# Systems Integration - Assignment #2

João Moreira - 2015230374
João Soares - 2009113071

2019/2020

## 1 Introduction

The intent of this assignment is to develop a web application to manage an online store of secondhand items, called MyBay and deploy it using WildFly Application Server. In order to develop this application, we build a Maven architecture and used Java Enterprise Edition (Java EE), dividing the system into three layers: presentation, business and data.

## 2 Presentation layer

To develop the presentation layer we used JavaServer Faces (JSF) because it is a MVC (model, view, control) framework.

A none logged in user can only go to the landing page, in this case the login, and access the sign up page, giving him the ability to register in the system and latter login to see the contents of the web-page. This is because we've created a session filter that only gives access to a restricted folder to logged in users. The filter is defined on the web xml.

In the relation of the password storage we have done this program thinking in the security from the front-end, all the way down to the persistence. The actual password never gets transmitted to our application as it goes under a MD5 hash on the clients browser using a JavaScript program, then we pass the hashed password received on more MD5 hashing rounds, so the password saved on the database can't be used to login even if the database information would get exposed.

After logging in, the user is presented with the home page (figure 1). All of the pages, from this point forward, use a template that we created so that they can have the same header, giving access to the user to the home page, add item page, profile page and logout option anywhere on the system.

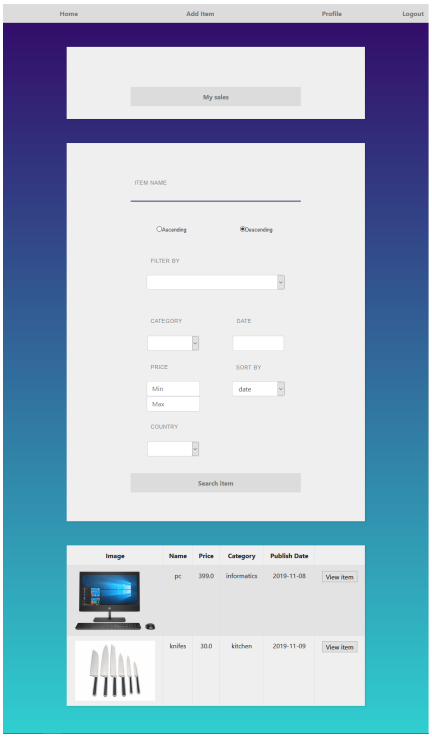


Figure 1: Home page layout after a search.

In the home page is where the user makes his searches. He's able to search for all items, by simply pressing the Search item button, he can see his own items, with the button my sales and can create various searches by using the input boxes, buttons and drop-down menus.

The add item section is the place where a new item can be created, and after that, by searching for his own sales, and clicking the view item button, the user can edit and delete his items.

In the profile area, the user can edit it's own name, country and password, as well as delete his account, deleting also his items for sale.

By logging out, the session is invalidated, so that the user needs to login again to see all of the pages above.

All of the information that the user inserts for searches, updates, etc. is treated in the controllers we've implemented to communicate with the business layer. This controllers call functions that are exposed by the business Enterprise JavaBeans (EJB)s interfaces and pass information through contracts that are known throughout the system.

One of the most challenging parts of the interface's implementation was the search part.

We wanted to make it obvious to the user how the categories to search by are selected, so we made a drop-down menu where the user selects the category he wants to search by. E.g.: I only want to see the items in the category wc. To do that, I need to click filter by, select categories and in the categories drop-down menu select the category wc. After this, I press Search item and will get all items in this category.

Another challenge was the item's image, resolved with an external API that lets us upload images and gives us the URL and the delete hash, for when the item is removed.

# 3 Business layer

The connection to the data layer is performed by the business layer, whom has three local stateless EJBs implemented. The account, sale and email. They are local because they are running in the same Java Virtual Machine (JVM) as the rest of the system and stateless since we don't need to maintain a conversational state, we only need to perform simple operations. The transactions for this beans are managed by the container, although we don't make any transactions on this layer, the transactions to the database are handled on the level of the data layer.

On the AccountEJB, most of the information that goes to the presentation layer is boolean, sense the operations mainly need confirmations on said layer, exceptionally on the login, where we send the User contract, created on the common package for transaction purposes, containing the information's about the user logged in (name, email, encrypted password and country).

The operations performed by this EJB are sign up, login, update account and delete account. This operations send to the data layer the User contract with only the information needed. In the sign up it sends a fully created user (name, email, password and country), in the case of the login just the email and password. For removing the account the email, and for updating it can bring a new name, country or password.

The SaleEJB, just like the AccountEJB, gathers booleans from the data layer, although, instead of User contracts it collects Item contracts to send to the presentation layer, and sends the mapped information to the presentation layer. The operations performed are create, list, update and delete sales, list an users items for sale and search sales.

To send the emails to the users with the three newest items, we have a third EJB called EmailEJB, a singleton that initiates upon deployment with a method called newsletter that is scheduled to run every Friday at 8AM. It gets the emails and the three newest items from the data layer (UserEJB and ItemEJB) and sends the emails with this new items through an API.

# 4 Data layer

The data layer works atop a database and exposes CRUD functionalities for the EJBs ItemEJB and UserEJB. Regarding transactions, we have left the management of all the transactions to the application container and on the data module specified the methods that require a new transaction to run (mainly all the crud operations) and by default all the EJB methods star an implicit transaction when called and end it upon returning from the method.

We've separated the entities from the persistence layer and the rest of the layers. This is why all the layers only know the user and item object classes that are not persistence entities, doing this we ensured isolation from the persistence entities from the rest of the application maintaining full control over them, performing all persistence operations in this layer.

To handle all the persistence we used hibernate 5.3 and javax.persistence.

We have created interfaces (contracts) to expose the EJBs methods and have put them in common modules with the name of the layer, this way the data layer exposes crud, search (to allow the reception of search parameters to refine searches) and list (to perform some list operations). The only layer that has the common data as a dependency is the business layer. This way only in the business layer the data EJBs are available. The same was done for the business / web interfaces.

The database interactions use two other entities to be performed, PersistenceUser and PersistenceItem. We can see how they interact in the following Entity-Relationship (ER) (figure 2).

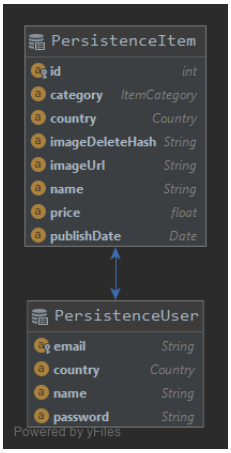


Figure 2: MyBay EJB ER diagram.

Both EJBs interface extends the crudable interface and the ItemEJB's extends Searchable, meaning they both implement create, read, update and delete, while user also implements a listing

of the user's items for sale and the item implementing a search method that has parameters as it's input.

In terms of the EJB security we have researched the use of role access base (but due to some problems trying to configure the users within WildFly with RBAC and the lack of time we couldn't persue this path)

# 5 Project management and packaging

All the cycle of (compiling packing and deployment) is done via Maven. The development system was a remote server running WildFly version 18 with java 13 under Ubuntu server 18.04lts. We've performed some configurations to the WildFly server in order to further customize the deployment and inserted the PostgreSQL driver in order to use a database (and configured a data source). Also, we've managed all our dependencies using WildFly. This way we have achieved a small deployment unit.

The project is divided into 7 folders common, data, common_data, business, common_business, web and ear. The common folder has support functions, such as enumerations, converters from and to enumerations and types, it's here where we have the Item and User class used to transfer data between layers. The common_data and common_business it where we store the interfaces of the EJBs of each layer. In the common_data we also have the Crudable and Searchable interface. The web, data and business store each of our layers. The ear folder has the pom file that creates the deployable file, packaged has an ear file.

For logging we used the tool Simple Logging Facade for Java (SLF4J) and created a logging profile to divide the logs by hierarchy and modules. We have a log-file common to all modules that registers all the information and above messages (this way we have a high level view of what's happening within our application). Then we have a separate logging file per module that records all the logs from trace and above, this way if we need to do some more deep investigation of what's happened within the system on a particular layer it's easier. The logging profile is operating in a rolling log files per day of operation.

Each of our folders has a pom fille. The root pom file has the modules of the project and the dependencies for the project, as well as the WildFly plugin configurations for the server that we're running WildFly on. The web pom has the URL where the system will be accessible from. The rest of the pom filles are used to define the packaging of the modules they're in, in this case both business and data are packaged has ejb, the web module is a war file and the common folders are jar files.

**Acronym list:**

**JVM** Java Virtual Machine

**Java EE** Java Enterprise Edition

**JSF** JavaServer Faces

**EJB** Enterprise JavaBeans

**JPA** Java Persistence API

**SLF4J** Simple Logging Facade for Java

**ER** Entity-Relationship

**RBAC** Role-Based Access Control