

Create a Web storefront using PHP, Derby and PayPal, Part 1: Pouring the foundation database

Learn how to configure the tools you need and set up two user types

Skill Level: Intermediate

[Tyler Anderson \(tyleranderson5@yahoo.com\)](mailto:tyleranderson5@yahoo.com)
Freelance Writer
Stexar Corp.

08 Nov 2005

This series chronicles the building of a Web storefront in PHP using PHP Data Objects to access a Derby database. The storefront includes a user manageable shopping cart that allows item purchases using PayPal, and includes the ability for merchants to notify customers via e-mail on successful orders automatically.

Section 1. Before you start

In this tutorial, Part 1 of a three-part series, you will set up both the user and the administrative sides of the storefront. The shopper will be able to browse items in the storefront, and administrators will be able to add or edit items in the storefront using a Web browser. User sessions are also covered. This tutorial assumes basic knowledge of PHP, including `if` statements, functions, `include` and `require` statements, as well as knowledge of variables submitted using `GET` and `POST`.

About this series

Our hypothetical situation for this series is Ghastly Computers, a computer hardware and software shop. Its owners want to set up a presence online, and for obvious

reasons, they want the online store to be constructed and managed with minimal fuss. This three-part series covers the whole process, from creating a storefront with an integrated shopping cart from scratch in PHP to using PayPal for payment. With e-commerce booming and a growing number of online shoppers using PayPal and other forms of payment to make purchases, creating an online storefront is a good way to attract customers from around the world. The benefit of creating a shopping cart from scratch is the absolute control it gives programmers.

Part 1 focuses on setting up a Derby database and creating the basic storefront. You will access Derby through PHP data objects. User sessions are started to help set up for Part 2 of the series, where a shopping cart will be associated with a user's session ID.

Part 2 covers creating and managing the shopping cart. The shopping cart will be stored in the Derby database, and the user's session ID will become the cart ID. The shopping cart, if containing items, will be shown under the listing of categories in the storefront with Manage cart and Checkout links. Checking out will involve taking a user's Ship-to/Bill-to information, which will be submitted for payment to PayPal. After completing the payment to PayPal, the user is shown a Thank you page and given a new session ID.

Part 3 covers viewing orders, shipping, and e-mail notification. Transactions using PHP data objects will also be introduced. Adding shipping to the checkout process will be incorporated via UPS, offering real-time shipping prices. E-mail notification will be sent to the customer showing an order summary. PayPal, by default, also sends a payment confirmation e-mail to the user and the merchant.

About this tutorial

In this tutorial, you prepare for creating a storefront from scratch by setting up a Derby database that houses the information of your storefront. Editing and querying the database is accomplished via PHP data objects (PDOs), as PDOs encapsulate much of the technical and database-specific details on connecting to and executing SQL queries and commands in a database. The owner of Ghastly Computers has requested an administrative (from here on referred to as "admin") and user entry into the storefront. Adding and editing items via a Web browser makes it easy for administrators anywhere, including the new programmer Ghastly Computers is planning to hire a few states away, to update the content of a storefront and the items contained therein. User sessions will also be introduced to set up the connection between a user and a shopping cart in Part 2 so that, as users move between areas of a storefront, their shopping carts will be carried with them.

Prerequisites

To follow along, you will need to install and test the following tools:

Web server

Pick any Web server and operating system. Feel free to use [Apache V2.X](#), or the [IBM HTTP Server](#).

PHP

Due to the use of PHP data objects, the latest version, [PHP V5.1](#), is necessary to fully follow along in this tutorial. Note that V5.1 is a release candidate (RC) and is not yet an official release. Be sure to configure PHP with the following option to include support for Derby:

```
--with-pdo-odbc=ibm-db2,/home/db2inst1/sqllib. See Resources for information about configuring Apache or the IBM HTTP Server with PHP.
```

Database

This tutorial uses Derby. [Download Derby V10.1](#), the current IBM DB2 JDBC Universal Driver, and the DB2® run-time client from IBM. Be sure to follow the instructions on each page carefully. Follow either the Linux® or Windows® instructions for downloading and installing the DB2 run-time client.

You may also use [Cloudscape](#) for this tutorial. The internals of Cloudscape are the same as Derby. However, the DB2 JDBC Universal Driver and other things are packaged into Cloudscape, and it is supported by IBM. Download Cloudscape V10.1, and the DB2 run-time client from IBM.

Java™

Derby requires Java technology. I have found the gcj provided in Red Hat Linux distributions insufficient. [Download Java technology](#) from Sun.

Section 2. Setting up: Derby and initial files

You will use a Derby database throughout this series to store information about users of your Web site. This section covers setting up the initial database, as well as adding the necessary tables, categories, and product attributes you will use throughout the series. Setting up the directory structure, and the functions of the administrative and user pages of the storefront will also be covered.

Setting up the Derby database

The folks at Ghastly Computers chose Derby for several reasons. Derby is built with Java technology, and, as such, it's portable and works well with other Java programs you will inevitably add later. Start the Network Server by typing:

```
java org.apache.derby.drda.NetworkServerControl start
```

Now that you have started the Network Server, you can begin communicating with it. Before you can communicate with the Network Server, however, you need to open the Derby command-line processor by typing the following in a new console window:

```
java org.apache.derby.tools.ij
```

You should now be at the `ij` prompt. Now it's time to create the new database:

```
connect
'jdbc:derby:net://localhost:1527/PAYPAL;create=true:user=paypaluser;
password=paypalpass;';
```

You have now created and set up the Derby database. The next thing to do is catalog the database in DB2.

Cataloging the database in DB2

You need to catalog the database so that when you connect to Derby via a PHP data object, PHP will be able to find the new database. Type the following in a new console window to start the DB2 command-line processor: `db2`.

You should now be at the DB2 prompt. Before you catalog the database, you need to catalog a TCP/IP node (`cns`), by typing the following:

```
catalog tcpip node cns remote localhost server 1527
```

The command-line processor should output the following:

```
DB20000I The CATALOG TCPIP NODE command completed successfully.
DB21056W Directory changes may not be effective
until the directory cache is refreshed.
```

Now you can catalog your new PayPal database by typing the following:

```
catalog db PAYPAL at node cns authentication server
```

The following output should be returned:

```
DB20000I The CATALOG DATABASE command completed successfully.
DB21056W Directory changes may not be effective
until the directory cache is refreshed.
```

Now that the database is cataloged, you should be able to connect to it via a PHP data object. For a sanity check and to make sure everything is working, and connect to the database at the DB2 prompt by typing the following:

```
connect to PAYPAL user paypaluser using paypalpass
```

If all is working, the following should be returned as output to the above command.

Listing 1. Output from connecting to the database

```
Database Connection Information
Database server      = Apache Derby CSS10011
SQL authorization ID = PAYPALUSER
Local database alias = PAYPAL
```

You are all set to begin setting up the necessary tables via the `ij` prompt.

Introducing the database tables

There are several tables in Derby you will use throughout this series. Each has its own role and function in the storefront application. Here are the tables needed by Ghastly Computer's storefront:

Categories

This table will have one field (a name) that will be the name of each of the categories in the storefront.

Products

Each product will have the following fields: productid, category, name, short description, long description, image name, attribute, and price. The category field exists so that when a category is visited, all the products that match the category are listed. Products may also have attributes, as often, there are several options when purchasing a product, including size, color, etc. The name of the attribute will be stored in the attribute field. Notice that for the purposes of this tutorial, each product can have at most only one attribute. This could be further extended by creating a new table with productid and attribute name fields, allowing multiple attributes for each product.

Attributes

The attributes table will hold the different attributes each product might have. It will correspond to the attribute field in the Products table. This table will have a

name and a type field. The type field designates how the attribute will be displayed. This tutorial supports only the radio type, however; check boxes or pull-down lists could easily be implemented.

Attribute values

The attribute values table will contain the values of each attribute in a name field and a value field. The name will correspond to the name field in the attributes table. For a color attribute, you could have several values, including red, yellow, and blue.

These are the tables used in this tutorial (Part 2 will set up the others):

- Customers
- Shopping carts
- Shopping cart contents
- Orders

Next, you will create the tables to use in this tutorial.

Creating the tables

The category table will hold the categories. Create this table at the `ij` prompt you have already created the database with by typing:

```
create table categories
      (name varchar(30) primary key);
```

The following output reveals success every time a table is created:

```
0 rows inserted/updated/deleted
```

The products table is next. This table will hold attributes and other product-specific information of products you will add. Create the table at the `ij` prompt.

Listing 2. Creating the products table

```
create table products
      (productid integer not null generated always as identity
        (start with 1, increment by 1),
      category varchar(30),
      name varchar(30),
      short_desc varchar(100),
      long_desc varchar(3000),
      image_name varchar(30),
```

```
attribute varchar(30),
price integer);
```

The attributes table connects to the products table through the attribute field in the products table and the name field in the attributes table. Create the attributes table as follows:

```
create table attributes
  (name varchar(30) primary key,
   type varchar(30));
```

The next table, attribute values, connects to the attributes table through the name field of each. Create the table by typing the following:

```
create table attribute_values
  (name varchar(30),
   value varchar(30));
```

Notice that the attribute values table also connects to the products table through the attribute field of the products table and the name field in the attribute values table. Learning how to connect these two tables on those fields will be covered in the PDO basics section.

Inserting the initial categories, attributes, and attribute values will be covered next.

Initializing the tables

Being a computer hardware and software store, Ghastly Computers has requested the categories below; however, you can add any you want. Add the following categories at the `ij` prompt:

```
insert into categories values ('Desktops'), ('Laptops'),
                             ('Monitors'), ('Software');
```

Confirmation of a successful insert yields the following output:

```
4 rows inserted/updated/deleted
```

Next, initialize attributes and the attributes values tables:

```
insert into attributes values ('color','radio');
insert into attribute_values values ('color','gray'),
                                   ('color','black');
```

Success on inserting the color attributes with radio type yields the following output:

```
1 row inserted/updated/deleted
```

The following output shows success on inserting into the `attribute_values` table:

```
2 row inserted/updated/deleted
```

You have now set up and initialized the tables.

Setting up the directory structure: Shared functions

The storefront will contain administrative and user sections.

The user section of the storefront will be contained at the root of your application (`./`). The admin section of the storefront will be contained the admin subdirectory (`./admin/`). There are, however, several functions that will be shared between the two sections of the storefront. Create a `shared_functions.php` file and place it at this subdirectory of the root of your application: `./includes/shared_functions.php`. Define it as shown:

Listing 3. The shared functions file

```
<?php
define('STORE_NAME', "TylerCo.");
define('IMAGEURL', "images/");

function db_connect($dbname='PAYPAL',
                   $username='paypaluser',
                   $password='paypalpass'){
}

function printViewItemHelper($admin, $pdo){
}

function getAttributeHTML($attr, $pdo){
}

function printCategorySectionHelper($admin, $pdo){
}

function printItem($row, $admin){
}

function displayTopBarHelper($admin){
    print("Home");
}
```

```
function processGETString($vetoString='') {  
}  
?>
```

Notice that all of the functions are defined, but they are empty. You will define the above functions throughout the tutorial. Notice, too, that some of the functions in Listing 3 have an `$admin` variable. These functions will use the variable to display information specific to the admin or user section of the storefront. For example, the `printItem` function will display a link to edit the item in the admin section and omit the link in the user section.

Next up is learning about the functions specific to the admin and user sections of the storefront.

Setting up the directory structure: Admin and user functions

Having an admin and user section of the storefront means there will be two entry points to the storefront: one for users and one for admins (See listings 6 and 7). The `$admin` variable will allow you to make the changes necessary for the admin section, as opposed to the user section. The `$admin` variable from the function declarations in Listing 3 will be set by functions in include files specific to the admin or user section of the storefront.

Create an `admin_functions.php` file at the following location:
`./admin/includes/admin_functions.php`. Define it as shown:

Listing 4. Admin functions

```
<?php  
  
function printCategorySection($pdo){  
    printCategorySectionHelper(1, $pdo);  
}  
  
function printViewItem($pdo){  
    printViewItemHelper(1, $pdo);  
}  
  
function displayTopBar(){  
    displayTopBarHelper(1);  
}  
  
?>
```

These functions, like the ones shown in Listing 5, are called by the functions in Listing 3, except they pass 1 as the value for the `$admin` variable, indicating the storefront is on the admin section.

Create a `user_functions.php` file at the following location:

./includes/user_functions.php. Define it as shown:

Listing 5. User functions

```
<?php
function printCategorySection($pdo){
    printCategorySectionHelper(0, $pdo);
}

function printViewItem($pdo){
    printViewItemHelper(0, $pdo);
}

function displayTopBar(){
    displayTopBarHelper();
}

?>
```

These functions will be called by functions in `shared_functions.php`, which will in turn call the Helper functions, passing in 0 as the value for the `$admin` variable, indicating the storefront is currently in the user section.

Next, you will move onto the basic storefront.

Section 3. The basic storefront

This section covers the basic storefront and how it is setup to display category and product information without querying the database; querying the database is saved for the PHP data objects section.

The processing point

There will be an administrative section and a user section to the storefront because Ghastly Computers would like a separate section for admins, making editing the storefront through a Web browser much easier. The user section is for displaying categories that display products and descriptions of these products to entice interested customers to add the product to their carts.

Both sections will display categories and category items. However, admins will be allowed to add and edit items, and users will be allowed to add items to the shopping cart (covered in Part 2 of this series).

Users that visit your storefront (see Figure 1) will always point to the same file. This

file is the processing point for your storefront that will take requests, such as a category, and return corresponding data to a user's browser.

Figure 1. Basic storefront in the user section



Create a storeFront.php file and place it at the root of your application (.). Define it as shown:

Listing 6. The storefront processing point for users

```
<?php
include('includes/user_functions.php');
include('includes/shared_functions.php');
$db = db_connect();

$title="Welcome to ".$STORE_NAME;
require('header.php');

print("
<p>Welcome to our PHP storefront!<br>
Checkout our products by browsing the categories to the left.");

require('footer.php');
?>
```

Notice that this is the processing point for users entering the storefront. Now create another file, storeFront.php, which will be used as the entry point for administrators. Place this file at the following subdirectory: ./admin/storeFront.php. Define it as shown:

Listing 7. The storefront processing point for administrators

```
<?php
include('includes/admin_functions.php');
include('../includes/shared_functions.php');
$pdo = db_connect();

$title = "Welcome ADMIN";
require('../header.php');

print("
<p>Welcome ADMINISTRATOR!<br>
  Select categories to add/edit items to the left.");

require('../footer.php');
?>
```

The processing points for users and administrators have now been established for your storefront. You will modify these files throughout this series to support additional functionality.

Next, you need to define header.php, the header file that will include the side panel, a top bar, and a footer.

Header file

The header and footer files are the data required by listings 6 and 7. These files add layout to the storefront by centering the view, creating panels and titles, and displaying copyright information. First, define the header file by creating a file header.php:

Listing 8. The header file

```
<?php

print('
<html>
<title>' . $title . '</title>
<body>
<table width="650px" align="center" valign="top">
<tr><td colspan="2">
');

displayTopBar();

print('
</td></tr>
<tr><td colspan="2">
<center><h1>' . $title . '</h1></
center></tr></td><tr><td height="100%">
<table width="150px" align="left" border="2px" height="100%"
><tr><td valign="top">
<table width="150px">
<tr>
<td><b>Categories</b></td>
</tr>
```

```

        <tr>
            <td>' );

print( "
<a href='storeFront.php?category=Desktops'>Desktops</a><br>
<a href='storeFront.php?category=Laptops'>Laptops</a><br>" );

print( '
        </td>
    </tr>
    <tr>
        <td>&nbsp;</td>
    </tr>
</table>
' );

print( '
</td></tr></table>
</td><td valign="top" width="100%">
' );
?>

```

Note that the header displays the `$title` variable, as defined from `storeFront.php`. Notice that the `displayTopBar` function is called. The information displayed by this function can be tailored to whether the storefront is in the admin or user section. Next, the `$title` is used again in big `<h1>` tags. The sidebar is then set up in the `<table width="150px" ...>` declaration. The Categories subheading is also shown, followed by a couple example category links.

Next up is the footer file.

Footer file

The footer file is needed to bring closure to the display of the storefront and display any remarks or unique information. Define the footer by creating a `footer.php` file.

Listing 9. The footer file

```

</td></tr>
<tr><td align="center" colspan="2">
<font size="2px"><br>
<center>Copyright 2005, <?php print(STORE_NAME) ?></center>
</font>
</td></tr></table>
</body></html>

```

This file is simple, aiming to close up the HTML and display any specific Web site or copyright information you would like to add. See the current application output, as shown in [Figure 1](#).

Next, you will cover displaying categories.

Processing view category requests

When you click on a category link, you would expect category items to display. Here, you create code to handle GET requests with the category variable set so you can display category items to the admin or user. Modify the admin and user storeFront.php files as shown in listings 10 and 11.

Listing 10. Capturing the category variable for administrators

```
...
require('../header.php');

if($_GET['category'] != ''){
    printCategorySection($pdo);
}
else{
    print("
<p>Welcome ADMINISTRATOR!<br>...
")
}
...
```

Listing 11. Capturing the category variable for users

```
...
require('header.php');

if($_GET['category'] != ''){
    printCategorySection($pdo);
}
else{
    print("
<p>Welcome to our PHP storefront!<br>
...
")
}
...
```

If the category variable is set in the GET array, the appropriate function will be called, `printCategorySection()`, in `shared_functions.php`. The `printCategorySection()` function currently does nothing. See Figure 2 for the current output of the application.

Figure 2. Displaying an empty category in the user section



Up to this point, you have built up a lot of infrastructure. Introducing PHP data objects is next.

Section 4. PHP data objects

This section will cover the syntax of PDOs, usage, and an example in the storefront application using PDOs to query the database you set up to display categories. This section also ramps up the next section, PDO basics, which covers adding and editing items in the storefront.

Introduction

A new feature in PHP V5.1 is PHP data objects (PDO). The folks at Ghastly Computers love the new PDOs because they make accessing the Derby database so much easier. PDOs abstract the low-level and database-specific details of connecting to databases. These are used in place of connecting to a database with database-specific functions, so if you use one database or another, the method of using the database is the same; only initializing the PDO is different.

The PDO extensions of PHP define a consistent interface for connecting to and querying databases in PHP. Each database driver implementing a PDO can add functions to the PDO accessed as regular function extensions.

PHP data objects greatly simplify the process of connecting to a database, as well as querying and executing statements on the database. Define the `db_connect()` function in `shared_functions.php`. Creating a connection to the database is as simple as the code shown in Listing 12.

Listing 12. Creating a new database connection with a PDO

```
function db_connect($dbname='PAYPAL',
                  $username='paypaluser',
                  $password='paypalpass'){
    $pdo = new PDO("odbc:$dbname", $username, $password);
    return $pdo;
}
```

The PDO is created by specifying the database driver, `odbc`, and the database you wish to connect to, `PAYPAL`. The username and password are also passed as parameters, which just says that you are an authorized user to the database. The function then returns the PDO, which can be later used to query and execute statements on the database. In fact, the next section covers querying the database to retrieve the category names, displaying them on the storefront.

Querying the database

Here, you will learn how to query the database using a PDO. This example will result in displaying the category names on the storefront. Modify the `header.php` by removing the two hard-coding category links you added earlier and replacing them with the following:

Listing 13. Retrieving and displaying category names from the database

```
...
    <td><b>Categories</b></td>
  </tr>
  <tr>
    <td>');

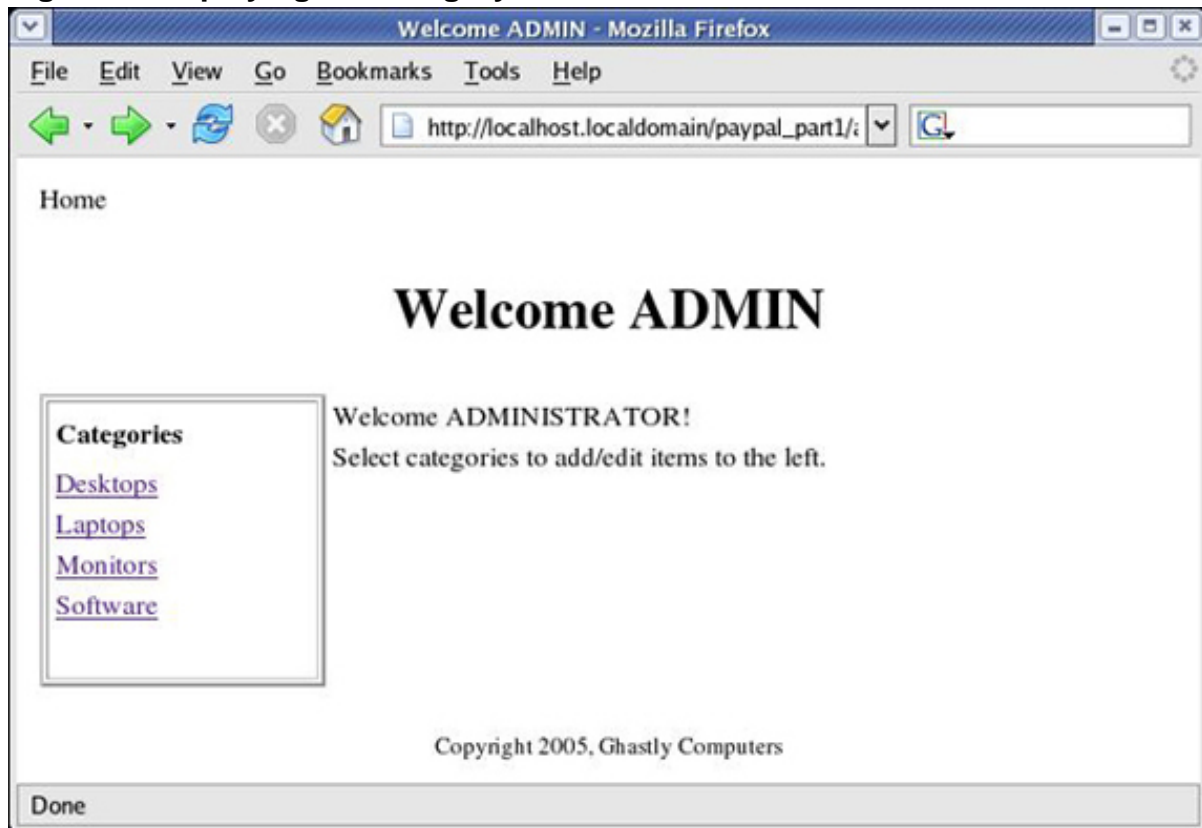
  $sql = "select name from categories;";
  foreach($pdo->query($sql) as $row){
    $name = $row['NAME'];
    print('
      <a href="storeFront.php?category='.$name.
        processGETString("cavi").'">'.$name.'</a><br />');
  }

  print('
    </td>
  </tr>
```

First, the SQL statement is set up, and all the results are looped through by using a `foreach` statement on the result of calling `$pdo->query($sql)`. The category link is then displayed, based on the name of the category, as returned from the SQL query. The function, `processGETString()` is then called, whose purpose is to block or propagate the transmission of variables in the `GET` array.

Now you can view the storefront and see all the category links.

Figure 3. Displaying the category links from the database in the admin section



You will cover the `processGETString()` method next.

Creating the URL: GET variables

Once a variable gets set in the `GET` array, it's often necessary to continue propagating it, unless otherwise specified. Define the `processGETString` in the `shared_functions.php` file, as shown:

Listing 14. Propagating or blocking variables currently in the GET array

```
function processGETString($vetoString='') {
    $string = '';
    if($_GET['category'] != '' && strpos($vetoString, 'ca') == '')
```

```

    $string .= "&category=".$_GET['category'];
    return $string;
}

```

Thus, if a category is defined in the GET array, and `ca` is not in the `$vetoString`, the category will be propagated, as you browse items within a category, for example. Otherwise, the presence of `ca` in the `$vetoString` blocks the category variable from propagating in the URL. Example usage of this function is shown in Listing 13.

Displaying categories

Now that the category links are displaying and your storefront is successfully processing the GET array for the presence of the category variable in the GET array, it's time to display the products. Define the `printCategorySectionHelper()` function in the `shared_functions.php` file, as shown in Listing 15. This function gets called by `printCategorySection()` in either `admin_functions.php` or `user_functions.php`, depending on whether the storefront is in the admin or user section.

Listing 15. Displaying category items

```

function printCategorySectionHelper($admin, $pdo){
    $category = $_GET['category'];

    $sql = "select productid, name, price,
            short_desc, long_desc,
            image_name
            from products
            where category='".$_category.'" ";
    $result = $pdo->query($sql);

    $str = "";
    $str .= "<center><h3>$category</h3></center>";
    $str .= "<table width='100%' border='0' cellspacing='5' ";
    $str .= "cellpadding='0'>";

    $totalItems = 0;
    foreach($result as $row){
        $str .= printItem($row, $admin);
        $totalItems++;
    }
    $str .= "</table>";
    print($str);

    if($totalItems <= 0){
        print("The $category category has no products");
        return;
    }
}

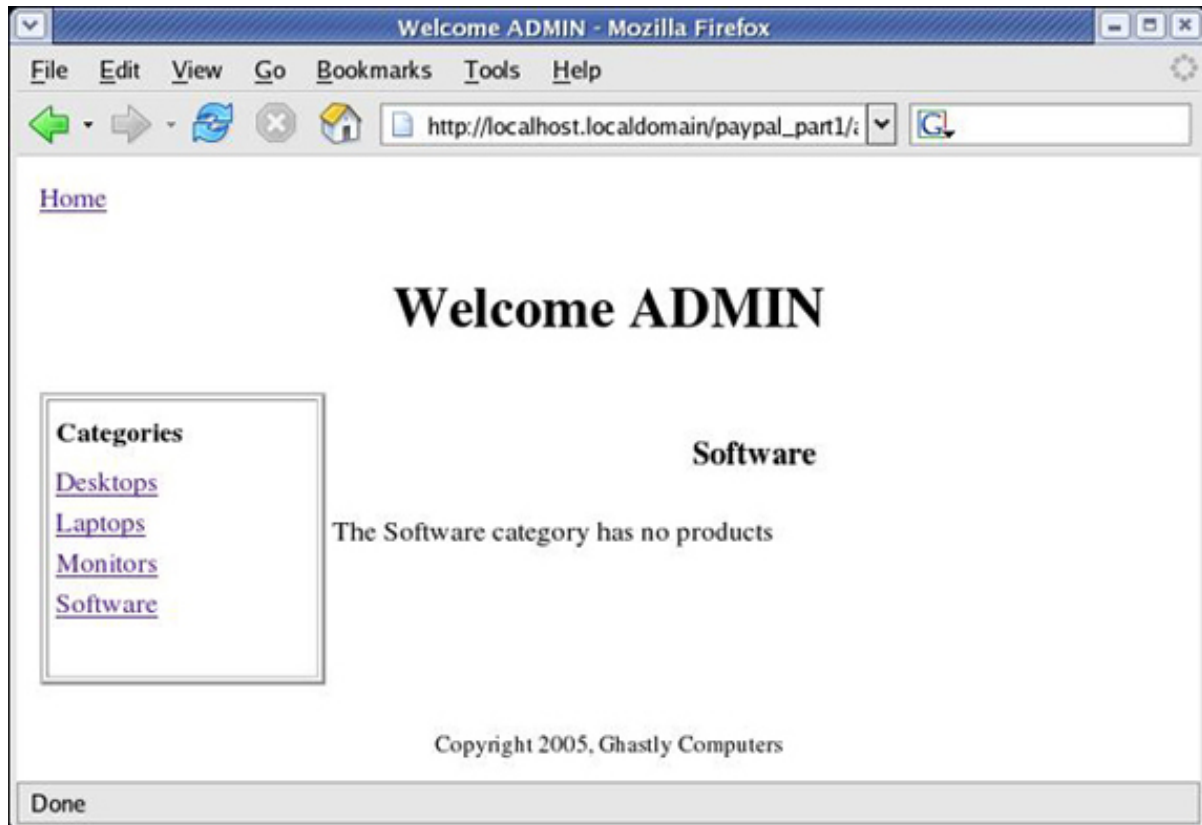
```

This function begins by querying the database on all products whose category name matches the the currently selected category in the GET array. The title of the category is then buffered in `$str`. The `foreach` statement loops through all the results of the query, and for each product in the list, that product is displayed by

calling `printItem()`. If no items exist (`$totalItems == 0`), that is displayed to the user.

Leave the `printItem()` function blank for now, since the database currently does not have any items in the products table. View an empty category, as shown in Figure 4.

Figure 4. Displaying a titled, empty category in the admin section



This function begins by querying the database on all products whose category name matches the currently selected category in the `GET` array. The title of the category is buffered in `$str`. The `foreach` statement loops through all the results of the query, and for each product in the list, that product is displayed by calling `printItem()`. If no items exist (`$totalItems == 0`), that is displayed to the user.

Configuring the top bar

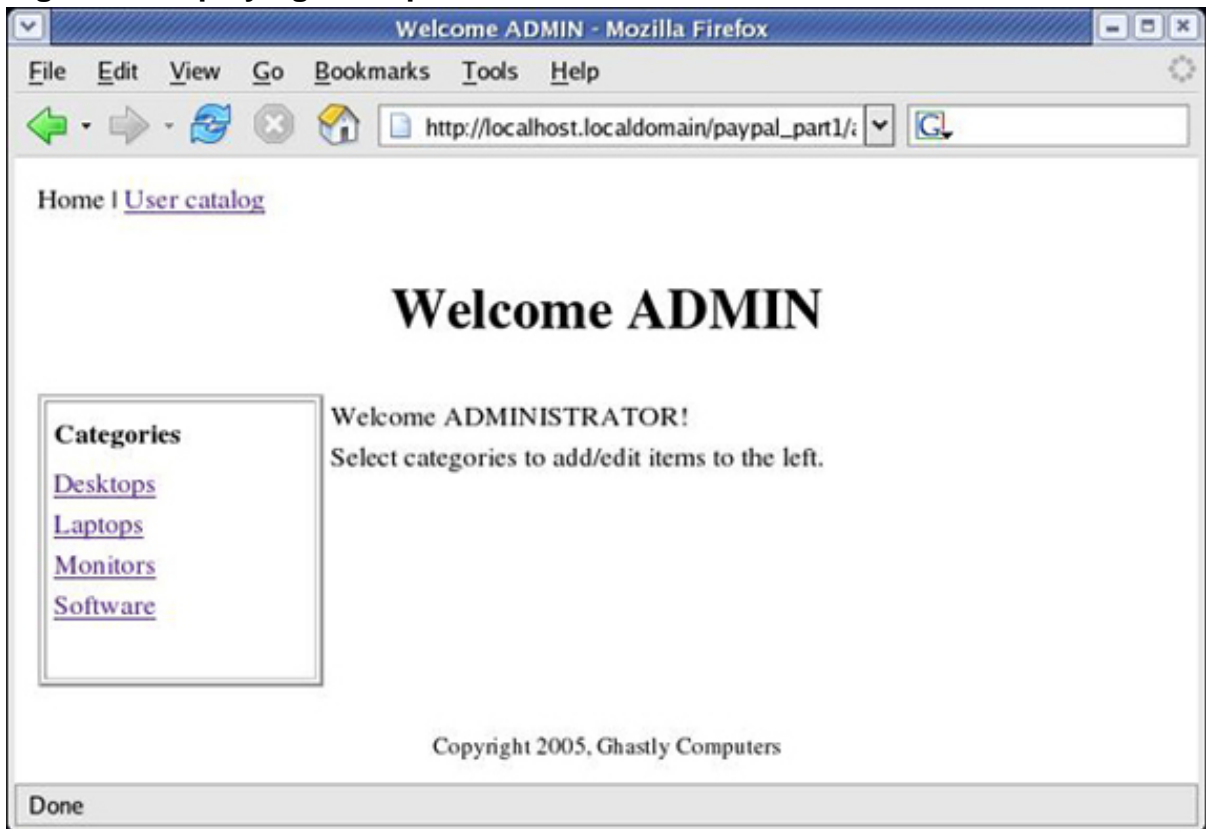
At the home page, notice that "Home" is displayed on the upper-left corner. When displaying a category, however, this should become a link that displays the home page, blocking most values in the `GET` array from propagating. Modify the `displayTopBarHelper()` function, as shown:

Listing 16. Implementing a dynamic top bar

```
function displayTopBarHelper($admin){
    if(!isset($_GET['category'])){
        print("Home");
    }
    else{
        print("<a href='storeFront.php?'.processGETString("cavi")");
        print(">Home</a>");
    }
    if($admin == 1)
        print(" | <a href='../storeFront.php?'.
            processGETString("").>User catalogue</a>");
}
```

If you're at the home page, the same will happen. If you're browsing categories, a link to go home will pop up, and if you're in the admin section, a link to go to the user storefront section will also appear, beside the "Home" link titled "User catalog." The owner of Ghastly Computers specifically requested this so he didn't have to enter the link manually while editing in the admin section of the storefront.

Figure 5. Displaying the top bar links of the admin storefront section



You should be getting used to querying PHP data objects. The next section covers a little more with PDOs, including executing insert and update statements.

Section 5. PDO basics: Adding items to the site

The next task is getting new products into the database in an easy and efficient manner. (Shoppers need something to buy!) This section focuses on the admin section of the storefront by showing how to add items to the database. We also learn how to edit existing items through the Web browser using a standard HTML form, as opposed to doing it at the `ij` prompt. The folks at Ghastly Computers love this idea because the `ij` prompt requires a little more knowledge, at a technical level, than they would like. Finally, we cover viewing items in the storefront.

Setting up functionality

Before you go into the form for adding and editing items in the storefront, you first need to set it up by displaying edit item links when viewing the storefront in the admin section. Modify the `printCategorySectionHelper()` function, as shown:

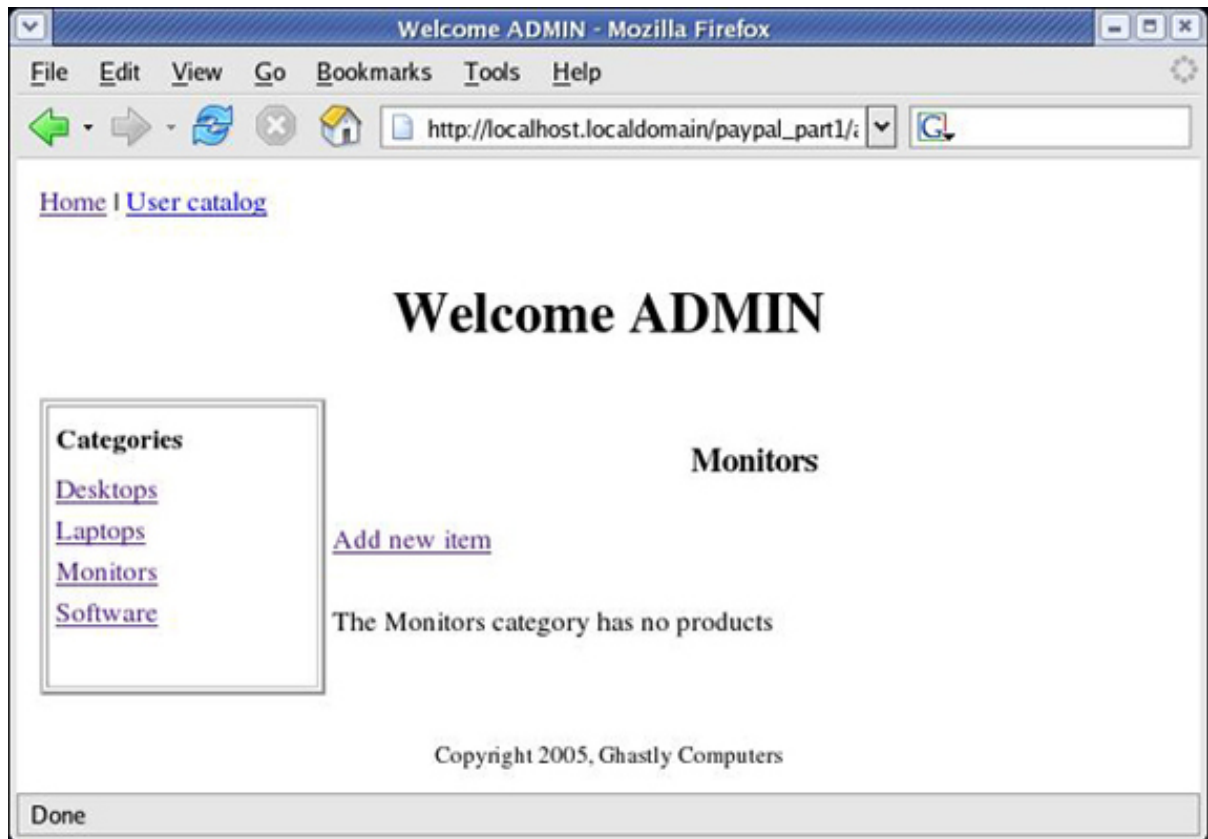
Listing 17. Displaying an Add new item link

```
...
    $str .= "cellpadding='0'>";
    if($admin == 1){
        $str .= "<a href='addEditItem.\
            php?category=$category'>";
        $str .= "Add new
item</a><br><br>";
    }
    $totalItems = 0;
...

```

This adds a link -- Add new item -- at the top of every category when the storefront is viewed in the admin section. Adding items is handled by a separate script, `addEditItem.php`, as opposed to the `storeFront.php` entry file. See the browser output, showing your new Add new item link.

Figure 6. Displaying an Add new item link



This new script and the form for entering new product information is up next.

The add items form

Adding or editing items to the storefront requires a form. This form is used to take information about a new product, and submitting it stores the information in the database. Create an `addEditItem.php` file and place it at this location: `./admin/addEditItem.php`. Define it, as shown:

Listing 18. Displaying a form for adding new product information

```
<?php
$category = $_GET['category'];

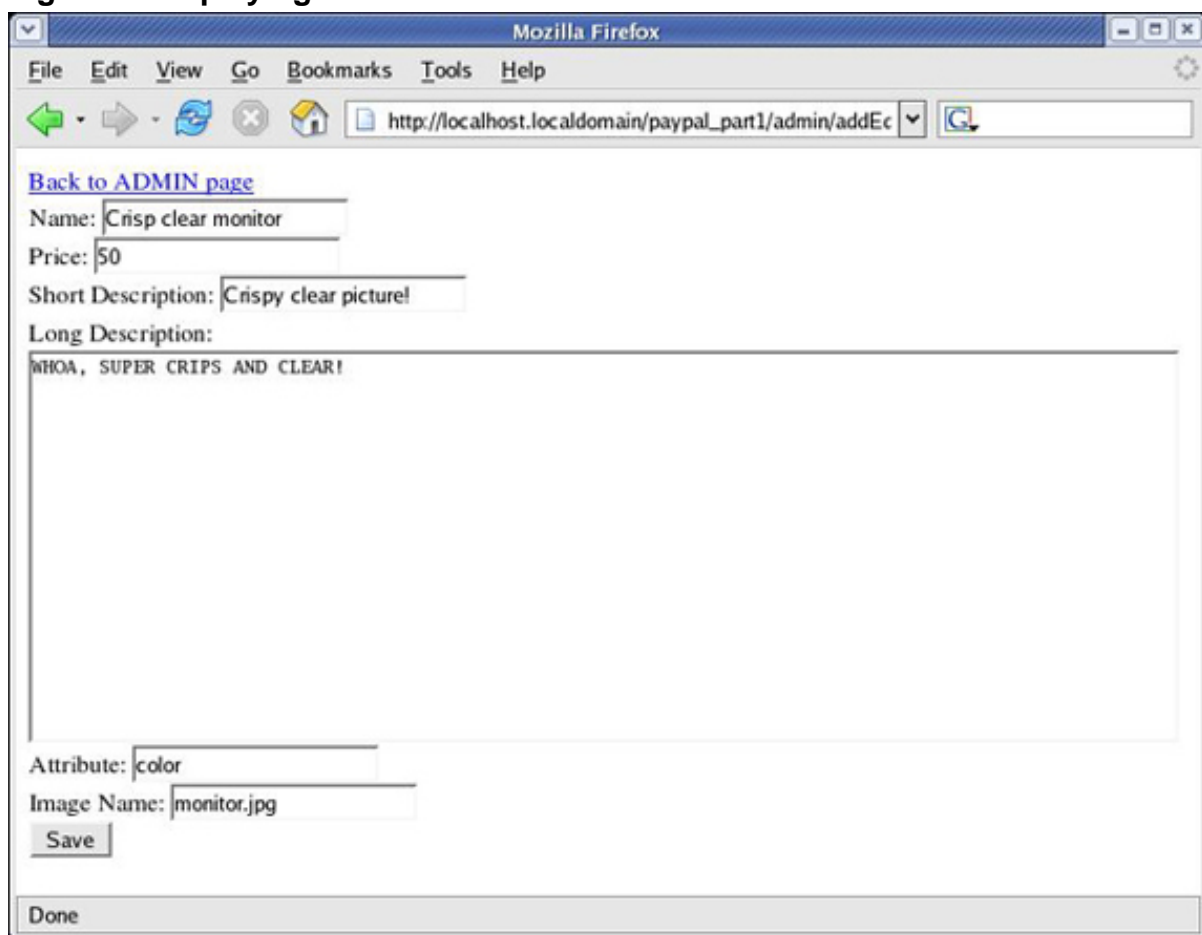
print("
<form action='saveItem.php' method='post'>
<input type='hidden' name='category' value='$category' />
Name: <input name='name' /><br>
Price: <input name='price' /><br>
Short Description: <input name='short_desc' /><br>
Long Description:<br>
<textarea name='long_desc' cols='100' rows='15'></textarea>
Attribute: <input name='attribute' /><br>
Image Name: <input name='image_name' /><br>
<input name='submit' type='submit' value='Save' />

```

```
</form>
");
?>
```

First, the category of the new item is captured from the `GET` array and is stored in the form as a hidden input. Hidden input items are "hidden" from the administrator. It is hidden in this case because the category was given when the administrator clicked the Add new item link in the category. The rest of the required input information is retrieved from standard text boxes, except the long description, which is in a text box, which allows for larger, multiline inputs. See the browser output of this form. (Note the misspelled description. We'll see how the admin user can edit this later.)

Figure 7. Displaying the Add new item form



The screenshot shows a Mozilla Firefox browser window with the address bar displaying `http://localhost.localdomain/paypal_part1/admin/addEc`. The page content includes a link [Back to ADMIN page](#) and the following form fields:

- Name:
- Price:
- Short Description:
- Long Description:
- Attribute:
- Image Name:

A **Save** button is located below the form fields. The browser status bar at the bottom shows "Done".

Clicking the **Save** button submits the new item to the `saveltem.php` script, covered next.

Saving new items

You have just saved an item, and now you need to define the script to handle the save process. Create a `saveItem.php` file and place it at the following location: `./admin/saveItem.php`. This script will take the form data posted by submitting the new item data in the previous section.

Listing 19. Saving the new item data

```
<?php
include('../includes/shared_functions.php');

$category = $_POST['category'];
$name = $_POST['name'];
$short_desc = $_POST['short_desc'];
$long_desc = $_POST['long_desc'];
$price = $_POST['price'];
$attribute = $_POST['attribute'];
$image_name = $_POST['image_name'];

$pdo = db_connect();
$sql = "insert into products (name, short_desc, long_desc, price,
                             image_name, category, attribute) values
        ('$name', '$short_desc', '$long_desc',
         '$price', '$image_name', '$category',
         '$attribute')";

$pdo->exec($sql);

header("Location: storefront.php?category=$category");
?>
```

The script begins by retrieving the posted variables from the `POST` array. Next, connect to the database and set up your SQL statement, placing the values of the variables into the statement. Finally, the SQL statement is executed by making a call to the `$pdo->exec()` function. Notice that you previously used `$pdo->query()`. The difference is that `exec()` modifies the database and returns the amount of affected rows. The `query()` function returns the results of your query. The last line sends you back to the category you added the new item to.

The new item should now be stored in the database. It won't display yet, and you can guess what's next: viewing items in the storefront.

Displaying item listings

Now that you have items in the database, it would sure be nice to view them and display them to interested users. This panel continues the description of the `printCategorySectionHelper()` function in the Displaying categories section by defining the called function, `printItem()`. Define the function, as shown:

Listing 20. Displaying individual items in list view

```
function printItem($row, $admin){
    $price = $row['PRICE'];
```

```

$imageURL = IMAGEURL . $row['IMAGE_NAME'];
$name = $row['NAME'];
$id = $row['PRODUCTID'];

$str = "";
$str .= "<tr><td align='center' valign='top' rowspan='3'>";
$str .= "<img border='0' width='60%' src='$imageURL'></td>";

$str .= "<td width='70%'><font size='-1'><b>";
$str .= "<a href='storeFront.php?viewItem=$id".
    processGETString(" ")."'>$name</a></b></font></td>";

$str .= "<tr><td><font size='-1'>$desc</font></td></tr>";

$str .= "<tr> <td align='left' valign='top'>";
$str .= "<font size='-1'>Price: $price</font></td> </tr>";

if($admin == 1){
    $str .= "<tr><td colspan=2><font size='-1'>";
    $str .= "<a href='addEditItem.php?id=$id'>";
    $str .= "Edit item</a></font></td></tr>";
}

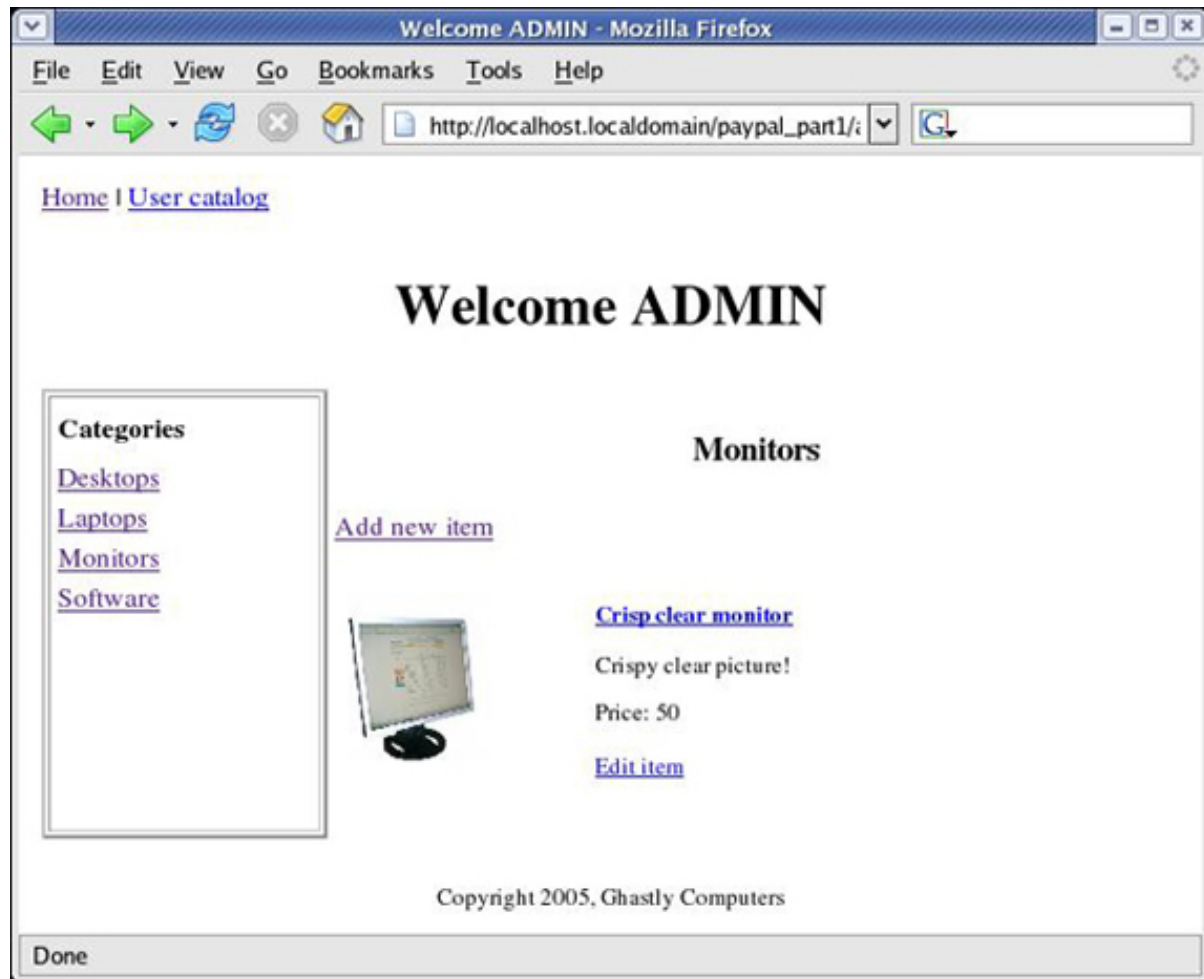
$str .= "<tr><td colspan=2>";
$str .= "<font size='-1'>&nbsp;</font></td></tr>";

return $str;
}

```

Remember in Listing 15 that the function passed the current `$row` and `$admin` variable to the `printItem()` function? The `$row` contains the data associated with a product, and the `$admin` variable will be used later for editing items. A link to view the item is given with the name of the product as the link text, setting the `viewItem` variable in the `GET` array to the `productid` of the item. Functionality for clicking on this link will be added next. Notice the use of the `$admin` variable: If its value is 1, meaning if the storefront is in the admin section, a link will be displayed to edit the item. See Figure 8 for the browser output of the new item in the listing.

Figure 8. Displaying a category with the new item



Next up is viewing single items.

Processing viewItem requests

When browsing a storefront, you often want to see more information about an item than shown in the traditional list view. This panel will cover implementing the functionality to view the long description of an item. Start by modifying the admin and user storefront.php files, as shown in listings 21 and 22.

Listing 21. Captures the viewItem variable for administrators

```

...
require('../header.php');

if($_GET['viewItem'] != ''){
    printViewItem($pdo);
}
else if($_GET['category'] != ''){
    printCategorySection($pdo);
}
...

```

Listing 22. Captures the viewItem variable for users

```

...
require('header.php');

if($_GET['viewItem'] != ''){
    printViewItem($pdo);
}
else if($_GET['category'] != ''){
    printCategorySection($pdo);
}
...

```

The `viewItem` variable has now been captured from the `GET` array, and the functionality is set up so individual items can be displayed. The code that will display individual items is in the next section.

Displaying individual items

Since you have already verified that the `viewItem` variable has been set in the `GET` array, you can define the function called by `printViewItem()` in the `admin` and `user` function files, `printViewItemHelper()`, whose role is to display individual items. Define this function in `shared_functions.php`:

Listing 23. Displays individual items

```

function printViewItemHelper($admin, $pdo){
    $id = $_GET['viewItem'];
    $str = "";
    $str .= "<a href='storeFront.php?' .
        processGETString("vi")."'>Up a level</a>";
    if($admin == 1){
        $str .= "<br><a href='addEditItem.php?id=$id&viewItem=$id'>";
        $str .= "Edit item</a>";
    }
    $sql = "select * from products
           where productid=$id";
    $row = $pdo->query($sql)->fetch();
    $name = $row['NAME'];
    $long_desc = $row['LONG_DESC'];
    $price = $row['PRICE'];
    $attribute = $row['ATTRIBUTE'];
    $imageUrl = IMAGEURL . $row['IMAGE_NAME'];

    $str .= "<center><h3>$name</h3></center>";
    $str .= "<table width='100%' border='0' ";
    $str .= "cellspacing='5' cellpadding='0'>";

    $str .= "<tr><td align='left' valign='top' rowspan='2'>";
    $str .= "<img border='0' width='60%' src='$imageUrl'></td>";

    $str .= "<tr> <td width='15%' align='right' valign='top'>";
    $str .= "<font size='-1'>Price: $price</font><br>";
    $str .= "</td></tr>";

    $str .= "<tr><td>.getAttributeHTML($attribute).</td></tr>";
}

```

```
$str .= "<tr><td colspan=2>$long_desc</td></tr>";  
  
$str .= "</table>";  
print($str);  
}
```

First, the `$id` of the item to view is retrieved from the `GET` array. Then a link is displayed that returns to the category view. If the storefront is in the admin section, a link to edit the item will be shown. Next, the database has `query()` executed for the product corresponding the `$id`. The variables from the resulting query are stored as local variables, and the display information is stored as HTML in the `$str` variable. See Figure 9 for the browser output of the new item in the listing without attributes.

Figure 9. Displaying a new item without attributes



Notice that the HTML of the attributes section is retrieved by making a call to `getAttributeHTML()`. This function is covered next.

Displaying the attributes

Items may have various attributes for purchase. Size and color are good examples of attributes. This is where you display the attribute associated with the current product, if any. Define the `getAttributeHTML()` function in the `shared_functions.php` file.

Listing 24. Displaying attributes

```
function getAttributeHTML($attr, $pdo){
    $sql = "select * from attributes, attribute_values
           where attributes.name = attribute_values.name
           and attributes.name='$attr'";

    $str = "";
    foreach($pdo->query($sql) as $row){
        $type = $row['TYPE'];
        $value = $row['VALUE'];
        $name = $row['NAME'];
        $str .= "$value<input name='attribute' type='$type'";
        $str .= "value='$name: $value'>";
    }

    return $str;
}
```

To display the attributes, the two tables -- `attributes` and `attribute_values` -- are joined where each name field matches and the name field matches the passed in attribute name. The result will contain the name and type for each attribute, and each resulting row will contain a different value of the attribute. The type field is important because it will determine how the attribute values will be displayed. This tutorial will show the radio type in use, but the application could be extended to use a pull-down menu by checking the type of attribute using an `if` or `switch` statement and displaying the attribute's values accordingly. See Figure 10 for the browser output of the new item in the listing, complete with attributes.

Figure 10. Displaying a new item with attributes



You have seen the Edit item link show up in the Displaying item listings and Displaying individual items sections. Next up is adding the functionality to support editing items.

Extending the form: Editing items

Often once an item has been entered in the database, things about a certain item will change over time, especially the price. Here, you cover the functionality to edit items. Modify the `addEditItem.php` file:

Listing 25. Editing items

```

<?php
include('includes/admin_functions.php');

$str = "";
if($_GET['id'] != ''){ // Edit existing item
    $viewItem=$_GET['viewItem'];
    if($viewItem != '')
        $str .= "&viewItem=$viewItem";
    $id = $_GET['id'];
    $pdo = db_connect();
    $sql = "select * from products
           where productid=$id";

    $row = $pdo->query($sql)->fetch();
    $category = $row['CATEGORY'];
    $name = $row['NAME'];
    $short_desc = $row['SHORT_DESC'];
    $long_desc = $row['LONG_DESC'];
    $price = $row['PRICE'];
    $attribute = $row['ATTRIBUTE'];
    $image_name = $row['IMAGE_NAME'];
    $buttonVal = "Edit";
}
else{ // Save new item
    $category = $_GET['category'];
    $buttonVal = "Save";
}

print("<a href='storeFront.php?category=$category$str'>");
print("Back to ADMIN page</a>");

print("
<form action='saveItem.php' method='post'>
<input type='hidden' name='category' value='$category' />
<input type='hidden' name='id' value='$id' />
<input type='hidden' name='viewItem' value='$viewItem' />
Name: <input name='name' value='$name' /><br>
Price: <input name='price' value='$price' /><br>
Short Description: <input name='short_desc' value='$short_desc'
/><br>
Long Description:<br>
<textarea name='long_desc' cols='100' rows='15'>$long_desc
</textarea>
Attribute: <input name='attribute' value='$attribute' /><br>
Image Name: <input name='image_name' value='$image_name' /><br>
<input name='submit' type='submit' value='$buttonVal' />

</form>
");
?>

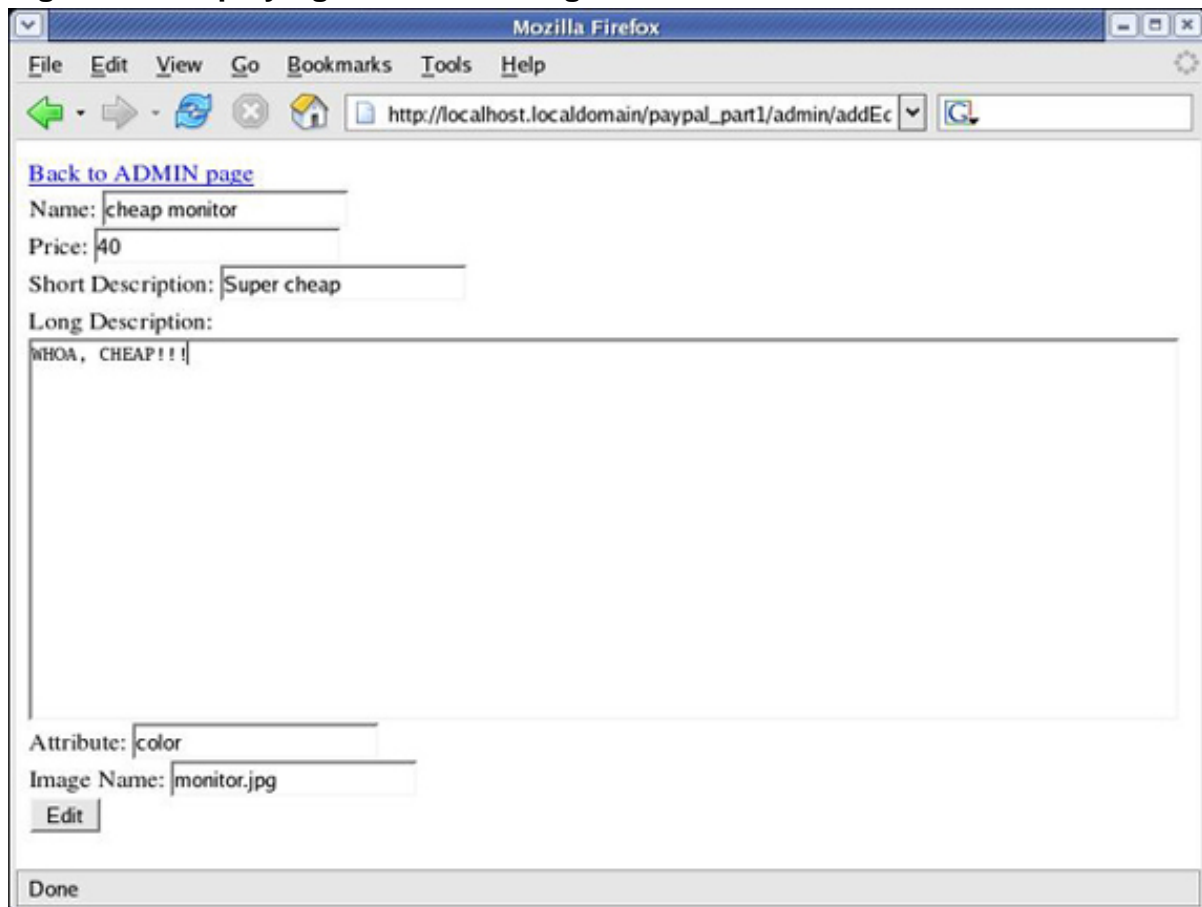
```

The Edit item links set an `id` variable in the `GET` array that corresponds to the `productid` of the item. Therefore, if this variable exists in the `GET` array, it must contain the ID of the item for editing. This causes the program execution to begin at the start of the new code in the `if` statement, which takes care of the editing items case. The item previously viewed is saved for propagation in the `GET` array if you decide to go back to item view.

Next, the database has `query()` executed, and the first row has `fetch()` executed to retrieve the product whose `productid` matches `$id`. The values are then stored

as local variables, used to initialize the text boxes that were previously uninitialized. The name of the button, `$buttonVal` is also parameterized, so the `saveltem.php` file can determine whether to insert a new row into the database or update an existing one. See the browser output of the form for editing items. (We fix the "CRIPS" typo by deleting it and the other superlative.)

Figure 11. Displaying the edit existing item form



The next section goes over handling an edit request in the `saveltem.php` script.

Updating edited items

After you edit an item, there needs to be a way to update the database. Here, we will cover the code to do just that. Modify the `saveltem.php` file:

Listing 26. Updating edited items

```
<?php
...
$image_name = $_POST['image_name'];
```

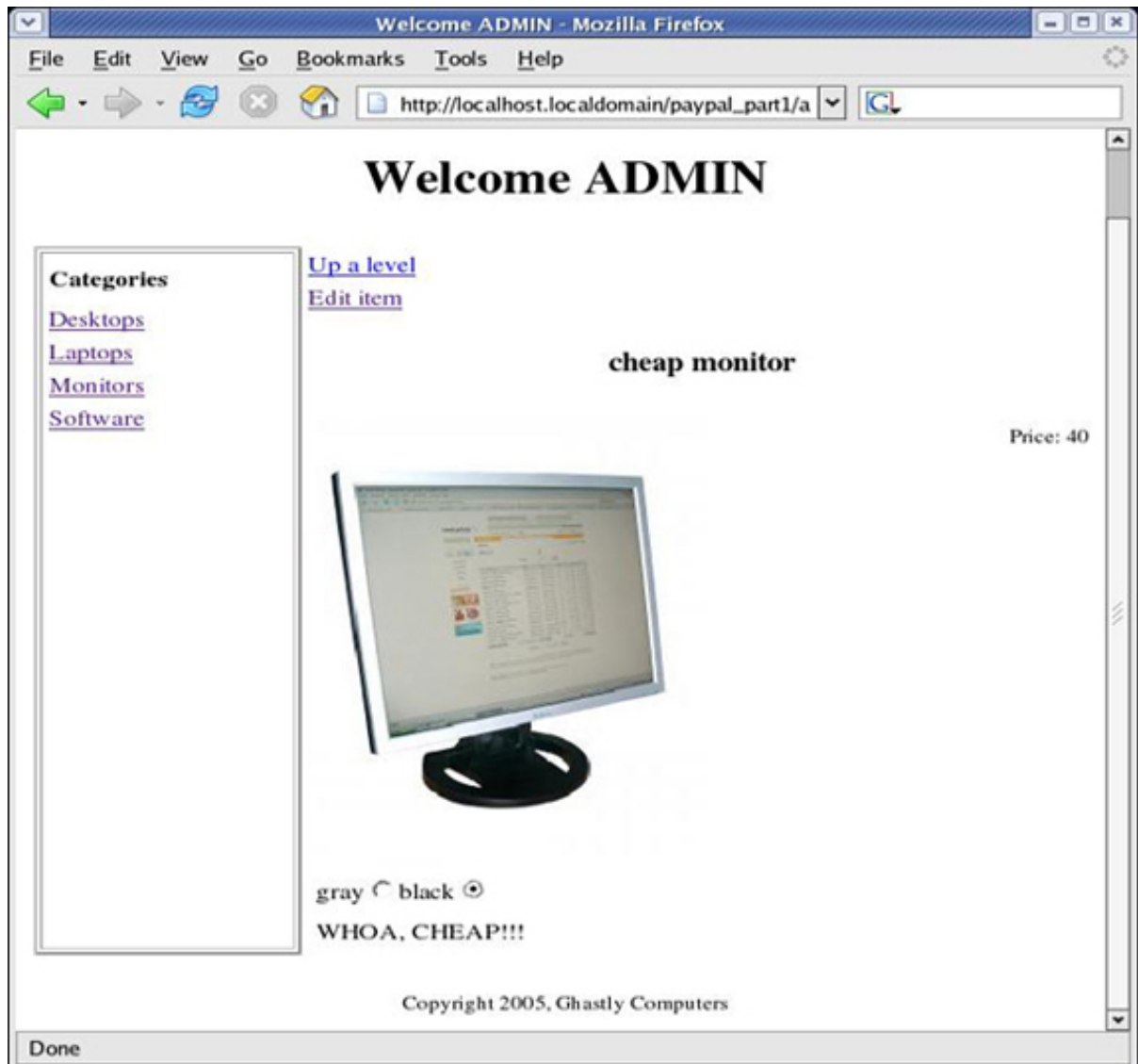
```
$str = "";
$pdo = db_connect();
if($_POST['submit'] == "Edit"){
    $viewItem = $_POST['viewItem'];
    if($viewItem != '')
        $str .= "&viewItem=$viewItem";

    $id = $_POST['id'];
    $sql = "update products set
            category='$category',
            name='$name',
            short_desc='$short_desc',
            long_desc='$long_desc',
            price=$price,
            attribute='$attribute',
            image_name='$image_name'
            where productid=$id";
}
else if(isset($_POST['submit']) == "Save"){
    ...
}

$pdo->exec($sql);
header("Location: storeFront.php?category=$category$str");
?>
```

If the value of the previously submitted button was Edit, the new code in the `if` statement executes. First, the `viewItem` variable in the `POST` array is extracted to return the user to the same item just edited previously. Next, the propagated `id` variable in the `POST` array is extracted for the SQL `UPDATE` statement. The product whose `productid` matches `$id` gets all its fields updated with the values retrieved from the `POST` array, the same values used when inserting new items into the database or in the `else` statement. Finally, the user is redirected to the item he was viewing, enabling him to view the changes immediately. See Figure 12 for the browser output of the newly edited item.

Figure 12. Displaying the newly edited item



Next, you will update the `processGETString` function to account for the new `viewItem` request from the `Displaying item listings` and `Displaying individual items` sections.

Creating the URL, revisited

Some links require that the `viewItem` variable in the `GET` array get passed through, and some require it to be blocked. Modify the `processGETString()`, as shown:

Listing 27. Blocking or passing through the `viewItem` variable

```
function processGETString($vetoString='') {
    $string = '';
```

```
if($_GET['viewItem'] != '' && strstr($vetoString, 'vi') == '')
    $string .= "&viewItem=".$_GET['viewItem'];
if($_GET['category'] != '' && strstr($vetoString, 'ca') == '')
    ...
}
```

This will act to pass through or block the `viewItem` variable, depending on the value of the `$vetoString`, just as you did to the `category` variable in the [Creating the URL](#) section.

Your storefront application is now set up for displaying categories and contained items, viewing single items, along with an item's attributes, as well as adding new and editing existing items to the storefront. Up next is user sessions.

Section 6. User sessions

When users visit your Web site, it's important to keep track of them and the items they add to the shopping cart (which we cover in [Part 2](#)). On top of that, it's important to keep track of values associated with checkout, rather than continuously querying the database (also covered in [Part 2](#)). This section uses session variables to keep track of a user's shopping cart and other variables through sessions variables in the user section of the storefront.

What's a session?

A session stores information about a user, from the last page visited to what the user entered as a favorite color on a survey. The goal of sessions is to track users as they visit your Web site. This is the usage of sessions you will implement in this tutorial.

You may think that the above can be accomplished with cookies. However, cookies are becoming unreliable due to the fact that they store data on local computers. A growing number of people do not want that data stored there. Many virus and registry-fixing programs mark cookies as security threats or vulnerabilities.

Instead, we store session information stored on the server with a unique session ID tied to each user's browser. Learn about using sessions to set a user's session identifier in the next section.

Setting a user's session identifier

It is important to keep track of users as they browse your Web site to keep their shopping carts attached to them. You will accomplish this through a session ID. Modify the storefront in the user section (./storeFront.php) by adding the following code:

Listing 28. Starting the session and setting the associated identifier

```
<?php
session_start();
if($_SESSION['sessid'] < 1002842 ||
   $_SESSION['sessid'] > 1410065407) // generate a new one
    $_SESSION['sessid'] = rand(1002842140, 1410065407);

include('includes/user_functions.php');
include('includes/shared_functions.php');
...
```

It's important to always start the session first because session variables are not accessible until a session has been started. Once started, however, the session variables, once set, always have the same values until changed. This makes tracking a user convenient and reliable because the session variables won't go away until a user closes the browser.

The `sessid` variable, stored as a session variable, is a random number (by choice, you can set the range to whatever you like) between 1002842 and 1410065407. On the computer this tutorial was tested on, 1410065407 was found to be the largest value returned by the `rand()` function. If this variable doesn't exist or is outside of the range of acceptable values, a new one gets generated.

What happens when a user bookmarks your Web site and returns? His shopping cart will be lost. How to stop this from happening is next.

Propagating the session identifier

If a user bookmarks a page on your Web site, then upon return, the session is lost because closing a browser causes session variables to be lost. Here, you will cover how to propagate the session in the `GET` array so when a user bookmarks your Web site and has a shopping cart, the shopping cart can be looked up in the database based on the value of the `sessid` variable. Modify the `processGETString()` function, as shown:

Listing 29. Propagating the sessid variable in the GET array

```
function processGETString($vetoString='') {
...
    $string .= "&category=" . $_GET['category'];
    if($_SESSION['sessid'] != '')
        $string .= "&sessid=" . $_SESSION['sessid'];
}
```

```

    return $string;
}

```

If the `sessid` variable is set, propagate its value as a variable in the URL. Click around on the storefront, and see the `sessid` variable in the URL.

Figure 13. Propagating the `sessid` in the GET array



Next, add the following code to the user section of the storefront in `./storeFront.php`:

Listing 30. Setting the `sessid` in the SESSION array from the `sessid` in the GET array

```

<?php
session_start();
if(isset($_GET['sessid']))
    $_SESSION['sessid'] = $_GET['sessid'];
else if($_SESSION['sessid'] < 1002842140 ||
        $_SESSION['sessid'] > 1410065407) //generate a new one
...

```

The new code in bold takes the `sessid` value in the GET array as the current `sessid` SESSION variable. This is a simple way of allowing customers to return to your Web site without having to re-enter their shopping cart information. You may want to extend your application to include a login, and only re-associate a shopping cart with customers that have created accounts.

Test the new functionality now by bookmarking any page on your storefront with the `sessid` variable set in the GET array. Now close, reopen the browser, and visit the bookmark you just made. Keep clicking around and notice that the `sessid` variable remains the same.

You have now set up user sessions for your application that prepares your application for creating a shopping cart, checking out, and making payments in Part 2 of this series by creating a session identifier to associate the users' shopping carts with until they have bought and paid for the items in their carts.

Section 7. Summary

Hooray! You completed Part 1. You have successfully set up and initialized a Derby database, you've created a basic storefront, and you can now add or edit new items via the Web browser using PHP data objects. You have also set up the application for Part 2 to aid in the development of the shopping cart, which is checking out and making payments.

Downloads

Description	Name	Size	Download method
Source code for Part 1	os-paypal1code_part1.zip	7KB	HTTP

[Information about download methods](#)

Resources

Learn

- For an excellent article on integrating PHP with Derby/Cloudscape, see "[Connecting PHP applications to Apache Derby.](#)"
- Learn how to configure IBM Cloudscape V10.0 and IBM DB2 Universal Database (DB2 UDB) V8.2 servers for access from PHP V4.x and PHP V5.x "[Develop IBM Cloudscape and DB2 Universal Database applications with PHP.](#)"
- To learn about the differences between the open source Apache Web server and IBM's version and view demonstrations of IBM's version running a well-known PHP application, see "[Hosting PHP applications on the IBM HTTP Server.](#)"
- For information about getting Apache V2 and PHP V4.x to work together on Linux, read the [Apache 2 and PHP Installation guide.](#)
- Learn more about PHP data objects and their capabilities in the [PHP Manual.](#)
- Various [HTML form input data](#) you may want to use for representing attributes is available.
- Visit the [IBM Tivoli information center](#) to learn more about the SQL syntax and items.
- For a series of tutorials designed to broaden your PHP skills, see: [Learning PHP, Part 1](#), [Part 2](#), and [Part 3](#).
- Visit the developerWorks [SOA and Web services zone](#) for extensive information.

Get products and technologies

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Tyler Anderson

Tyler Anderson graduated with a degree in computer science from Brigham Young University in 2004 and is currently in his last semester as a master's student in computer engineering. In the past, he worked

as a database programmer for DPMG.com, and he is currently an engineer for Stexar Corp., based in Beaverton, Ore.