23 Library management with EJB and JPA

Bărcan Florin

Bîrsan Diana

Contents

[Theoretical aspects 3](#_Toc95851578)

[EJB 3](#_Toc95851579)

[JPA 3](#_Toc95851580)

[JSP 3](#_Toc95851581)

[Wildfly application container 3](#_Toc95851582)

[Project configuration 4](#_Toc95851583)

[Application architecture 5](#_Toc95851584)

[Database 6](#_Toc95851585)

[Authentication 7](#_Toc95851586)

[Login 7](#_Toc95851587)

[Register 8](#_Toc95851588)

[Dispatcher 9](#_Toc95851589)

[GET method 9](#_Toc95851590)

[POST method 9](#_Toc95851591)

[Deploying the application 10](#_Toc95851592)

[Personal contribution 11](#_Toc95851593)

[Bibliography 12](#_Toc95851594)

# Theoretical aspects

## EJB

**Enterprise Java Beans** or EJB are server-side software components that encapsulate the business logic of an application. An EJB web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services.

## JPA

**Java Persistence API** or JPA is a set of concepts that is concerned with persistence inside a Java application, which loosely means any mechanism by which Java objects outlive the application process that created them. Not all Java objects need to be persisted, but most applications persist key business objects. The JPA specification lets you define which objects should be persisted, and how those objects should be persisted in your Java applications.

## JSP

**Java Server Pages** or JSP is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. This tutorial will teach you how to use Java Server Pages to develop your web applications in simple and easy steps.

## Wildfly application container

**WildFly** is a flexible, lightweight, managed application runtime that helps you build amazing applications.

# Project configuration

First, we had to connect our database to the wildfly server.

For the database we have used mysql and apache.

We needed to define a JDBC driver, along with a Datasource for the Wildfly server, and include them in the persistence file of our project, under /resources/META-INF/persistence.xml. Later, we will also declare our entities there, in order to be able to use them.

The configuration for the **standalone.xml** file of the wildfly server looks like this:

<xa-datasource jndi-name="java:jboss/datasources/**MySqlDS**" pool-name="MySqlDS">

<xa-datasource-property name="ServerName">

localhost

</xa-datasource-property>

<xa-datasource-property name="databaseName">

la\_biblioteca

</xa-datasource-property>

<xa-datasource-property name="port">

3306

</xa-datasource-property>

<xa-datasource-class>com.mysql.cj.jdbc.MysqlXADataSource</xa-datasource-class>

**<driver>mysql</driver>**

<security>

<user-name>root</user-name>

</security>

<validation>

<valid-connection-checker class-name="org.jboss.jca.adapters.jdbc.extensions.mysql.MySQLValidConnectionChecker"/>

<validate-on-match>true</validate-on-match>

<exception-sorter class-name="org.jboss.jca.adapters.jdbc.extensions.mysql.MySQLExceptionSorter"/>

</validation>

</xa-datasource>

The persistence file looks like this:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<persistence xmlns="https://jakarta.ee/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence\_3\_0.xsd"

version="3.0">

<persistence-unit name="ejb">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

<jta-data-source>java:jboss/datasources/**MySqlDS**</jta-data-source>

<class>entities.User</class>

<class>entities.Carte</class>

<class>entities.Inchirieri</class>

<properties>

<property name="hibernate.dialect" value="org.hibernate.dialect.MariaDBDialect"/>

<property name="hibernate.show\_sql" value="true"/>

</properties>

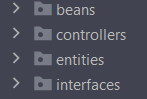
</persistence-unit>

</persistence>

# Application architecture

The backend side of our project is using EJB and JPA for managing 3 tables from a database.

This way, the backend side of the project is structured like this:

The **beans** folder is holding the beans logic, that follow their respective interfaces from the **interfaces** folder and entities are the base entities, declared accordingly to their respective table.

If we take the **user\_bib** table for example, we will have the following flow for entity->interface->bean

**ᐯ entities/User**

@Entity

@Table(name="user\_bib")

public class User implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int uID;

private String nume;

....

}

**ᐯ interfaces/IUser**

public interface IUser {  
 User add(User user);

...

}

**ᐯ beans/UserBean**

@Stateless

@Remote(IUser.class)

public class UserBean implements IUser {

@PersistenceContext(unitName = "ejb")

private EntityManager entityManager;

@Override

public User add(User user) {

entityManager.persist(user);

return user;

}

....

}

Next, we have the **controllers** package, which holds our Dispatcher object, that controls requests and responses and manages our JSP pages and the LoginController.

# Database

For this project we have used a mysql database, named **la\_biblioteca**, which contains three tables:

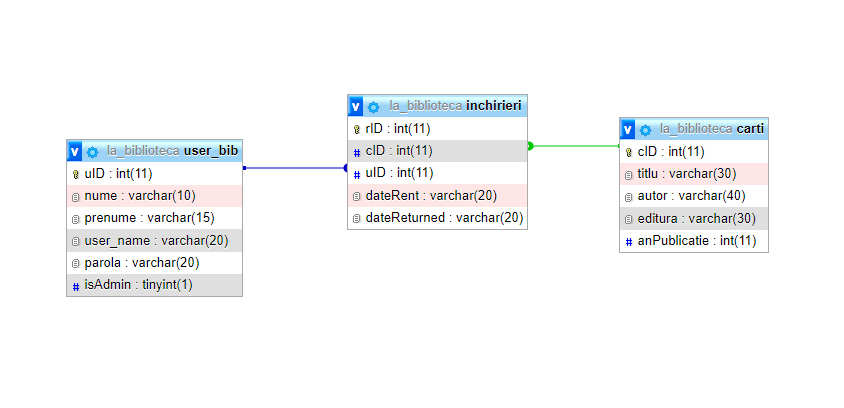
1. The **user\_bib** table represents the users of the application, which can be of two types
   1. Admins – they can manage the list of books by adding a new book, deleting or editing an existent book
   2. Normal users – they can rent or return a book; so they will be able to see a list of books that are available for renting, along with a list of books they have rented
      1. a user cannot rent a book they have not returned and cannot rent more than 5 books at a time

A user has multiple properties: **nume**, **prenume**, **user\_name** and **parola**, all strings, and a boolean field that represents the admin rights of the user.

1. The **carti** table is responsible for our books

A book has the following fields: **titlu**, **autor**, **editura**, all strings and an integer field, **anPublicatie**.

1. The **inchirieri** table represents the intermediary table for the many-to-many relationship a normal user can have with books. ( A user can rent more books, and a book can be rented by many users). This table holds information about the rentDate and returnedDate (strings), as long with the book and user’s id.

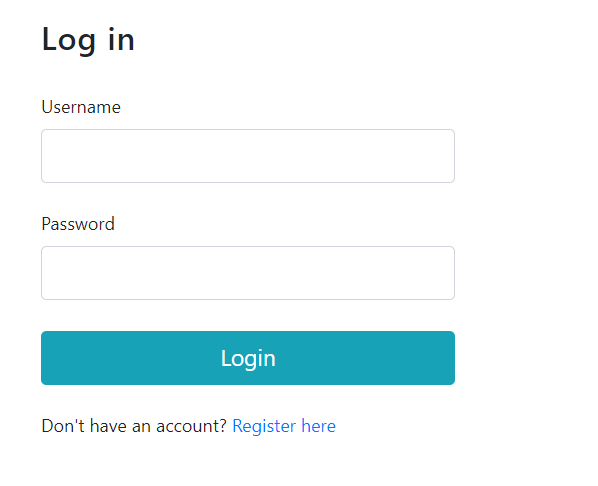


# Authentication

The login mechanism operates on its own HttpServlet, **ServletLogin** and is responsible for the login and register mechanism.

## Login

This servlet works using two main methods, get and post, both accessible on the ***/login*** path.

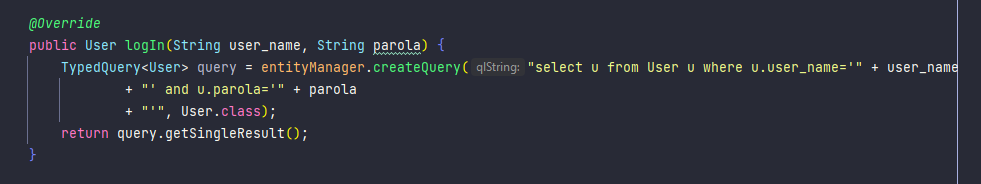
The **GET** method checks if the user is logged in and redirects the application to the dispatcher if so.

The **POST** method is the one that ”authenticates” the user. Because the servlet uses the user specific EJB object, in the POST method we check if there exists a user in the database with the username and password provided by the login form. If the answer is yes, we store the username, as well as a variable, **user\_logged**, stored in the session object of the servlet, then we redirect the user to the dispatcher.

In the same way, if the logout action is sent to the servlet from within the application, we remove the *user\_logged* variable from the session and redirect the user to the login page.

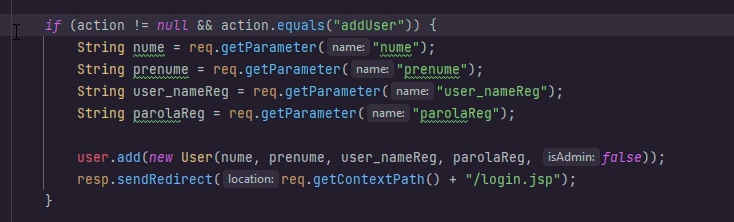


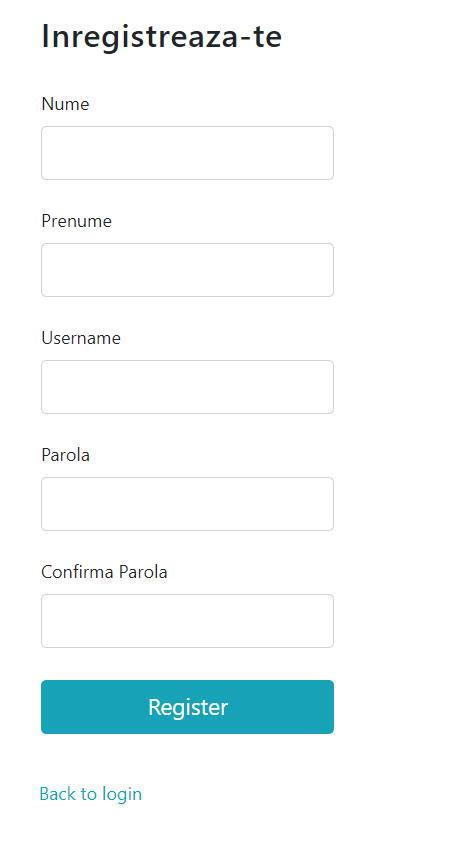
The logIn method within the User bean is the following:



## Register

The register logic also uses **ServletLogin**. Because we already have the User EJB object instantiated, we will use the add function, that persists a new user object in the database. When the register button is clicked (or when the action is equal to ”addUser”), we create a new user and redirect the application to the login page)





# Dispatcher

The navigation is made possible throughout the application with the use of a dispatcher, a HTTP servlet with two methods.

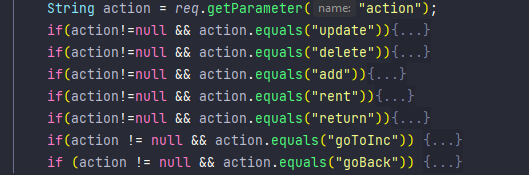
## GET method

The get method is responsible for checking the authentication state of the user and redirecting them to the login page if not authenticated, getting the lists of books and rents from the servlet session and redirecting them to their specific pages ( for a **normal user** that means the books list page, from which they can choose a book to rent, and for the **admin user** that means the page from which they can add/ modify or delete any book).

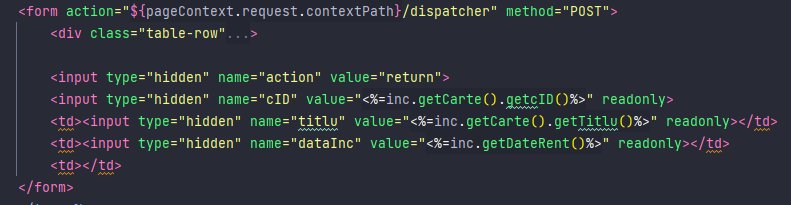


## POST method

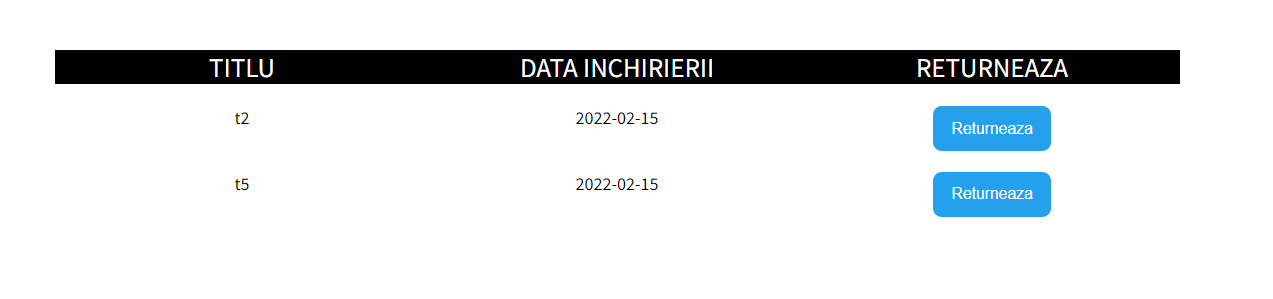
The POST method is responsible for the actions performed by any user with a POST method from the specific form tags of the JSP file. These actions can be modifying the books list, renting of returning a book, going to the list of rented books or going back into the application.



This action parameter, as long as any other important information we may need are sent to the servlet using html forms, in the jsp, like this:



This specific form is meant to manage the returning a book from the list of rented books.



# Deploying the application

We have chosen Wildfly as the container for deploying our application. After configuring both the project and the Jboss Server, the application can be deployed by being placed in the /standalone/deployments folder of the server, or by using the console provided by it.

It can be accessed in the browser at: <http://127.0.0.1:8080/wildfly-1.0-SNAPSHOT/dispatcher>

# Personal contribution

Diana – Styling, documentation and creation of corresponding entities

Florin – Login logic, java beans and interfaces logic, general validation

Both – Dispatcher, database creation, frontend JSP

# Bibliography

<https://docs.wildfly.org/>

<https://www.tutorialspoint.com/jsp/index.htm>

<https://www.tutorialspoint.com/ejb/index.htm>

<https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>

Lecture notes