page. Both of these pages include the name of the application in the header, and both of these pages include a copyright notice in the footer. These pages also include borders that separate the header and footer from the body of the page.

The Product List page lets the user view and delete products. It displays a list of categories and a table of products for the current category. This is similar to the main page of the Product Viewer application. However, this list includes a Delete button for each product that lets the user delete the product. In addition, this page includes an Add Product link below the table that lets the user display the Add Product page.

The Add Product page lets the user add a new product. This page starts by displaying a drop-down list that lets the user select a category for the product. Then, it displays three text boxes that let the user enter data for the product's code, name, and price. Next, it displays an Add Product button that lets the user submit the data for the product. Finally, this page displays a View Product List link that lets the user return to the Product List page without adding a product.

The code

The ***index.php*** file works similarly to the ***index.php*** file for the Product Viewer application. As a result, you should already understand most of it. However, there are a few important differences.

To start, the first statement uses a `require_once()` function. As a result, this code only loads the ***database.php*** file once. That means it won't load the file if it has already been loaded by the ***delete_product.php*** or ***add_product.php*** files. This improves the efficiency of the application.

Next, the first block of code begins by checking whether the `$category_id` variable has already been set. As a result, this code only attempts to get the category ID from the `$_GET` array if the `$category_id` variable hasn't already been set by the ***delete_product.php*** or ***add_product.php***.

In the HTML, this code uses `<header>` and `<footer>` tags to define a header and footer for the page. Here, the header displays the name of the application (Product Manager), and the footer displays a copyright notification for the application.

In the aside block in ***index.php*** file, the `foreach` loop that displays links for each category is the same as you saw in the Product Viewer application, with one important difference. In this

version, the URL in the anchor's href attribute is preceded by a single dot (`.`). This dot makes sure the URL starts with the current directory. Without it, the application would create incorrect links when including the ***index.php*** file from the php files that add or delete products.

In the HTML table of products, the code displays four columns. Here, the fourth column contains a form that displays a Delete button that allows the user to delete the corresponding product. Note that this form uses the POST method to call the ***delete_product.php*** file, and it uses hidden fields to pass two variables to this file: `product_id` and `category_id`.

The ***delete_product.php*** file begins by loading the ***database.php*** file to get a connection to the database. Then, it gets the IDs for the product and category from the `$_POST` array. To do that, it uses the `filter_input()` function to get the IDs. This protects against malicious input. Since these values are stored in hidden fields and an HTTP POST request, you might think that they are safe. However, it's still possible for an attacker to change these values. As a result, it's a good practice to check them to make sure they don't contain malicious input.

Next, it checks the values returned from the `filter_input()` functions to make sure that the validation didn't fail, and then it executes a `DELETE` statement that deletes the specified product.

Finally, it runs the ***index.php*** file again. As a result, the ***index.php*** file can access the `$category_id` variable and use it to display a list of products for the current category.

The ***add_product_form.php*** file consists mostly of the HTML tags that display the Add Product page. However, this file begins by executing some PHP code that gets all categories from the database. Later, in the HTML for the page, a foreach loop uses these categories to create a drop-down list that lets the user select a category. Note that this drop-down list uses the category ID as the value for the option and that it displays the category name to the user. As a result, when the user selects the category name, the application gets the category ID that corresponds to that name.

When the user clicks on the Add Product button, the form uses the POST method to pass four product variables to the ***add_product.php*** file. This file begins by getting these four variable from the `$_POST` array, again using the `filter_input()` function.

Then, it checks all four variables to make sure they are valid. First, it checks to make sure the category id exists and is an integer. Second, it checks to make sure the user has entered a value for the code and name variables. Third, it checks to make sure the user has entered a price and that the price is a floating-point number. If any of these checks fail, the code defines an appropriate error message and uses a page named ***error.php*** to display this message. Otherwise, this code uses an `INSERT` statement to add the product data to the products table.

Finally, the code runs the ***index.php*** file again. This causes the ***index.php*** file to use the `$category_id` variable to display a list of products for the current category. As a result, if the user adds a product to the Drums category, the application displays the Product List page for the Drums category.

This application use echo statements to display the data retrieved from the database without escaping it. In other words, this application assumes that no malicious data has been stored in the database. This keeps the code simple.

However, since the data in the database originally came from users, it's possible, though unlikely, that it could contain malicious data that could lead to an XSS attack. To guard against that, you could use the `htmlspecialchars()` function with your echo statements.