THE SPRING APPLICATION
“NewVideoClubProject” - different story

1. INTRODUCTION

The starting point for the application “**NewVideoClubProject**” was previously designed project “**VideoKlubProjekat**” [21], and the bunch of technologies for further application improvements (according to the chapter 6. “Guidelines For Further Application Improvements” in **doc/VideoKlubProjekat.pdf**). With new technologies for application improvements we can consider this project to be a “different story” in comparison with the previous one.

This document contains a short description of programming technologies used in application design, application rules, configuration and design details.

2. LIST OF TECHNOLOGIES

“**NewVideoClubProject**” is a web application which demonstrates a usage of the following technologies:
- **Spring MVC framework** – business layer
- **Spring Security framework** – for logging and access activities
- **Spring WebSocket and messaging**– for async message-driven multi-user communication
- **Hibernate** (ORM)
- **MySQL** – RDBMS data layer
- **JSP** + **JSTL** + various **TAG** libraries: presentation (View) layer
- **Bootstrap** + **JQuery**: front-end framework (css + js)
- **jQuery-validate Plugin** - for input data validation
- AJAX in front-end via jQuery standard library AJAX support
- JSON data handling with **Jackson** library
- Back-end logging with **Slf4j-with-Logback** library

A standard Java servlet container (Tomcat) is used as a server platform.

Application is set-up to use Maven as its build system.

The code is Java8 – compliant.

3. APPLICATION'S FUNCTIONALITY

In comparison to the previous application [21], this one has an improved functionality and minor additions.

User authorization model uses the classic 'role' paradigm. There are 2 roles implemented:

a. Administrator role:
- Has a CRUD functionality on films on a single webpage (with the support of AJAX, JavaScript, Jquery, Bootstrap modals and Jquery-validation for input data)
- Rents available films
- Returns rented films
- Has a complete overview of users and their activities – names, rented films etc.
- CRUD functionality on users on a single webpage (with the similar design and the same technologies used for films)
- Has the statistics view of users and rented films
- CRUD on admins on a single webpage (with the similar design and the same technologies used for films)

b. User role:
- Can browse through the list of all the films
- Can rent the films that are available
- Has the complete view of his/her own rented films
- Can change the old password

NOTE:
- User rents films by application and after that he physically takes over the film.
- He cannot return film(s) by application. He returns films physically to administrator, then, administrator returns film(s) by application.

Application controls:
- The renting time (for example 7 days, but for demonstration purposes here it has been downsized to 10h)
- Max number of renting films per user (max 5 films per user at the same time).
- Login data (login control with the support of Spring Security framework and BCrypt password encoding feature)
- User page access (access control with the support of Spring Security framework)
- Interactive work admin/multi-user roles with information update in real-time (with the support of Spring WebSockets mechanism)
- Session timeout in another way (with HTTP Session Listener and Spring Security)
- Multiple concurrent logins prevention (with the support of Spring Security)

4. APPLICATION DESIGN

Application has several parts that are inter-connected:

- First step was to create a (MySQL) database "**video_club**", similar to the one that was used for the previous project.
- "**NewVideoClubProject**" was designed as a Maven Project in Eclipse IDE. The integration of Hibernate and Spring framework was performed; Spring Security and Spring WebSockets capabilities were introduced.
- For web pages views, jsp files were used (with HTML code); CSS resources and Bootstrap front-end framework was used alongside. Additional features such as JavaScript, Jquery, Ajax and JSON were added for js supporting files which improve front-end application design.

4.1. DATA STRUCTURE

The scheme of database "**video_club**" is given in the next figure:

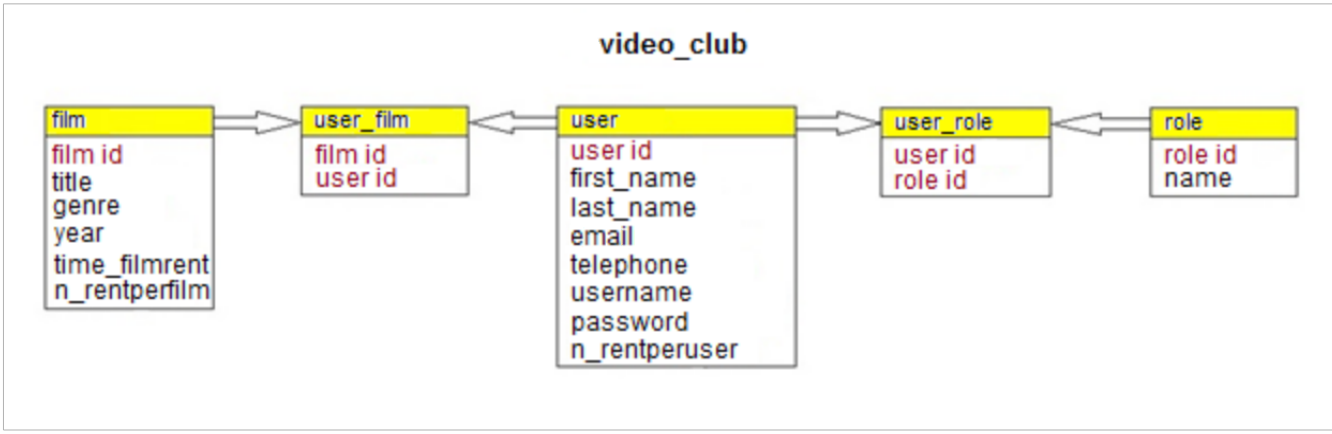

*Figure 1 - "**video_club**" database schema*

In comparison with the database "**dbvideo_club**" there are minor changes. For **user** table the **jmbg** column was replaced with **email** one, and the major change was made for the length of password datatype. The length was increased from 45 to 100 due to the fact that Bcrypt hashing algorithm generates a hash value of length 60.

SQL script file **`video_club.sql`** is given in folder **`sql`**.

1.For application start we need to populate the **`role`** table:

```
INSERT INTO 'video_club'. 'role' ('id', 'name') VALUES ('1', 'ADMIN')
INSERT INTO 'video_club'. 'role' ('id', 'name') VALUES ('2', 'USER')
```

Real applications should never store passwords in plain text format. Passwords should always be encoded using a secure hashing algorithm. Spring Security provides *BCrypt* password encoder and **`PasswordEncoder`** interface that uses *BCrypt* hashing function to encode the password.
If we want to store encoded password, first we should create the Maven skeleton project. Utility class **`QuickPasswordEncodingGenerator.java`** creates first encrypted administrator password.

2. After creation of administrator password, we populate data in table '**`user`**' for the admin:

```
INSERT INTO 'video_club'. 'user' ('id', 'username', 'password',
'first_name', 'last_name', 'email', 'telephone')
VALUES ('1', 'sneza', '***', 'Snežana', 'Snežić',
'snezasn@mycomp.com', '9343897');
```

Here, '**`***`**' - represents encrypted administrator password given from utility class **`QuickPasswordEncodingGenerator.java`**

3. In table '**`user_role`**' we insert id of admin and id of his role (ADMIN):

```
INSERT INTO 'video_club'. 'user_role' ('us_id', 'rol_id')
VALUES ('1', '1');
```

4.2. PROJECT CREATION AND CONFIGURATION FILES

This application was created and developed as a Maven Project in Eclipse IDE.

Skeleton application has the main package **`com.snezana.videoclub`** and **`pom.xml`** – a main unit of work in Maven.
`pom.xml` file should include dependencies for Spring, Spring Security, Spring WebSockets and messaging, JSON, Jackson, MySQL, Hibernate, Slf4j, Logback and Servlet.

Also, in the **`src/main/webapp`** the **`WEB-INF`** directory was created. In this directory we create **`context`** folder and **`web.xml`** file as an entry point for web application.

In **`context`** folder we create following xml configuration files:

- **`applicationContext.xml`** as context component with base package
- **`springmvc-dispatcher-servlet.xml`** with ViewResolver and resources
- **`spring-security.xml`** – configuration file for Spring Security
- **`spring-database.xml`** – configuration file for database

Additional definitions are placed in **`web.xml`** for Spring Security Filter, Character Set Filter for UTF-8 charset and HTTP Session.

Under the **Java Resources folder** (**src/main/resources**) we create **application.properties** file which provides property values for database connection. This file can be accessible to Spring through registering properties in **spring-database.xml.**

In **spring-security.xml** we define bean for Bcrypt encoding. In **util** package we create class **QuickPasswordEncodingGenerator.java** for encrypted password generation (4.1. chapter). The other class **CharacterSetFilter.java** is used for UTF-8 character set support.

4.3. OTHER DIRECTORIES AND FILES FOR SUPPORT

webapp/static/ folder contains subdirectories for front-end activities: **css, images,** and **js**.

In **WEB-INF** alongside with **context** we create **/tld/** subdirectory for tag library descriptor files and **/views/** subdirectory for **jsp** files.

As a logger solution we use ***slf4j*** and ***logback***. The most common way to configure logback is through **logback.xml** configuration file. We create this file in **src/main/resources/** directory. (You can read more about that in [13] - [16]).

5. DESIGN PARTS

The basic application was formed considering several approaches:
- Reducing the number of **jsp** pages and improving front-end capabilities (model [1]).
- Introducing ***Spring Security*** for login features and access control (the article series [2] -[6]).
- For better design of simultaneous work (admin/user and user/user) the Spring ***WebSocket*** support was introduced (the article series [7] -[11]).

5.1. SPRING SECURITY AND HTTP SESSION

We have additional definitions for Spring Security Filter and HTTP session in **web.xml** file.

In **spring-security.xml** file we define url access mapping with *intercept-url* parameter. Other definitions are also included. For authentication failure we have login error page. We also define **accessDenied** page, **csrf** protection and **multiConcLoginsExp** expired page for concurrent logins attempted by the same user. For session timeout detection we define **CustomFilter**.
We also define a bean for **customUserDetailsService** responsible for providing authentication details to Authentication Manager.

Finally, we define a bean **customSuccessHandler** for url determination where to redirect the user after login, based on the role of user.

According to this we create in **service** package **CustomUserDetailsService.java** class and in **configuration** package **CustomSuccessHandler.java** class.

For HTTP session tracking and session timeout detection we create in **configuration** package **SessionListener.java** class and **CustomFilter.java** class.

5.2. DAO

Package **dao** has DAO interfaces and implementation classes:

AbstractDao.java
LoginDao.java
LoginDaoImpl.java
AdminDao.java
AdminDaoImpl.java
UserDao.java
UserDaoImpl.java

Methods involved in DAO interface are commented in code. Also, in **spring-database.xml** we define beans for DAO implementation **adminDao** and **userDao**.

In **service** package there is **LoginService.java** service interface and its implementation **LoginServiceImpl.java** that handle login authentication and authorization.

5.3. CONTROLLERS

In comparison with the previous project, controller reorganisation for better performances was introduced. Login controller controls login activities. According to the activities based on the role, we have administrator controller and user controller. So, in package **controller** there are three controllers:

LoginController.java
AdminController.java
UserController.java

Every service method of each particular controller was commented in the code. **AdminController.java** has **@ResponseBody** annotation in some service methods. These methods are related to **allAdmins.jsp**, **allUsers.jsp** and **allFilms.jsp** pages.

In **model** package additional files are:

FilmData.java
UserData.java

These classes were used in **AdminController.java** for data transfer (**allUsers, allFilms, allAdmins jsp**).

5.4. SPRING WEBSOCKET FEATURE

For message-driven communication we use Spring WebSocket capabilities. The **WebSocket** protocol defines full-duplex, two-way communication between the client and server. It is an important capability for web applications that makes the web more interactive. *WebSocket* is used in web applications where the client and server need to exchange events at high frequency and with low latency.

In **model** package we create **EvtWSMessage.java** class as a message model. Messages as Java Objects will be exchanged between the clients and the server.

In **controller** package we create **EventWSHandler.java** class as a *WebSocket* server that extends **TextWebSocketHandler** class. In method **sendEventWS()** we perform serialization of Java Object into JSON string.

In **springmvc-dispatcher-servlet.xml** we define bean **eventWSHandler**, map *WebSocket* handler to a specific URL and enable SockJS functionality. SockJS lets application to use a WebSocket API, but also enables fallback options for browsers that don't support WebSocket. SockJS uses JSON formatted arrays for messages.

In **service** package there is a **EventWSService.java** class for service that handles *WebSocket* events.

The second part of Spring *WebSocket* functionality is related to front-end activities. The specific code written in JavaScript is put in jsp pages (chapter 5.6.)

5.5. TAG LIBRARY DESCRIPTOR FILE

In the previous project, time functions related to films were put in **Film.java** class. For clarity we can move time functions from Film.java to utility methods. For that reason we create in **util** directory file **FilmTimeUtil.java** and transfer time static methods from Film.java to it.

The second step is to create tag descriptor library file in **WEB-INF/tld** subdirectory - **timeFunctions.tld** tag librarie descriptor file which defines the configuration of the time utility functions for Film object.

Also in a **web.xml** file we create an entry for this tld file.

Finally, in jsp file, we can call these functions using jstl (chapter 5.6.). (The procedure of creation a custom function for JSTL is given in [12]).

5.6. JSP PAGES

Yes, JSP is used as front end in this project. I know, it's not so fancy these days, but it gets the job done, especially for a full-stack development.

Here's a list of pages:

General jsp pages:

- **`welcome.jsp`** – welcome application page
- **`login.jsp`** – entrance form with data checking
- **`header.jsp`** – basic header
- **`multiConcLoginsExp.jsp`** – expired page due to multiple concurrent logins being attempted by the same user

Admin pages:

- **`admin.jsp`** – the main admin page with links to the other admin pages
- **`allAdmins.jsp`** – list of all admins with add/edit/delete/info actions
- **`allFilms.jsp`** – list of all films with add/edit/delete/info actions
- **`allUsers.jsp`** – list of all users with add/edit/delete/info actions
- **`availableFilms.jsp`** – list of all available films that you can rent
- **`rentedFilms.jsp`** – list of rented films with return functionality
- **`rentNewFilm.jsp`** – on this page admin selects user who wants to rent new film
- **`statistics.jsp`** – statistics data
- **`usersWithFilms.jsp`** – list of users with their rented films
- **`usersWithoutFilms.jsp`** – list of users without rented films

User pages:

- **`home.jsp`** – the main user page with links to the other user pages
- **`userRentNewFilm.jsp`** – list of available films that he can rent
- **`changeUserPassw.jsp`** – the page for doing the old password change-
- **`rentedFilmsPerUser.jsp`** – review of films that the user has rented
- **`accessDenied.jsp`** – information about access attempt to admin pages

In **`css`** directory we create **`main.css`** for style definitions.

To reduce the number of pages, we implement CRUD actions on the same page. To achieve that approach the jsp files **`allAdmins.jsp`**, **`allFilms.jsp`** and **`allUsers.jsp`** have support in Bootstrap modals [20], in jQuery Validation Plugin [17] for input data validation and in js support files. Js files are in **`js`** directory - **`allAdmins.js, allFilms.js,`** and **`allUsers.js`**. These files have AJAX support for CRUD actions. AJAX allows web pages to be updated asynchronously, which means that it is possible to update parts of a web page, without reloading the whole page.

`changeUserPassw.jsp` has input data validation with *jquery-validate* plugin.

On the browser side, a client might connect using sockjs-client. **`availableFilms.jsp`**, **`rentedFilms.jsp`** and **`userRentNewFilm.jsp`** have the support of Spring *WebSocket* feature with **SockJS** functionality. This code is written in JavaScript. Also, data received from web server is always a string. Parsing the data with **`JSON.parse()`**, data becomes a JavaScript object that can be used in our page.

CSRF feature was used for preventing Cross-Site Request Forgery attacks [18]-[19].

6. IN SHORT...

In this document the development of Spring MVC application "**NewVideoClubProject**" in Eclipse IDE with Maven dependencies was presented. As a starting point we use the previous one project "**VideoKlubProjekat**" and additional features. Through the various steps the improvements were performed: login control with encrypted password, page access, the jsp pages reduction, controller reorganization and simultaneous work improved by Spring WebSocket feature (admin/user and user/user).

7. REFERENCES

[1] http://sindhitutorials.com/blog/spring-mvc-hibernate-eclipse-maven/

[2] http://websystique.com/spring-security-tutorial/

[3] http://websystique.com/spring-security/spring-security-4-hibernate-annotation-example/

[4] http://websystique.com/spring-security/spring-security-4-hibernate-role-based-login-example/

[5] http://websystique.com/spring-security/spring-security-4-password-encoder-bcrypt-example-with-hibernate/

[6] http://websystique.com/springmvc/spring-mvc-4-and-spring-security-4-integration-example/

[7] https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html

[8] https://www.sitepoint.com/implementing-spring-websocket-server-and-client/

[9] https://github.com/bbranquinho/wpattern-frameworks-spring-mvc-websocket

[10] https://sunitkatkar.blogspot.com/2014/01/spring-4-websockets-with-sockjs-stomp.html

[11] https://stackoverflow.com/questions/27158106/websocket-with-sockjs-spring-4-but-without-stomp

[12] http://findnerd.com/list/view/How-to-create-a-custom-Function-for-JSTL/2869/

[13] http://www.codingpedia.org/ama/how-to-log-in-spring-with-slf4j-and-logback/

[14] https://www.baeldung.com/logback

[15] https://examples.javacodegeeks.com/enterprise-java/maven/logback-maven-example/

[16] https://wiki.base22.com/btg/how-to-setup-slf4j-and-logback-in-a-web-app-fast-35488048.html

[17] https://jqueryvalidation.org/

[18] https://www.baeldung.com/spring-security-csrf

[19] https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html#csrf-include-csrf-token-ajax

[20] https://stackoverflow.com/questions/10626885/passing-data-to-a-bootstrap-modal

[21] https://github.com/petroneris/video-club-project

NOTE: Last access to the references: 10th January 2019.