2020

Critical Assessment of Enterprise Programming Coursework Project

MANCHESTER METROPOLITAN UNIVERSITY
ANTE TOMICIC

Table of Contents

Introduction

In this paper I will be critically assessing the coursework project for Enterprise Programming at Manchester Metropolitan University (MMU).

The primary goal of this project was for students to single-handedly create a web service using sound Software Engineering techniques and a wide variety of technologies and programming languages such as: MySQL, Java, XML, JSON, HTML, CSS, JavaScript, Ajax, jQuery, JAXB, JSP. Besides languages utilities such as: Eclipse EE IDE, MySQL Workbench and a web browser, mostly Google Chrome, were used. The secondary goal of the coursework was to evaluate the extent to which the students learned and applied all the techniques taught at the university and on the course.

The goal of this paper is to go through all the sections of the project and provide a critical view of the work done using examples and snippets of code alongside detailed commentary and definitions based on proven Software Engineering (SE) practices and techniques. This paper will try to discuss all the good programming techniques that were and weren’t used, and all the bad programming techniques that were and were not used. The first thing I will be covering is the front-end part of the project followed by the implementation of the database and then I will bring both of those parts together with a detailed discussion about the main logic of the project i.e. the back-end. After all that I will finish with a conclusion and will break down all the sections that were not completed.

Ajax based web front-end

In this section we will go through the front end of the web service (i.e. the project). That includes the HTML, CSS and JavaScript files used to get the data from the user and make a POST request to the server based on the Google Cloud Platform.

HTML and CSS

For the visual representation of the web service I used an HTML and CSS template from https://templated.co/, along with some JavaScript effects and functions, which was changed to fit the purpose of making the website more visually pleasing. Credits and license text files are included in the project as well as in the README text file in order to ensure copyright rules were met.

For the main functionality of the web service, regarding the front-end, jQuery User Interface demo templates were used and modified. That is an easy and a convenient way to make the website functional with less effort and in a much shorter period of time.

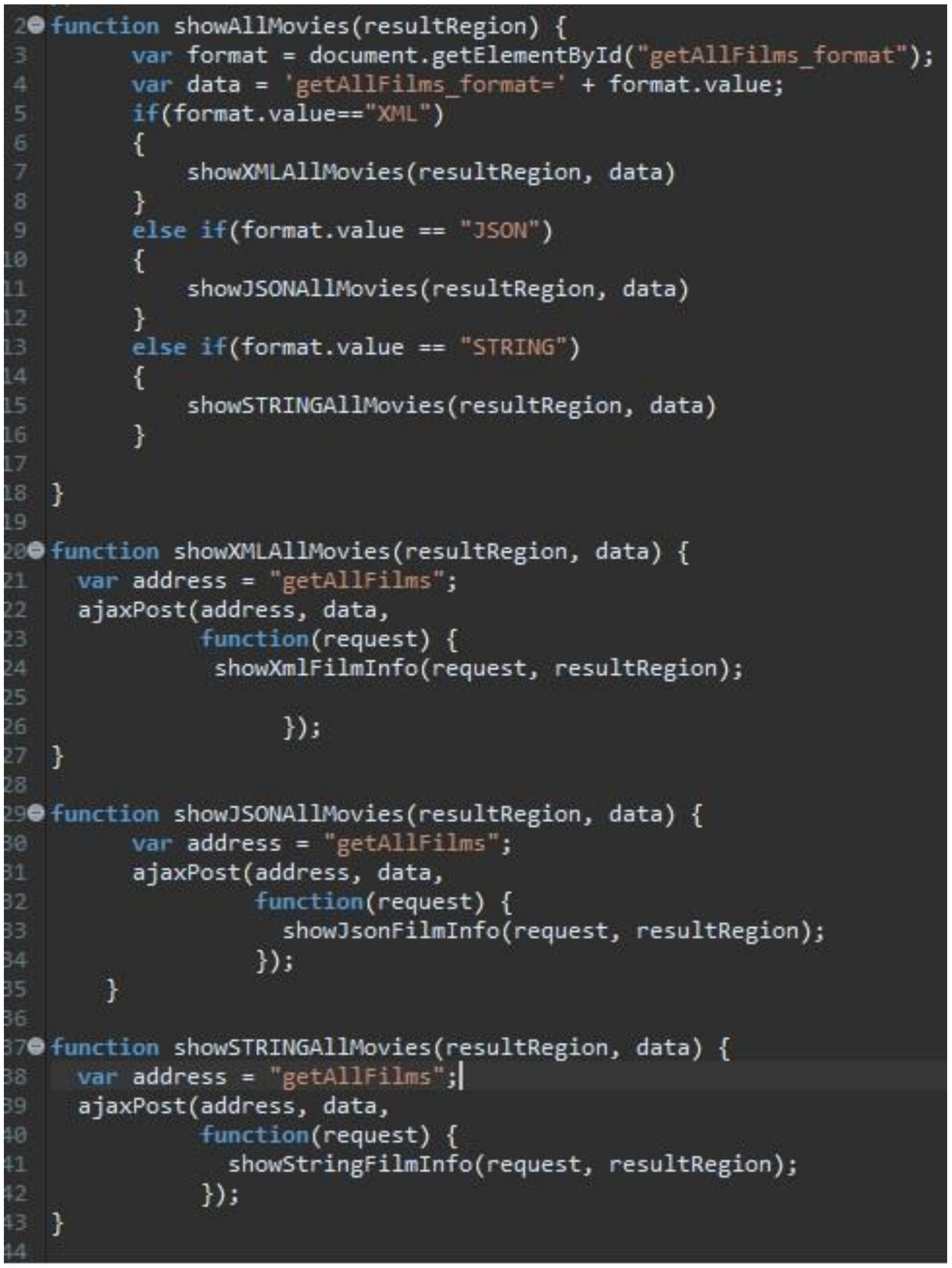
Other options, like making the whole website by myself should have been put into consideration, not only does that bring authenticity and uniqueness to the web service and provides training grounds for skill improvement in the area of Front-end Development, but it also can make for a good representation for future hiring opportunities and representation of past work.

Ajax and jQuery

The finished project does conform to the specification of the assignment and does use dynamic web requests and displays the content on the screen in a form of a table. Not only that, but all three formats: xml, json and string were implemented and are fully functional for each method and CRUD operation specified.

However, when looking at the .js (JavaScript) files, in otherwise well-structured and organized folder system in the project, it is easily visible that the JavaScript part of the assignment is heavily “hacked” and just made to work without any real sense of applied SE techniques. The code is really bunched up, badly organized into a few .js files containing Ajax functionality using jQuery libraries to make a request to the server and then receive and display the content. The code, on the other hand, should have been spread out over more .js files with each file having its own level of abstraction, instead of piling up all the functions with different levels of abstraction into two .js scripts. It makes the scripts very unorganized and would be considered a nightmare to maintain and refactor, the code is only made to work. The scripts lack in the areas of maintainability, usability, simplicity, dependability and mostly conformity to the relevant standards. Another principle that is completely ignored in the scripts is the redundancy of same functions, functions like “function showXMLAllMovies(resultRegion, data)”, “function showJSONAllMovies(resultRegion, data)” and “function showSTRINGAllMovies(resultRegion, data)”. Picture 1. shows a small snippet of code that describes the exact point that is being made in this paragraph, note that this shortcoming is repeated for each method and CRUD operation of the assignment.

A few good things about the scripts is that they hold true to the Single Responsibility Principle (SRP), meaning each function does one thing and one thing only, the scripts also have good indentation and naming convections and are, in themselves, rather simple and concise.



```
function showAllMovies(resultRegion) {
      var format = document.getElementById("getAllFilms_format");
      var data = 'getAllFilms_format=' + format.value;
      if(format.value=="XML")
      {
          showXMLAllMovies(resultRegion, data)
      }
      else if(format.value == "JSON")
      {
          showJSONAllMovies(resultRegion, data)
      }
      else if(format.value == "STRING")
      {
          showSTRINGAllMovies(resultRegion, data)
      }

}

function showXMLAllMovies(resultRegion, data) {
  var address = "getAllFilms";
  ajaxPost(address, data,
           function(request) {
            showXmlFilmInfo(request, resultRegion);

                 });
}

function showJSONAllMovies(resultRegion, data) {
      var address = "getAllFilms";
      ajaxPost(address, data,
               function(request) {
                 showJsonFilmInfo(request, resultRegion);
               });
    }

function showSTRINGAllMovies(resultRegion, data) {
  var address = "getAllFilms";
  ajaxPost(address, data,
           function(request) {
             showStringFilmInfo(request, resultRegion);
           });
}
```

Picture 1. Redundancy issue

Database implementation

Data base is a simple part of the assignment; however, the implementation wasn’t fully completed. Since the web service was supposed to be cloud based, the database doesn’t meet the requirements because it is not stored on the cloud. The database used in the assignment is the database stored at MMUs local database servers. That means that the web service is not fully cloud based and therefore can face mayor obstacles and problems in the future.

The main benefit of a cloud based system is its scalability, real enterprise web services must be scalable because of traffic requirements. In practice, this way of implementation would not be sufficient for professional deployment since it would most likely crash because of the traffic demands and would most likely be very slow and inefficient. Besides scalability, flexibility is one of the main concerns for this system and making the service completely cloud based should be the first upgrade to make when considering improving the finished product.

The only good thing is that the database works as it is supposed to, it is MySQL based which is acceptable even though there are more modern data storage solutions, and is convenient for development and testing.

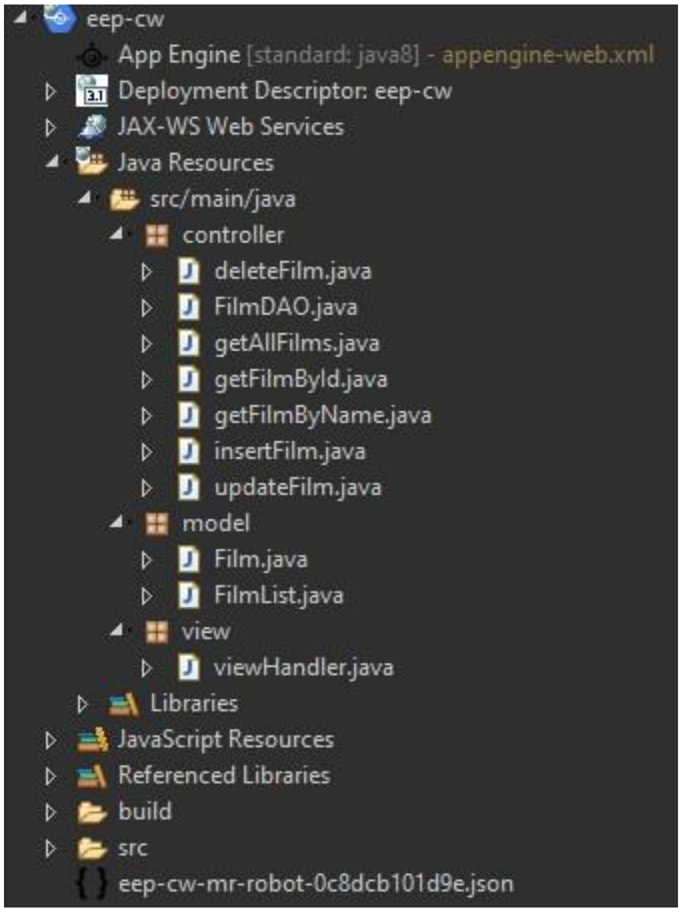
Back-end

In this section I will go through all the back-end implementation i.e. the main logic where most of the heavy lifting and work is usually done. The back-end uses Java as its main language for all the operations and it was developed in Eclipse EE IDE which is suitable for this kind of development. Besides Java, technologies like xml, json, jsp, jaxb were used in order to format and send data back to the client-side from the server in a form of a http response.

The first thing to go over is the principle of the highest level of abstraction which is a design pattern called Model-View-Controller (MVC).

Model-View-Controller

The MVC design patter was correctly applied on this project, it is clearly visible that the structure of the folder system and packages is successfully completed, as show in Picture 2.



```
eep-cw
    App Engine [standard: java8] - appengine-web.xml
  Deployment Descriptor: eep-cw
  JAX-WS Web Services
  Java Resources
    src/main/java
      controller
        deleteFilm.java
        FilmDAO.java
        getAllFilms.java
        getFilmById.java
        getFilmByName.java
        insertFilm.java
        updateFilm.java
      model
        Film.java
        FilmList.java
      view
        viewHandler.java
    Libraries
  JavaScript Resources
  Referenced Libraries
  build
  src
  eep-cw-mr-robot-0c8dcb101d9e.json
```

Picture 2. MVC package structure

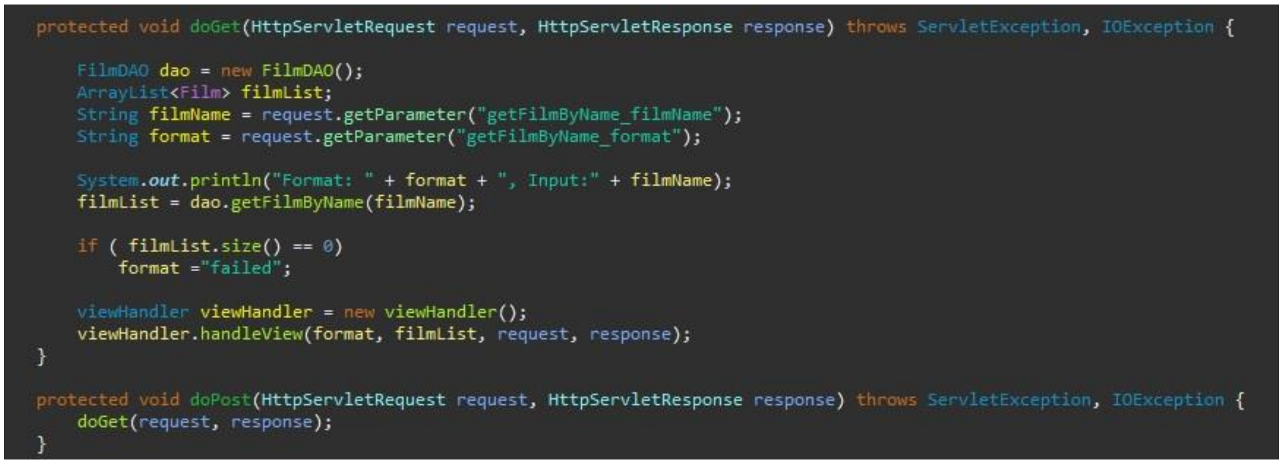
The MVC design has one fundamental principle, and that is that the controller is the only package that interacts with the model and the view, view can not interact with the model directly and vice versa. Within the controller package there is only one class inside FilmDAO.java file (DAO stands for Data Access Object) which can access the model and is used to interact with the database, and it uses the model Film.java to successfully achieve that. One downside the DAO has is that the database connections aren’t using Connection Pooling but rather open and close a connection for each communication interaction to the database. On the other hand, there is the package called view which has a viewHandler.java class which takes care of formatting and forwarding the data to the client and is called upon by each servlet.

That reinforces the redundancy principle since the code for the View is now written only once and is invoked by the servlets, which would otherwise each have a copy of the same code that does the same thing. That would contribute negatively to the simplicity, readability and especially maintainability of the code in the project.

Controller

From all the servlet names it is easily visible that good constant naming principles were applied and that is also visible from the servlets code itself. Names are clear and concise, they tell exactly what each servlet does and are symmetrical, meaning for an “Insert” and “Update” functions there is a “Delete” function. Another principle that these servlets follow is the SRP principle, each servlet in the assignment does only one thing as it should. This decreases complexity and increases maintainability, simplicity and usability.

Besides all mentioned above, good indentation practices were applied along with great grouping of code and also good separation of code from the data. The code is well written, all the main methods are defined within the FilmDAO.java class and therefore makes the servlets much easier to program and clean. All that makes the code readable and well put together, easy to maintain and refactor if needed later on. Comments aren’t present in the code, which is a good thing since they are not needed because the code is self-explanatory. Picture 3. serves as an example of all the positives I have stated for this section. Also visible from the example is that the doPost() method is redirected to doGet() meaning it is overloaded, which is a positive technique. There is no deep nesting of the if statements, no obvious and unnecessary comments, line length is well limited and the methods aren’t long but rather short and easily understandable.



```java
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

    FilmDAO dao = new FilmDAO();
    ArrayList<Film> filmList;
    String filmName = request.getParameter("getFilmByName_filmName");
    String format = request.getParameter("getFilmByName_format");

    System.out.println("Format: " + format + ", Input:" + filmName);
    filmList = dao.getFilmByName(filmName);

    if ( filmList.size() == 0)
        format ="failed";

    viewHandler viewHandler = new viewHandler();
    viewHandler.handleView(format, filmList, request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    doGet(request, response);
}
```

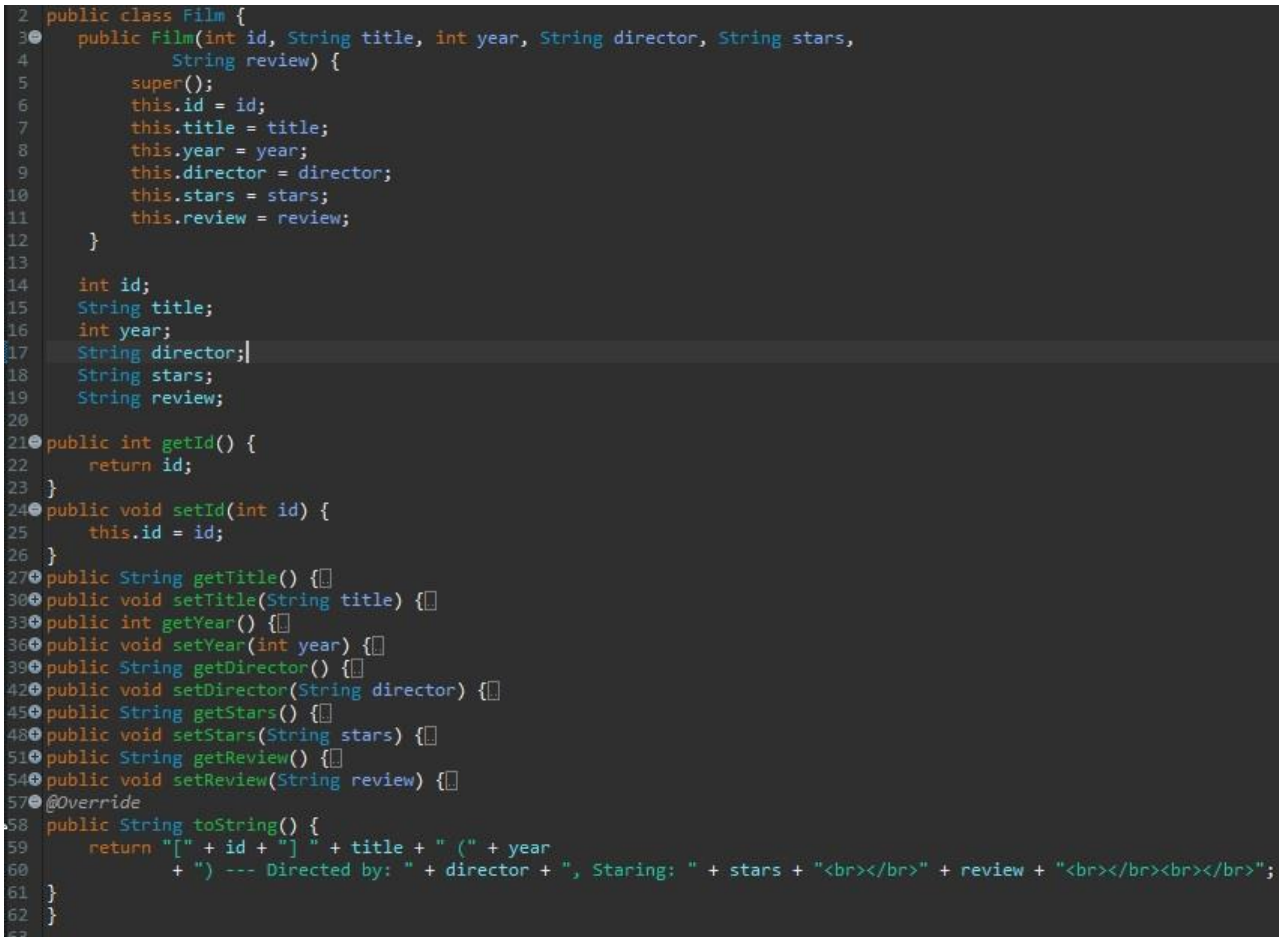
Picture 3. Code from getFilmByName.java

Other design patterns that could have been used was the Singleton principle for making a class responsible for its own instantiation using a hidden or blank constructor. Another SE method that wasn’t used was the Factory method pattern for dealing with the problem of creating objects without having to specify the exact class of the object. Also, Data Transfer Object (DTO) method was not used and could have been a great addition to the project because it takes care of carrying data between the processes on the client side and the server side. Lastly, not very important but minor is the capitalization of SQL statements in the FilmDAO.java file is not done.

Model

The model is defined by a java file called Film.java which contains a Film class which is responsible for mirroring and storing the data from the database and therefore enabling the controller to manipulate the data and easily use it.

The model is, beside missing the application of Singleton principle, written well. All the variables have properly defined and written getters, setters and toString() methods as you can see in the Picture 4.



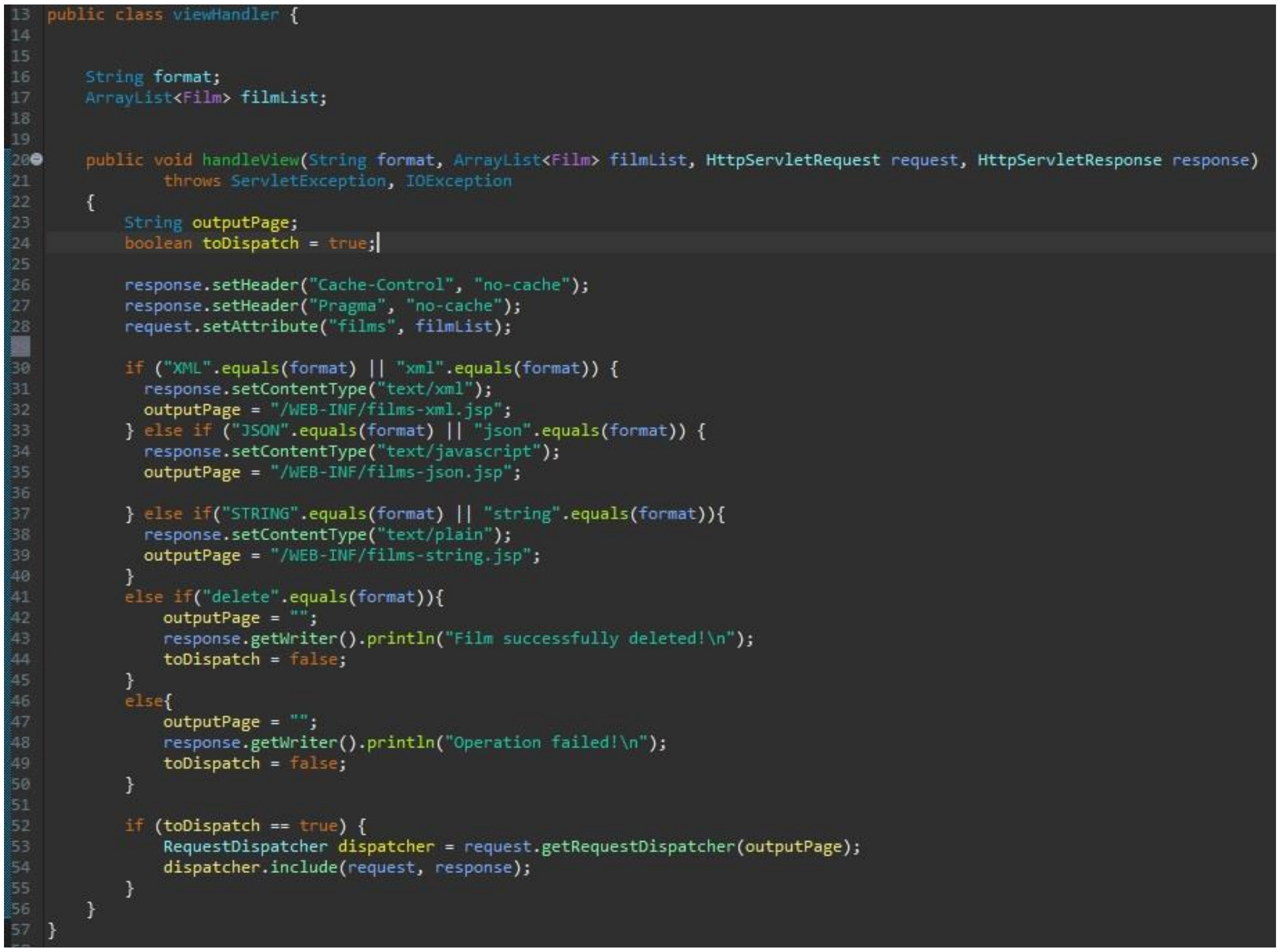
```java
public class Film {
    public Film(int id, String title, int year, String director, String stars,
             String review) {
        super();
        this.id = id;
        this.title = title;
        this.year = year;
        this.director = director;
        this.stars = stars;
        this.review = review;
     }

    int id;
    String title;
    int year;
    String director;
    String stars;
    String review;

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getTitle() {
public void setTitle(String title) {
public int getYear() {
public void setYear(int year) {
public String getDirector() {
public void setDirector(String director) {
public String getStars() {
public void setStars(String stars) {
public String getReview() {
public void setReview(String review) {
@Override
public String toString() {
    return "[" + id + "] " + title + " (" + year
            + ") --- Directed by: " + director + ", Staring: " + stars + "<br></br>" + review + "<br></br><br></br>";
}
}
```

Picture 4. Film.java

View

The view is defined by a java file called viewHander.java which contains a viewHandler class which is responsible for receiving and forwarding the output data to an appropriate .jsp file. Those .jsp files are well written and preform their function well.

However, as it is visible from Picture 5., it also lacks the implementation of the Singleton Principle.

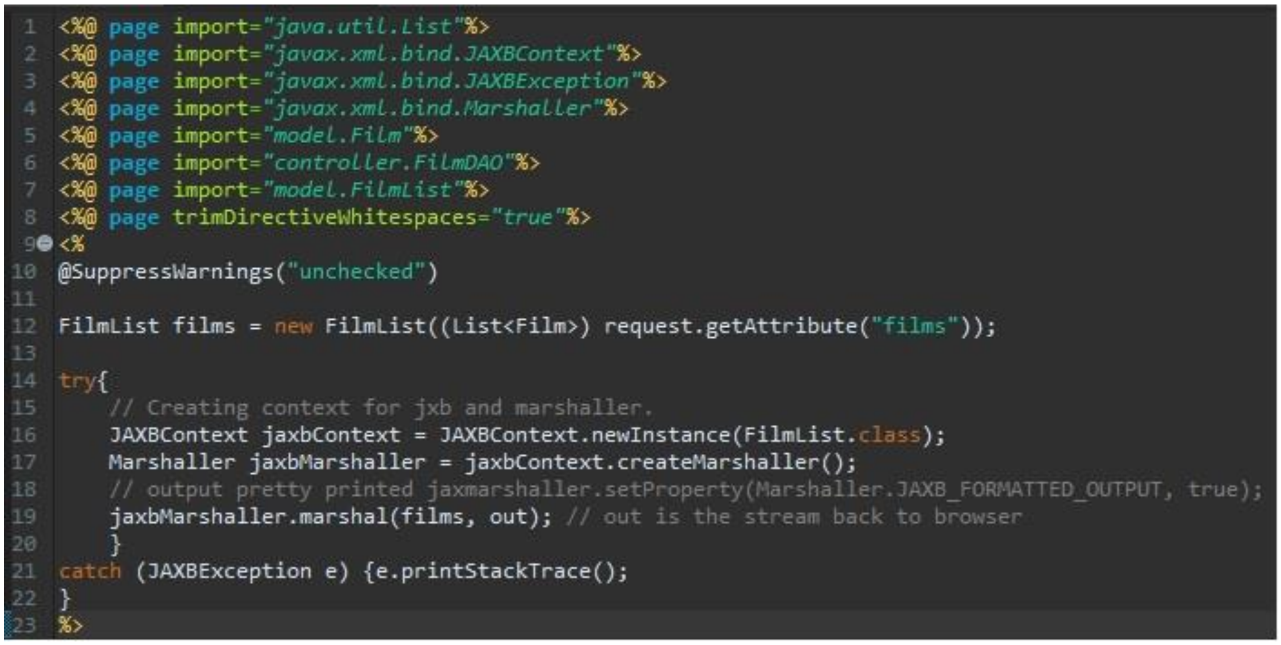


```java
public class viewHandler {


    String format;
    ArrayList<Film> filmList;


    public void handleView(String format, ArrayList<Film> filmList, HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        String outputPage;
        boolean toDispatch = true;

        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma", "no-cache");
        request.setAttribute("films", filmList);

        if ("XML".equals(format) || "xml".equals(format)) {
          response.setContentType("text/xml");
          outputPage = "/WEB-INF/films-xml.jsp";
        } else if ("JSON".equals(format) || "json".equals(format)) {
          response.setContentType("text/javascript");
          outputPage = "/WEB-INF/films-json.jsp";

        } else if("STRING".equals(format) || "string".equals(format)){
          response.setContentType("text/plain");
          outputPage = "/WEB-INF/films-string.jsp";
        }
        else if("delete".equals(format)){
            outputPage = "";
            response.getWriter().println("Film successfully deleted!\n");
            toDispatch = false;
        }
        else{
            outputPage = "";
            response.getWriter().println("Operation failed!\n");
            toDispatch = false;
        }

        if (toDispatch == true) {
            RequestDispatcher dispatcher = request.getRequestDispatcher(outputPage);
            dispatcher.include(request, response);
        }
    }
}
```

Picture 5. viewHandler().java

The formatting was done using .jsp files as a formatting tool which would use JAXB and Java snippets along with XML and JSON formatting tools. The code is short, concise and fairly commented in order to describe JAXB syntax since it is highly abstract.

An example of one of the formatting .jsp files is provided in Picture 6, to be exact the file with code that formats a list of objects Film into an xml structure and sends it back to the client side using the variable out, which is the stream back to the browser.

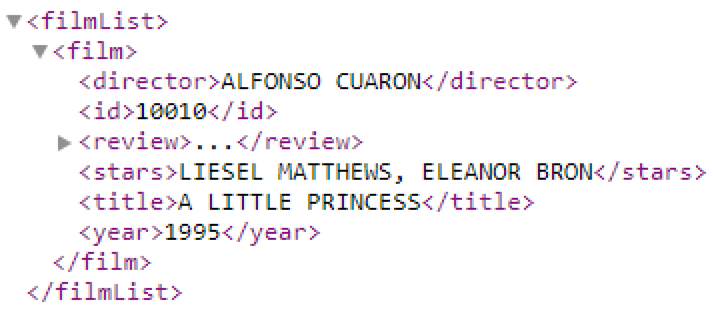


```
<%@ page import="java.util.List"%>
<%@ page import="javax.xml.bind.JAXBContext"%>
<%@ page import="javax.xml.bind.JAXBException"%>
<%@ page import="javax.xml.bind.Marshaller"%>
<%@ page import="model.Film"%>
<%@ page import="controller.FilmDAO"%>
<%@ page import="model.FilmList"%>
<%@ page trimDirectiveWhitespaces="true"%>
<%
@SuppressWarnings("unchecked")

FilmList films = new FilmList((List<Film>) request.getAttribute("films"));

try{
    // Creating context for jxb and marshaller.
    JAXBContext jaxbContext = JAXBContext.newInstance(FilmList.class);
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    // output pretty printed jaxmarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    jaxbMarshaller.marshal(films, out); // out is the stream back to browser
    }
catch (JAXBException e) {e.printStackTrace();
}
%>
```

Picture 6. films-xml.jsp

Application Programming Interface

The Application Programming Interface (API) was quite simply introduced to the system, without any special libraries, but using the FilmDAO.java file from the controller part of the MVC design pattern, third parties are able to access the database and all the methods that the web service provides. Picture 7. shows an example of that, web browser URL field was used to forward the data to the server by making a request and then receiving the data in a chosen format, whether XML, JSON or string. The request is as follows: “https://eep-cw-mr-robot.appspot.com/getFilmById?getFilmById_format=xml&getFilmById_Id=10010” and the result is displayed in Picture 7. For finding a film by Identification number (ID) of 10010 and the resulting format to be set to XML.



```
▼<filmList>
  ▼<film>
     <director>ALFONSO CUARON</director>
     <id>10010</id>
   ▶<review>...</review>
     <stars>LIESEL MATTHEWS, ELEANOR BRON</stars>
     <title>A LITTLE PRINCESS</title>
     <year>1995</year>
   </film>
 </filmList>
```

Picture 7. XML formatted output of getFilmById() method

Cloud implementation

The cloud implementation part wasn't done to the full extent. The application itself, the project, the web service is available on the cloud, however the database is not. The database is stored at the MMUs database server; therefore, the project doesn't meet the full criteria for this part of the assignment.

Uncompleted tasks

Uncompleted tasks of the assignment were:

- Generating a WSDL description
- REST implementation
- Storing the database on the cloud

Conclusion

The task of the assignment was completed however not in full, there is a lot of room for improvement especially on the cloud database side and the client side. Uncompleted task should have been done in order to achieve the highest possible mark. Besides uncompleted sections, the code on the back-end, meaning Java, was written fairly nicely; easy to read, easy to maintain and refactor. The patterns used, like MVC give a good structure to the project, along with nicely distributed files and a clean folder system.

With all that said, the assignment or the end product is satisfactory, but it is far from perfect or even well done, there is a lot more room to improve upon this web service and a lot more opportunity to learn.