SUGGESTED ALTERATIONS

The goal of this book is to teach sound e-commerce code and methodologies. In thinking of the two sites I'd use to achieve that goal, I tried to come up with examples that best portray the breadth of what e-commerce can be. Being who I am, however, I also dreamt up about three dozen other ideas for each one that actually made it into the book. Rather than discard good brainstorming, I thought I'd finish this chapter with some suggestions as to how this particular example could be expanded. While none of these ideas will be fully developed in the text, you'll see more than enough, in terms of MySQL and PHP, to realize them yourself, should you choose.

tip

Some of these ideas may also be partially developed in the downloadable scripts from **www.DMCInsights.com/ecom/**.

Logging History

One addition you could add to the site would be to log all the pages people visit. You'd need to create a **history** table, defined as:

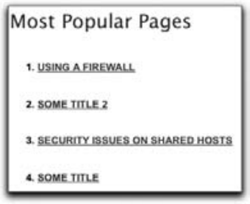
```
CREATE TABLE history (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
`user_id` INT UNSIGNED NOT NULL,
`type` ENUM('page', 'pdf'),
`page_id` MEDIUMINT UNSIGNED DEFAULT NULL,
`pdf_id` SMALLINT UNSIGNED DEFAULT NULL,
`date_created` TIMESTAMP  NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
KEY (`page_id`, `type`),
KEY (`pdf_id`, `type`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

The **page.php** script would add a new record to this table every time the page is loaded (by an active user):

```
INSERT INTO history (user_id, type, page_id) VALUES ($_SESSION['user_id'],
➥'page', $_GET['id'])
```

The **view_pdf.php** page would do the same thing, but change the type to *pdf* and use the **id** value from the **pdfs** table for the *pdf_id*.

The **history** table would then be able to provide the user with a history of every page and PDF they've seen. This table would provide the administrator with indications of the most popular pages. That information could also be used on the public side, perhaps to display the 10 most popular articles on the home page (**Figure 5.19**). Here's the **JOIN** query that would return the 10 most frequently viewed pages:


Most Popular Pages

1. USING A FIREWALL
2. SOME TITLE 2
3. SECURITY ISSUES ON SHARED HOSTS
4. SOME TITLE

Figure 5.19

tip

You could decide whether each time a user visits the same page should be reflected in the **history** table or not.

```
SELECT COUNT(history.id) AS num, pages.id, pages.title FROM pages,
➥history WHERE pages.id=history.page_id AND history.type='page'
➥GROUP BY (history.page_id) ORDER BY num DESC LIMIT 10
```

Making Notes

Another added service would be allowing users to make notes on pages. The table would be:

```
CREATE TABLE notes (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
`user_id` INT UNSIGNED NOT NULL,
`page_id` MEDIUMINT UNSIGNED NOT NULL,
`note` TINYTEXT NOT NULL,
`date_created` TIMESTAMP  NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE (`user_id`, `page_id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Next you add a form on each page containing just a textarea. When the form is submitted, the information is stored in that table. On the user's viewing history page, you could easily indicate those pages for which the user has left notes (by performing a left outer join from the **pages** to the **notes** table). On a specific content page, you could easily retrieve the notes a user previously made on its content.

Recording Favorites

To make the site easier to use, especially as you create more content, users might appreciate being able to bookmark their favorite content. To do that for the HTML pages is simple. First, create a table defined like so:

```
CREATE TABLE favorite_pages (
`user_id` INT UNSIGNED NOT NULL,
`page_id` MEDIUMINT UNSIGNED NOT NULL,
`date_created` TIMESTAMP  NOT NULL,
PRIMARY KEY (`user_id`, `page_id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

This is a *junction* table, used to manage the many-to-many relationship between **users** and **pages**. It has only three columns and the first two together constitute the primary key (that is, it's a compound primary key). Each favorite for each user will represent one record in this table.

Next, create an image displayed with the content that, when clicked, passes the page ID along in the URL. On the **add_to_favorites.php** page, you'd store the user's ID from the session and the page ID from the URL in the **favorites** table.

You could then create a **favorites.php** page, linked through the Manage Account section for logged-in users, which displays all the favorites. It would need to perform a **JOIN** to get the favorite content. Or you could mark the user's favorites next to each item in a history page.

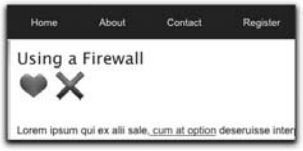
tip

An even better addition would be to use Ajax to submit the user's favorites indication to the server behind the scenes.

On each content page, you could add a check to see if the page is already in the user's favorites. If it is, you would display text and an image indicating such, and perhaps clicking the link would remove it from the user's favorites by sending the page ID along to **remove_from_favorites.php**.

Here's what the HTML for adding and removing favorites might look like (**Figure 5.20**):

```
echo '<p><a href="add_to_favorites.php?id=' . $_GET['id'] . '">
➥<img src="/images/heart_48.png" border="0" width="48"
➥height="48" /></a>
<a href="remove_from_favorites.php?id=' . $_GET['id'] . '">
➥<img src="/images/cross_48.png" border="0" width="48"
➥height="48" /></a></p>';
```



Lorem ipsum qui ex alii sale, cum at option deseruisse inter

Figure 5.20

tip

The images used in this example are freely available at **www.wefunction.com/2008/07/function-free-icon-set/**.

Making a note of favorite PDFs requires a bit more thought. You can't easily add links to the PDF itself, so you'd have to put the *Add to Favorites* link somewhere else, like on the page that lists all the PDFs. This means the user would have to read the PDF, then go back to that page to flag it. As for storing favorite PDFs in the database, you could create a **favorite_pdfs** table, just like **favorite_pages**, or create a **favorites** table that stores both, like the history example.

Rating Content

Continuing on this same theme, users could indicate a rating on each page. You would store this in a table like:

```
CREATE TABLE page_ratings(
`user_id` INT UNSIGNED NOT NULL,
`page_id` MEDIUMINT UNSIGNED NOT NULL,
`rating` TINYINT UNSIGNED NOT NULL,
`date_created` TIMESTAMP  NOT NULL,
PRIMARY KEY (`user_id`, `page_id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Unlike with the **history** table, where you may want to record every time every user visited a page, you would only want each user to be able to rank a page once. You can accomplish that using this query:

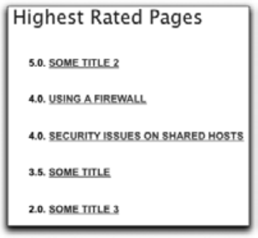
```
INSERT INTO page_ratings (user_id, page_id, rating) VALUES
➥($_SESSION['user_id'], $_POST['id'] , $_POST['rating'])
```

tip

An even better addition would be to use Ajax to submit the user's rating to the server behind the scenes and immediately update the page to reflect the rating.

The rating would be an integer, from say 1 to 5. The easiest way to submit the rating would be to use a drop-down menu and the form would post the rating back to **page.php**. The **page.php** script would then need an extra block at the top of the script that checks for a POST request and adds the record to the database if all the data successfully passes the validation tests.

The ratings could then be displayed to the user on their favorites and history pages. The ratings could also be used by the administrator to see the best reviewed content, which again might be turned into a listing on the home page (**Figure 5.21**):



Highest Rated Pages

5.0. SOME TITLE 2

4.0. USING A FIREWALL

4.0. SECURITY ISSUES ON SHARED HOSTS

3.5. SOME TITLE

2.0. SOME TITLE 3

Figure 5.21

```
SELECT ROUND(AVG(rating),1) AS average, pages.id, pages.title FROM
➥pages, page_ratings WHERE pages.id=page_ratings.page_id GROUP BY
➥(page_ratings.page_id) ORDER BY average DESC LIMIT 10
```

Again, this could also be done for the PDFs, but the logic would need to be added to the **pdfs.php** script.

Making Recommendations

Implementing a recommendations system is a fantastic way to encourage people to use your site and increase your business. Whether it's something like Netflix that recommends titles based upon your body of ratings and viewing history, or Amazon that recommends related and alternative products to those you're looking at, are in your cart, or were just purchased, there's a lot you can do with recommendations. The logic with a recommendations system may take some effort, though.

tip

If you want some concrete SQL and PHP to start implementing a recommendation system, contact me through my support forum (**www.DMCInsights.com/phorum/**).

One implementation would have the administrator making the recommendations: For each page, the administrator could use a drop-down menu (which allows for multiple selections) to associate related content. The **recommendations** table would store for each page an ID of other recommended pages. If you wanted to allow for recommendations across content types, that would require a more complex table structure.

An alternative way to implement a recommendation system would be to base recommendations on user rankings. For example, if user Alice gave page 1 five stars and page 2 four stars, and user Bob gave page 2 four stars, Bob *might* really like page 1, as well. Such a recommendation system could be more accurate than the administrator-created one, but relies on lots of sound logic and filtering. The more accurately you can equate Bob's tastes to other user's tastes, the better you can make recommendations.

Placing HTML Content in Multiple Categories

Just like a blog can file posts under multiple categories (and multiple tags), you may decide that some of your HTML content should be listed in multiple categories, too. That would result in a many-to-many relationship between the **pages** and the **categories** tables, so you would have to take the **category_id** out of the **pages** table and use a junction table instead. That table would be defined as:

```
CREATE TABLE pages_categories (
`page_id` MEDIUMINT UNSIGNED NOT NULL AUTO_INCREMENT,
`category_id` SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
PRIMARY KEY (`page_id`, `category_id`)
)
```

You wouldn't need to change the footer, which links to every category, but the **category.php** page would have to perform a join across **pages_categories** and **pages**:

```
SELECT id, title, description FROM pages, pages_categories WHERE
➥pages.id=pages_categories.page_id AND pages_categories.category_id='
➥. $_GET['id'] . ' ORDER BY date_created DESC
```

The **add_page.php** script would need to allow for multiple categories to be selected:

```
<select name="category[]" multiple="multiple" size="5"
➥<?php if (array_key_exists('category', $add_page_errors))
➥echo ' class="error"'; ?>>
```

The logic to make the categories sticky would have to be changed as **$_POST['category']** is now an array:

```
if (isset($_POST['category']) && (in_array($row[0], $_POST['category']) )
➥echo ' selected="selected"';
```

Then, the handling part of the script would have to validate that **$_POST['category']** has a **count()** greater than zero. And after the record was added to the **pages** table, you would insert one record into **pages_categories** for each selected category.