Web Application Assessment
Bookstore

Victor Godayer
HW ID: H00107316

Contents

List of Figures

Introduction

This report stands for summarizing the purpose of the website developed and its specifications. It also provides documentation for both developer and user.

The main goal of this application is to provide the functionalities of a bookstore through a web interface. The web interface provides the following functionalities: registration of a customer, connection of a this latter, search an item, view the details of an item and add or update items to a shopping cart. The application runs of course with a backend data storing system which is a MySQL database.

An archive containing the project source is available at http://www.macs.hw.ac.uk/~vg55/GODAYER-PHP.tar. The website is accessible at http://www.macs.hw.ac.uk/~vg55/bookstore/

1 Requirements' checklist

The list below enumerates the requirements through the pages composing the web application:

- Home Page ✔

  This page provides an input box to identify the customer thanks to his ID. If he is not a registered customer he can register thanks to the link provided on the page or access the Search Request page.

- Search Request ✔

  This page provides a select input which lists criterions to choose and a text input for typing the request.

- Search Result ✔

  The page returns the results corresponding to the search request. If no results are found a message noticing it is displayed. It contains links to Search Request and Product Detail pages.

- Product Details ✔

  On the page, we can find the details of a book thanks to a GET parameter *id*. The page contains links to Search Request and Shopping Cart pages.

  From this page it is possible to add or update the current viewed viewed to the shopping cart.

- Shopping Cart ✔

  This page contains the items which have been added to the cart. If no items are recorded in the cart yet a message is displayed to warn the user. When an item is in the cart, two actions are available : update the quantity, or add a certain amount of this item.

2 Design considerations

The application follows the MVC pattern on the outlines. The View and the Controller concepts have been correctly respected. Since we need to use a database, the Model concept is much more difficult to follow because of the ORM (Object Relational Mapping) issue, and I could not implement the mapping layer on my own with the slot of time attributed for this assessment.

An MVC implementation implies to use object oriented paradigm. For web application field such a pattern turns out to be like a web framework providing the common features necessary to develop web applications.

The main entities composing this little framework are 3 abstract classes. Each of them gathers the common methods and properties respectively for the controllers, the model and the modules. The latter is not fully related to MVC pattern but the same abstract class feature was needed.

Further more 3 other files has to be taken into account. A file containing the settings about the application has been defined (database, PHP path). A *bootstrap* file gathers all the common operations to do when the application is invoked. Then an *index.php* was required in order to access the website without providing a specific php file to the browser. In this project we make an heavy usage of abstract classes and methods, and inheritance. Another feature used that you will can see is the references.

On the HTML hand, the YAML (Yet Another Multicolumn Layout) CSS framework has been used.

More details about each component of this little framework and the application using it will be given in the Developer section.

3 User guide

Here is a set of screenshots showing the main pages of the application.

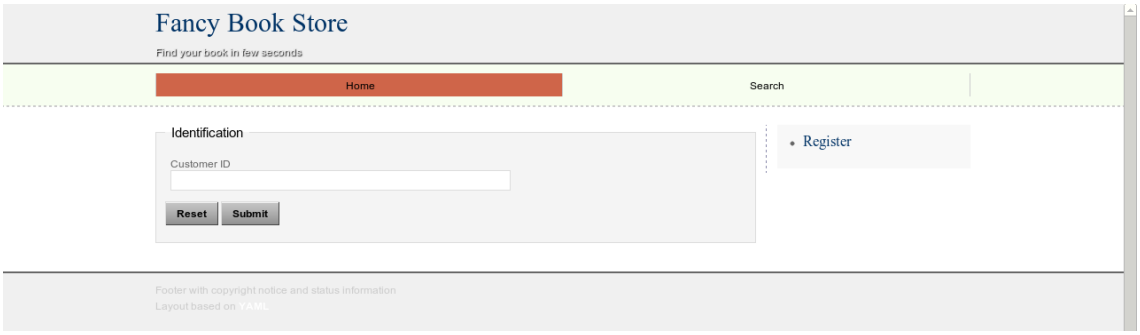

Fig. 1: Home page

The home page presents a form on which the user can login with his ID. It requires of course he has registered before. If he wants to register he can go on the *Register* page through the displayed link. When a user is already logged in, the home page displays the customer details. When the customer logins he is redirected to his customer details.

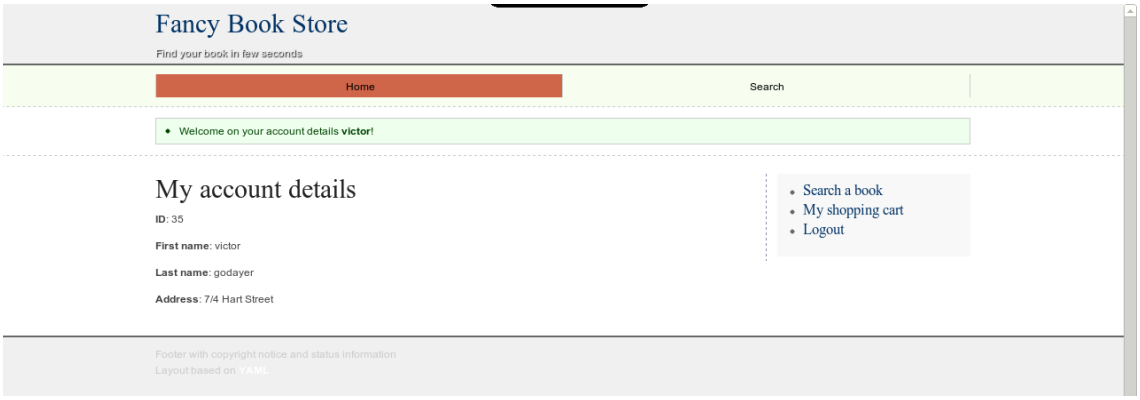

Fig. 2: Customer Details page

The Customer Details page shows the ID, the first and last name and the address of the customer. It provides also links to the search request page, the shopping cart and a logout link.

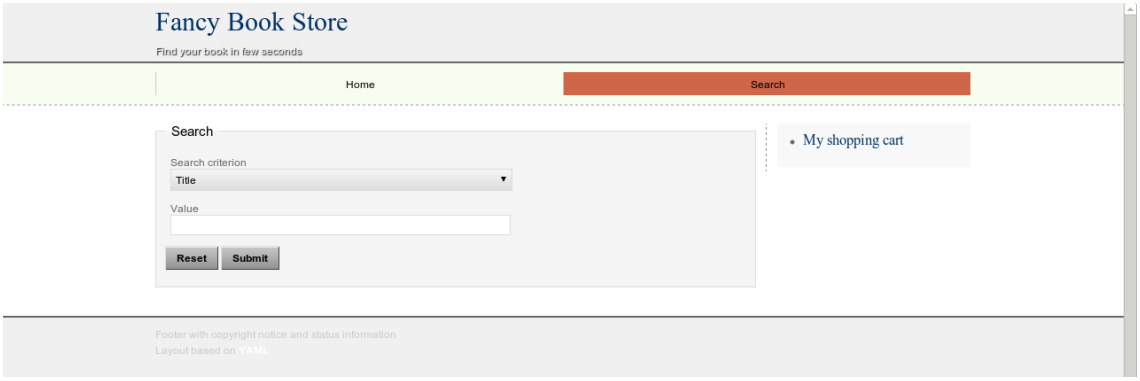

Fig. 3: Search Request page

This is the page for searching a book. A filter by *Title, Author,* or *Subject* is available. All the field are required.

Fig. 4: Search Result page

The Search Result page shows the books matching with the request submitted. The related author linked to the book is also displayed. For each item a Details link leads to the page providing more information about the item selected.

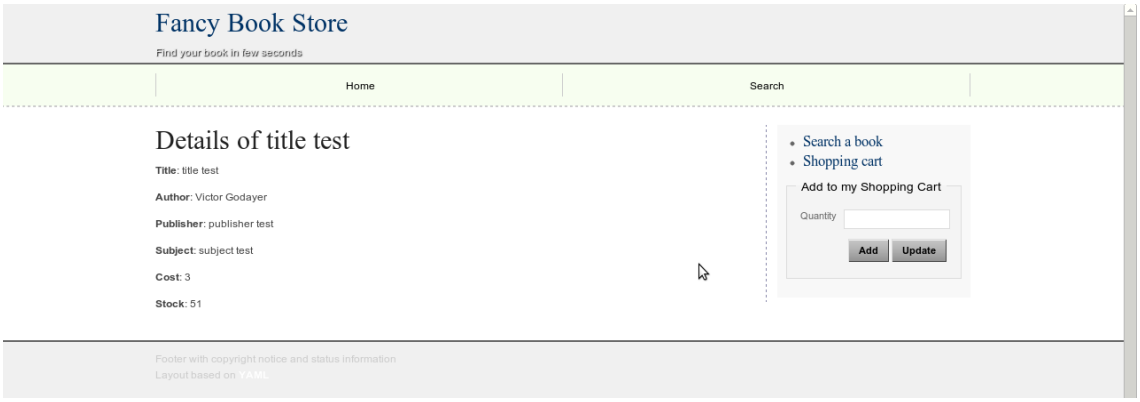

Fig. 5: Product Details page

The page shows the current stock level, the price and all the other information already shown in the Search Result page. From here, it is possible to add the current item viewed to the shopping cart through the form on the right part. Once the quantity submitted the Shopping Cart page is displayed.

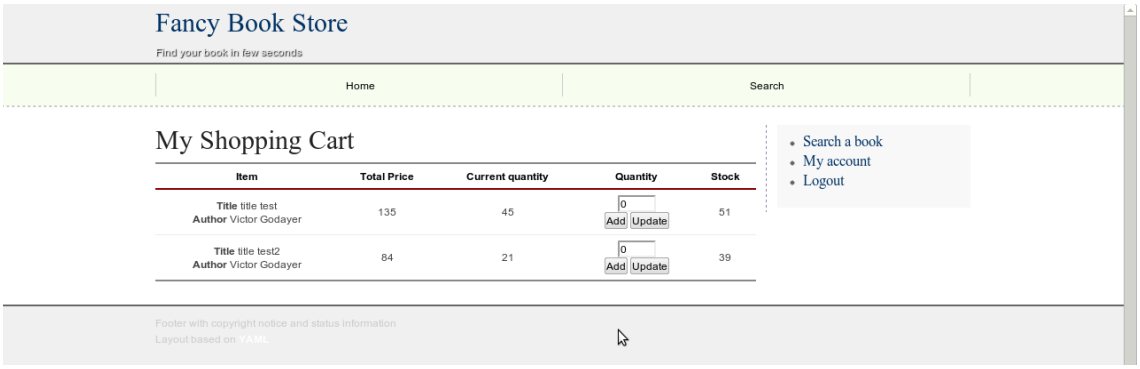

Fig. 6: Shopping Cart page

Here is listed all the items the user has added to his shopping cart. He can add or update the number of items from this page. The submission of the new quantity redirects to the same page with the values updated.

4 Developer guide

4.1 Application settings

In order to parametrize external and internal components the application uses a *settings* file defined in the *conf* directory. It contains the information needed to connect to the database, the root path of the application on the machine it is executed. Moreover the *php include path* is extended with the commonly used such as the ones containing the controller classes, the models classes and the modules classes.

4.2 Bootstrap procedure

The application can be viewed as a sequence of class loadings and execution of hooks defined in these classes. This procedure is always the same except that it depends on the URL the user is browsing. Then a file named *bootstrap*, in the conf directory serves this task: it loads the settings file and the abstract classes (controller, model, and module), declare the modules which has to be loaded by default, set the parameter *c* of the super global variable $_GET if it is not set, and then creates the controller bounded to the value of the *c* parameter. Once the controller instantiated, it just sets the *request* controller attribute according a specific format.

4.3 Overview of the framework core

Below are the three abstract classes composing the hard core of the framwork.

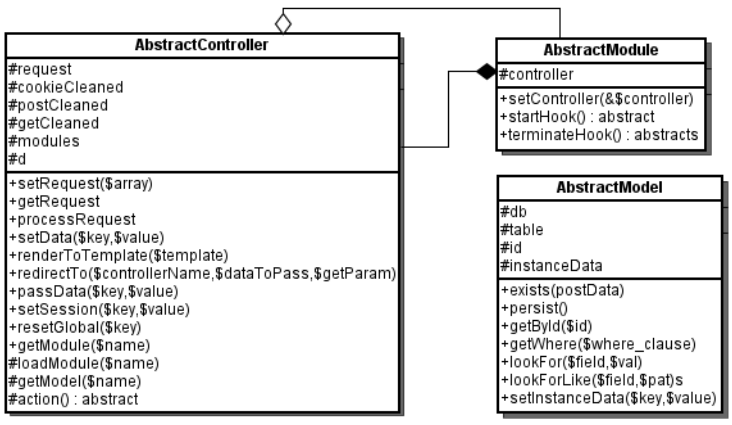

Fig. 7: Global UML of the application

As mentioned in the previous section, all the chain of class loadings and context settings are managed by the *bootstrap* and *settings* files.

The entry point allowing to trigger these procedures is the *index.php* file. It includes the *bootstrap* file which includes the *settings* file. We choose the *index.php* file as entry point because it is a standard on the web and because this is the default value defined in Apache server configuration file.

After all the inclusion are made, the *index.php* file invokes the *processRequest* hook on the controller instantiated in the *bootstrap* file.

Before describing more precisely each of the core abstract classes we needs to peruse the naming standards defined.

4.4 Standards

4.4.1 Naming standards

In order to allow a dynamic instantiation of the appropriated classes such as controller, model, module, or template a naming standard is necessary.

- *URL:*

  Each URL must contain a *c* GET parameter. If not the *IndexController* is instantiated by default. This *c* GET parameter must be the same (case sensitive) as the suffix composing the name of the associated controller. For instance, if the value of this parameter is *CreateAccountForm* then the controller implemented for this URL must be named *CreateAccountFormController.class.php*

- *Controller classes:*

  Each controller class must extends the *AbstractController* class and implements the *action* abstract method. The file and class names must begin with the value of the *c* GET parameter and then be prefixed by *Controller* for the class name and *Controller.class.php* for the file name.

- *Templates:*

If you want a template rendering without being obliged to give the name of the template associated with the URL processed, you have to name your template with the value of the *c* GET parameter followed by the extension *.php*.

- *Models:*

  Each model class must extends the *AbstractModel* class, implements the abstract methods and set the attributes *table, id* accordingly. The concrete class file must be named according this pattern : *A*nyname*Model.class.php*. The first letter must be capital and it has to be prefixed by *Model.class.php*. The class name must be the same without the *class.php* extension.

- *Modules:*

  Each module must extends the *AbstractModule* class and implements the abstract hooks. Each module implementation must repect this : class name : *AnyName,* filename : *AnyName.class.php*

Of course each new extension of one of these abstract classes must be stored in a file saved in the same place as the abstract class it refers to.

4.4.2 Informal standards

Each controller dealing with data submitted through a form has the same name as the controller rendering the form, extended by the prefix *Post* before the *Controller.class.php* prefix. By this way it makes a clear separation between rendering the page which contains the form and the controller dealing only with the data submitted and redirecting to the appropriate page afterward. It is also a way to avoid multiple post data submission by the user when he refreshes a page.

4.5 AbstractController

This class contains the different super global variables in an array attribute named *request*. This choice has been made in order to allow to add some abstraction layer on each of this super global if further development is made on the framework. Here are the description of the main methods of this class:

- *processRequest()*

  This method is invoked from the *index* file has mentioned earlier. It is a wrapper which handle some recurrent jobs on each request received. It executes jobs before the invocation of the *action* hook and after this latter is finished. For instance it loads the declared modules in the *bootstrap* file, loads the data stored in the session variable into the *d* attribute dedicated to be used in the templates. Then it calls the *action* hook which is implemented in the child controller instantiated. If the *action* hook does not redirect to another page, then the *processRequest* method does the jobs following the end of the *action* hook. That is to say it invokes the *terminateHook()* method on each modules loaded, cleans the session data which are not needed anymore.

- *action() : abstract*

  This method has to be implemented in each controller extending the AbstractController. It must contains the business logic of the of the resource (URL) it implements.

- *setData($key,$value)*

  This method set a new entry in the *d* array attribute dedicated for displaying data in the templates. It is particularly useful from within a module implementation.

- *passData($key,$value)*

  This method set data in the session variable which persist until the next browsed page. Then the data will be deleted. This is useful for displaying notifications for instance. The *redirectTo* method makes an heavy usage of it.

- *renderToTemplate($template)*

  This method is just a wrapper for including the template corresponding to the *c* parameter value defined in the super global $_GET, but it can also include a template passed as argument.

- *redirectTo($template,$dataToPass,$getParams)*

  This method redirects from a page to another one. It is used after processing form data for instance. It is possible to pass non-persistent data as an associative array and also an array of *GET* parameters to be set in the redirecting URL.

- *setSession($key,$value)*

  This method allows to set persistent data in the session super global variable. It is useful for keeping information about a logged in user for instance.

The *modules* attribute is an array containing the loaded module. The *-cleaned attributes are arrays containing data ready to be used in a SQL statement. They are generated after an invocation of the *cleanData* method. As noticed above the *d* attribute is the variable dedicated to be used in the templates to displayed dynamic data.

4.6 AbstractModel and Models

The definition of the AbstractModel class is a little bit fuzzy. It mixes the model and the mapping concepts, the latter being nearly non-existent. Then it is quite confusing to use it as an external user.

The abstract class contains the *table* attribute which is the name of the table of the model implementing the abstract class. The *id* attribute has the same purpose but with the name of the id field in the table. The *db* attribute is only the database module which is an interface to operate with the database for simple queries.

- *exists($postData)*

  This method checks if an entry already exists in the database according post data passed in argument and the table name defined in the *table* attribute. This method makes use of the implementation of the *uniquenessClause* method.

- *persist()*

  Save in the database the data held in the *instanceData* attribute if the entry does not already exist. Otherwise it updates the entry corresponding. Make use of the *exists* method.

- *setInstanceData($key,$value)*

  Allows to modify a value in the *instanceData* associate array attribute holding the data for a retrieved database entry.

- *uniquenessClause() : abstract*

  This abstract method is implemented in the concrete models classes. It returns a SQL *where* clause which determines the uniqueness of an instance of the model.

  For instance the uniqueness clause for the model user is : *"WHERE C_FNAME=fname AND C_LNAME=lname"*. It means that a customer with his first and last name can register only once, whatever the address is.

The other methods are simple interfaces to get data with the WHERE or LIKE statement in an easier way.

4.7 AbstractModule and Modules

- *setController(&$controller)*

  This method is a setter for the *controller* attribute of the abstract class. The attribute is protected in order to be accessed by the module extending the abstract class. Also, it is a referenced variable. It means modification operated on it are referenced to the original variable which is the current instantiated controller.

- startHook() & terminateHook()

  Each of them are implemented by the module extending the abstract class.

4.8 Data structure choices

First of all the class concept is surely the feature the most used in this project, because it allows to separate the different concepts composing the web interactivity. The class inheritance is heavily used in order to relieves the coder from many recurrent tasks.

Also the choice of arrays as main data structure is led by its simplicity of use and the plurality of built-in functions coming with it. Especially, the associative arrays allowing to give a string key for an element is very handy. It allows to access easily any of its items by specifying the key corresponding.

4.9 Business logic

In this section we are going to deal with the controllers which implements the requirements. It will be a brief overview because it is just the translation of the requirements into PHP. Most of controllers make use of modules. The choice to implement a part of the requirements in modules is made because it externalizes some code which is recurrent in many controllers. Then the code written can be used directly from different controllers.

- *IndexController:*

  It checks whether the current user is logged in or not. If he is, then it renders the *CustomerDetails* template, otherwise the login page which is *Index* template.

- *CreateAccountFormController & CreateAccountPostController:*

  The first of them returns the form to create an account if the user is not already logged in. The second processes the post data, checks if the user does not already exists in the database and creates a new entry accordingly.

- *CustomerDetailsController:*

  This controller simply display the data of the user currently logged in, such as his id, address, first name and last name.

- *SearchFormController & SearchPostController:*

  The first of them renders the form allowing to search a book according a selected criterion and a value specified. It uses the *LIKE* SQL feature in the SQL request. The result (which can be empty of course) is displayed in the *SearchResult* template with links to the details pages of each product listed.

- *ProductDetailsController & ProductDetailsPostController:*

  The first one renders the template *ProductDetails* according an *id* GET parameter value. It displays all the information related to the item, even the "foreign data" such as the author first and last name. It also displays a form allowing to add a certain quantity or update the amount of the item viewed to the shopping cart.

  On submitting this form, the second controller handles the post data and update the shopping cart accordingly.

  We will notice that the shopping cart concept has been externalized in the *SessionCart* module.

- *ShoppingCartController & ShoppingCartPostController:*

  The first one displays the items stored in the shopping cart if there is any. When items exist in the shopping cart it is possible to update the quantity or add a certain amount to the current number stored in it.

After this overview of the controllers implementing the solution, let us have a look to the template system and the CSS.

4.10 Templates, HTML integration and CSS

4.10.1 Templates

As we mentioned earlier a dedicated php variable is used to display the dynamic data. This is the *d* variable contained as a protected attribute in the *AbstractController*. The variable name choice is related to is little size and it is the first letter of the word *data*. On a type view, this variable is an associative array. In the template we can access this variable as usual with php variable: *$d*. In the controller if you want to add some data in this variable you can access it by *$this->d*. We can see that it is not the same way to access it than in the template. This is because before calling any template a copy of the array *$this->d* is made into the variable *$d*. This is only to make the HTML writing phase easier and avoid to type each time $this->d or even do the copy manually in each template.

4.10.2 HTML integration

Each template which is stored in the *views* directory is made of includes part such as the meta, the top navigation, the navigation and the footer parts. These several "parts" are gathered in a sub-directory named *includes*. Then the only HTML code to write has to be done between three HTML tags that you must add manually:

```
<div id="main">
  <div class="page_margins">
    <div class="page">
        Your HTML here.
    </div>
  </div>
</div>
```

The reason why you need to write your HTML code within these 3 tags is justified by the usage of a YAML CSS framework.

4.10.3 CSS

The CSS is a specific thing that it can be done properly only by expert, knowing the issues with the different HTML web browser engines. That is why I decided to use a CSS framework discovered recently which is the YAML framework. It gives several CSS classes allowing to build multiple columns layouts for a website within a short integration time. For any documentation and advices to use it properly check out this website: http://www.yaml.de/en/documentation.html.

Every static files such as CSS, javascript or images files, are supposed to be stored in the sub directories *css, js, images,* of the directory *media.*

5 Testing

All the tests are gathered in a *tests.php* file. Tests performed in this file are related only on a concrete class of *AbstractController* (*TestController*). No tests has been made on the model part.

Function	Result
setRequest / getRequest	✔
getModule	✔
initializeData	⚠(1)
setData	✔
passData	✔
setSession	✔

(1) Thisi error was due to the fact that the variable *d* was not initialized in the constructor of the *AbstractController*. The correction has been made accordingly.

Conclusion

First of all I'm proud to be able to implement a basic web framework even if it is not a complete framework because of a lack of generic implementation on the abstract model side. I have been dealing with web frameworks for a long time now, that is why I felt like to try to implement my own framework for "playing around". I mainly took inspiration and concepts from the Django framework and the Symfony2 framework.

We can see that the powerful of the object oriented programming is strongly used to centralize repeated code and process, to specify compulsory specifications but unknown till the implementation of it (*hooks* thanks to abstract methods, initialization of parent class attributes in child classes). Also, associative arrays are the most used features since PHP does not have many other built in types.

Nonetheless we can see that the model side is not well implemented, it should be done once again from the scratch for this part. Also, we could implement the possibility to configure the different URLs available for the web application in a separate file, and code a layer allowing to access these URLs through a friendly name defined in the URL file. By this way we could increase the decoupling between template, controller and url, and also increase the facility to call URLs from within templates and controllers.

References

[1] http://www.php.net/

[2] http://www.yaml.de/en/documentation.html