2

Student: Czako Zoltan

**Group: 30235**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 6

4. UML Sequence Diagrams 10

5. Class Design 13

6. Data Model 15

7. System Testing 16

8. Bibliography 17

1. Requirements Analysis

# Assignment Specification

In this assignment we implemented a web application. The subject of this application is a bookstore app, in which helps the employees in their daily work, helps them to manipulate the book’s data, to make sells of books and so on. Besides helping the employees, it also helps the administrator of the bank, who can hire new employees, he can modify employee’s data and he can get reports about the actions of the workers.

# Functional Requirements

Our application will have two types of users. The first type is the Employee. An employee has access to the data of the books in the store. He can insert new book, he can modify the information of an existing book and if he must, he can delete a book. An employee can see all the data of a book, he can view the title of a book, the genre, the author, the price, number of pages, the price and the quantity on stock.

The second type of user is the Administrator. This type of user has the ability to view the list of the bookstore’s employees, he can select an employee and view his personal information like username, first name, last name, mobile and mail. He can modify the user data of an employee and he can also delete an employee if needed. The admin can generate a report of the books out of stock.

# Non-functional Requirements

In order to obtain all this functionality we created an ASP.NET MVC 4 Web Application. Talking about accessibility, to differentiate the two type of user we created a login page. After login there are two different views, one for the admin and one for the client. We used list to show all the users and all the books in an ordered way. For creating and updating employees/books data we created another two views with text boxes in which we can introduce or modify data.

To obtain availability we created a GitHub repository and after each modification that we made or new features that we created, we committed out changes to this repository and in this way we prevented data loss, wrong modifications or accidental code deletion. This repository helps us in backup, because we can backup our system striating with a well specified date(last time when our system was stabile).

Talking about dependency, we have an MVC Architecture, we have three separate project one for the XML Database, one for the Services and one for the MVC Project. Besides these functional layers we have another project for testing our system.

We work really much on failure management and we tried to cover all the situations when a user can enter wrong data, or tries to make some illegal operation.

Talking about portability, our application is platform dependent, it work only on Windows operation systems because it is a Windows native application.

In the view of usability, this software is intended to be used in bookstores, it can help in book data management and for create reports of the out of stock books.

To grant a well working and bugs free application, to keep business needs well satisfied we created Unit Tests for the Services, in which we have data validations or conditional statements. We tried to have a 100% of covering, we test corner cases as well as happy flow situations.

2. Use-Case Model

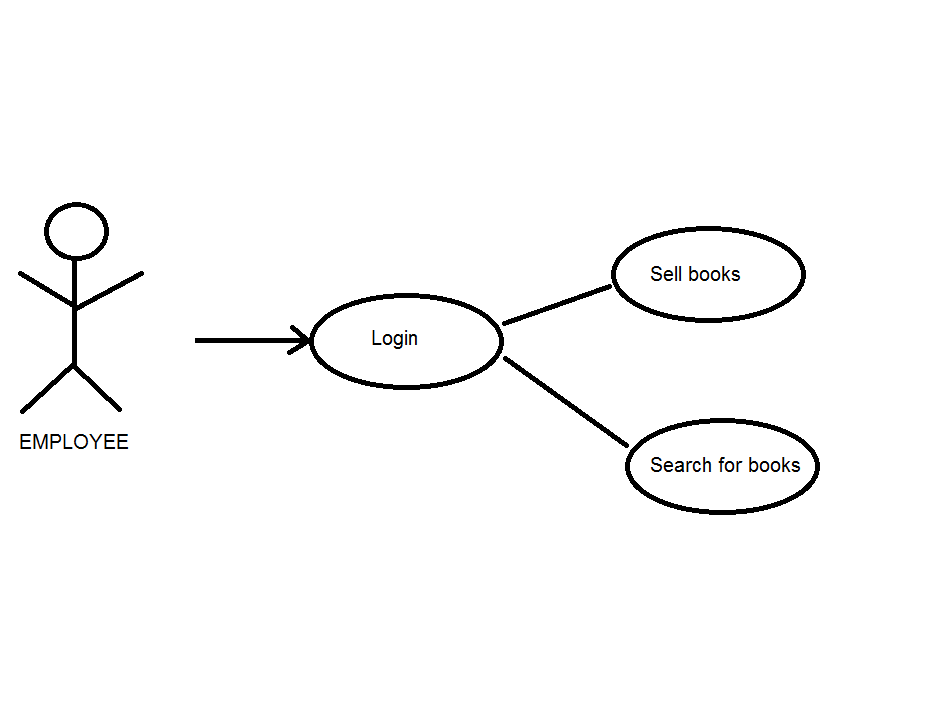
Use Case: Describe the possible actions of the employees

Level: user-goal level

Primary actor: Employee

Main success scenario:

1. Login
2. Sell books
3. Search for books by genre, title, description, author.



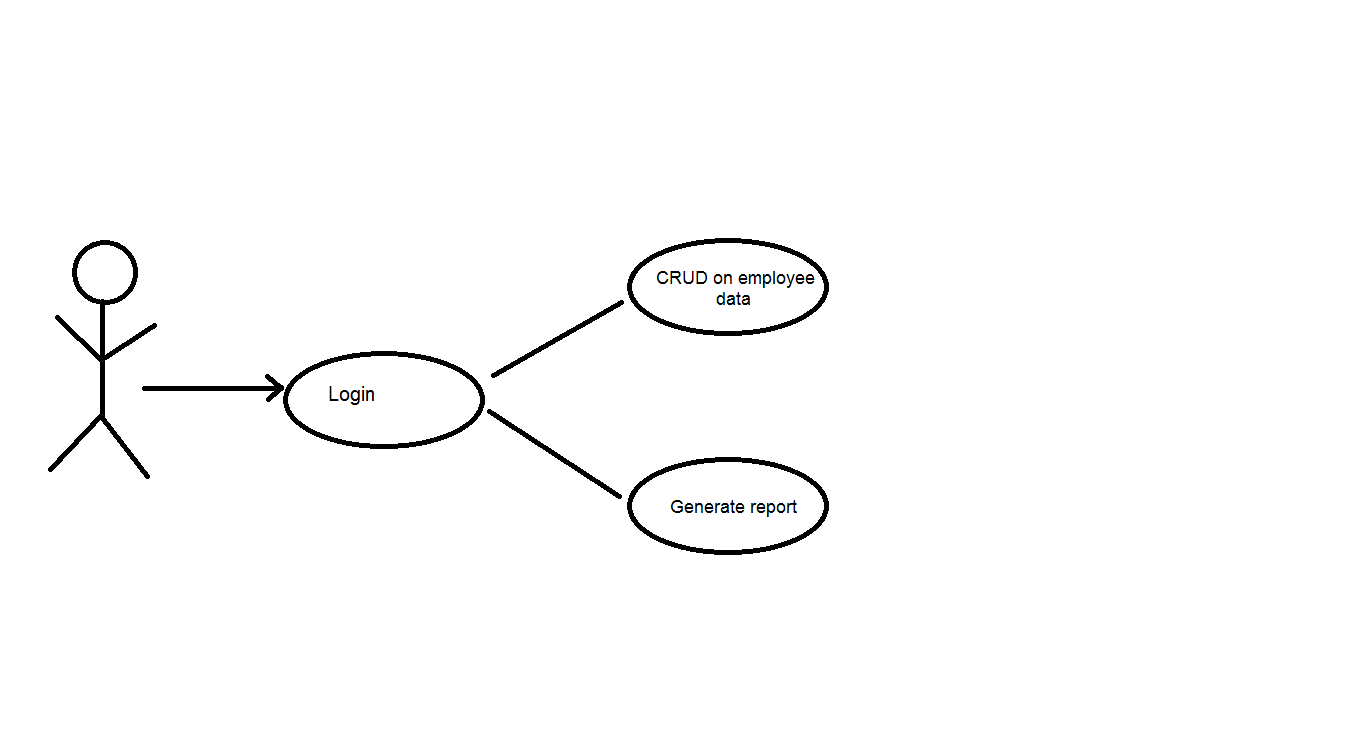
Use Case: Describe the possible actions of the aministrator

Level: user-goal level

Primary actor: Administrator

Main success scenario:

1. Login
2. CRUD on Employees
3. Report on books out of stock in .xml and .txt format



Administrator

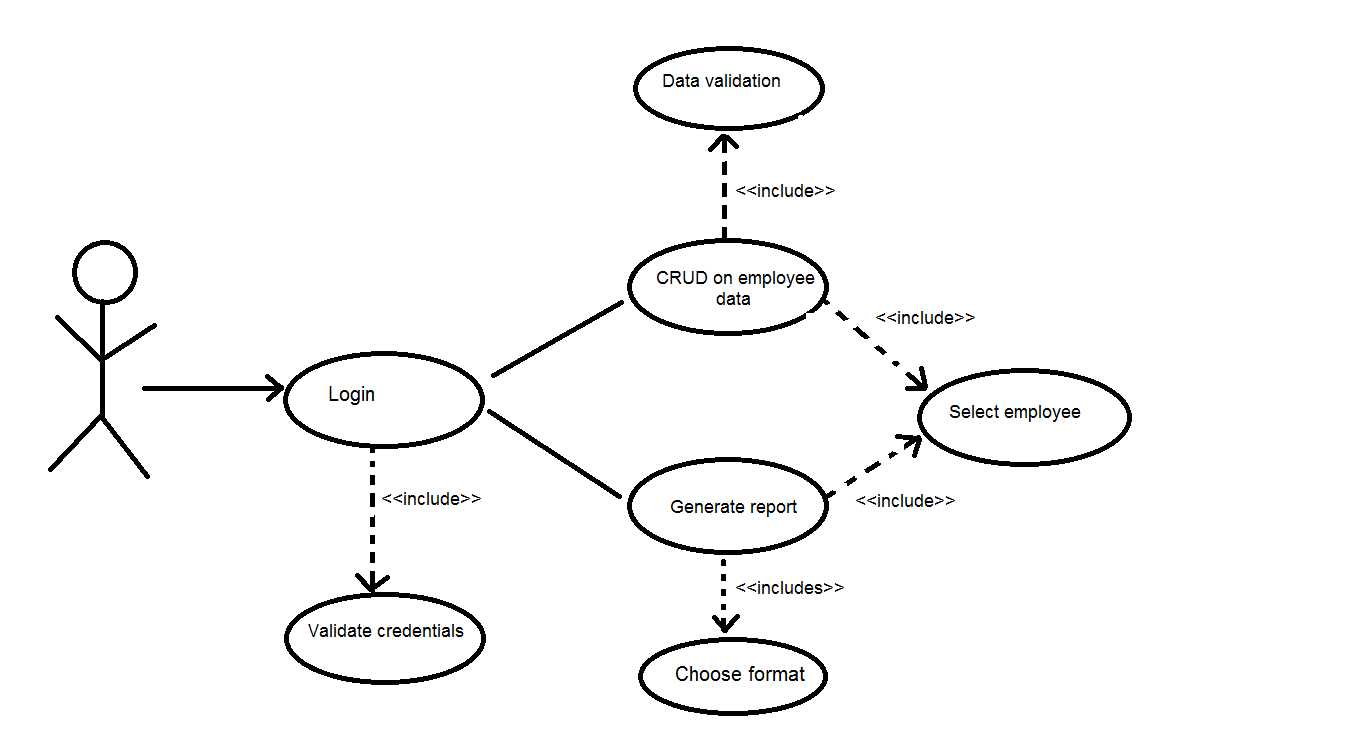
Use Case: Describe the possible actions of the aministrator

Level: sub-function level

Primary actor: Administrator

Main success scenario:

1. Login 🡪 Validate credentials
2. CRUD on Employees 🡪 Select employee 🡪 Validate data
3. Report on books with a specific format (.txt or .xml) 🡪 Select format



Administrator

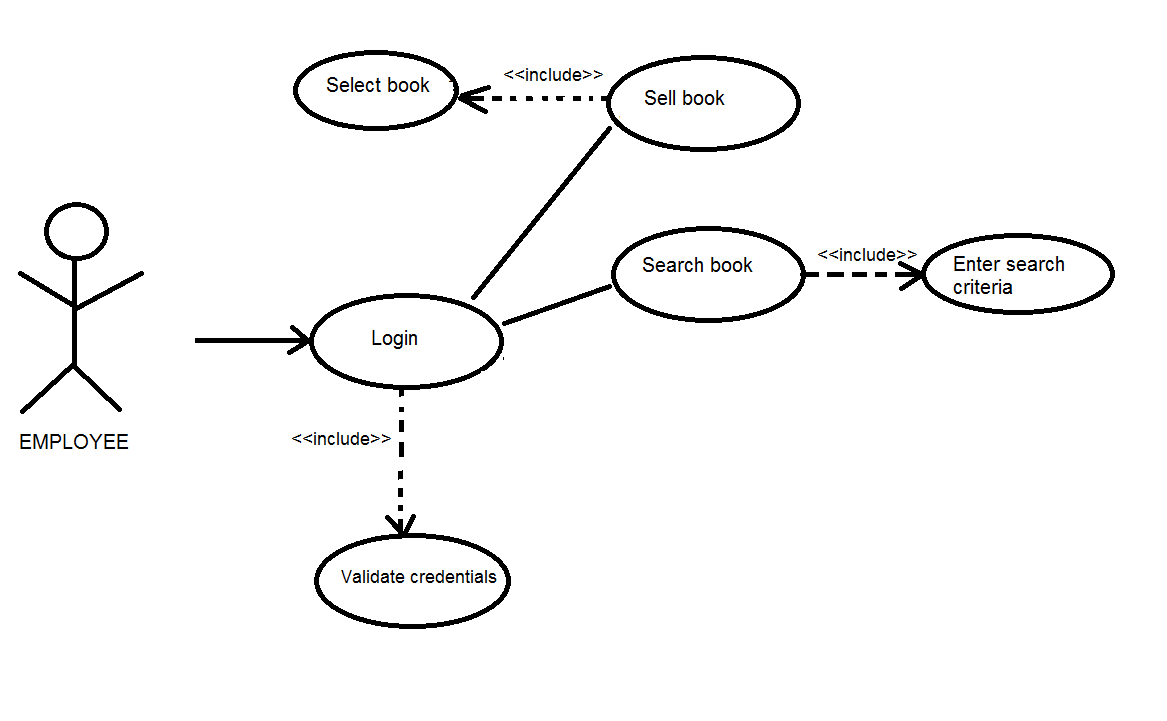
Use Case: Describe the possible actions of the employees

Level: sub-function level

Primary actor: Employee

Main success scenario:

1. Login 🡪 Validate credentials
2. Sell on book 🡪 Select book
3. Search for books 🡪 Enter criteria



3. System Architectural Design

**3.1 Architectural Pattern Description**

The Model-View-Controller (MVC) architectural pattern separates an application into three main components: the model, the view, and the controller. The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms pattern for creating Web applications. The ASP.NET MVC framework is a lightweight, highly testable presentation framework that (as with Web Forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication. The MVC framework is defined in the **System.Web.Mvc** assembly.

MVC is a standard design pattern that many developers are familiar with. Some types of Web applications will benefit from the MVC framework. Others will continue to use the traditional ASP.NET application pattern that is based on Web Forms and postbacks. Other types of Web applications will combine the two approaches; neither approach excludes the other.

The MVC framework includes the following components:

* Models. Model objects are the parts of the application that implement the logic for the application's data domain. Often, model objects retrieve and store model state in a database. For example, a Product object might retrieve information from a database, operate on it, and then write updated information back to a Products table in a SQL Server database.

In small applications, the model is often a conceptual separation instead of a physical one. For example, if the application only reads a dataset and sends it to the view, the application does not have a physical model layer and associated classes. In that case, the dataset takes on the role of a model object.

* Views. Views are the components that display the application's user interface (UI). Typically, this UI is created from the model data. An example would be an edit view of a Products table that displays text boxes, drop-down lists, and check boxes based on the current state of a Productobject.
* Controllers. Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render that displays UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles query-string values, and passes these values to the model, which in turn might use these values to query the database.

The MVC pattern helps you create applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the application. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an application, because it enables you to focus on one aspect of the implementation at a time. For example, you can focus on the view without depending on the business logic.

The loose coupling between the three main components of an MVC application also promotes parallel development. For example, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

## [Support for Test-Driven Development](javascript:void(0))

In addition to managing complexity, the MVC pattern makes it easier to test applications than it is to test a Web Forms-based ASP.NET Web application. For example, in a Web Forms-based ASP.NET Web application, a single class is used both to display output and to respond to user input. Writing automated tests for Web Forms-based ASP.NET applications can be complex, because to test an individual page, you must instantiate the page class, all its child controls, and additional dependent classes in the application. Because so many classes are instantiated to run the page, it can be hard to write tests that focus exclusively on individual parts of the application. Tests for Web Forms-based ASP.NET applications can therefore be more difficult to implement than tests in an MVC application. Moreover, tests in a Web Forms-based ASP.NET application require a Web server. The MVC framework decouples the components and makes heavy use of interfaces, which makes it possible to test individual components in isolation from the rest of the framework.

### **Advantages of an MVC-Based Web Application**

The ASP.NET MVC framework offers the following advantages:

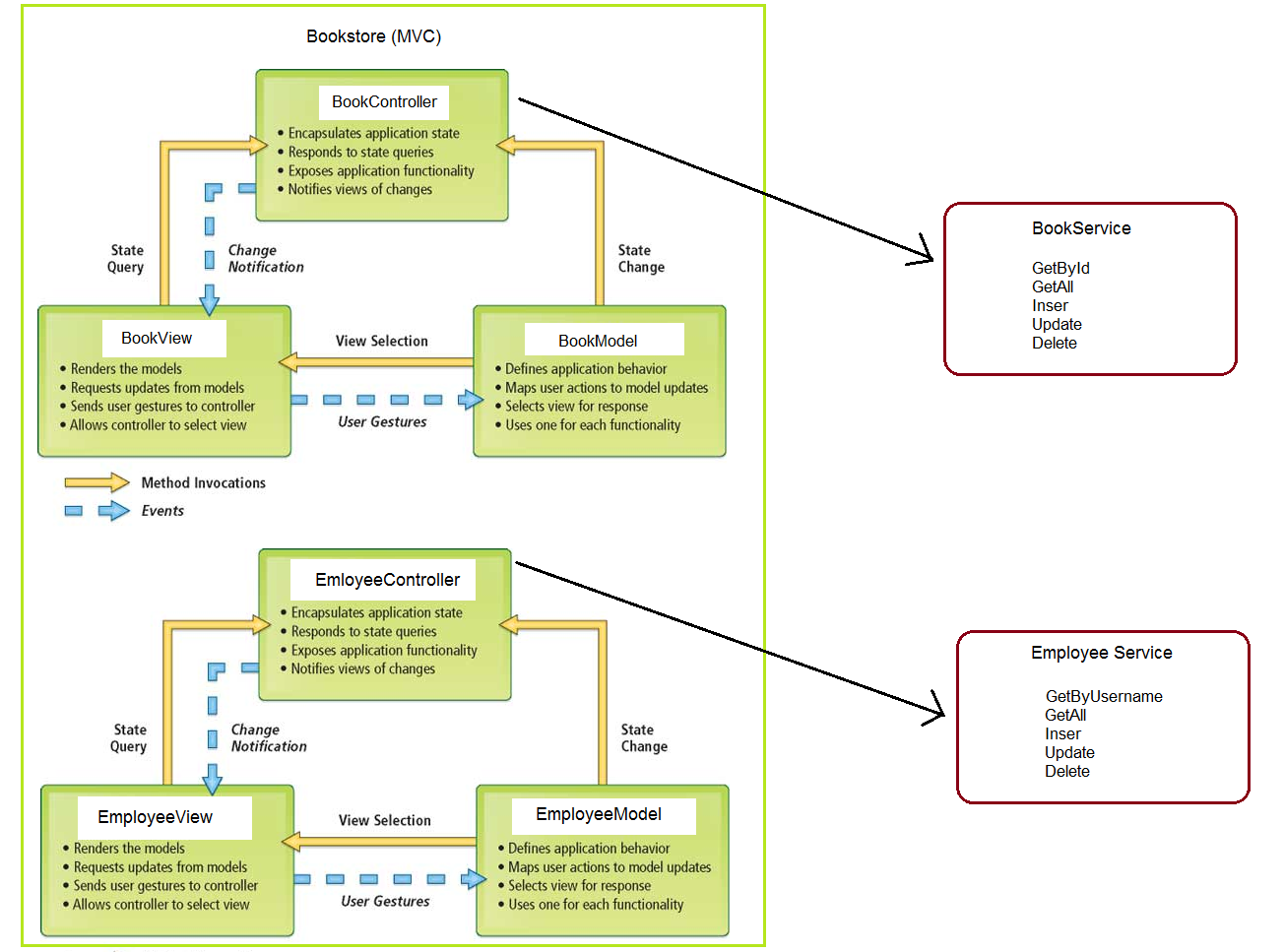
* It makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
* It does not use view state or server-based forms. This makes the MVC framework ideal for developers who want full control over the behavior of an application.
* It uses a Front Controller pattern that processes Web application requests through a single controller. This enables you to design an application that supports a rich routing infrastructure. It provides better support for test-driven development (TDD).
* It works well for Web applications that are supported by large teams of developers and for Web designers who need a high degree of control over the application behavior.

### **Advantages of a Web Forms-Based Web Application**

The Web Forms-based framework offers the following advantages:

* It supports an event model that preserves state over HTTP, which benefits line-of-business Web application development. The Web Forms-based application provides dozens of events that are supported in hundreds of server controls.
* It uses a Page Controller pattern that adds functionality to individual pages. It uses view state on server-based forms, which can make managing state information easier.
* It works well for small teams of Web developers and designers who want to take advantage of the large number of components available for rapid application development.
* In general, it is less complex for application development, because the are tightly integrated and usually require less code than the MVC model.

**3.2 Diagrams**



**Package Diagram**

Bookstore

(MVC)

<<references>>

Services

BankTest

<<references>>

<<references>>

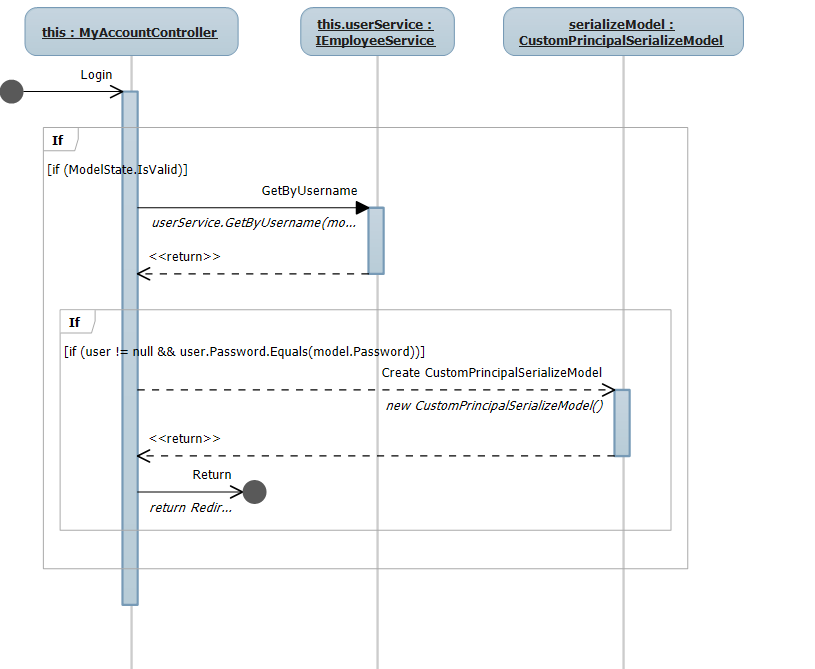
Xml Database Accessor

<<references>>

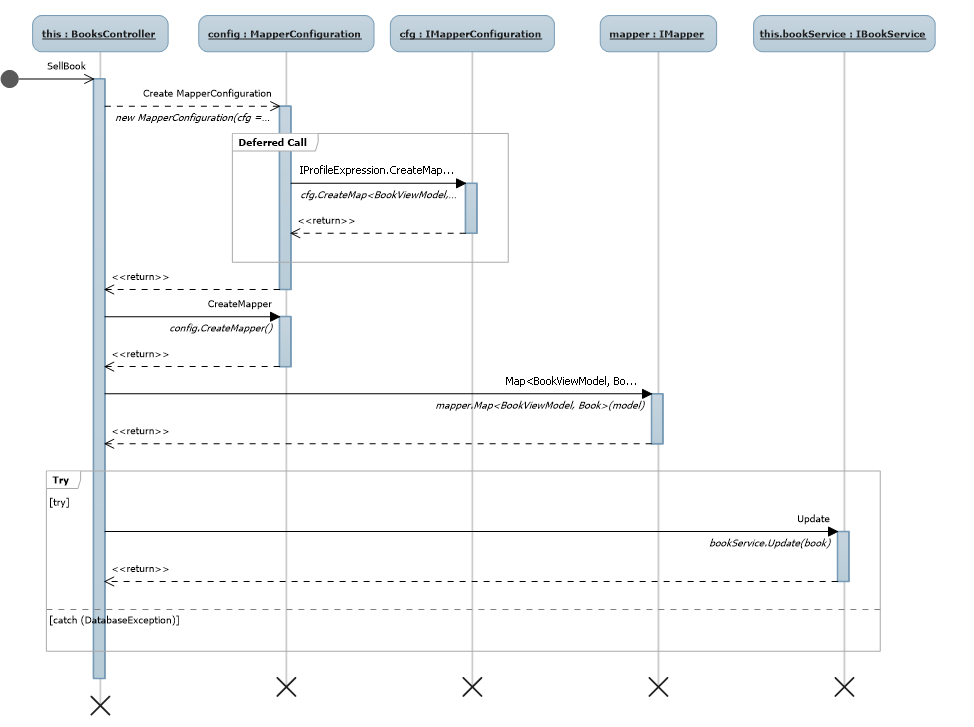
Xml Database

4. UML Sequence Diagrams

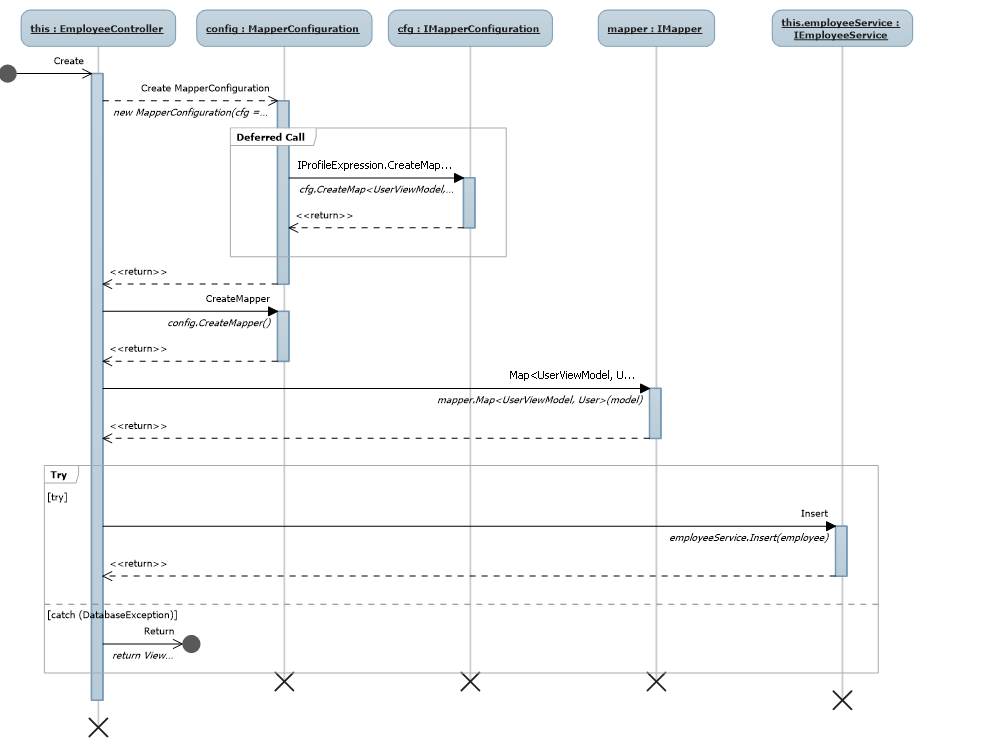
In the next image we will present a sequence diagram in which we describe the flow of login:



The next sequence diagram shows the way an employee can sell books.



The next sequence diagram shows the way the admin can create a new employee.



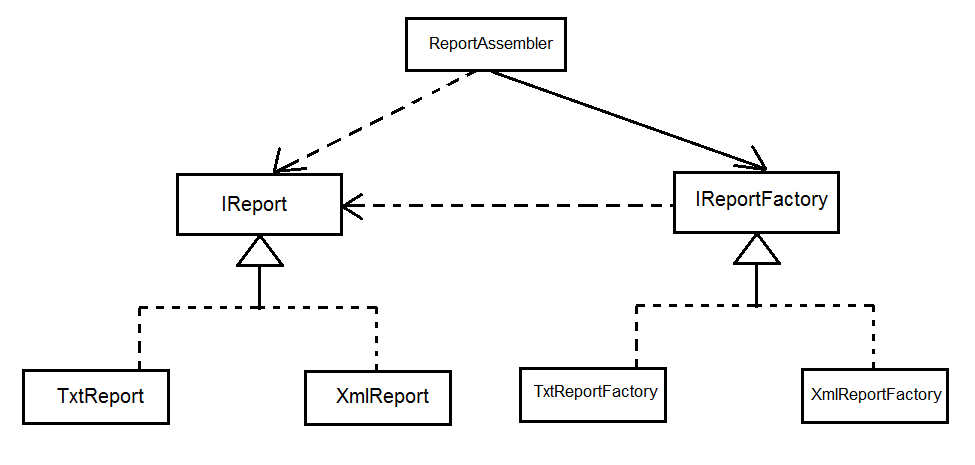
5. Class Design

**5.1 Design Patterns Description**

The factory completely abstracts the creation and initialization of the product from the client. This indirection enables the client to focus on its discrete role in the application without concerning itself with the details of how the product is created. Thus, as the product implementation changes over time, the client remains unchanged.

While this indirection is a tangible benefit, the most important aspect of this pattern is the fact that the client is abstracted from both the type of product and the type of factory used to create the product. Presuming that the product interface is invariant, this enables the factory to create any product type it deems appropriate. Furthermore, presuming that the factory interface is invariant, the entire factory along with the associated products it creates can be replaced in a wholesale fashion. Both of these radical modifications can occur without any changes to the client.

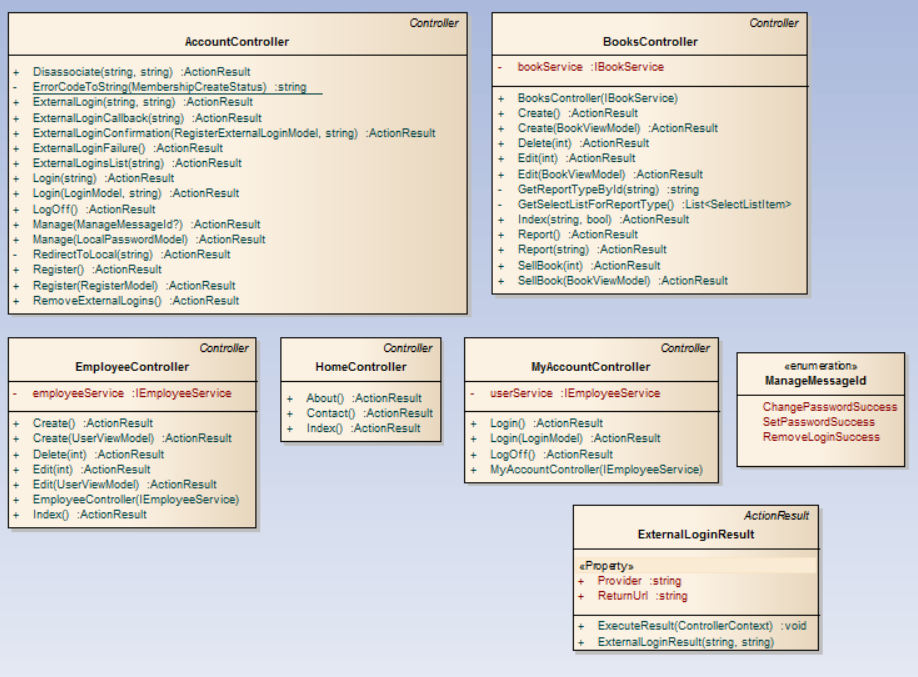
In the next image we can see this pattern is use, the picture shows the classes and interfaces of our particular case.



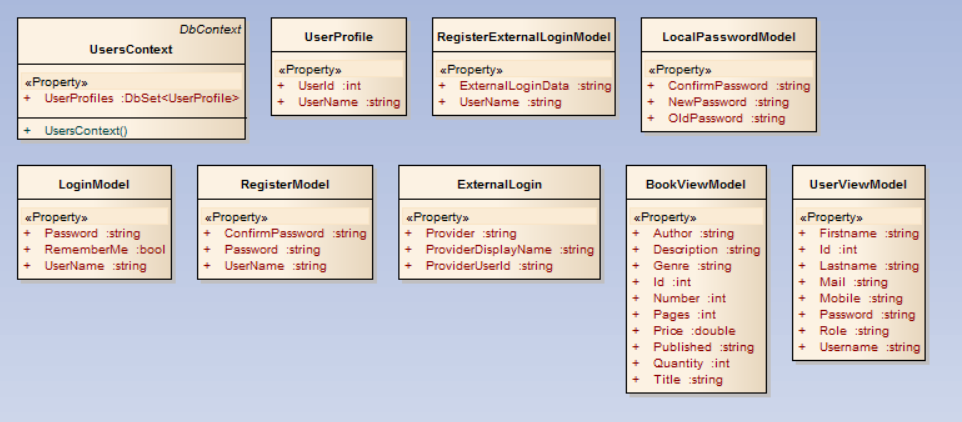
**5.2 UML Class Diagram**

**Class diagram for Bookstroe (MVC)**

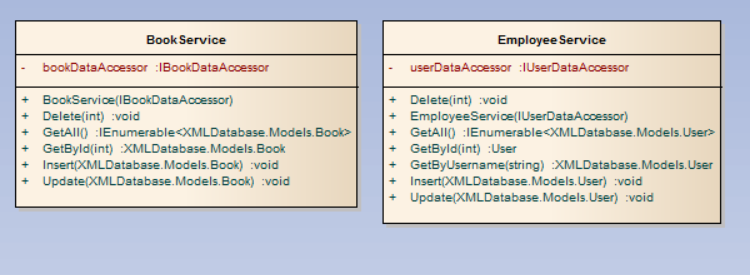
*Controllers*

**

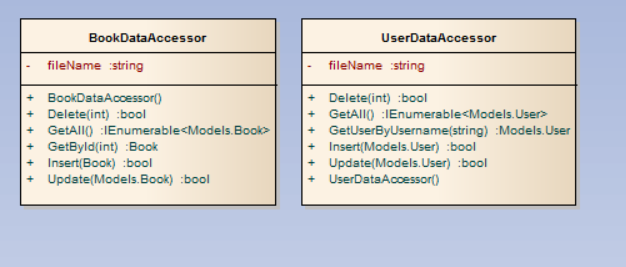
*Models*

**

**Class diagram for Services**

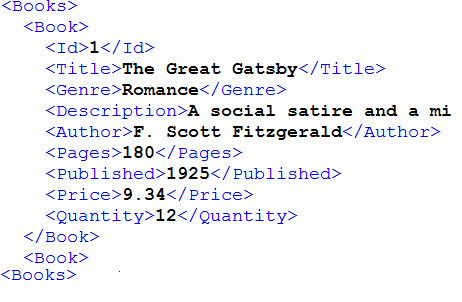


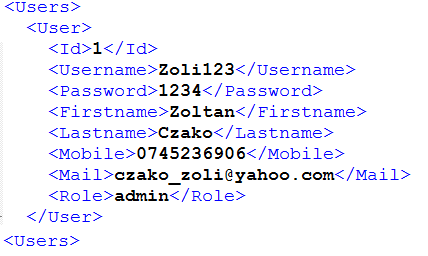
**Class diagram for DataAccessor**

****

6. Data Model

The data of the bookstore is stored in .xml file. The next images shows the structure of the xml files with an example.

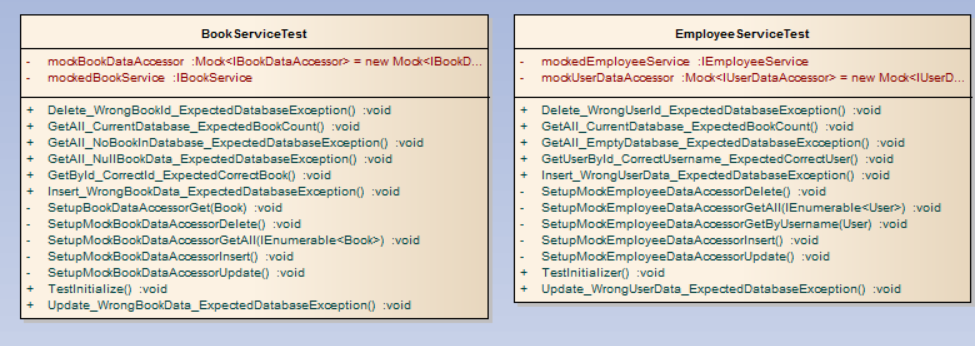




7. System Testing

We wanted to create a reliable and functionally correct project so we created test for each functionality, for each data validation and conditional statements. We used the Unit test framework and we tested the Services, because in this part of the project we had a lot of logic. We tested the corner cases as well as the happy flow situations.

The next pictures shows our test units, classes and methods that we used to cover as many situations as we could.



8. Bibliography

[1] <https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx>

[2] <https://msdn.microsoft.com/en-us/library/ee817667.aspx>

[3] <http://www.asp.net/mvc/overview/older-versions-1/nerddinner/create-a-new-aspnet-mvc-project>