Final Report Part II

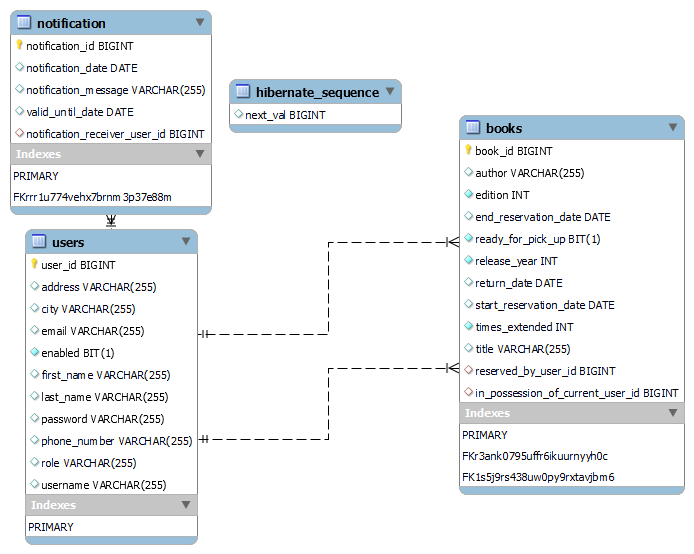
Student: Neremzoiu Ionuț-Dragoș

ACE UCV

**Introduction**

This part II of the report is meant to present and explain the overall design and implementation of this particular Spring MVC Web application.

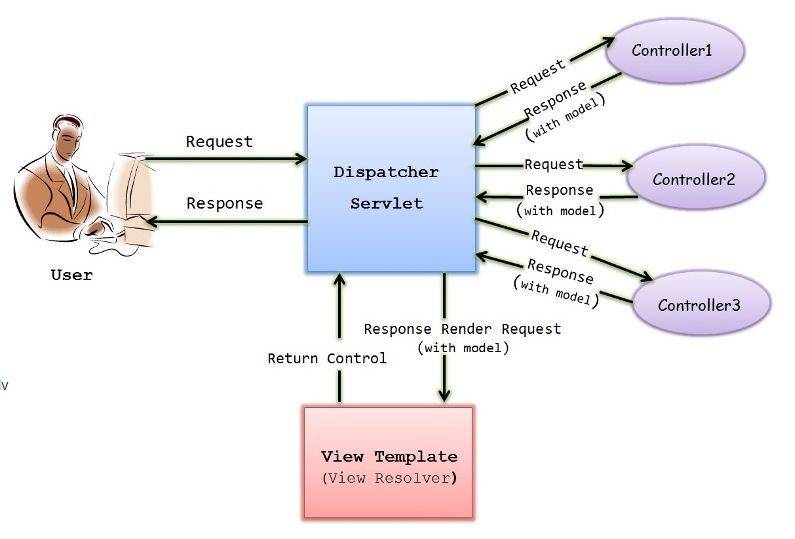
**Database Design**



As it can be seen in the database diagram above, there are only 4 tables, i.e.: **notification**, **users**, **books** and **hibernate\_sequence**. In order to clarify any possible doubts or possible misunderstandings, the following observations will be mentioned:

* **Users** and **notification** tables have a *one-to-many* relationship, meaning that a user can have multiple notifications.
* **Users** and **books** tables have a *one-to-many* relationship, meaning that a user can reserve or currently have in possession multiple books.
* The **reserved\_by\_user\_id** field is meant to specify which user has currently reserved a book, whereas **in\_possession\_of\_user\_id** is meant to reveal the fact that even though the reservation period has expired for an user, the book hasn’t been returned to the library by that user.
* **Hibernate\_sequence** table is a table that is generated the moment the **strategy** for a *@GenereatedValue* of a field (usually an id) is set to *GenerationType.AUTO*. The way Hibernate interprets *AUTO* generation type has changed starting with Hibernate **version 5.0**. When using Hibernate **v 4.0** and *Generation Type* as *AUTO*, specifically for MySql, Hibernate would choose the *IDENTITY* strategy (and thus use the *AUTO\_INCREMENT* feature) for generating IDs for the table in question. Starting with **version 5.0** when *Generation Type* is selected as *AUTO*, Hibernate uses *SequenceStyleGenerator* regardless of the database. In case of MySql Hibernate emulates a sequence using a table, because MySql doesn’t support the standard sequence type natively. Hence, that’s why if strategy *AUTO* is used, Hibernate will generate a table called hibernate\_sequence to provide the next number for the ID sequence. (Such details will also be found in the code)

**Spring MVC Architecture**



**Flow of a Spring MVC application**:

1. User makes a request through an URL.
2. URL is passed to dispatcher servlet.
3. Dispatcher servlet passes the request to the corresponding controller based on URL mapping.
4. Controller performs the task and returns the model and view.
5. Dispatcher servlet maps the view name to the corresponding jsp (any view technology) using View Resolver.
6. View renders the model and displays it.

**Project Structure and Parts of Implementation**

Text

Description automatically generated

As it can be seen in the screenshot above, the project consists of multiple files, each of them with their own content for specific purposes. In this report, however only the most important files will be discussed in more details, namely: src/main/java, src/main/resources, src/test/java, JRE System Library, Maven Dependencies, and pom.xml. And still, even for these directories not all the packages will be presented in depth, but only the most important.

Text

Description automatically generated

In the screenshot above, we have src/main/java, where the backend development is made. In this particular case it is modularized into 8 packages, each of them handling different tasks.



Text

Description automatically generated

In this particular *LibraryApplication.java* class, it’s usually the *main(String[] args)* function in which the application runs and sometimes a **@Bean** annotation such *CommandLineRunner runner()* can be added and it runs as soon as the application starts as well. Such beans are usually used to run methods once the application runs or to hardcode tests such as creating and inserting new instances of a DAO class in the database and so on.

Text

Description automatically generated

Text

Description automatically generatedIn this screenshot, it is revealed the way the controllers are modularized. The role of the controllers is usually to handle specific request which is mapped by its request mapping.

Text

Description automatically generated

Above, we can see an example of a *HomeController* which handles the redirection of a user to a based-role home page depending on the role he/she was assigned (e.g.: admin, user, employee).

In this screenshot, it is revealed the way the reporistories or DAOs (Data Access Objects) are modularized. Their role is to provide access Text

Description automatically generatedto a particular data resource without coupling the resource’s API to the business logic.

Text

Description automatically generated

As it can be seen in the screenshot above, to make a repository for a particular database entity, it is enough to add @Repository above an interface and make it extend the JPARepository (Java Persistance Access Repository) interface. Apart from the general way a repository interface works, in this case it also extends JPASpecificationExecutor class of a generic type(in this case Book), which basically allows us to retrieve queries in a more efficient way, by providing “specifications”. Specifications will be discussed in further detail in the following part of the report, where services will be presented.

The example below also extends JpaSpecificationExecutor in order to perform different search filter.

Text

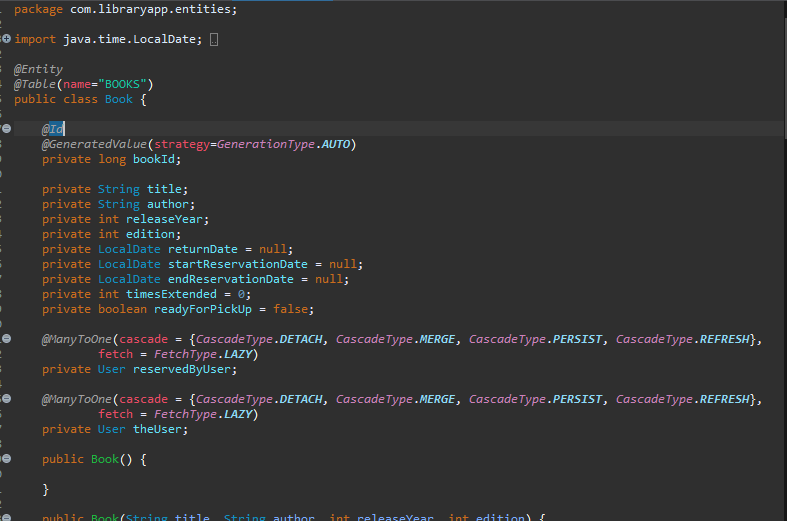
Description automatically generated

The following important package is the one in which all database entity classes are stored:

Text

Description automatically generated

The following screenshots show the class for Book Entity:







Text

Description automatically generated

Text

Description automatically generated

The following screenshots show the class for Notification Entity:

Text

Description automatically generated Text

Description automatically generated

And finally, the following screenshots illustrate the User Entity class:

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generatedIn this particular screenshot, it is revealed the package that contains the service layer of each entity the DAO package. This additional layer acts a communicator between controller and repository layer and contains business logic.

The following screenshots will reveal details about how each service class of each entity was developed thus far. Keep in mind that even though it may seem complete, I believe that some methods might need some improvements, others need to be removed and maybe replaced with completely methods, some need some code refactoring for better code reading, etc.

The following screenshots illustrate the services which handle books.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

A screenshot of a computer

Description automatically generated

Text

Description automatically generated

The last two methods illustrated in the last two screenshots handle the pagination after performing a search on books with different specifications such as all books starting with letters ‘c’, ’o’, ’d’ and ‘e’, the full name of an author or a part of the name of the author etc.

The following screenshots show the services of Notification entity.

Text

Description automatically generated

The following screenshots highlight the services of the User Entity:

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

The last two methods work the same way as mentioned in Book services.

In the Specifications package, there are Specifications classes in which the queries can be made in a custom way according to a regex rule (such as the string provided followed by anything that comes after that, which is also the rule by which the search is made by). The instances which play a role in making this possible are root, query and cb (criteria builder) which are used in a lambda expression, which is what is ultimately returned in a Specification method.

Text

Description automatically generated

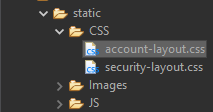
Text

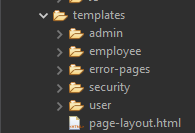
Description automatically generated In this particular directory it is found the implementation of the views, where also models come in play. Apart from that we can also find an *application.properties* configuration file

Text

Description automatically generated

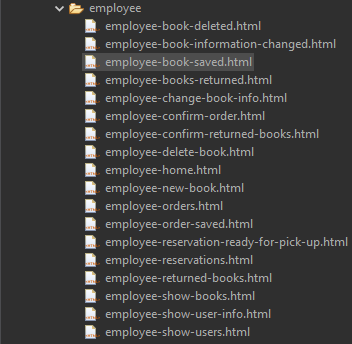
The screenshot above illustrates the way the database is configured such that the application can access its data.

 This particular screenshot shows the location of CSS files and Images used for the web design.

 Text

Description automatically generated Text

Description automatically generated



Text

Description automatically generated

The screenshots show a template for each database entity, each possible error page, for security and a general page-layout.



The screenshot above illustrates the files for where each maven dependency file is and which java plugins are used. The maven dependencies have a .jar extensions whereas JRE files don’t have a particular extension.

Text

Description automatically generated

Text

Description automatically generated

And finally at last, but not least the last most important file worth mentioning is pom.xml , where basically all maven dependencies can configured/added/removed, etc by simply copy-pasting their xml from <https://mvnrepository.com> and once any kind of such action is done, the Maven dependencies file is updated.

Example of how a dependency can be added:

A screenshot of a computer

Description automatically generated

Text

Description automatically generated

The following screenshots illustrate a part of how the pom.xml configuration looks like overall:

Text

Description automatically generated

Text

Description automatically generated

**Frontend implementation**

Here are some examples of the current user interfaces, because not all of them are 100% done or not presented yet due to other reasons (such as not making time to properly make all screenshots of all functionalities which are a lot) as I want them to be and still require some improvements:

**Login Form**

As the interface title suggests, this is a login in form through which users can either **log in** to the application or **register** if they don’t have an account already.

Graphical user interface, website

Description automatically generated

**Register Form**

As the title above suggests, here unregistered users can create an account by filling in the the following form with different private data (username, password, email, first name, last name, adrress, city, phone number).

Graphical user interface

Description automatically generated

**Home page of an employee user**

If a user logs in with an employee account, he will be recognized as an actual **employee**, and will thus be redirected to a home page in which **only** employees have access to. In this home page, the employee can mainly access the **catalog**, **users** (as **customers**), place order for **customers**, **return books** and **process reservations** or come back to the **home** page.

A screenshot of a computer

Description automatically generated

**Home page of an admin user**

If a user logs in with an employee account, he will be recognized as an actual **admin**, and will thus be redirected to a home page in which **only** **library managers** or **administrators** have access to. In this home page, the admin, for now, can mainly **manage accounts** or come back to **home** page.

Graphical user interface, text, application

Description automatically generated

**Manage Accounts**

As an admin, he can edit settings of a user account by either selecting it from the list of users or searching it either by first name or last name.

A screenshot of a computer

Description automatically generated

**Home page of a customer user**

If a user logs in with a regular registered account, he will be recognized as an actual **customer**, and will thus be redirected to a home page in which **only** **customers** have access to. In this home page, the customer can mainly **manage accounts** or come back to **home** page.

A screenshot of a computer

Description automatically generated

**Catalog**

In this section, the employee has multiple choices. He or she can :

* search for a book by **title** or **author**, choose **a number of registered books per page** from search results, sort the results by **title** or **author** or **sort the results in ascending or descending direction**.
* return all paginated books
* **add** a new book
* **change** book info
* **delete** a book

A screenshot of a computer

Description automatically generated

**Users**

In this section, the employee has multiple choices. He or she can :

* search for a user by **first name**, **last name**, **e-mail** or **phone number**, choose **a number of registered users per page** from search results, sort the results by fields previously mentioned or **sort the results in ascending or descending direction**.
* return all paginated users
* **show** user info

A screenshot of a computer

Description automatically generated

**Manage Account**

In this page, when an admin selects a user, the admin can either enable or disable an account (a disabled account cannot log in) or change the role of a user account.

Text

Description automatically generated

Last Remark: The only thing that wasn’t fully respected in this architecture was the fact that I didn’t manage to update the admin role to have the feature to manage books instead of the employee.