# Chapter Four

## Focus on the Database Store

The Database store generates product pages dynamically from an ASCII text file (flatfile) that contains a pipe-delimited database. The Database store has several important advantages over the HTML store. First and foremost, using a database means that the store administrator need not code any HTML. This can save a lot of time and energy for stores with extensive inventories of hundreds or even thousands of items (see Figure 4.1). Because the Web store is designed to generate HTML pages dynamically from the database, the store administrator need only prepare the database (often a preexisting database reformatted to be delimited by the pipe (|) symbol). There is no need to code an HTML page for every product category. And as we will see in Chapter 5, the database back end facilitates more complex search algorithms.

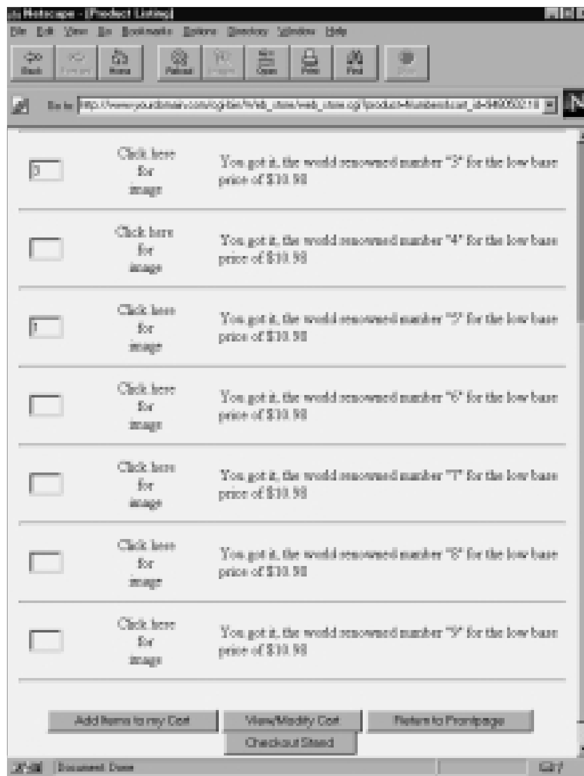


**Figure 4.1** The Database product page.

The second benefit is the ability to lessen dependence on HTML code that can be manipulated by the sly customer. Instead, cart contents are generated and checked from your local database and not from hidden elements in the HTML. There is continual error checking at all points in the shopping process to make sure that prices of items in the cart match prices in the database.

Finally, through the use of the format strings in the Setup file and coded URLs in the data file, you can devise quite complicated and creative displays for products. Even though every product must be presented in a standard format, that format can be quite complex or graphic-intensive, even mimicking some of the more standardized HTML store displays.

As with the HTML store, many companies have opted for the Database version in the past. These versions can all be found in the second half of the

Scripts in Action list at http://www.eff.org/~erict/Scripts/web_store.html. If you are interested in seeing how others have utilized the Database back end, we recommend taking a few minutes to browse through the examples there.

# Customizing the Setup File

As with all versions of this script, the first thing you must do is to customize the Setup file. For the most part, the setup file for the Database store will be like any other implementation. Eighty percent of the setup should remain constant throughout all the versions. However, there will be some differences. In the following section, we will go through **web_store.setup.db** and highlight variables specific to the Database store and explain their usage. Variables not mentioned should be considered globally important to all versions of the store and will be defined the same way regardless of which implementation you choose. They are also discussed in Chapter 2.

## Global File Location Variables

The first variable of interest in **web_store.setup.db**, which is an example setup file in the distribution, is **$sc_html_search_routines_library_path**. **$sc_html_search_routines_library_path** defines the location of **web_store_html_search.pl** which is used to perform searches in the Products subdirectory of the HTML directory. Because we will be searching the data file directly, there will be no need to search the product pages in the Products subdirectory and no need for that variable. However, we will still have to define **$sc_html_product_directory_path,** because we will store list of product pages in that directory.

Since we will be searching the data file in order to generate product pages, **$sc_data_file_path** must be defined and pointed at the location of the data file and **$sc_db_lib_path** must be defined and set equal to the location of **web_store_db_lib.pl**.

**$sc_options_directory_path** defines the locations of the Options files used by the Database store to create option-related HTML. This must be set

relative to the script or absolutely based upon the server's directory structure. The options files themselves will be discussed later in this chapter.

**$sc_store_front_path** must point to a frontpage specific to a Database implementation. Because hyperlink references differ slightly in the Database and HTML implementations, you may not use the same front page for both. When composing hyperlinks, we will use the **product** keyword when pointing to product pages instead of the **page** keyword. The **page** keyword will be reserved for pointing to list of product pages, which are simply HTML pages that contain hyperlinks to product pages.

An example of a hyperlink to a list of products page can be seen in **outlet_frontpage.db.html** in the hyperlink referencing, **Letters.html**.

Figure 4.2 shows **Letters.html** as it appears on the Web:

```
<A HREF = "web_store.cgi?page=Letters.html.db&cart_id=">
```

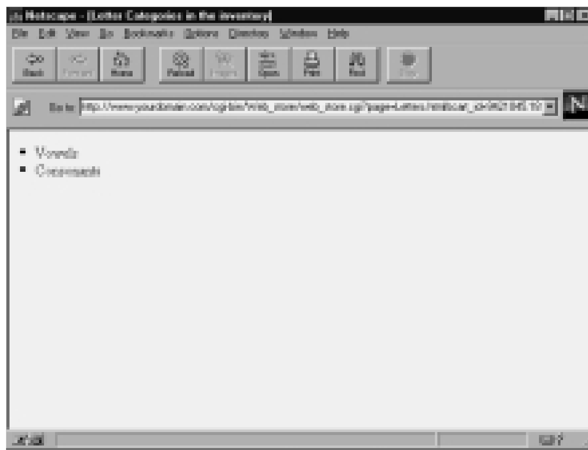


**Figure 4.2** Letters.html

Notice that in the above URL, we have passed to the script the variable **page** with the value of **Letters.html**. (We will discuss the **cart_id=** tag later). **Letters.html** is an example of a list of products page, which is simply a subcategory with links to two product pages, **Vowels** and **Consonants**. Because we do not want to generate that page from the database, we tell the script to go to the **Html/Products** subdirectory and grab **Letters.html** for display.

**$sc_html_product_directory_path** provides it with the information it needs to find **Letters.html**. Thus, if you set this variable equal to:

```
/usr/local/httpd/cgi-bin/Web_store/Html/Products/
```

the script will expect to find the file using the following location:

```
/usr/local/httpd/cgi-bin/Web_store/Html/Products/Letters.html
```

You may also define subdirectories in your hyperlink calls if you decide to partition your HTML product pages. Thus, the following hyperlink would access **Lower_case.html** in the subdirectory:

```
/usr/local/httpd/cgi-bin/Web_store/Html/Products/Letters/Vowels/
<A HREF =
"web_store.cgi?page=Letters/Vowels/Lower_case.html&cart_id=">Vowels</A
>
```

Unlike the HTML store, however, the Database store cannot use the **page** variable to display product pages because no HTML product pages actually exist. The Web store must be responsible for generating product pages on the fly based upon which products the customer wants to see.

This chore is handled by using the product keyword in the hyperlinks leading to product pages. Consider the following hyperlink to the "Numbers" Page:

```
<A HREF = "web_store.cgi?product=Numbers&cart_id=">
```

Notice that we use the keyword **product** to specify those items in the data file we want displayed for the customer. The value associated with the product's keyword will correspond to a field in the database that you will use to group products that should be displayed on the same product page. Thus, every item with a Products field equal to **Numbers** will be displayed when one clicks on the above link.

There will be more details on how to set up your products fields and how the script knows which field to search when we discuss the Database definition

variables in the Setup file, but it is important to drive home the fact that the **page** variable is used only for list of products pages and that the **products** keyword is used to point to product pages.

## Understanding Tags for Maintaining State

Now that we have opened the list of products discussion and introduced the strange **cart_id=** tag, we need to take a moment to explain state maintenance tags in more detail. The trick about list of product pages such as **Letters.html** is that they are tagged so that **web_store.cgi** can automatically insert state information when it filters them.

As we said in Chapter 2, all HTML pages must be filtered through **web_store.cgi** if the customer's state information is to be maintained. When using list of product pages, there are two items of state information that must be passed: (1) the product page from the list that the customer wants to see and (2) the location of the customer's unique shopping cart.

In fact, every internal-store hyperlink (most commonly used in list of product pages) must communicate that information. The problem is that when you write the list of product pages in HTML, there is no way for you to know in advance what the cart i.d. will be. In fact, many different customers, all with different cart i.d.s will be navigating through the same page. There is no way to hard code that information into your HTML. But you can provide a tag which the script will recognize and dynamically exchange for the current value it has for the customer's cart.

This line is handled in the **display_page** subroutine in **web_store.cgi**:

```
s/cart_id=/cart_id=$cart_id/g;
```

This line assures that every HTML page passed to **web-store.cgi** will be filtered for these two state variables. Let's consider the line in **outlet_front page.db.html** discussed above:

```
<A HREF = "web_store.cgi?page=Letters.html.db&cart_id=">
```

In this file, we see that we left the URL encoded hyperlink incomplete. Specifically, we did not set a value for **cart_id**. We simply left it dangling. However, the script is prepared for that and is actually looking for the **cart_id=** flag.

As it is going through the file, preparing it to send to the Web browser, it checks to see if it finds that flag. If it does, it substitutes the incomplete phrase **cart_id=** with **cart_id=[THE ACTUAL CART VALUE]** using the regular expression. Thus, every hyperlink within the store *must* have the cart_id= flag.

## Defining Products with the Database Definition Variables and the Datafile

When the customer clicks on a link for a product page such as **Numbers**, the script must search through the data file and dynamically generate a products page with the appropriate items and the HTML <FORM> code necessary for the customer to order those products. In essence, the script must dynamically generate the HTML product pages discussed in Chapter 3.

However, because the script generates the Name values for all the product input fields, the Database store administrator need not go through the process of coding all the Name attributes for products. That is, no HTML product pages need be created. Instead, she must prepare a data file and describe each field in the setup file so that the script can do it by itself.

Let us first look at the data file. A data file, again, is a simple flatfile ASCII file which has pipe delimited fields and newline separated rows. Below is an example:

```
ID|Category|Price|Description|Image
1|Vowel|10.98|A|~lt~IMG SRC = ~qq~Images/a.gif~qq~~gt~
2|Vowel|10.98|E|~lt~IMG SRC = ~qq~Images/e.gif~qq~~gt~
3|Cons|11.98|T|~lt~IMG SRC = ~qq~Images/t.gif~qq~~gt~
4|Number|13.98|1|~lt~IMG SRC = ~qq~Images/1.gif~qq~~gt~
```

There are a few important things to note about a data file.

First, every data file is a set of standardized items with separate fields in a predictable order. Notice that in every item above, the price is the third field. The standardization of fields is essential. If you misplace fields, the script will

not know how to display your items when it generates product pages. Thus, even blank fields must be defined. For example, if there was no description associated with the number 1 above, you would still have a row that looked like the following:

```
4|Number|13.98||~lt~IMG SRC = ~qq~Images/1.gif~qq~~gt~
```

The field would still be in the data row, it would just be blank.

Second, several special tags must be used to represent characters which may cause trouble within your HTML code (specifically characters used in the HTML code itself). There are three in particular:

**~qq~** represents a double quote mark ("), **~gt~** represents a *greater than* symbol (>), and **~lt~** denotes a *less than* symbol (<). The script knows how to translate these when it uses them to display the customer's cart but you must encode them here so that they will not confuse the <INPUT> tag in which they are embedded. After all, how would the browser interpret the following **NAME** value?

```
<INPUT TYPE = "text" NAME = "<IMG SRC = "Images/1.gif">">
```

The extra quote marks and the greater than and less than symbols would be too confusing for the Web browser to interpret!

Third, every row (item) is separated by a newline character. Thus, you may not include a newline character within the data. However, this should not be a problem since newline characters translate to a space in HTML. If you wished to include a line break, you would use the <BR> tag.

Fourth, each item must be uniquely identifiable with some form of product i.d. field. We must do this so that the script will have some absolute way of differentiating items that it must display (specifically, how to apply options).

Finally, no pipe characters are allowed within the data because the pipe is used as a field delimiter. If you included a pipe character in your data, it would cause the script to incorrectly display your item.

Once we have created our data file, we must describe it for the script in the setup file. That way the script will know how to define each product in the HTML it must dynamically generate.

**%db** is an associative array that contains a mapping of your own customer defined fields to the index number of the fields as they appear in the flatfile database.

For example, in our sample data file above, we would create the following associative array:

```
$db{"product_id"} = 0;
$db{"product"}  = 1;
$db{"price"}    = 2;
$db{"name"}     = 3;
$db{"image_url"} = 4;
```

NOTE

Remember, fields start counting at 0.

**@sc_db_display_fields** is an array containing the descriptive headers for the fields in the database we wish to display to the user when they do a query search. Notice that these headers need not include every field defined in **%database**. You may display only some of the information contained in the database row if you so desire. The array takes advantage of the indexes defined in **%database** and each descriptive element in this array corresponds to the index numbers in **@sc_db_index_for_display**.

**@sc_db_index_for_display** is an array containing the index numbers of the **db** fields that correspond to the **display_fields** array. To access those index numbers, we just utilize the **%darabase** associative array. There must be one index number in this array for every descriptive element in **@sc_db_display_fields**.

**@sc_db_index_for_defining_item_id** is an array containing the **database** fields that correspond to the fields from the database that you wish to associate with a customer's cart when he or she selects that item for purchase. These field values will be used to determine the fields that make

up each item row in the customer's cart. If you do not put a database field in this array, it won't be put in the cart and will not be available for display when customers view their carts. It is mandatory that the Price and Options database fields become incorporated into the cart as these are used for cart subtotal calculations.

**$sc_db_index_of_price** is an index to the field in the database that contains the price. This setupvariable is used by the Web store to decide how to calculate and display money. In the example above, this value would be 2. Remember that even if you do not display price, it must be defined here so that the script will be able to do subtotaling.

**@sc_db_query_criteria** is an array containing the criteria that can be used to search on the database. This is a powerful search mechanism. Though searching is discussed in greater detail in Chapter 5, we will touch on it here as it relates to developing product pages on the fly. The array contains pipe-delimited fields inside each list item. The fields are the following:

1. *Form variable name* This is the variable name which you want to associate with the products to display. For example, if you have the following hyperlink for getting a product page:

   ```
   <A HREF = "web_store.cgi?product=Numbers&cart_id=">
   ```

   then the form variable name would be **product**.
2. *Index into the database that this criteria applies to* This list corresponds to the **%db** associative array in the same way that **@db_index_for_display** does. Thus, if you want the product field to be searched by keyword, the hyperlink variable would be **product** as above and the index into the database would be **1** according to our sample **%db** associative array.
3. *operator for comparison* This field is used by the script to determine what logical criteria to apply when searching the database. Possible values include >, <, >=, <=, =, != (not equal),

   and the operator is compared the following way:

   ```
   form_variable OPERATOR database_field_value
   ```

That is, item 1 above is the left-hand side of the operator and item 2 above is the right-hand side of the operator. Typically, we'll use "=" for generating product pages since we will be searching by keywords.

4. *data type of the field* This field determines how the operator in (3) gets applied to the data. Typically we will use the string comparison.

Perhaps some examples are in order. Typically, you just want to do a search on a product category and include that search term within URLs in a frontpage such as the following:

```
web_store.cgi?product=Vowels.
```

To do so, we must set the form variable equal to the above (**product**), set the second field equal to the field in the database corresponding to a product name (for example, 1), set operator =, data type **string** to do a keyword search that is case insensitive. Thus, **@sc_db_query_criteria** would be equal to:

```
("product|1|=|string")
```

**$sc_db_max_rows_returned** is the maximum number of rows you will allow to be displayed to the user as the result of a query. If the query gets above this number, customers are presented with a message letting them know that they need to narrow their query.

# Cart Definition Variables

Now that we have gone to the trouble of defining our data file, it is time to make it pay off. The Cart Definition variables will be used to index the product information coming in to the script in the form of text field Name form data. Let's go through the cart definition for the sample data file proposed previously.

Given the structure of the data file and the value of:

```
@sc_db_index_for_defining_item_id
```

%cart must be

```
$cart{"quantity"}            = 0;
$cart{"product_id"}          = 1;
$cart{"category"}       = 2;
$cart{"price"}               = 3;
$cart{"image_location"}             = 4;
$cart{"options"}             = 5;
$cart{"price_after_options"}        = 6;
```

All other variables will depend upon utilizing these array elements.

# Order Form Definition Variables

These variables will basically be the same for all versions of the Web store regardless of whether they take their information from product pages or a data file because they will all have the same order processing interface.

# Store Option Variables

Of course the most basic variable definition for the Database store is **$sc_use_html_product_pages**. The only way to dynamically generate product pages is to set this variable to **no**. If it is set to **yes**, the script will look liek the predesigned HTML product pages in the **Html/Products** subdirectory.

# HTML Search Variables

Because we will use the search routines in the Database search library, we will not need to worry about any of these variables. They are all specific to the HTML store.

# Error Message Variables

These variables are global for all versions of the Web store and have already been discussed in sufficient detail.

## Miscellaneous Variables

**$sc_product_display_title** is the title that you would like to appear on your product pages. Unfortunately, one of the limitations of the Database store is the inability to custom design titles.

**$sc_product_display_header** is the header HTML used to display products in the database based implementation. Notice that we use **%s** to substitute for any given product information. The script will substitute product data for each product in place of the **%s** when the variable is actually used.

<NOTE> There must be a **%s** for every item in **@sc_db_index_for_display** because those elements will be what gets substituted for each **%s** in the order they are defined in the array.

**$sc_product_display_footer** is the footer for each product

**$sc_product_display_row** is the **%s** embedded product row variable.

## Understanding Options

Options are unique characteristics that can be applied to a product. For example, a basic T-shirt might come in black, blue, or green or small, medium, or large. Options modify a generic product. We need a way to communicate to the script what an option is, which item it belongs to, what effect it will have on the base price of that item, and the value set by the customer. Figure 4.3 depicts a product page with options:

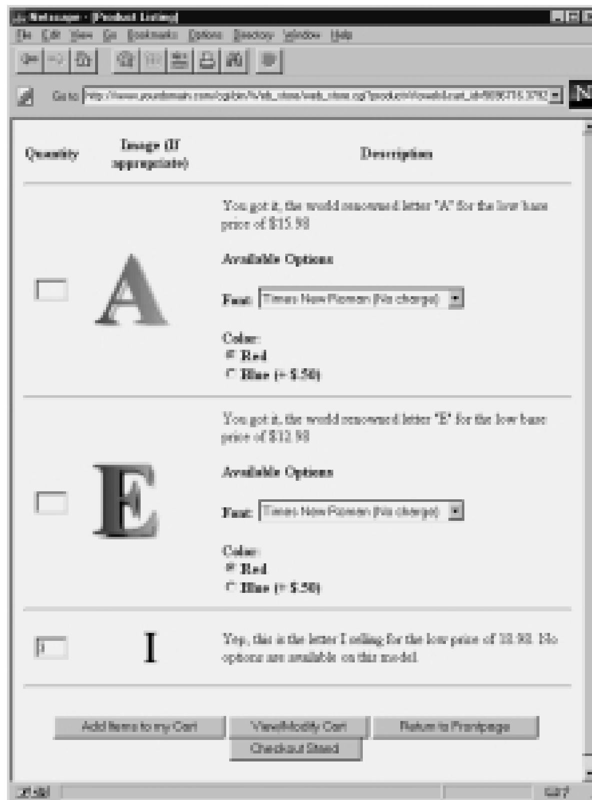


**Figure 4.3** Product page with options.

The first part in this process is to make sure that we associate options with items for sale in the data file. To do so, we will use a special database field with a specific format. Here is an example row from the distributed version of the data file.

```
0010|Vowels|15.98|The letter A|~lt~IMG SRC = ~qq~Html/Images/a.jpg~qq~
ALIGN = ~qq~left~qq~~gt~|You got it, the world renowned letter
“A”|%%OPTION%%option.html
```

Notice that the fifth field reads **%%OPTION%%option.html**. The special way that options are denoted in the database is by using the format **%%OPTION%%filename** in the data file. This string includes two important pieces of information.

First, the string begins with **%%OPTION%%**. This is a flag that lets the script know that it needs to deal with this database field as if the field were an option. When the script sees the flag, the string will then look to the string following the flag to see which file the string should load. Thus, in this example, the script would load the file **option.html** for display.

NOTE

Why go through all the trouble? Basically, we need to create a system that will handle large chunks of similar HTML code within the database. Options on product pages are likely to be repeated fairly often. For example, every item in a database might have an option like tape, CD, or LP. By creating one **option.html** file, we could easily put all the code into one shared location and not worry about typing it in for every single database entry.

The option file is opened and read and every line of the option file is appended to the product when it is displayed and the file is closed again.

However, options files themselves have another important flag, the **%%PRODUCT_ID%%** flag. Because options must be associated with the items they modify and because you do not know in advance what those items will be, you must let the script make this connection at run time. The connection is achieved by the flag. The current product i.d. number will be substituted by the script for the **%%PRODUCT_ID%%** flag, which is a mandatory tag contained in all options files.

With the exception of this **%%PRODUCT_ID%%** flag, the option definitions follow those in the HTML store. First you must associate options with the items they modify. We make this connection by using the NAME argument of the form tag that defines each option. Below is an example of using a Select menu for options.

```
<P><B>Available Options<B><P>
Font: <SELECT NAME = "option|1|0001">
<OPTION VALUE = "Times New Roman|0.00">Times New Roman (No
charge)
<OPTION VALUE = "Arial|1.50">Arial (+ $1.50)
<OPTION VALUE = "Chicago|2.00">Chicago (+ $2.00)
</SELECT>
<P>
```

In this case, the NAME syntax breaks down as follows:

1. *Option flag* This flag tells the script that the incoming data is an option, not an item. Thus, the first field in a pipe-delimited option Name value will *always* be **option** just as item Name tags *always* begin with **item-**.
2. *Unique sequence number of the option* Each item for sale may have several options associated with it. It is essential that each gets its own number. If item #0001 had all its options called **option|0001**, it would be impossible to parse them separately. So we will name them uniquely such as **option|1|0001** for color, **option|2|0001** for size, or **option|3|0001** for brand name.
3. *I.D. of the item that the option is associated with* Notice that this i.d. is the same as what was used in the Name argument for the quantity text box in a product field box. This is deliberate and essential. Options must be associated with the items they modify. This is where we must use the **%%PRODUCT_ID%%** flag.

Finally, notice that options also contain Values, which are two field pipe-delimited lists containing an option description and an option price. The option description will be used for display in the user's cart and the price option will be used to modify the base price of an item.

The following is an example of using a radio button to create an option. It uses the same naming conventions as the <SELECT> tag but is included here for variety:

```
Color: <BR>
<INPUT TYPE = "radio" NAME = "option|2|0001"
    VALUE = "Red|0.00" CHECKED>Red<BR>
<INPUT TYPE = "radio" NAME = "option|2|0001"
    VALUE = "Blue|.50">Blue (+ .50)
```

In other words, product number **0001** has two possible options. Font type is option number 1 and color is option number 2. The customer may choose from three font types. If she chooses Arial, $1.50 will be added to the base price of the item. She can also order red or blue. If she chooses red, nothing will be added to the base price of the item.

If this option modified the letter *A* discussed above, and a customer ordered three *A*s, all red with Arial font, the cart row would appear as follows:

```
3|1|Vowels|10.98|A|~lt~IMG SRC = ~qq~Html/Images/a.jpg~qq~ ALIGN =
~qq~left~qq~~gt~|Arial 1.50, Red 0.00|12.48|1
```

Below, for you to review, is the code for the sample **option.html** file:

```
<P>
<B>Available Options<B>
<P>

Font:
<SELECT NAME = “option|1|%%PRODUCT_ID%%”>
<OPTION VALUE = “Times New Roman|0.00”>Times New Roman (No charge)
<OPTION VALUE = “Arial|1.50”>Arial (+ $1.50)
<OPTION VALUE = “Chicago|2.00”>Chicago (+ $2.00)
</SELECT>

<P>

Color:
<BR>
<INPUT TYPE = “radio” NAME = “option|2|%%PRODUCT_ID%%”
    VALUE = “Red|0.00” CHECKED>Red<BR>
<INPUT TYPE = “radio” NAME = “option|2|%%PRODUCT_ID%%”
    VALUE = “Blue|.50”>Blue (+ $.50)
```

# Summary

To utilize the Database-based interface, you must satisfy several requirements. First, you must modify the following variables in the Setup file as discussed above:

```
$sc_html_product_directory_path
$sc_data_file_path
$sc_db_lib_path
$sc_options_directory_path
$sc_store_front_path
```

```
@sc_db_display_fields
@sc_db_index_for_display
@sc_db_index_for_defining_an_item
$sc_db_index_of_price
@sc_db_query_criteria
$sc_db_max_rows_returned
$sc_use_html_product_pages
$sc_product_display_title
$sc_product_display_header
$sc_product_display_footer
$sc_product_display_row
```

Second, any list of product pages that you create for navigation must be hard-coded with the **page** and **cart_id=** flags for filtering.

Third, you must create an ASCII text file (flat file) database that includes pipe-delimited database rows separated by the newline charcter. The fields of this database must correspond to the **db_index** variables defined above.

Finally, options files must be created for any options that you wish to use to modify products in your database. The options must be prepared for filtering with the **%%PRODUCT_ID%%** flag and must be included as specially flagged fields in the data file.