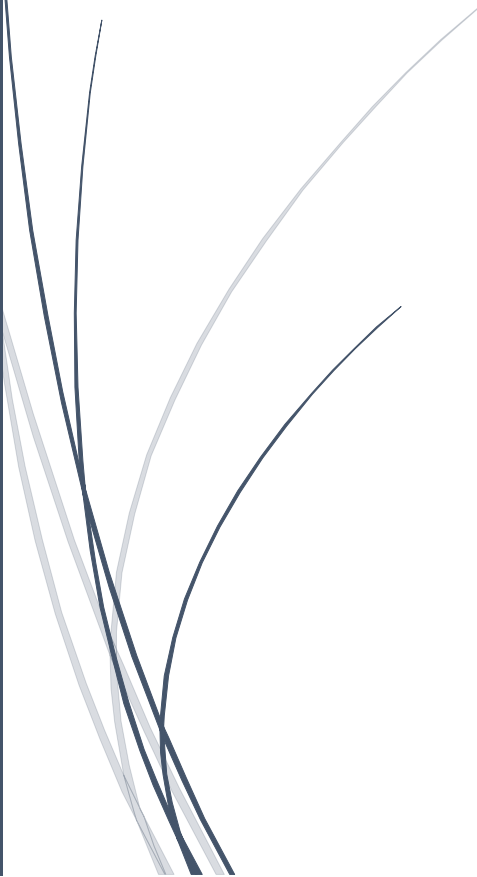
A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date 4/8/2019.

4/8/2019

Public library

Group-2

Several thin, curved lines in dark blue and light grey originate from the bottom left and curve upwards and to the right.

Devanshu Srivastava, Nirav Solanki, Ravi
Zala, Eugene Shishlannikov

B00810667 | B00808427 | B00805073 | B00806895

Devanshu Srivastava, owner for the below modules. Lines of code:

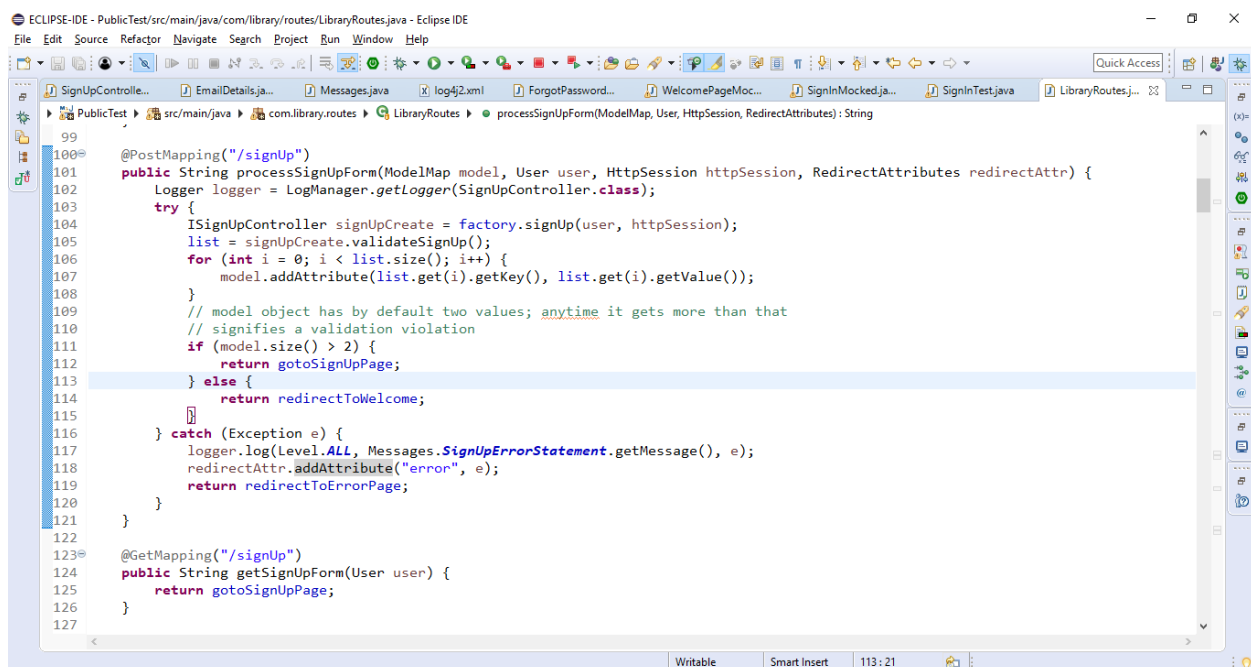
Registration Page

I needed to register our users into our library system. Hence, my module took whole responsibility of the following:

- Registering users
- Configurable business logic
- Registration form validations
- Sending user an email on successful registration to our application (Bonus functionality)
- On successful registration user is redirected to Welcome page
- Adding SALT to the password

Implementation

For implementing this module end to end I have developed multiple test cases but at first our database was not fully developed hence I created my own mock data that I tested. Hence, this was how I followed TDD in this module. Test cases can be found under `src/test/java/com/library`. Added insightful logging; Refactoring was done. I have followed the approach of throw early - catch late. If anything goes into catch I am redirecting the user to ErrorPage.



```
ECLIPSE-IDE - PublicTest/src/main/java/com/library/routes/LibraryRoutes.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

SignUpController... EmailDetailsJa... Messages.java log4j2.xml ForgotPassword... WelcomePageMoc... SignInMockedJa... SignInTest.java LibraryRoutes.j...
PublicTest > src/main/java > com.library.routes > LibraryRoutes > processSignUpForm(ModelMap, User, HttpSession, RedirectAttributes) : String

99
100 @PostMapping("/signUp")
101 public String processSignUpForm(ModelMap model, User user, HttpSession httpSession, RedirectAttributes redirectAttr) {
102     Logger logger = LogManager.getLogger(SignUpController.class);
103     try {
104         ISignUpController signUpCreate = factory.signUp(user, httpSession);
105         list = signUpCreate.validateSignUp();
106         for (int i = 0; i < list.size(); i++) {
107             model.addAttribute(list.get(i).getKey(), list.get(i).getValue());
108         }
109         // model object has by default two values; anytime it gets more than that
110         // signifies a validation violation
111         if (model.size() > 2) {
112             return gotoSignUpPage;
113         } else {
114             return redirectToWelcome;
115         }
116     } catch (Exception e) {
117         logger.log(Level.ALL, Messages.SignUpErrorStatement.getMessage(), e);
118         redirectAttr.addAttribute("error", e);
119         return redirectToErrorPage;
120     }
121 }
122
123 @GetMapping("/signUp")
124 public String getSignUpForm(User user) {
125     return gotoSignUpPage;
126 }
127
```

The above is the HTTP verb function to registration module, this function which is shown in the above figure takes care of routing to different pages and it takes care of setting up the validations at the UI of registration page from reading XML file. The logic is simple so to say the process is that user arrives at registration page fill the form details and then the data of the user is checked against validations that I fetch from reading an xml file and then I use configurable business logic for setting form validation rules.

Then, after validations are successfully done, user data is traversed to DAO layer where it connects with DB and voila User is finally inside the library system (If everything goes happy path.) I am also sending the user a welcome email anyways if anything goes wrong, I am logging information and customized ErrorPage is always there because I don't want my user to see JAVA errors which is pathetic.

Java files that I worked on while developing the registration module are as follows:

- a. LibraryRoutes
- b. SignUpController and ISignUpController
- c. com.library.validations all the files inside it contains configurable business logic
- d. XMLParser
- e. Some part of Messages as I need to create enums like the ones that are used for redirecting routes etc.
SignInForm("SignInForm"),
SignUpForm("SignUpForm"),
Welcome("Welcome").
- f. com.library.email all the files inside.
- g. Created singleton and factory for initializing Controller classes like ISignInController/ISignUpController etc.
- h. com.library.mockDB except BrowsePageMocked.
- i. Test cases.

Login Page

I needed to login our users into our library system. Hence, my module took whole responsibility of the following:

- Login users
- Configurable business logic
- Login form validations
- Forgot password redirection.
- Sending user an email on successful registration to our application (Bonus functionality)
- Checking on login if the user is admin or a regular user.
- On successful registration user is redirected to Welcome page
- Adding SALT to the password and then checking the password which is stored in the Database.

processSignInForm() is the first method that will be called when user clicks on the signIn button. This function takes care about calling the validations and processing the signIn for the user.

Java files that I worked on while developing the login module are as follows:

- a. LibraryRoutes
- b. SignInController and ISingInController

- c. `com.library.validations` all the files inside it contains configurable business logic
- d. `XMLParser`
- e. Some part of Messages as I need to create enums like the ones that are used for redirecting routes etc.
`SignInForm("SignInForm"),`
`SignUpForm("SignUpForm"),`
`Welcome("Welcome").`
- f. Created singleton and factory for initializing Controller classes like `ISignInController` etc.
- g. `com.library.mockDB` except `BrowsePageMocked`.
- h. Test cases.

```

194 @GetMapping("/signIn")
195 public String getSignInForm(User user) {
196     return gotoSignInPage;
197 }
198
199 @PostMapping("/signIn")
200 public String processSignInForm(HttpSession httpSession, ModelMap model, User user,
201     RedirectAttributes redirectAttr) {
202     Logger logger = LogManager.getLogger(SignInController.class);
203     try {
204         SignInController signIn = factory.signIn(user, httpSession);
205         list = signIn.validateSignIn();
206         for (int index = 0; index < list.size(); index++) {
207             model.addAttribute(list.get(index).getKey(), list.get(index).getValue());
208         }
209         // ModelMap by default has two values and anytime it gets more than that
210         // signifies validation violation
211         if (model.size() > 2) {
212             return gotoSignInPage;
213         }
214         return signIn.checkUserCredential();
215     } catch (Exception e) {
216         logger.log(Level.ALL, Messages.SignInErrorStatement.getMessage(), e);
217         redirectAttr.addAttribute("error", e);
218         return redirectToErrorPage;
219     }
220 }
221
222
223

```

Welcome Page

I needed to welcome the registered, non-registered and admin user to this module. The feature I supported are as follows:

- The ability to login or registers for the site (link to other pages)
- A logo with the site's name
- A simple text search box, the search text would be clickable to go to an advanced search page with the ability to search each individual field in the database.

- Searching takes you to a search results page which lists any items that match your search
- The New Arrivals section lists the newest X (where X is configurable) items added to the library, showing the photo for the item and some way of knowing it is a movie, book, or music.
- The Most Popular section lists the X (where X is configurable) items that are the most popular at the library (as determined by number of borrows + number of holds).
- The Browse section has logos to take them to the browse page for books, movies or music.
- A new section appears labeled “Admin Tools” with links to the Add Library Items page and the Loan Management page.

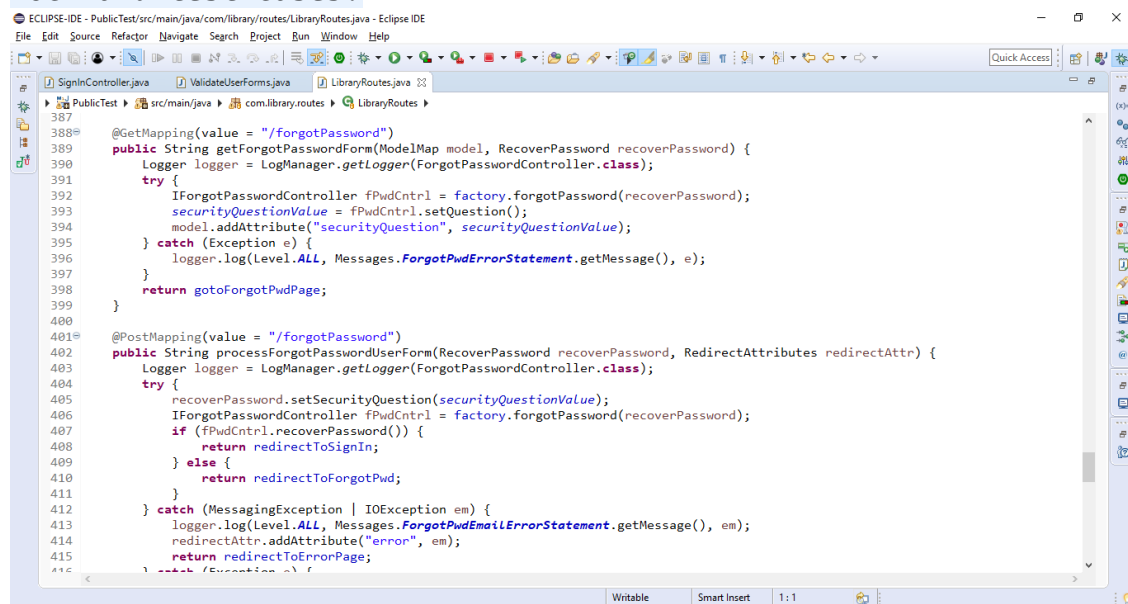
File altered are as follows:

- Files under welcome package.
- Created LibraryItemDAO / ILibraryItemDAO that take care of connecting to and from database
- Most popular and latest items are also taken care by me.
- Test cases and mock data.

Forgot password Module

I created this module to email the user their password and for this I had created test cases and mock data and below are the files I altered.

- forgotPassword package.
- Mock and test cases.



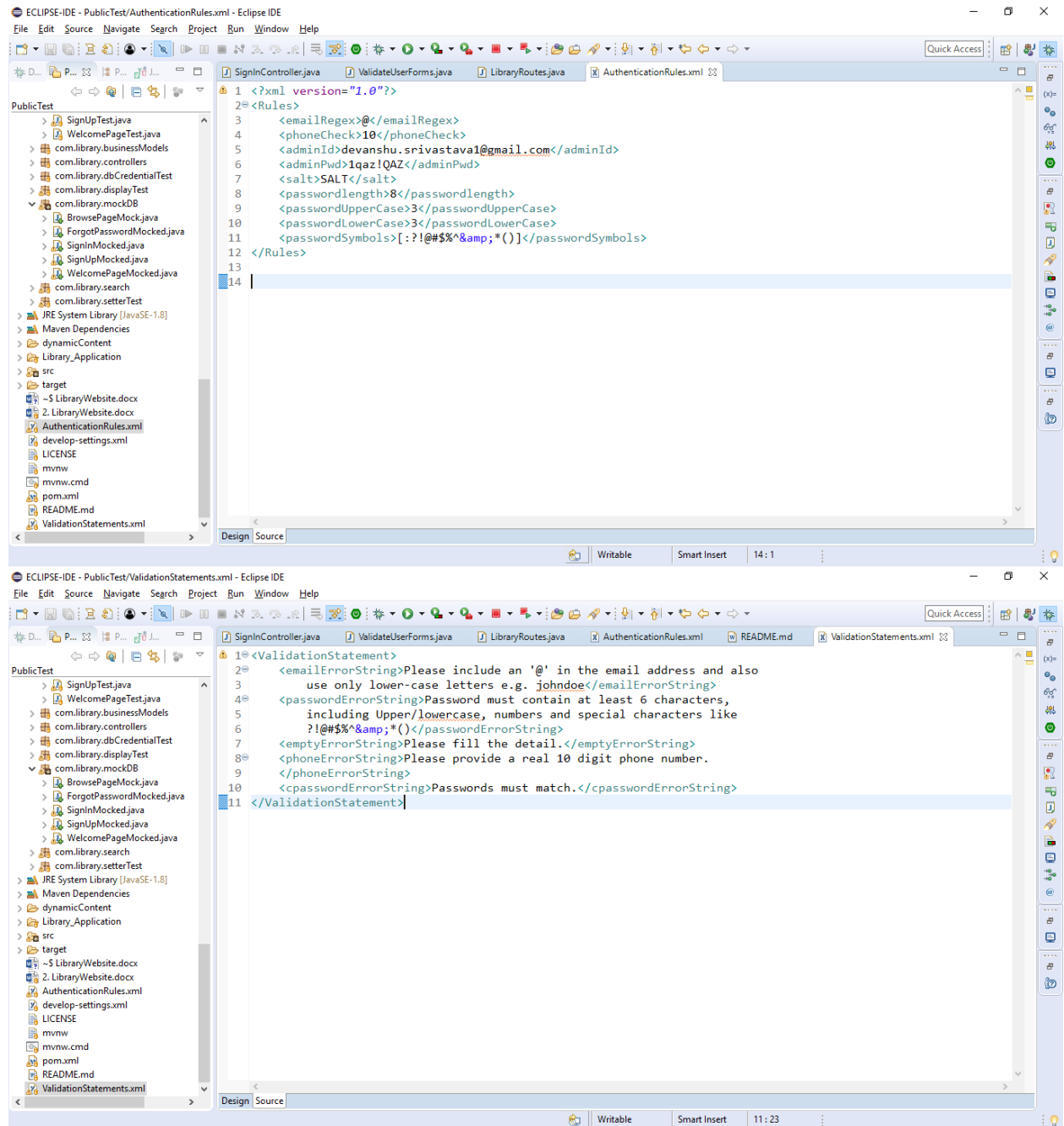
```

ECLIPSE-IDE - PublicTest/src/main/java/com/library/routes/LibraryRoutes.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
PublicTest src/main/java com.library.routes LibraryRoutes
387
388 @GetMapping(value = "/forgotPassword")
389 public String getForgotPasswordForm(ModelMap model, RecoverPassword recoverPassword) {
390     Logger logger = LogManager.getLogger(ForgotPasswordController.class);
391     try {
392         IForgotPasswordController fPwCtrl = factory.forgotPassword(recoverPassword);
393         securityQuestionValue = fPwCtrl.setQuestion();
394         model.addAttribute("securityQuestion", securityQuestionValue);
395     } catch (Exception e) {
396         logger.log(Level.ALL, Messages.ForgotPwErrorStatement.getMessage(), e);
397     }
398     return gotoForgotPwPage;
399 }
400
401 @PostMapping(value = "/forgotPassword")
402 public String processForgotPasswordUserForm(RecoverPassword recoverPassword, RedirectAttributes redirectAttr) {
403     Logger logger = LogManager.getLogger(ForgotPasswordController.class);
404     try {
405         recoverPassword.setSecurityQuestion(securityQuestionValue);
406         IForgotPasswordController fPwCtrl = factory.forgotPassword(recoverPassword);
407         if (fPwCtrl.recoverPassword()) {
408             return redirectToSignIn;
409         } else {
410             return redirectToForgotPw;
411         }
412     } catch (MessagingException | IOException em) {
413         logger.log(Level.ALL, Messages.ForgotPwEmailErrorStatement.getMessage(), em);
414         redirectAttr.addAttribute("error", em);
415         return redirectToErrorPage;
416     } catch (Exception e) {
417

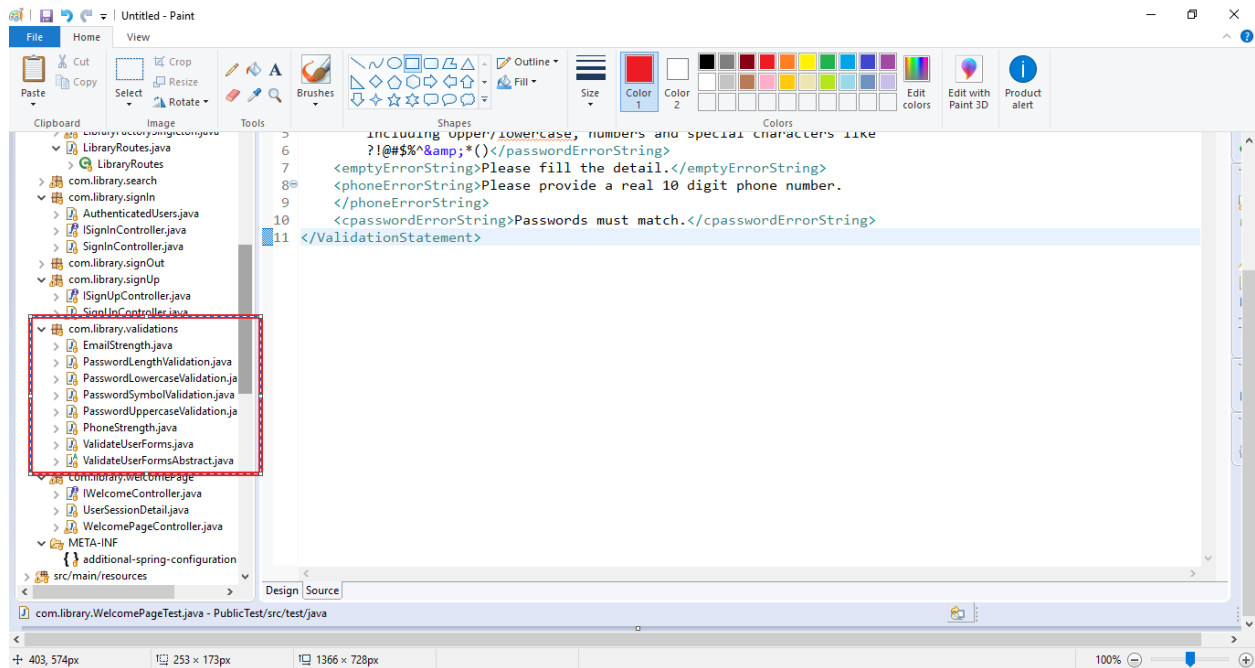
```

Validation Feature (Configurable business logic)

XML parser parses the xml file and in the xml file there are many rules and validation strings that are fetched in the configurable business logic flow where the classes gets initialized only of there are rules defined for them in the XML.



The above are the XML files I parse and read for the validation of the two forms.



The above are the files I altered for the whole application validations. Also, I have created spring router that had the sole responsibility of routing the HTTP verb calls in our application

Table regarding packages and the work they perform developed by Devanshu Srivastava.

| Package (com.library.) | Work | Tests for modules |
|------------------------|--|--|
| mockDB | SignUpMocked data that I created for test cases. | Registration |
| mockDB | SignInMocked data that I created for test cases | Login |
| mockDB | ForgotPasswordMocked data that I created for test cases that helped me in developing this module | ForgotPassword (Bonus feature) |
| mockDB | WelcomePageMocked data that I created for testing the cases and understanding in developing welcome module. | Welcome page both admin and normal user. |
| signUp | This package contains class file and interface for the registration module and here the real business logic for registration initiates. After form validations the user is registered with SALT added to its password. Then this data is sent to the DAO layer where it connects to DB layer and finally user is registered. | Done |
| route | This package contains most important part of spring it contains all the routes to business logic, it is the backbone for our application. | None required |
| welcome | This package contains class file and interface for the welcome module and here the real business logic resides. The main logic for the routes to other pages and show the admin page, to show all the required features. | Done |
| forgotPassword | This package contains class file and interface for the ForgotPassword module and here the real business logic resides. Emailing the user their passwords. | Done |
| parser | Parses XML | Done |
| email | This package takes care of sending emails to the user | Done |
| validations | All the business logic regarding validations and configurable business logic resides here | Done |
| signIn | This package contains class file and interface for the login module and here the real business logic resides. After form validations and salted password is checked with the salted one in the Database. | Done |
| UserDAO.java | In this, I created two functions checkEmailExist(),getEmailRelatedPassword() | Done |
| Salt.java | This is bean class having setter getter functions | Done |

| | | |
|---|--|------|
| User.java, UserBasicInfo,UserExtendedInfo (Interfaces) | These were developed by me for the user data accumulation. | Done |
| LibraryItemDao,enums,Interfaces | All these were developed by me for populating welcome page data. | Done |
| Xml's | Xmls regarding validations are taken care by me they are positioned in the parent directory. | Done |

Feature wise table

| Module name | Owner | Refactoring | Pattern | Logging | Tests | Technical Debt |
|-----------------|---------------------|-------------|------------------------------|---------|-------|--|
| Registration | Devanshu Srivastava | Done | Configurable Business logic. | Done | Done | I could have applied Template pattern for both registration and login modules validation. Although I tried to use template method but couldn't use it as efficiently as possible. I could have used a single method that would be responsible for putting validations to both registration/login module. For now, I have two different methods for validating different form validations. I will refactor it. Also, I could have created more insightful test cases. |
| Login | Devanshu Srivastava | Done | Configurable business logic | Done | Done | I could have applied Template pattern for both registration and login modules validation. Although I tried to use template method but couldn't use it as efficiently as possible. I could have used a single method that would be responsible for putting validations to both registration/login module. For now, I have two different methods for validating different form validations. I will refactor it. Also, I could have created more insightful test cases. |
| Welcome | Devanshu Srivastava | Done | None | Done | Done | None |
| Forgot password | Devanshu Srivastava | Done | None | Done | Done | None |

Eugene Shishlannikov owner of the features described below

Search and Search Results Pages

Feature Description

The search feature consists of two modes of search (basic and advanced) and the search results page that is the same for both search modes. Basic search is available to both logged in and guest users, whereas advanced search is available to the logged in user only. I have understood

that from the advanced search page wireframe in the project description where user email is depicted. Basic search is available on the main page. The implementation satisfies the requirements of the project definition.

Implementation

During the code refactoring I have made a design choices that has allowed me to write the code in the generic form, so once the need to add a new search category will emerge, that can be done by extending base classes and implementing interfaces in the way that is like I have implemented it for the book, video and music categories. For adding a new search category some changes will be required in the `LibraryRoutes.java` and the HTML templates.

Unit tests were implemented for most of the classes and their methods. Unit tests are located at the convenient for the Spring projects directory - `src/test/java`. The name of the test package is the same as the name of the code package.

I have also implemented the basic tests for the `com.library.routes.LibraryRoutes.java` class. This class is the Spring application routing controller. Spring Controller is a Spring bean; therefore, all its dependencies must be Spring beans, hence `DBSearchController` is a bean. Because of this I did not integrate the `DBSearchController` into the `com.library.routes.LibraryControllerfactory.java` factory class, but instead I have introduced the `DBSearchControllerBean.java` class which constructs the `DBSearchController` bean.

Because `DBSearchController` is a bean, this forced any class in the chain of dependencies be a bean. This way `DBSearchController`, `SearchResultsCoverImageProxy` and `CoverImageLoader` were made Spring beans.

Technical Debt

1. I have defined the Spring beans in the code. The better way is to define them in the configuration file. The use of the configuration files requires further exploration of Spring framework.
2. The methods `executeAdvancedSearch()` and `executeBasicSearch()` (POST mapping) in the class `com.library.routes.LibraryRoutes` has 6 arguments and this number will grow if a new search category as DVD search will be introduced. Part of the arguments probably might be replaced by one argument containing the others, but this requires deeper learning of Thymeleaf (Java template engine), as attempts to search the web didn't yield a solution. I would be able to avoid this situation completely if I was using JSON for the backend – frontend interaction.
3. We have several classes that implement the singleton pattern. In Spring framework, bean classes are singletons and we could use the Spring approach, but the learning curve of the Spring concepts and approaches to the unit testing is not deterministic in time of ongoing project implementation, so I didn't offer it to the team members.

Though I consider the knowledge of Spring framework I have gained in the implementation as of high value.

4. Item cover image loading from the database to the disk and its further managing. I have defined an immutable string that determine the paths to the loaded images. The proper implementation should include loading the path tokens from the configuration file and their concatenation with the appropriate OS separator (/ or \).
5. The strategies of managing search requests and results and the algorithm of the execution of the search in the database are erroneous. Using the right approach was simplifying the implementation, reducing the number of code lines both in the code and unit tests.
6. In the class `com.library.search.HttpSessionMonitor` I have defined constants. The proper way to have these constants is to read the from the configuration file on the application start. Same applies to the `DISPLAY_ROW_SIZE` in the `com.library.search.SearchResults` class

Classes that I have partially or completely implemented

| Package/class/interface name | Details |
|--|---|
| <code>com.library.dao.CoverDAO.java</code> | implementation of the class methods |
| <code>com.library.dao.ICoverDAO.java</code> | Complete implementation |
| <code>com.library.dao.MovieDAO.java</code> | methods <code>getMoviesBySearchTerms()</code> and <code>prepareSearchQuery()</code> |
| <code>com.library.dao.MusicDAO.java</code> | methods <code>getMusicBySearchTerms()</code> and <code>prepareSearchQuery()</code> |
| <code>com.library.dao.BookDAO.java</code> | methods <code>getBooksBySearchTerms()</code> and <code>prepareSearchQuery()</code> |
| <code>com.library.businessModels.BusinessModelsfactory.java</code> | Complete implementation |
| <code>Com.library.businessModels.Cover.java</code> | partial implementation of getters and setters |
| <code>com.library.businessModels.LibraryItem.java</code> | I removed code duplication in the existing code and have introduced this class. For unknown reason it was overridden by one of the group members. Using the blame feature shows it. |
| <code>com.library.routes.LibraryRoutes.java</code> | methods <code>getAdvancedSearchPage()</code> , <code>executeAdvancedSearch()</code> , <code>executeBasicSearch()</code> , <code>executeSearch()</code> |
| <code>com.library.businessModelSetter.CoverSetter.java</code> | Most of the implementation |
| <code>com.library.businessModelSetter.ICoverSetter.java</code> | Complete implementation |
| <code>com.library.localStorage.CoverImageLoader.java</code> | Complete implementation |
| <code>com.library.localStrorage.CoverImageLoaderBean.java</code> | Complete implementation |
| <code>com.library.localStrorage.ICoverImageLoader.java</code> | Complete implementation |
| <code>com.library.search</code> | All classes/interfaces in the package |
| <code>Com.library.signIn.AuthenticatedUsers.java</code> | Complete implementation |

| | |
|--|---------------------------------------|
| src.test.java.com.library.controllers.LibraryControllerTest.java | Complete implementation |
| src.test.java.com.library.search | All tests in the package |
| com.library.signOut | All classes/interfaces in the package |

**Nirav Solanki of the features described below
(B00808427)**

Browse page

Feature Description

The page is accessed by clicking on an item type (Book, Music, Movie) on the Main page. This page does not require user to be logged in to be fully functional. It works the same for both general user and administrator. Following features are included in this module:

- Category: On page load list of all category for the item Type are displayed to the user.
- Item Display: On clicking a category list of all the item belonging to a category is displayed to the user.
- Breadcrumbs: On clicking the item type breadcrumb should show the item type the user has accessed and a link to return to Main page. When a category is clicked, the breadcrumb displays the category along with a link to go back to category list or Main Page.
- Item Click: By clicking on a display item, user is taken to item detail page for that item.

Implementation

Supporting Files Used in this Module

1. BookDAO, MovieDAO, MusicDAO: Two functions from each of the above mentioned three java files were used. Function to call list of categories from the database and function to call list of items belonging to a category.

2. DAOFactory: n abstract factory to initiate DAO classes.
3. BookSetter, MovieSetter, MusicSetter: Has function which takes resultset and converts them into Book, Movie or Music objects.
4. Book, Movie, Music: Getter-Setter classes that contains information of an item.
5. CoverDAO: Used function to fetch cover image in the form of blob by using item ID is used.
6. DefaultImageFetcher: If image is not fetched by the CoverDAO, image fetched from the DefaultImageFetcher is used instead.
7. LibraryRoute: Contains functions to map java objects to the front end.

Files built for this module

1. Display: Getter-Setter class containing display Information (item ID, item type, title, image). The image setter uses CoverDAO to fetch image for the item ID and stores it in Base64 form. The image setter is only called when image getter is used.
2. DisplaySetter: Used functions to take list of Book, Movie or Music and convert them into Display Objects.
3. BrowseBooks, BrowseMovie, BrowseMusic: Contains two functionalities:
 - i. Get List of Categories for the item Type
 - ii. Fetch List of Display objects according fetched according to category

4. BrowseDisplayFactory: An abstract factory to make appropriate BrowseDisplayComponent.
5. DisplayComponentInitializer: Has functions that takes string as input and returns appropriate BrowseDisplayComponent using BrowseDisplayFactory.
6. BrowsePageCategory.html: Front end page to display categories
7. BrowsePageItems.html: Front end page to display items of a category

How does the code work?

From the welcome page, when the Book, Music or Movie is clicked the URL to fetch Browse Display Page is called. The URL is appended with a parameter which states which item Type needs to be fetched. This parameter is further used to make object following IBrowseDisplayComponent, used for fetching categories and items. Abstract factory is used to create this object for Book, Music or Movie. The URL parameter is also appended in the breadcrumb. If in the future a new item Type emerges it could be easily integrated.

Categories are fetched using database interface, stored inside the Web Page model and finally mapped into front end. By clicking on a category, user is directed to a new URL which has the item type and the category imbedded as parameters.

The parameter of item type and category is imbedded into the breadcrumb. The item type in the breadcrumb is treated as a link to back to the previous page. Also, a link to go back to the main page is included in the breadcrumb. Appropriate object implementing IBrowseDisplayComponent is made again using the item type parameter. The object communicates with the Database interface, fetches items according to the category and converts it into display objects. This display is appended to the Web Page Model and finally mapped to the front end.

The reason for using display objects is to make the code generic for all three item types. Display object title and image, which are mapped to the front end. The Display object also contains item type and item ID, which will be further used for fetching details for that item. When image is requested from the display object, it communicates with the database interface to fetch cover image using the item ID. If pagination is performed in the future, images will only be

fetches when the item needs to be displayed on the screen. If a cover is not found mapped to the item ID, then a default image is loaded instead.

By clicking on the title in the front end, user is directed to item Detail page with parameter item type and item ID appended into the URL.

Testing

Most of the code for this module only required to fetch information from the database and map it to the front end. Because of this reason, the code was tightly coupled with the database interface. Hence, most of the code was not feasible for creating unit test. Unit tests were temporarily created to test out if the code was working but were then removed to avoid issues if changes are made to the data present in the database.

The only unit tests written were for the module DisplaySetter. A mock class was created to make list of Book, Movie and Music objects to perform the testing. Each list was passed to the methods present in DisplaySetter and testing was performed to check whether they return appropriate Display objects.

Technical Debt

The method to fetch categories and fetch all items of a category are separate concerns. They need to be separated out into individual Classes.

The user interface for this module needs to be improved.

The image loading is very slow, a more optimum solution could be implemented.

When transition is done to the item details page, all the data needs to be fetched again. Instead a way to send display object data further should be used. Currently due to limited understanding of Thymleaf, I was unable to implement that.

Item details page

Feature Description

This page can be accessed by clicking on an item in Main Page, Search Result Page or Browse Display Page. Following functionalities are included in this module:

- Fetching appropriate item information
- Displaying cover image for the item
- Showing the user status borrowed if the user has already borrowed the item
- Showing the user status Reserved if the user has already reserved the item
- Showing the user status Borrow if the item is available for borrowing, and by clicking on a button borrowing the item for the user. User is also sent a mail if borrowed successfully. Status is changed to Borrowed.
- Showing the user status Reserve if the item is available for reserving, and by clicking on a button reserving the item for the user. User is also sent a mail if reserved successfully. Status is changed to Reserved.

Implementation

Supporting Files Used in this Module

1. BookDAO, MovieDAO, MusicDAO: Two functions from each of the above mentioned three java files were used. Function to call list of categories from the database and function to call list of items belonging to a category. Functions to change count and availability and check item availability are also used.
2. UserItemDAO: Used to borrow or reserve item and check status.
3. DAOFactory: An abstract factory to initiate DAO classes.
4. BookSetter, MovieSetter, MusicSetter: Has function which takes resultset and converts them into Book, Movie or Music objects.
5. Book, Movie, Music: Getter-Setter classes that contains information of an item.

6. CoverDAO: Used function to fetch cover image in the form of blob by using item ID is used.
7. DefaultImageFetcher: If image is not fetched by the CoverDAO, image fetched from the DefaultImageFetcher is used instead.
8. LibraryRoute: Contains functions to map java objects to the front end.
8. AuthenticatedUsers: Used to fetch user Email ID from the session.

Files built for this module

1. DisplayDetailed: A getter-setter class containing display Information (item ID, item type, title, description, image). The image setter uses CoverDAO to fetch image for the item ID and stores it in Base64 form. The image setter is only called when image getter is used. It extends from the Display class and has additional description String.
2. DetailedDisplaySetter: Used functions to take list of Book, Movie or Music and convert them into DisplayDetailed Objects.
3. DetailsSetter: Takes Book, Movie or Movie object, extracts relevant information and joins them into a single string.
4. DetailedDisplayFetcher: Communicates with the database interface to fetch appropriate Book, Movie or Music object and then converts it into DisplayDetailed object.
5. Item Status: Takes the Email Address and item ID and return appropriate status of the user for the item.
6. BookItem: Borrow or reserves item for the user according to appropriate status.

7. Email Sender: Sends email to user according to context (Borrowed or Reserved)
8. ChangeItemCount: Changes the count of the item in the database.
9. DecreaseAvailability: Decreases Availability of item in the database

How does the Code Word?

On clicking an item, user is directed to the item detail page. The URL is appended with two parameters item Type and item ID. User Email Address is also fetched from the session. Using the parameters DetailDisplayFetcher fetches appropriate DisplayDetailed object. The DisplayDetailed contains title, id, image, type and description. Description contains appropriate information for Book, Music and Movie. DisplayDetailed is used instead of Book, Movie or Music to provide more generic code. DisplayDetailed is mapped at the front end.

The status of the item is checked by sending user Email Address and item ID. The status is mapped into a button at the front end. If the status is Borrowed or Reserved no action is taken on button click. If the status is Borrow or Reserve, item is borrowed or reserved using BookItem Class. Two additional action are taken: appropriate email is sent to the user and count of the item is increased. Additionally, if the item is Borrowed, availability of the item is decreased.

Testing

Most of the code for this module only required to fetch information from the database and map it to the front end. Because of this reason, the code was tightly coupled with the database interface. Hence, most of the code was not feasible for creating unit test. Unit tests were temporarily created to test out if the code was working but were then removed to avoid issues if changes are made to the data present in the database.

Two unit test module remained. A unit test module to check if appropriate DisplayDetailed object are returned for Book, Movie and Music by DetailedDisplaySetter. And another to check if appropriate description string is returned for Book, Movie and Music by DetailsSetter.

Technical Debt

The User Interface needs improvement.

When the page is called information is fetched again from the database. A way to use information fetched on the previous page needs to be implemented.

Preventing the admin from booking or reserving an item needs to be implemented.

Exception handling and printing error messages to the user is not implemented.

Currently concrete classes are used for item status and booking item. Interfaces needs to be used instead.

Classes that I have partially or completely implemented

| Package/class/interface name | Details |
|--|---|
| com.library.borrowItem | Completely Implemented |
| com.library. browsePage | Completely implemented |
| Com.library.bussinessModel.Book com.library. Movie, com.library. Music | Initially implemented by me, correctly refractured by Eugene to prevent code duplication |
| Display | Completely implemented |
| User | Initially implemented by me, slightly modified by Devanshu to cater to his module |
| bussinessModelSetter package | Implemented everything except CoverSetter |
| BookDAO | Implemented following methods: getBookByID, getBookByCategory, getBookCategories, createBook, getAvailability, increaseCount |
| MovieDAO | Implemented following methods: getMovieCategories, getAvailability, increaseCount |
| MusicDAO | Implemented following method: getMusicCategories, getAvailability, increaseCount |
| UserDAO | Implemented following method: checkPassword, registerUser |
| DatabaseCredential | Completely implemented |
| itemDetail package | Completely implemented |
| DefaultImageFetcher | Completely implemented |
| BrowserPageMock | Completely implemented |
| LibraryRoutes | Implemented following method: |

| | |
|--------------------------|---|
| | borrowItem, browsePageItems, itemDetailed, browsePageCategory |
| dbCredentialTest package | Completely implemented |
| displayTest package | Completely implemented |
| setterTest package | Completely implemented |

Owner Of the All The Below Modules is Ravi Zala(B00805073)

Administrator add item page (Member)

Member : Ravi Zala (B00805073)

Feature Description

Add Item feature facilitates the Administrator to add the various types of books, movies and music. Moreover administrator can also add the cover image of an item. This page displays the form of adding items according the values in content type. Clicking on Save Item will save the record in Database if an Item is not present in database. If an item is already present in database, application will show an error. Clicking on cancel button will redirect the Admin to welcome page.

Task Flows for this feature :

Welcome → Login As Admin → Welcome page for an Admin → Add Library Items Button → Add Item

Implementation

This feature is developed purely as Full-Stack by avoiding monolithic modules for each layer of MVC architecture. This feature helps developer in improving the skills in front end and backend frameworks as none is working just on frontend.

Thymeleaf framework for Springboot is used for separating the frontend and business logic of the application. As Front end is deciding which category to choose, so the possibility of using Strategy, Command or State pattern is eliminated.

Two types of Mappings, GET and POST are done for this feature. When the thymeleaf parse the AddItemPage.html page it checks the GET mapping of the forms for example, /addBook, /addMovie, and /addMusic. On clicking on submit button, POST mapping is done according to the selection of the content type. For example, if the book is selected, it will go to /addBook post mapping. Each mapping calls the abstract factory to create a concrete library factory reference. Which subsequently create the controller of interface type. For example, addBook mapping method will use, LibraryFactory reference to create IAddBookController.

LibraryRoutes class is used only for routing of the application. Particular controller is used for actual logic of adding items. These controllers have layer of abstraction to make it easy to change afterwards. These controllers calls the DAO classes methods for checking duplicacy, creating an item, getting item

categories. Creating an Item returns the newly generated key(itemId) which helps in storing the cover to separate table. This helps in normalizing database. Categories in front end are fetched from the database for configurability.

Cover image from the thymeleaf is sent as a MultipartFile which needs to be converted to BLOB type in order to store it in covers table. By Serializing the bytes fetched from MultipartFile, image is converted into BLOB type. Using BusinessModels classes, which are nothing but POJO(Plain Old Java Object) object or Value Object, provides the getter and setter methods of the attribyte. This helps in interacting with the DAO classes.

Code Refactoring

Code refactoring is done in terms of Extract Class to maintain adherence to Single Responsibility Principle, Prevent Duplication of Code, Prevent Large Class, and Divergent Change.

Replace Method with Method Object : Moving the method isCoverAddedToDatabase() to separate class ItemCoverSetter helps in eliminating the code smells. So whenever this method changes, controller classes do not have not changed.

Rename Method : Rename method is useful for reducing the use of comments.

Parameter Object : BusinessModels classes(Book,Movie,Music etc) are useful in removing the long parameter code smell.

Enums : . Enums are used for the messages in a particular module which are constants.

Decompose Conditional : Method like checkBookDuplicacy() is useful for decomposing conditional. Useful for maintaining the code later on.

Classes that I have partially or completely implemented

| Package/class/interface name | Details |
|-------------------------------|---|
| com.library.dao.MovieDAO.java | Complete Implementation of most of the methods except for increaseCount , getMoviesBySearchTerms, and getMoviesCategories. For this module, I used checkMovieDuplcy and createMovie methods. |
| com.library.dao.MusicDAO.java | Complete Implementation of most of the methods except for for increaseCount , getMusicsBySearchTerms, and getMusicCategories.. For this module, I used checkMusicDuplcy and createMusic methods |
| com.library.dao.BookDAO.java | I implemented updateAvailability, checkBookDuplicacy in this class. For this module, I used checkBookDuplcy and createBookmethods |

| | |
|--|---|
| Com.library.businessModels | Complete implementation of Book, LibraryItem, Movie, Music, User, UserItem and Cover classes. For this module it is used for eliminating the long parameter smell and Setting the values in Frontend. |
| com.library.routes.LibraryRoutes.java | Partial Implementation.For this module, Methods under GetMappings and PostMappings of (/addBook,/addMovie,/addMusic) are implemented. |
| com.library..addItem.AddBookController.java | CompleteImplementation |
| com.library..addItem.AddMovieController.java | Complete Implementation |
| com.library..addItem.AddMusicController.java | Complete implementation |
| com.library..addItem.IAddBookController.java | Complete implementation |
| com.library..addItem.IAddMovieController.java | Complete implementation |
| com.library..addItem.IAddMusicController.java | Complete implementation |
| com.library..addItem.ItemCoverSetter.java | Complete Implementation |
| com.library..addItem.AddItemMessagesEnum | Complete implementation |
| com.library.dao.BookDAOEnums,CoverDAOEnums,Mov ieDAOEnums, MusicDAOEnums, UserDAOEnums, UserItemDAOEnums | Complete Implementation |

6.9.4 SOLID principles evaluation

Code refactoring helps in achieving the goal of maintaining the adherence to Single Responsibility Principle. Moreover, using the different interfaces for different controllers maintain the Interface Segregation principle. Modules and Classes are open for extension but closed for modification so it also maintains the Open/Closed principle.

6.9.5 Screenshots

Add Item

Content Type

Category

Title

Author

ISBN

Publisher

Description

Cover Image: Jason.jpg



Add Item

Content Type

Category

Title

Author

ISBN

Publisher

Description

Cover Image: No file chosen



Book Successfully added!

6.9.6 Technical Debt

Test cases not needed as most of the functions are using DAO classes. Trying to implement the tests using MockDB. Logging coverage is 85% done. Remaining logging is left due to time constraints.

Administrator loan management page (Member)

Member : Ravi Zala (B00805073)

Feature Description

Loan Management feature facilitates Administrator to return the outstanding borrowed items to the library. On returning these items, this feature also check if any user is on hold for this item. If there are multiple users are on hold for a single item, the item will be automatically borrowed for the user by First Come First Serve (FCFS) basis.

User will be notified via email for the booking of an item and it will appear in the borrowed item. Admin can also return to welcome page on clicking cancel button. After returning the item if there are not any users on hold, it increases the availability for the item. Administrator can select/deselect all the items from the table. Administrator can also sort the table alphabetically for each columns.

Task Flow for this feature :-

Welcome → Login As Admin → Welcome page for an Admin → Loan Management Button → Loan Management

Implementation

After carefully designing the schema for this module, Thymeleaf and Bootstrap table are used in creating the front end part. This feature also follows the full-stack convention. Routes for the feature are mentioned in LibraryRoutes file. Main function of this feature is to return item to library or some user. So route will call controller to make this happen. JQuery is used for getting the selected checkbox values to the UserItem object of business model. JSON parser is implemented as these values are in JSON String format.

From the route page, controller takes the list of items to be removed. This remove item will be a long method as actual return process is to send email, remove item from database, add item to database, increase availability, checkItemOnHold etc. for each content type. Strategy pattern is used in order to prevent the endless switch statement. BookReturnStrategy, MovieReturnStrategy and MusicReturnStrategy are called by LoanManagementContext to execute the particular strategy. Controller will have a context for selecting the strategy. Controller will remove the item from the table but the whole return item process will be done by ReturnStrategy concrete classes.

ReturnStrategy classes implements the IReturnStrategy interface. Return strategy has methods for increasing availability, check if item on hold, remove user from hold, and add user to borrowed item. This strategies interacts with UserItemDAO to execute these methods. SendBookingEmail is an extract class for sending the email as this function is same in all the strategies. Category enum is used for selecting the strategy.

Classes that I have partially or completely implemented

| Package/class/interface name | Details |
|------------------------------|---------|
|------------------------------|---------|

| | |
|--|---|
| com.library.dao.MovieDAO.java | Complete Implementation of most of the methods except for increaseCount , getMoviesBySearchTerms, and getMoviesCategories. For this module, I used updateAvailability and getAvailability methods. |
| com.library.dao.MusicDAO.java | Complete Implementation of most of the methods except for for increaseCount , getMusicsBySearchTerms, and getMusicCategories. For this module, I used updateAvailability and getAvailability methods. |
| com.library.dao.BookDAO.java | I implemented updateAvailability, checkBookDuplicacy in this class. For this module, I used updateAvailability and getAvailability methods. |
| Com.library.businessModels | Complete implementation of Book, LibraryItem, Movie, Music, User, UserItem and Cover classes. For this module it is used for eliminating the long parameter smell and Setting the values in Frontend. |
| com.library.routes.LibraryRoutes.java | Partial Implementation. For this module, Methods under GetMappings and PostMappings of (/loan) are implemented. |
| com.library.LoanManagement.BookReturnStrategy | Complete Implementation |
| com.library.LoanManagement.MovieReturnStrategy | Complete Implementation |
| com.library.LoanManagement.MusicReturnStrategy | Complete Implementation |
| com.library.LoanManagement.IReturnStrategy | Complete Implementation |
| com.library.LoanManagement.LoanManagementContext | Complete Implementation |
| com.library.LoanManagement.Select | Complete Implementation |
| com.library.LoanManagement.CategoryEnum | Complete Implementation |
| com.library.LoanManagement.SendBookingEmail | Complete Implementation |
| com.library.LoanManagement.LoanManagementController | Complete Implementation |
| com.library.LoanManagement.ILoanManagementController | Complete Implementation |

6.10.3 Code Refactoring

Remove Middle Man : LoanManagementContext doesn't tell other class to call this function instead it calls the all the methods of the ReturnStrategy classes.

Replace Method with Method Object : sendBookingEmail method of ReturnStrategy classes is replaced by SendBookingEmail method object. Useful for isolating a long method and removing code duplication.

Decompose Conditional : Method like isItemOnHold() is useful for decomposing conditional. Useful for maintaining the code later on.

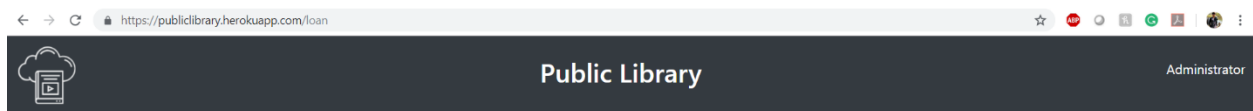
Parameter Object : BusinessModels classes(UserItem, Book,Music,Movie etc) are useful in removing the long parameter code smell.

Enums : . Enums are used for the messages in a particular module which are constants.

6.10.4 SOLID principles evaluation

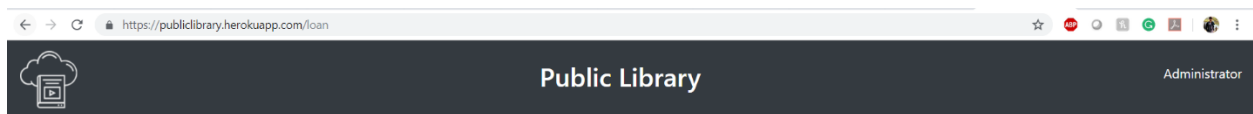
Strategy pattern helps this module to make it open for extension closed for modification. It helps in maintaining Open/Closed Principle. It maintains the Single Responsibility Principle by doing the code refactoring mentioned above.

Screenshots



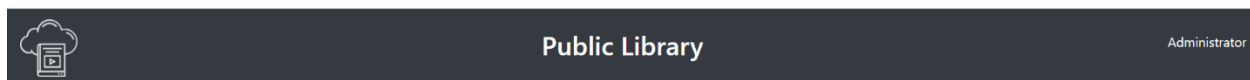
| <input type="checkbox"/> | Content Type | Title | User |
|--------------------------|--------------|--------------|------------------------|
| <input type="checkbox"/> | Movie | Interstellar | ravizala.emp@gmail.com |
| <input type="checkbox"/> | Movie | Inception | dv960112@dal.ca |
| <input type="checkbox"/> | Music | Despacito | nr952727@dal.ca |
| <input type="checkbox"/> | Music | Thunder | ravizala.emp@gmail.com |
| <input type="checkbox"/> | Book | Alchemist | ravizala.emp@gmail.com |
| <input type="checkbox"/> | Book | Love Story | ravi.zala@dal.ca |

[Return Items](#) [Cancel](#)



| <input checked="" type="checkbox"/> | Content Type | Title | User |
|-------------------------------------|--------------|--------------|------------------------|
| <input checked="" type="checkbox"/> | Movie | Interstellar | ravizala.emp@gmail.com |
| <input checked="" type="checkbox"/> | Movie | Inception | dv960112@dal.ca |
| <input checked="" type="checkbox"/> | Music | Despacito | nr952727@dal.ca |
| <input checked="" type="checkbox"/> | Music | Thunder | ravizala.emp@gmail.com |
| <input checked="" type="checkbox"/> | Book | Alchemist | ravizala.emp@gmail.com |
| <input checked="" type="checkbox"/> | Book | Love Story | ravi.zala@dal.ca |

[Return Items](#) [Cancel](#)



Loan Management

| Content Type | Title | User |
|----------------------------|-------|------|
| No Matching Records Found! | | |

Return Items

Cancel

Technical Debt

Test cases not needed as most of the functions are using DAO classes. Trying to implement the tests using MockDB. Logging coverage is 85% done. Remaining logging is left due to time constraints.

Logging (Member : Ravi Zala)

We have used log4j2 as our logger, for this we have created a XML file that looks like below figure attached,

```
ECLIPSE-IDE - PublicTest/src/main/resources/log4j2.xml - Eclipse IDE
File Edit Source Navigate Search Project Run Window Help

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN" monitorInterval="30">
  <Properties>
    <Property name="LOG_PATTERN">
      %d{yyyy-MM-dd HH:mm:ss.SSS} %5p ${hostName}---
      [%15.15t] %-40.40c{1.} : %m%n%ex
    </Property>
  </Properties>
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      follow="true"
      <PatternLayout pattern="${LOG_PATTERN}" />
    </Console>
    <JDBC name="SQLAppender" tableName="LMS_LOGS">
      <ConnectionFactory>
        class="com.Library.dbConnection.DatabaseConnection" method="getConnection" />
      <Column name="LOG_ID" pattern="%u" />
      <Column name="ENTRY_DATE" isEventTimestamp="true" />
      <Column name="LOGGER" pattern="%logger" />
      <Column name="LOG_LEVEL" pattern="%level" />
      <Column name="MESSAGE" pattern="%m" />
      <Column name="EXCEPTION" pattern="%throwable" />
    </JDBC>
  </Appenders>
  <Loggers>
    <Logger name="com.Library.dao" level="all" additivity="false">
      <AppenderRef ref="SQLAppender" />
    </Logger>
    <Logger name="com.Library.signIn" level="all" additivity="false">
      <AppenderRef ref="SQLAppender" />
    </Logger>
  </Loggers>
</Configuration>
```

Ex Custom Logger :-

```
<Logger name="com.Library.signIn" level="all" additivity="false">
  <AppenderRef ref="SQLAppender" />
</Logger>
```

```
logger.log(Level.ALL, "Check the SQL syntax of :"+query, e);
```



Most of the logging activities are done in DAO classes. Using throw early and catch late, logging is done where the exception can be occurred. With logger, meaningful message with variables are provided. Moreover the stacktrace of the all the calls upto this exception is provided with the logger.

List of Implemented Design Patterns:

- Strategy Pattern
- Proxy pattern
- Singleton pattern
- Observer pattern
- Factory pattern
- Bridge pattern
- Template method pattern
- Abstract Factory pattern