A General Purpose E-commerce Engine

CS470 / Final Writeup

Jake Feasel

4/25/2005

1.0 Introduction

This application is intended to be a " boxed-product" type of system, for any small business wanting to sell their inventory online, at the lowest possible cost. The goal is that a very simple, generic system can be quickly installed and easily customized. It is also expected that the store owners and site visitors aren't technically savvy, so the product must be intuitive. Since this is expected to be as low of a cost as possible, all components are built to work with Free software systems.

2.0 Project Overview

This system started as a service for my parents-in-law and their small business. I t is this base system built for them that I made generic and improved. It was a PHP 4 application coded around an early "Fusebox" framework, with a MySQL back-end. The features it had served the basic business needs for the original client, but I added some important new features that should make it more useful.

Features

The system has all of the expected features needed for a small scale site. T he site administrator can create new products, complete with multiple in-line images. Shipping costs are unrestricted, either by weight levels or by units to measure the weight (lbs., oz., kg, etc...). General site settings are stored in a database table and are also managed by the administrative site. In-coming and historical orders are all available for processing and record-keeping.

From the public perspective, t he site shows a navigation menu with choices of products in each category. As currently implemented, the products for the current category show up under that category. Different categories can be chosen to display and shop from. A s hopping cart is maintained in the user's session. Prices are also saved in the session - this is to prevent the problem of a site administrator altering the prices of a product at the same time that a user is purchasing that product. Storing the price as the user has seen it prevents any problem that could arise in this scenario. After the user reviews their "cart", t hey can either remove items from it or proceed to checkout. The checkout screen prompts the user to enter the necessary contact information for payment processing and delivery. An email receipt is generated, as well as displayed on the final screen. Notice is also sent to the site operator at this point, prompting them to process the newly arrived order.

There are two types of changes I made on this base system - code cleanup and feature addition.

Code Cleanup

The first step in this project was porting the existing code base to the newly released Fusebox 4 .1 framework. In addition, this step involved porting the display logic to the Smarty template system. This was a very lengthly portion of the project. The first problem faced was overcoming the interface between Fusebox and Smarty. I soon discovered that there were no documented

methods available on the Internet describing their use together. F usebox itself didn't support Smarty templates natively, so some customization was required.

Fusebox uses XML as a stripped-down flow-control language, used to connect different "fuses" of code (individual files). The latest version of the Fusebox core files include the ability to extend the base XML grammar with a custom lexicon. This feature is still considered "beta", but it was the only option that I had available to make use of the Smarty system. Using this option, I defined two new XML verbs - "smt:display" and "smt:assign". These verbs mapped very closely to the most important base smarty commands - "display" is used to render a template and "assign" is used to pass variables into the smarty namespace. The finished project may serve as a quite useful example application for others who wish to use these technologies together.

Feature Addition
I had hoped to add a photo gallery feature to the the application. I had some progress toward this goal - necessary tables, framework stubs, initial code, etc... Unfortunately time did not permit completion of this feature. Given its current status, however, I estimate that it would not take many hours to complete.

An important feature did get added to the system - namely, tight integration with Google's Froogle shopping site. This is accomplished by querying the database and formatting the results in a tab-delimited, spreadsheet-type file. This file then gets sent to a google-operated ftp site. This feature is triggered by the click of a link in the admin menu. The result of this FTP transaction is displayed on the screen following the click of the link.

3.0) Project Requirements

There are two motivating factors in this project - the needs of my in-laws, and the desire to make the general application. My in-laws came to me asking for a site to sell their product, and all they had was a printed brochure to base it on. Most of the requirements were left to my discretion. Since they don't really know how an online business operates, I a m the main motivator behind these new features and code cleanup.

3.1) Functional Specifications

- The application should be relatively easy to configure and customize. While I don't expect a complete novice to be able to set it up, I don't want to require someone to be an expert in my code for them to use it. T he only requirements would be some experience with HTML design and the (well-documented) smarty template language. T hese skills will be used to update the global site template, which has things like the site headers and footers. Inner content could also be updated, a lthough this probably wouldn't deviate much from the provided files.
- The Fusebox structure focuses on the separation of Model, View, and Controller components. I will provide a standard, documented set of engine calls that can be made in the controller and given to the " view", which in this case will be the smarty templates. These engine calls will make it easy for the web designer to incorporate whatever look they would like, since they can loop and process over these results in any way.
- The integration with Froogle involved creating a file that provides a spreadsheet conforming to the standard defined by Froogle. This spreadsheet contains all relevant

inventory information needed to include in their index. Merchant registration with Froogle will be all that is required to complete this process.

3.2) System Requirements

The only hard requirement of this application will be that the servers it runs on will need to have PHP installed and a database accessible. PHP is Free Software, and is available for nearly every operating system. Whatever server the code is installed on, it will need to have a port open to the public internet. It is recommended that the server uses port 80, the standard web port. It is also recommended that the web-server have an SSL certificate installed, so secure transactions can occur. The server with the secure certificate must be the same server running the rest of the code base.

The database used can be nearly any vender, but there will be SQL scripts provided to create the schema in MySQL and PostgreSQL. These servers will also be considered "supported". Any database that is supported by the Pear DB project should work, however. MySQL and PostgreSQL both are Free software and available for nearly every operating system.

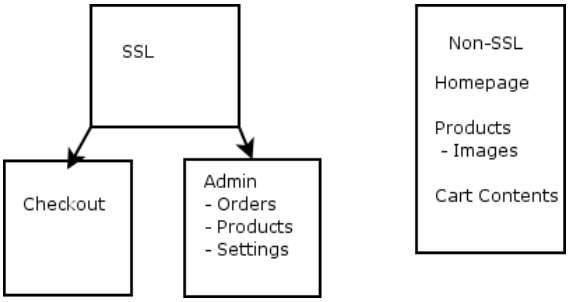
4.0) System Design

Fusebox 4.1 will allow a Model-View-Controller system that isn't a true Object Oriented design, but has many of the same benefits. Most importantly, this break down allows for the decoupling of logic from display that is the typical shortcoming of a web application.

4.1) Data Structures

The most complex data structures will be the query objects that get created and passed to the smarty templates. These queries will be looped over in the display. In the engine, there may be some looping done to calculate total costs. In general, though, there won't be many complex data structures needed.

4.2) System Architecture

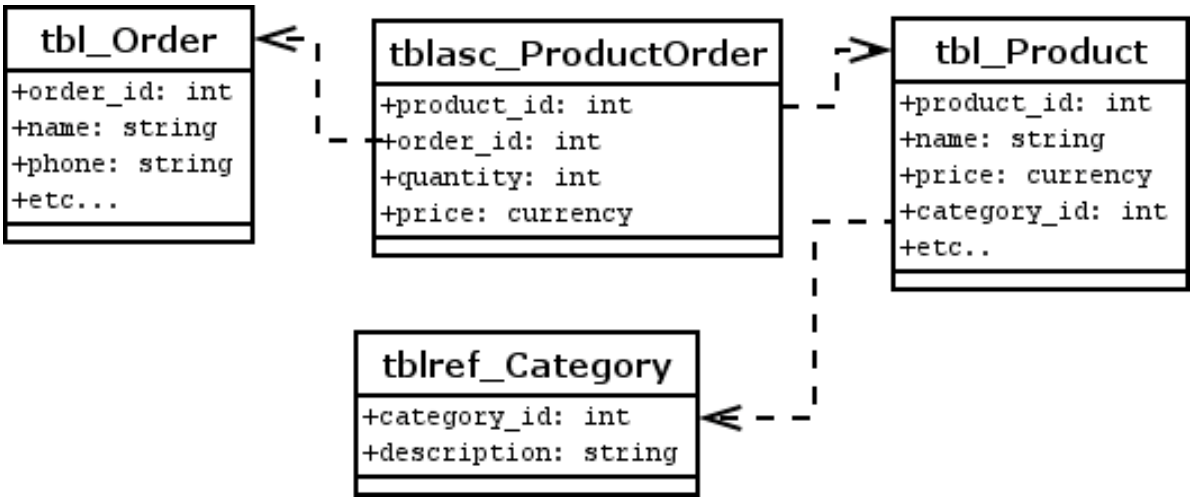


Basic division of application code

The code is arranged in two parts - SSL and non-SSL. The non-SSL code handles basic, non-private data that anyone can access. The SSL code is for the protected business transactions. In order for sessions to be shared from one part to the other, both parts of this code must be served by the same PHP server.

In Fusebox terminology, major groups of functionality are called " circuits". The circuits I defined are listed below:

- Admin: This circuit contains all of the code for the administrative back-end. It is the largest circuit, as every aspect of the database needs to be updated from this area.
- Checkout: This circuit has all of the logic used in completion of an order by a user. Notably, the receipt code has been unified and is kept in this circuit.
- Store: This circuit contains all of the information that a public user sees in their general site browsing.



Core database tables

5.0) Development Process

Most of the time was spent on porting the existing code into the new framework. Both Fusebox and Smarty are very strict in how the allow you to code. This strictness has many advantages - primarily, that the code is clean and simple. The process of moving the existing, less strict code into this more strict environment was a source of difficulty. Also, the shear size of the code-base being ported was more than was originally anticipated. Many details of the existing system were forgotten in the planning phase, so estimates were off. This caused a few of the features originally planned for to be delayed. These currently unfinished features include integration with PayPal and Amazon, as well as the Photo Gallery (which was started, but not completed).

6.0) Results

A complete, functional demo site is now available with the new framework. Revamping the code with the Fusebox framework has allowed many previously sloppy workarounds to be cleaned up and more maintainable. For example, in the old code base, the receipt display code was copied and slightly modified in three or four places throughout the system. Now, that logic has been unified into a single, much more maintainable location. This will make it far easier for future developers to customize the look of the receipt.

The use of smarty templates was a bit of a challenge. Many of the nice features of the base grammar provided by the Fusebox core files are not available when using the corresponding smarty tags that I built. This is especially noticeable when defining "XFAs" - Exit FuseActions. These are variables that point to other fuseactions in the application. Normally, the XFA tag is used to define them. Those variables wouldn't be available in the smarty namespace, however, so I had to use a much weaker work-around - setting "fake" XFAs like they were normal variables.

Another hope that I had was to incorporate "Fusedocs" into all of my fuse files. These are comments that describe the purpose, history, and variable IO for each of the files. Fusedocs are a standard way to do this, using XML at the top of every file. I included Fusedocs in several of my files, but this was a very time-consuming practice that eventually was abandoned. Before I can present my application as a sample for the Fusebox community, I will need to go through the rest of my fuses and add these into the files.

7.0) Conclusions

While it can be a lot of work to migrate existing systems to Fusebox and Smarty, there is significant benefit to using a mature framework. Fusebox 4.1 for PHP is very new, so there isn't a large base of users for it yet. Smarty has a very large user base, and is quite mature and widespread. This application, when released back to the Fusebox community, should serve as a useful example of how to bring these two technologies together in a real-world system. In addition to serving as an example for future Fusebox and Smarty developers, this application itself may find wider deployment. This would be due to its sole dependence on free software and a clear, well designed code-base.

Appendix

Installation
To install, an existing PHP server must be available (recommend 5 or greater). This PHP server must be configured to support PEAR::DB and Smarty templates. Also, either PostgreSQL or MySQL must be available. Ideally, this web server would also have SSL available, and SMTP server configured for PHP, and HTTP level authentication enabled for the site. Installation consists of unzipping the code archive into a PHP server setup this way. Then, the files "mysql_build_database" or "pgsql_build_database" (whichever appropriate) should be ran inside an otherwise empty database. The file "myGlobals.php" in the root folder will need to be updated to reference the database server. The folders "parsed", "templates_c", "images" and "Admin/templates_c" need to be set to be writable by the web-server. At this point, the demo application should be up and running. HTTP level authentication should be applied for the Admin folder. An example .htaccess file is provided in that directory for Apache users.

Configuration
A web designer will need to be available to provide a custom look-and-feel for the particular store to be operated by the system. Updates to the style.css Cascading Stylesheet file should be done for most style color schemes. Most of the files in the View folders (named dsp_*.php) will most likely need to be updated to reflect the site operator's language. The administrative section has an area for site settings - these are defaults that need to be changed to reflect the operator's site. Site categories will also need to be entered into the database back-end prior to use.

Usage
After a site has been customized by a web designer, t he site operator can start creating products to be available on the site. First, the operator browses to the Admin website, where they are prompted to login with an HTTP Authentication username/password prompt. After authenticating correctly, they would then click on "Manage Products" and "Add a product". At this point they can start entering prices, weights, a nd descriptive text. A main photo for the item should be uploaded, as well as paragraphs of text (each with their own optional image). T he paragraph function works by repeatedly editing the product page. Each time the page is entered, a new paragraph entry form is available for further input. After all of the values are completely entered, the product needs to be " activated". T his is done on the main product list screen. Disabled products are shown with a checkbox that is checked by their name. Un-checking this box will enable the product and it will then be listed on the public page. Repeat this process for all products. After all products have been entered, Shipping and Handling rates need to be adjusted. If the site operator has a froogle merchant account (recommended), they enter their froogle username and password in the "Site Settings" area. T hey can at this point click on the "sync with froogle" link in the Admin navigation to send their newly entered site content to the froogle system. At this point, the store is operational and orders can be placed. Every time an order is placed, t he site operator will be notified via email. They then login to the Admin site. The number of incoming orders awaiting attention is displayed in the menu. The operator clicks on the "Process Incoming Orders" link and process the order by either accepting it (spawning and email to the client) or rejecting (with an option email notice explaining why). Historical records are kept to help the operator manage past orders. A third-party credit-card processing system will be needed, as the system doesn't support automatic transactions.

References:
http://smarty.php.net
http://www.fusebox.org