

PROJECT REPORT

CSCI – 5308 Quality Assurance



**DALHOUSIE
UNIVERSITY**

Dashboard - Library Management System

Submitted by

Anitt Rajendran

B00783632

Mary Ann Joji

B00783053

Yuvaraj Subramanian

B00754715

***Submitted on
AUG 03, 2018***

***Faculty of Computer Science
Halifax, Nova Scotia***

Deployment and configuration

Application URL - <http://dashboardlibrary.azurewebsites.net/>

Deployment was done in Azure cloud platform through visual studio IDE, the application was published and hosted in Azure Platform. As part of continuous integration, the deployment was setup through Jenkins. Through Jenkins we were able to do deployment of the application in a such way that each check in of and update to project would get directly deployed and application would be up and running. Furthermore, the complete configuration of the project is given below.

Business Configuration logic was done by doing setting in *web.config* file, the main logic of the app was configured in such way that the application adapts to the logic

Configurations

For the project we have used the below tools and configurations:

- Front-end: HTML, CSS, JavaScript
- Backend: ASP.NET MVC (C#)
- Database: MySQL
- Frameworks: Bootstrap
- Continuous integration: Jenkins
- Version Control: GitHub
- Cloud: Azure
- Project management: Trello

For the project development, all the team members discussed and agreed to use Microsoft visual studio community edition.

Hardcodes: Mostly 60 percent of the code is removed from hardcodes, there still some constants like log messages, some parameter bindings which are not removed because of time constraints.

Design patterns

Abstract Factory

Abstract factory design pattern was used for our backend validation logic of all the classes. Abstract factory pattern was mainly implemented to separate the details of the implementation of group of objects and rely on object creation through the methods exposed in the factory interface. Furthermore, it's not recommended to create platform objects directly rather the best possible approach is to use abstract factory pattern where a super factory or factory generator creates a group of factories and these factories methods depends on interface for creating factory related objects without explicitly specifying the class name. Additionally, it's these factories are responsible for creating objects for the individual classes.

Chain of Responsibility

This design pattern was used to implement password validation when a user registers to the system. Password validation involves validating different properties like the use of uppercase, lowercase, special character, digit and meet the required password length. Chain of responsibility pattern was apt to implement this as each of these properties could be set as different receiver objects in a chain having the responsibility of validating their respective property, and if the validation succeeds, pass it to the next object in the chain. If a property is violated in the chain, the chain breaks and returns a failure message to the user.

Singleton

The Singleton pattern was used in the 'MySQL Database' class which manages all the connections to the database. Use of this pattern ensured that the global access was provided to the object of the database class and one single instance managed and coordinated all the SQL connections across the application. All the responsibilities of the database connection layer were encapsulated in the single class.

Data/business logic/user interface

The project is implemented using the MVC framework – ASP.NET MVC 5. Use of MVC helped to achieve the desired separation of model, view and controller classes. The Below diagram can be used for explaining the MVC framework we implemented in the project.

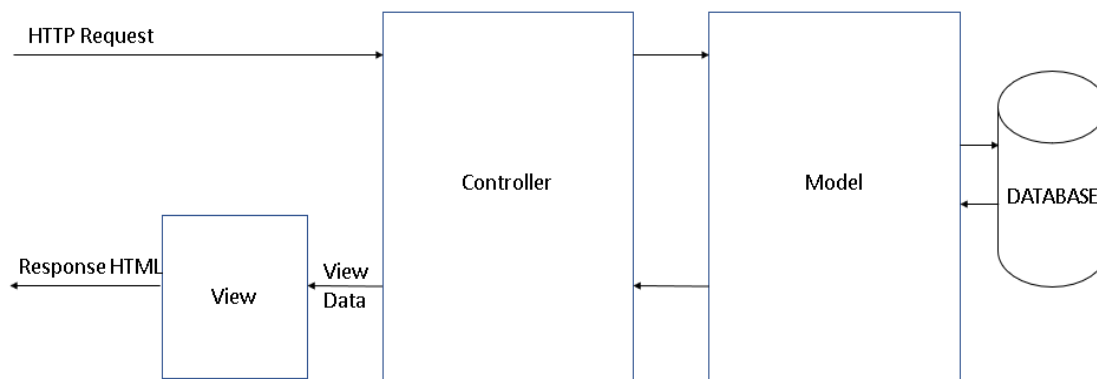


Figure 1. Application's MVC Architecture

HTTP request first goes to controller which contains the application logic and it interacts with the model and view to serve the request. Controller is responsible for storing the logic needed to return the view to the user. The model part of the architecture contains the business logic, validation logic and data. The main functionality of model is to retrieve and store data from the database. The View layer contains the HTML and content which is used as the front-end for the application. It is responsible for displaying the data in GUI to the user. As demonstrated in the diagram the controller interacts with model for getting the data and interacts with View for rendering the application in the browser.

Naming Strategy and coding conventions

Naming Strategy

The naming conventions that we followed while developing the application is as given below.

Main project: Dashboard-Library Management

- **Class name:** We followed *Pascal Case* naming convention for the class names i.e. the first letter of each word in a class name is capitalized.
 - E.g. BookTable, BookController
- **Methods:** We followed *Pascal Case* naming convention for methods like the class names naming convention.
 - E.g. AddBook, RenewBook
- **Variable names and Class instances:** We followed camel case for the variable name and class instances.
 - E.g. bookId, currentTransaction

According to the common naming convention, if the name contains multiple words, the first letter of each word would be capitalized to make the words distinct. We made sure that there was difference between naming classes/methods and other variables/class instances.

- *Interface:* All interface names are prefixed with 'I'.
 - E.g. IDatabase, IPassword

Test project: Dashboard-LibraryManagement-UnitTest

The structure of the test project is similar to the main project. For each class in the main project, we have written a test class to test the functions within the main project class.

- *Class name:* The classes in the test project have the same name of its respective class in the main project followed by the word 'Test'.
 - E.g. BookTableTest, BookControllerTest

Coding convention

Test driven development

We tried our maximum to follow the test-driven development to produce quality code and we have more than 80 percent of code coverage of unit test.

Peer code review

As a team, we reviewed each other's code before pushing it to the repository. To ensure that the requirements, coding standards were properly followed.

Code readability

We have added appropriate comments for the code to make the code readable. The naming convention we followed also assist to the code readability.

Refactoring

Introduce Parameter object:

This refactoring technique was used to pass a group of parameters to the functions. It has been used in many controller functions and other functions that perform some core business logic functions. This refactoring was mainly used when function logic manipulated the

data in the parameters of a model class. In this case, an object of the model class would be passed as parameter to the functions.

E.g. The functions performing CRUD operations accept the objects of model class to execute the functions.

For example, in all the Validation logic of the application we have passed the object of the class instead of passing the longlist parameters.

```
public class BookValidation : IBookValidation
{
    8 references | Anitt Rajendran, 4 days ago | 2 authors, 4 changes
    public Boolean ValidateBook(Book newbook) {

        try
        {
            BookTable bookTable = new BookTable(new MySQLDatabase());
            IEnumerable<Book> ListofBooks = bookTable.GetBooks();
            // validate and check whether the Book is already existing
            foreach (var book in ListofBooks) {

                if (newbook.Name == book.Name && newbook.Author == book.Author) {

                    return false;
                }
            }
        }
    }
}
```

Figure 2. Introduce parameter object for ValidateBook method

Extract Class:

The core logic of the application consists on the various functions of users, library members and books. Instead of putting the entire functionalities in three different classes, we have further divided the responsibilities of the classes. Initially, we started with three main classes for Book, User and Library member. As the project evolved, the class responsibilities were further divided.

Example: Book Module

BookController - Application logic

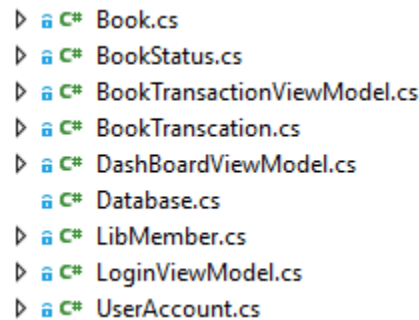
BookTable - CRUD operations (Insert, Delete, Update)

BookTransactionTable - Functions related to issue and renew of books

Similarly, the responsibilities of Users (Librarians) and library members have been written in separate classes.

Model classes:

Another instance where we used extract classes were on the model class and their properties, there was one model book class which had properties of book insertion as well as book transaction, both the classes were separated and put into separate classes and the relevant fields were moved from old class to new classes.












```
▸  C# Book.cs
▸  C# BookStatus.cs
▸  C# BookTransactionViewModel.cs
▸  C# BookTranscation.cs
▸  C# DashBoardViewModel.cs
   C# Database.cs
▸  C# LibMember.cs
▸  C# LoginViewModel.cs
▸  C# UserAccount.cs
```

Figure 3. Model classes in code

In the above image, all classes related to books were extracted and made into new classes giving properties which followed single responsibility. Furthermore, Method names were refactored to provide meaning information.

Extract Interface:

This technique has been used for Password Validation logic.

Rename method:

We have refactored the code to ensure that all method names in the code is self-explanatory and reveal its purpose with the name.

Technical debts

Some hardcodes like log messages and parameter bindings are not removed because of time constraint.

More code coverage could be attained.

Member contributions

Every Member in the team contributed equally for the success of the project, all the functionality information and high-level folder structure and coding in model, view, controller, validation classes, exception handling, design pattern are mentioned.

Team Member	Contributions
<p style="text-align: center;">Anitt Rajendran</p>	<p>1.Application functionality - Login, List Admin, Manage Admin, Add Admin, Issue Book, RenewBook(partial)</p> <p>2.Exception handling and error logging were equally contributed by all team members</p> <p>3.DataAccessLayer – MySQL Database connection and User Table classes.</p> <p>4.Model – Book Model,BookStatus, BookTransationModel.</p> <p>5.Design pattern - Chain of Responsibility – Was Equally shared by all the team members , the team discussed and implemented design pattern together.</p> <p>6.Abstract Factory Design pattern - Was Equally shared by all the team members, the team discussed and implemented design pattern together</p> <p>7.Configurable Logic – Renew Book and Password Validation logic was designed and implemented by all members in team, equal contribution.</p> <p>8. All the backend logic was equally shared between the team members.</p> <p>9. Stored Procedures – Stored procedures were split according to controller functions and written and put into repository. However due to time constraints, Stored procedures were not replaced in all functions.</p> <p>10.Deployment and Cloud operations was taken care.</p> <p>11.Continuous integration on Jenkins equal contribution.</p> <p>12.TestCases –Equal contribution of test cases from all the team members. 84 percent of code coverage was achieved, and all test cases passed.</p> <p>13.Refactoring was done at few areas.</p> <p>14.Front design and development is equally shared, and ideas were exchanged.</p>
<p style="text-align: center;">Mary Ann Joji</p>	<p>1.Application Functionality – Registration, add member, List Members, Edit Member, Delete Member, Dashboard Logic.</p> <p>2.Exception handling and error logging were equally contributed by all team members</p> <p>3.DataAccessLayer – Member Table, Book Transcationtable.</p> <p>4.Model – User account model, Lib Member model.</p> <p>5.Design Pattern - Chain of Responsibility – Was Equally shared by all the team members, the team discussed and implemented design pattern together.</p> <p>6.Abstract Factory Design pattern - Was Equally shared by all the team members, the team discussed and implemented design pattern together</p> <p>7.Configurable Logic – Renew Book and Password Validation logic was designed and implemented by all members in team, equal contribution.</p> <p>8. All the backend logic was equally shared between the team members.</p> <p>9. Stored Procedures – Stored procedures were split according to controller functions and written and put into repository. However due to time constraints, Stored procedures were not replaced in all functions.</p> <p>10.Continuous integration on Jenkins equal contribution.</p> <p>11. Testcases –Equal contribution of test cases from all the team members. 84 percent of code coverage was achieved, and all test cases passed.</p>

	<p>12.Constants classes.</p> <p>13.Refactoring was done at few areas.</p> <p>14.Front design and development is equally shared, and ideas were exchanged.</p>
<p>Yuvaraj Subramanian</p>	<p>Application Functionality – AddBook, Edit Book, List Book, Delete Book, List Book, RenewBook, Dashboard View.</p> <p>2.Exception handling and error logging were equally contributed by all team members</p> <p>3.DataAccessLayer – BookTable, Dashboard Table classes.</p> <p>4.Model –, BookTransactionViewModel, DashboardViewModel.</p> <p>5.Design Pattern - Chain of Responsibility – Was Equally shared by all the team members, the team discussed and implemented design pattern together.</p> <p>6.Abstract Factory Design pattern - Was Equally shared by all the team members, the team discussed and implemented design pattern together</p> <p>7.Configurable Logic – Renew Book and Password Validation logic was designed and implemented by all members in team, equal contribution.</p> <p>8. All the backend logic was equally shared between the team members.</p> <p>9. Stored Procedures – Stored procedures were split according to controller functions and written and put into repository. However due to time constraints, Stored procedures were not replaced in all functions</p> <p>10. Continuous integration on Jenkins equal contribution.</p> <p>11.TestCases –Equal contribution of test cases from all the team members. 84 percent of code coverage was achieved, and all test cases passed</p> <p>12.Refactoring was done at few areas.</p> <p>13.Front design and development is equally shared and ideas were exchanged.</p>

Presentation Feedback

As part of the demo, we got feedback from Rob saying that we were half way through the business logic as deleting the business logic rule didn't work for us with the validation rule. We have fixed the issue in such way that deleting the rule from the configuration would result in removing the validation logic itself thus the registration of user happens without any password restrictions. When again the business logic is added the application follows the password restrictions.

During our demo presentation to the TA, we noticed a minor bug with the user permission. Later, we have worked on the bug and fixed it.