1/8/2021

Developing, Deploying, Publishing and Finding your own Webservice

Critical Analysis of Code

Nathan Ainsley
18028669

Contents

Software Engineering Techniques

Waterfall

When creating my http interface which took up the bulk of the development time for this assignment, I used the waterfall approach to manage my workflow. I set out a list of things that I needed to achieve and the order that I would go through them. Starting with the background work and working up towards the working solution. The reason I went with a waterfall approach to creating this project is because a lot of the modules required others to work and so it made logical sense to start with base classes and work up to the servlets and JavaScript. I started off by creating the Film object and DAO classes and then started work on the servlets. Once the servlets were working and doing as they were expected too, I was then able to start work on the front-end design in HTML and JavaScript. Following this order of work and using a waterfall mentality if I needed to go back and change something the parts following it would also have to be changed as well, this meant that I was more careful and made sure that the modules worked before starting the next one. If I had instead used an agile approach I would have been going back and forth between modules in the assignment trying to get everything working at once and this would have led to it being harder to identify the root cause of an issue. Overall, I believe using a waterfall approach to the entire assignment not just this individual project was the best course of action as having to juggle everything with agile would have been overwhelming.

Design Patterns

Singleton

During the creation of the DAO I decided to utilise the Singleton design pattern to reduce the resources needed by my application. This was done by creating a connection pool so only 1 instance of the DAO would ever exist at once. After the first time the DAO object has been created, every subsequent call to the function will return the same object. This reduces the amount of recourses needed as there will only ever be 1 DAO object and not 1 for every time the database needs to be accessed. As seen bellow in Figure 1 Shows the Singleton Object There is a private constructor so that only this class can call it. The Getter for the object is a public static synchronized method which means that it is available to other classes. The synchronized keyword in the declaration ensures that if multiple classes on different threads then only 1 of them will receive the object and the rest wont. This issue if not tackled would completely remove the point of having using the singleton dp as two classes could, if called at the same time, retrieve a different DAO object each.

```
12      private FilmDAOHib() {}
13      public static synchronized FilmDAOHib getFilmDAOHibobject() {
14          if (FilmDAOHibobject == null) {
15              FilmDAOHibobject = new FilmDAOHib();
16          }
17          return FilmDAOHibobject;
18      }
```

Figure 1 Shows the Singleton Object

Within the Class of the servlets that need to call this object instead of setting a new object as you normally would I am instead now calling this getFilmDAOHibobject() method as it will return the

object and not create a new one as you normally would do. This can be seen bellow in Figure 2 Calling the Singleton.

```
30          //creates new FilmDAO object using the singleton DP
31          FilmDAOHib Hib = FilmDAOHib.getFilmDAOHibobject();
```

Figure 2 Calling the Singleton

The use of singleton dp was a good idea because if I hadn't used it and this project was deployed to the cloud and used by a lot of people then there would be exponentially more amounts of DAO objects created to the point where the resources needed by the application could lead to unwanted charges and cost a lot of money for the additional resources needed.

DAO

My method of connecting to the database was via a Data Accessing Object. This object was created using the Singleton DP as mentioned before to ensure that only a single instance of the object would exist at any 1 time. This DAO class contained all the methods that directly accessed the database. This meant that I would only have to have this single class connecting to the database and all other classes that needed access to the database would use the DAO to do so. This allowed my code to be more modular and the addition of classes easier as instead of adding multiple lines to connect to and query the database I could just call the DAO. I initially had my own method to connect to the database however I changed this to use the Hibernate framework towards the end of the project. This allowed me to use the Film object to insert films directory into the database and generally interface with the database easier. This can be seen bellow in Figure 3 DAO add film on line 183 where a new film object is created using the parameters passed to the method. That film object is then saved to the database in line 184. This is a lot cleaner than writing a long Sql statement as using hibernate it already knows what fields the values in the object go into.

```
170     /* Method to CREATE a film in the database */
171     public Integer addFilm(int id, String title , int year, String director, String stars, String review){
172         try {
173             factory = new AnnotationConfiguration().configure().addAnnotatedClass(Film.class).buildSessionFactory();
174         }catch(Throwable ex) {
175             System.err.println("Failed to create sessionFactory object. " + ex);
176             throw new ExceptionInInitializerError(ex);
177         }
178         Session session = factory.openSession();
179         Transaction tx = null;
180         int success=0;
181         try{
182             tx = session.beginTransaction();
183             Film newfilm = new Film(id, title, year, director, stars, review);
184             session.saveOrUpdate(newfilm);
185             tx.commit();
186             success = 1;
187         }catch (HibernateException e) {
188             if (tx!=null) tx.rollback();
189             e.printStackTrace();
190         }finally {
191             session.close();
192         }
193         return success;
194     }
```

Figure 3 DAO add film

The old method of adding a film is still used in the rest service to demonstrate the alternative method without using hibernate. This can be seen bellow in Figure 4 Rest DAO InsertFilm, you can

see that a string is generated that is very long and messy that is then ran against the Sql server. This approach works but is very messy compared to using hibernate despite being less overall code.

```
86  public int insertFilm(int id, String title , int year, String director, String stars, String Review) {
87      int success=0;
88      openConnection();
89
90      try{
91          String selectSQL = ("Insert into films " + "Values ("+id+", '"+title+"', "+year+", '"+director+"', '"+stars+"', '"+Review+"');");
92          stmt.executeUpdate(selectSQL);
93          stmt.close();
94          closeConnection();
95          success = 1;
96       } catch(SQLException se) { System.out.println(se); }
97
98      return success;
99  }
```

Figure 4 Rest DAO InsertFilm

The DAO contains methods for all the search modes my application can provide as well as adding a film to the database, updating a film in the database and deleting a film from the database. The methods such as adding, updating and deleting return either a 1 or a 0 after the operation is complete, This binary value indicates if the operation was successful with the return of a 0 being that the operation failed and 1 being that it succeeded. This is useful for debugging and providing error messages to the user. This can be seen bellow in Figure 5 DAO Delete Method where on line 203 an integer success is set as 0. If the try clause is successful, then success is then set to 1 as the operation has been successful. At the end of the method the integer success is then returned. Only if the operation was successful is the integer returned as 1 or true

```
195     /* Method to Delete a film in the database */
196     public int deleteFilm(Integer FilmID){
197         try {
198             factory = new AnnotationConfiguration().configure().addAnnotatedClass(Film.class).buildSessionFactory();
199         }catch(Throwable ex) {
200             System.err.println("Failed to create sessionFactory object. " + ex);
201             throw new ExceptionInInitializerError(ex);
202         }
203         int success=0;
204         Session session = factory.openSession();
205         Transaction tx = null;
206         try{
207             tx = session.beginTransaction();
208             Film delFilm = (Film)session.get(Film.class, FilmID);
209             session.delete(delFilm);
210             tx.commit();
211             success=1;
212         }catch (HibernateException e) {
213             if (tx!=null) tx.rollback();
214             e.printStackTrace();
215         }finally {
216             session.close();
217         }
218         return success;
219     }
```

Figure 5 DAO Delete Method

The add Film Servlet before adding a film will search to see if there are any films in the database with the ID that the user want to use. If that search finds something, then the operation will fail, and the user will be told to pick a different ID. This is because the ID is a unique identifier for the film as multiple films could have different titles or directors. This prevents the case of multiple films having the same ID in the database. If the Search doesn't return a value, then the DAO is then called, and the record is added to the database.

Model View Controller

The servlet side of my code, the java code, is done with the MVC principle in mind. This is reflected with the 3 packages my code contains being; Model, Controller and View. The Model package contains the java files for the film object, the DAO, and the Film Array object which was used for xml generation. The next package, controller, contained all the servlets that controlled the application. These servlets served the user by accessing the model package, this could be done by requesting all the films from the database via the DAO. Then the controller had acquired data from the model it then needed to be represented to the user, this was the job of the view package with the Formatter class which would format the data into either XML, JSON or STRING format. The formatter would then pass the data back the controller and is passed to the user via the response print writer. Seen bellow in Figure 6 MVC is the file structure of the java files within the project.

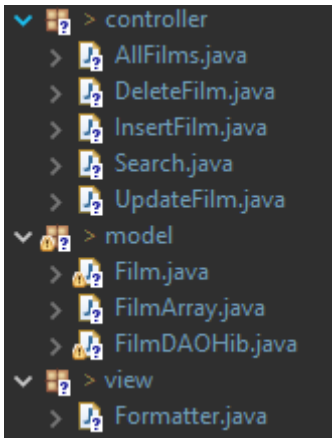

Figure 6 MVC

By laying the project out like this I can make the project more modular and easier to come back to perform changes to because it is clear what each class is doing and how they interact with each other. Without using this layout for my project, the code would have been a lot harder to come back to and change as everything would have been muddled into classes that contained a lot of methods that might not all be related to each other. So, by using MVC I would be able to come back and do maintenance on the application in a much easier and simpler fashion.

Refactoring

Objects

Throughout the project side my java objects have retains a cohesive naming convention with appropriate names and relevant getters and setters. This allows for the easy understanding of the code and how it operates and allows for it to be followed in its logic with ease.

Common Code

An example of some common code that I moved to its own class would be the formatter class, this contains the code that was used in both the AllFilms.java and Search.java class. The formatter class takes an input and returns the input as either an XML object or json string or a regular string. This meant I was calling the class instead of having the 3 methods in the code a total of 6 times. This helped to streamline the code base and make it more modular.

On the JavaScript side I have multiple different utility functions that are located within a single class that are used throughout the project. These are within the Data_utils.js file, this contains methods to extract values from xml data and then also to create a parameter search string depending on what

search method was used. As seen bellow in Figure 7 MakeParamString() is the function to generate the search string depending on the value in the dropdown box. By using this method im able to have a single search function instead of having different ones for different thing, this is because the Search Types will always be in the format seen so I can just see what the variable Search Type contains and then build a search string from that. An example of this is the first if statement on line 16 where if the Search Type is title then I will make the string for finding a film name which is filmname={?}&format={?}. This utility method allowed me to have a lot more power over the way the user can search.

```
15  function makeParamString(SearchType,SearchVal,format) {
16      if (SearchType == "title"){
17          var paramString =
18          "filmname=" + SearchVal +
19          "&format=" + format;
20          return(paramString);
21      }
22      else if (SearchType == "id"){
23          var paramString =
24          "filmid=" + SearchVal +
25          "&format=" + format;
26          return(paramString);
27      }
28      else if (SearchType == "year"){
29          var paramString =
30          "filmyear=" + SearchVal +
31          "&format=" + format;
32          return(paramString);
33      }
34      else if (SearchType == "director"){
35          var paramString =
36          "filmdirector=" + SearchVal +
37          "&format=" + format;
38          return(paramString);
39      }
40      else if (SearchType == "stars"){
41          var paramString =
42          "filmstars=" + SearchVal +
43          "&format=" + format;
44          return(paramString);
45      }
46
47      else {
48              var paramString =
49          "format=" + format;
50          return(paramString);
51      }
52  }
```

Figure 7 MakeParamString()

Another example of some common code would be all the libraries I used, these made it a lot easier to perform certain actions as I didn't have to create the methods myself.

Lab Code

My projects were created with the labs as help however all the code was created from the ground up. Any aspects used from lab files were reformatted to fit into my assignment perfectly. The code as stated was created from the ground up thus I didn't start with a lab file and reformat that into my assignment. This was because I knew how much more was going to be needed on top of whatever a lab could provide me and so it was a lot easier to make it its own project instead of a Frankenstein of labs.

Libraries

JAXB

To create my XML objects to be returned to the user I used the JAXB library. Using this library I was able to pass it an array list containing all the films and then it would create an XML object using this and the notations I had set within the Film object as seen in Figure 8 JAXB Film Object.

```
14 //setting the root element for the xml object
15 @XmlRootElement(name = "film")
16 //setting the order of data in the xml object
17 @XmlType(propOrder = { "id", "title", "year", "director","stars","review" })
```

Figure 8 JAXB Film Object

By adding notation into the object of the film I can provide JAXB with an array list of these objects and create an XML document as I have defined what the nodes are within the Film object. Normally you may use a format document to set the format with JAXB but because the project had to run on the cloud I couldn't create or delete files so I had to have the xml object written out into a byteArrayOutputStream which was then passed back to the servlet to be passed to the user which can be seen on line 28 in Figure 9 JAXB.

```
17    public static ByteArrayOutputStream FormatXML(ArrayList<Film> filmsFromDAO) {
18        FilmArray filmarray = new FilmArray();
19        System.out.println("Format: XML");
20        //populates the filmArray object with the arraylist
21        filmarray.setFilmObj(filmsFromDAO);
22        try {
23            //Uses Jaxb to sort the xml data
24            JAXBContext context = JAXBContext.newInstance(FilmArray.class);
25            Marshaller m = context.createMarshaller();
26            m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
27            ByteArrayOutputStream baos = new ByteArrayOutputStream();
28            m.marshal(filmarray, baos);
29            //returns the xml data to the user
30            return baos;
31        } catch (JAXBException e) {
32            // TODO Auto-generated catch block
33            e.printStackTrace();
34        }
35        return null;
36    }
```

Figure 9 JAXB

JAXB made it a lot simpler than I first imagined it would be to turn the film objects returned by the database into an xml object. When doing my research this stood out as the clear winner in how simple it was to apply to the project and get working.

GSON

The GSON library allowed me to easily create a json object to return the data back to the user with. This is seen within the formatter.java class and bellow in Figure 10 GSON. In this class I am passing it an array list object which is the films retrieved from the DAO. This is then converted into a json object using the GSON library. This is done by creating a new GSON object and then simply creating a string using that object and the toJson() function within the class.

```
42      public static String FormatJSON(ArrayList<Film> filmsFromDAO) {
43          //creates a new Gson obkect
44          Gson gson = new Gson();
45          String jsonResult;
46          //creates a json string with the arraylist of films objects
47          jsonResult = gson.toJson(filmsFromDAO);
48          //returns the data to the user
49          return jsonResult;
50      }
```

Figure 10 GSON

This allowed me to easily create a json object that could be returned to the user in as little as 4 lines of code which I was really happy with and saved a lot of time and effort by the fact that I didn't have to make my own methods that would parse the array and generate a json string from it.

Jquery.js

When doing the labs for the module I used a file of self-made ajax methods that I could use to perform ajax calls for the html page. However, for the assignment I decided to use the jQuery library. This is because it offers a lot more utility than my own code ever would, and it is Widley used in the industry and is well known. By using jQuery to perform my ajax calls I was able to condense my code into a much easier to understand format. Take this example here from when the add film button is pressed as seen in Figure 11 Add Film JS. Upon pressing the button, the function takes the data from within the fields and creates a film json object with them. This object is then passed via the $.ajax() method which is the ajax call that jQuery allows me to perform. It allowed me to specify what kind of http call it was, in this case being a POST request. I give it the URL of the servlet that the data needs to be sent to and then the data itself. With the success field I am specifying what to do when the operation is successful. In this case it renders a template thanks to Mustache that specifies that it was successful by passing the return statement from the servlet.

```
42  $('#AddFilm').on('click',function(){
43
44      $resultRegion.empty();
45      var film = {
46          id: $FilmID.val(),
47          title: $FilmTitle.val(),
48          year: $FilmYear.val(),
49          director: $FilmDirector.val(),
50          stars: $FilmStars.val(),
51          review: $FilmReview.val()
52      }
53      var errorTemplate = $('#InsertErrorTemplate').html();
54      $.ajax({
55          type:'POST',
56          url: 'InsertFilm',
57          data: film,
58          success: function(request) {
59              data = {data: request};
60              $resultRegion.append(Mustache.render(errorTemplate, data));
61          },
62          error: function(){
63              alert('Error adding film, try another ID');
64          }//close error function
65      })
66  });//close click
```

Figure 11 Add Film JS

It was a lot easier to create this project using the jQuery library rather than writing my own code to perform the same actions as it saved time that I was able to invest into other areas of the assignment such as the Mustache.js templates.

Mustache.js

My choice of library to help create my Graphical user interface was Mustache.js, this is because it allowed me to create templates that I could then use to perform some of the powerful operations that my application has the ability to do so. The first step of using Mustache.js was to create a template in HTML, this can be seen bellow in Figure 12 Template. This template is for a single film record with it being with a li and having multiple spans and inputs that may not be shown at once. All the attributes within this template have classes that are used alongside CSS to hide and show certain ones at certain time.

```
<template id = "filmTemplate">
    <li>
        <button data-id='{{id}}' class='remove'>X</button>
        <p>
        <strong>ID: </strong>
        <span class="noedit id">{{id}}</span>
        <input class="edit id" size="6" autocomplete="on">
        </p>
        <p>
        <strong>Title: </strong>
        <span class="noedit title">{{title}}</span>
        <input class="edit title" size="30" autocomplete="on">
        </p>
        <p>
        <strong>Year: </strong>
        <span class="noedit year">{{year}}</span>
        <input class="edit year" maxlength="4" size="4" autocomplete="on">
        </p>
        <p>
        <strong>Director: </strong>
        <span class="noedit director">{{director}}</span>
        <input class="edit director" size="20" autocomplete="on">
        </p>
        <p>
        <strong>Stars: </strong>
        <span class="noedit stars">{{stars}}</span>
        <input class="edit stars" size="50" autocomplete="on">
        </p>
        <p>
        <strong>Review: </strong>
        <span class="noedit review">{{review}}</span>
        <textarea class="edit review"  rows="4" cols="100" autocomplete="on"> </textarea>
        </p>

        <button class="editFilm noedit">Edit</button>
        <button class="saveEdit edit">Save</button>
        <button class="cancelEdit edit">Cancel</button>

    </li>
</template>
```

Figure 12 Template

Within the ShowData.js class is the function that will insert this template into the screen. Within the HTML file is a ul called results which is what the $resultRegion is. When this method is called it will append a copy of the template onto the screen containing the data from the film object it was given.

```
function addResult(film, $resultRegion){
    //collects the template from the html page
    var resultTemplate = $('#filmTemplate').html();
    //adds on a film record to the ul
    $resultRegion.append(Mustache.render(resultTemplate, film));

}
```

Figure 13 addResult

As seen in Figure 14 ShowJson within the ShowJsonCustomerInfo function which is located in the same class as the addResult the addResult function is called for every record within the json object. This means for every film returned to the webpage, a new template is added to the webpage in the $resultRegion at the end of the current list.

```
function showJsonCustomerInfo(request,$resultRegion) {
    if (request != null) {
        var rawData = request;

        console.log(request);
        var films = eval("(" + rawData + ")");
            //validating if a result was returned so an error msg can be shown if no result was found
            if(films.length >= 1){
                //iterates through records in the json object
                for(var i=0; i<films.length; i++) {
                    var film = films[i];
                    //creates an object to store the single film to pass to the mustache template
                    var film_obj = {
                        id: film.id,
                        title: film.title,
                        year: film.year,
                        director: film.director,
                        stars: film.stars,
                        review: film.review,
                        }//close film_obj
                    //sends the film and the location to insert it to the method that will show the record to the user on the webpage
                    addResult(film_obj, $resultRegion)
                    console.log("Search Complete, result returned");
                }
            }
            else{
                $resultRegion.html("<li> No Results Found with Search Values, Please Try again with a different Search</li>");
                console.log("Search Complete, No result found");
            }
        }
    }
```

Figure 14 ShowJson

So far it is reasonable to assume this isn't anything special because this can be done in html just printing all the records to the screen. However, where this differs is the edit and delete functions within the application. This is shown best when the editFilm button is pressed on a single template. What this button does is it gets the closes li to the button, which is the li that surrounds the template and sets the class to 'edit' which can be seen bellow in Figure 15 Edit Button on line 93.

```
85      $resultRegion.delegate('.editFilm','click',function(){
86          var $li = $(this).closest('li');
87          $li.find('input.id').val($li.find('span.id').html());
88          $li.find('input.title').val($li.find('span.title').html());
89          $li.find('input.year').val($li.find('span.year').html());
90          $li.find('input.director').val($li.find('span.director').html());
91          $li.find('input.stars').val($li.find('span.stars').html());
92          $li.find('textarea.review').val($li.find('span.review').html());
93          $li.addClass('edit');
94      });//close edit button
```

Figure 15 Edit Button

Now with the CSS settings seen in Figure 16 CSS edit Classes when the li has the class of noedit then anything with the class of edit is hidden. And when the class has the class of edit then items with noedit are hidden. This means that when the edit film button is pressed the input boxes are shown and the spans are hidden. Additionally, when the button is pressed the default value of the inputboxes are set to the values of the spans. This means that it appears as though you are directly editing the values in the database instead of having to copy the data yourself.

```css
ul li.edit .edit{
    display:initial;
    background-color: green;
}

ul li.edit review{
    line-height: 4em;
}

ul li.edit .noedit{
    display:none;
    background-color: yellow;
}
```

Figure 16 CSS edit Classes

In the edit mode the edit button gets replaced with a save and a cancel button. The cancel button simply removes the class of edit from the li therefore removing the textboxes and bringing the spans back. This is seen in Figure 17 Delete Button. However if you decide to instead save the data then pressing the Save button will run an ajax call to the UpdateFilm servlet and on success will revert the li back to being noedit but it will also retrieve the data from the film object created that is passed to the servlet to update the spans as they are out of date as the record has just been updated.

```
$resultRegion.delegate('.cancelEdit','click',function(){
    $(this).closest('li').removeClass('edit');
});//close canceledit button
```

Figure 17 Delete Button

```
$resultRegion.delegate('.saveEdit','click',function(){
    var $li = $(this).closest('li');
    var film = {
        id: $li.find('input.id').val(),
        title: $li.find('input.title').val(),
        year: $li.find('input.year').val(),
        director: $li.find('input.director').val(),
        stars: $li.find('input.stars').val(),
        review: $li.find('input.review').val()
    }
    console.log(film);
    $.ajax({
        type: 'POST',
        url: 'UpdateFilm',
        data: film,
        success: function(request){
            $li.find('span.title').html(film.title);
            $li.find('span.year').html(film.year);
            $li.find('span.director').html(film.director);
            $li.find('span.stars').html(film.stars);
            $li.find('span.review').html(film.review);
            $li.removeClass('edit');
        },
        error: function(){
            alert('error updating film');
        }
    })

});//close saveedit button
```

Figure 18 Save Button

By using mustache.js for this operation it makes editing the record much easier as you don't have to physically remember the values in each of the record fields and can make small edits a lot easier and not have to worry about copying data over incorrectly.

The Final Operation that Mustache.js helped a lot was deleting a record. If you cast your eyes back to Figure 12 Template you will notice on line 60 a button is created with the class of remove and a data-id of the id of the film. Pressing this button on a specific li will send that id, which is unique to the film that the li is displaying, to the delete film servlet as seen in Figure 19 Delete Button.

```
68    $resultRegion.delegate('.remove','click',function(){
69        console.log($(this).attr('data-id'));
70        var data = "id=" + $(this).attr('data-id');
71        var $li = $(this).closest('li');
72        $.ajax({
73            type:'GET',
74            url:'DeleteFilm',
75            data:data,
76            success: function(request) {
77                if(request == "Successfully Deleted Film"){
78                    $li.remove();
79                }
80                console.log(request);
81            }
82        })
83    });//close remove button
```

Figure 19 Delete Button

When the button is pressed it retrieves the id of the film that was set as the data-id attribute within the button itself. Again, this id is the if of the film that was searched, it's not visible on the button but it is stored within it. After pressing the button, it takes this id and creates a string being 'id={?}', it then runs the ajax call to the servlet passing it the id parameter. When the operation is successful it then removes the li of the film as there is no reason to show the film record on the page anymore as It has been deleted from the database.

Mustache.js was a big part of my HTTP interface as it allowed me to simplify the interface into a search and an add box and then once you have searched for a film you can then either edit or delete the film. I didn't want the interface to be overpopulated with fields or must rely on alert boxes with fields in it to change data. This way everything is completely within the browser and is Asynchronous .

Hibernate

Initially I used my own code and connection methods to connect to the database and these are still in use within the REST service to display this skill as seen in Figure 20 REST SQL Connection as it connects to the mudfoot server at University. However, towards the end of the HTTP interface I decided to change the connection method to use the Hibernate Library.

```java
public class FilmDAO {

    Film oneFilm = null;
    Connection conn = null;
    Statement stmt = null;
    String user = "ainsleyn";
    String password = "Flac7term";
    // Note none default port used, 6306 not 3306
    String url = "jdbc:mysql://mudfoot.doc.stu.mmu.ac.uk:6306/"+user;

    public FilmDAO() {}


    private void openConnection(){
        // loading jdbc driver for mysql
        try{
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch(Exception e) { System.out.println(e); }

        // connecting to database
        try{
            // connection string for demos database, username demos, password demos
            conn = DriverManager.getConnection(url, user, password);
            stmt = conn.createStatement();
        } catch(SQLException se) { System.out.println(se); }
    }
    private void closeConnection(){
        try {
            conn.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

Figure 20 REST SQL Connection

With Hibernate I had to edit the Film object class by adding some annotation to the class to help hibernate identify the data structure of the data I want to deal with on the SQL server. This can be seen bellow in Figure 21 Hibernate Annotations. There I am setting the name of the table that the data is regarding and then the name of the columns that the variables are related to. This means that I could then pass Hibernate a full film object and it knows where exactly to insert the data into and what columns take what data.

```java
@Entity
//defines the table
@Table(name="films")
public class Film {
    //constructor for the Film object
    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;
    }
    //defines as the ID
    @Id
    //defines the table column to id
    @Column(name="id")
    int id;
  //defines the table column to title
    @Column(name="title")
    String title;
  //defines the table column to year
    @Column(name="year")
    int year;
  //defines the table column to director
    @Column(name="director")
    String director;
  //defines the table column to stars
    @Column(name="stars")
    String stars;
  //defines the table column to review
    @Column(name="review")
    String review;
```

Figure 21 Hibernate Annotations

Hibernate then needs a configuration file which can be seen bellow in Figure 22 Hibernate Config. This is the file that tells Hibernate all the connection information of the server. Notice that the connection details are different in this file as to the RESTful connection data. This is because the HTTP interface connects to a cloud database whereas the RESTful interface connects a Sql server located on MMU Grounds.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password">root</property>
        <property name="hibernate.connection.url">jdbc:mysql://35.189.101.68:3306/Movies</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    </session-factory>
</hibernate-configuration>
```

Figure 22 Hibernate Config

If we compare the two connection methods for the methods to get all the films then the difference between the Hibernate framework and the old method is stark.

```
20      /* Method to return a list of all films in the database*/
21      public ArrayList<Film> ListAllFilms() {
22          try {
23              factory = new AnnotationConfiguration().configure().addAnnotatedClass(Film.class).buildSessionFactory();
24          }catch(Throwable ex) {
25              System.err.println("Failed to create sessionFactory object. " + ex);
26              throw new ExceptionInInitializerError(ex);
27          }
28          Session session = factory.openSession();
29          Transaction tx = null;
30          ArrayList<Film> films = new ArrayList<Film>();
31          try {
32              tx = session.beginTransaction();
33              String hql = "FROM Film";
34              films.addAll(session.createQuery(hql).list());
35              return films;
36
37          }catch (HibernateException e) {
38              if (tx!=null) tx.rollback();
39              e.printStackTrace();
40          }finally {
41              session.close();
42          }
43          return null;
44
45      }
```

Figure 23 Hibernate AllFilms

```
44      private Film getNextFilm(ResultSet rs){
45          Film thisFilm=null;
46          try {
47              thisFilm = new Film(
48                      rs.getInt("id"),
49                      rs.getString("title"),
50                      rs.getInt("year"),
51                      rs.getString("director"),
52                      rs.getString("stars"),
53                      rs.getString("review"));
54          } catch (SQLException e) {
55              // TODO Auto-generated catch block
56              e.printStackTrace();
57          }
58          return thisFilm;
59      }
60
61
62
63
64      public ArrayList<Film> getAllFilms(){
65
66          ArrayList<Film> allFilms = new ArrayList<Film>();
67          openConnection();
68
69          // Create select statement and execute it
70          try{
71              String selectSQL = "select * from films";
72              ResultSet rs1 = stmt.executeQuery(selectSQL);
73          // Retrieve the results
74              while(rs1.next()){
75                  oneFilm = getNextFilm(rs1);
76                  allFilms.add(oneFilm);
77              }
78
79              stmt.close();
80              closeConnection();
81          } catch(SQLException se) { System.out.println(se); }
82
83          return allFilms;
84      }
```

Figure 24 REST all Films

Now as seen above in both Figure 23 Hibernate AllFilms and Figure 24 REST all Films we have the two different Sql methods. In the Hibernate one we start by creating the factory that we will use to create the session and setting the class that we are dealing with which is the Film class. Once we have this factory, we create the session which we will use to connect to the Sql server. We then create a transaction variable and an array list. Lastly, we Start the transaction using the session, set our Sql query string and then perform the query which is located on line 34. We then return the films array list to the servlet for processing. Compare this to the other method as shown in Figure 24 REST all Films we create the array list, open the connection using a method then run a full string on the server to retrieve the films. We then must process these films here and add them to the array

list one at a time. Whilst the none hibernate method is less code it is not quite as simple to do as the other method as there is also the openconnection() and closeconnection() methods as seen in Figure 20 REST SQL Connection.
Hibernate was an amazing choice of library to use as it allowed for much simpler calls to the database in an easier fashion without having to open my own connections and allowed me to provide the database with an entire film object as seen bellow in Figure 25 Hibernate Add Film.

```
170     /* Method to CREATE a film in the database */
171     public Integer addFilm(int id, String title , int year, String director, String stars, String review){
172         try {
173             factory = new AnnotationConfiguration().configure().addAnnotatedClass(Film.class).buildSessionFactory();
174         }catch(Throwable ex) {
175             System.err.println("Failed to create sessionFactory object. " + ex);
176             throw new ExceptionInInitializerError(ex);
177         }
178         Session session = factory.openSession();
179         Transaction tx = null;
180         int success=0;
181         try{
182             tx = session.beginTransaction();
183             Film newfilm = new Film(id, title, year, director, stars, review);
184             session.saveOrUpdate(newfilm);
185             tx.commit();
186             success = 1;
187         }catch (HibernateException e) {
188             if (tx!=null) tx.rollback();
189             e.printStackTrace();
190         }finally {
191             session.close();
192         }
193         return success;
194     }
```

Figure 25 Hibernate Add Film

Lack of Spring

Upon looking over the lab files for the spring framework, I decided not to use it as I liked the modularity of having each servlet within their own java file and having the ability to easily add a new servlet without effecting another one. Therefore, I decided to write my own servlets apposed to using the spring framework.

This was entirely down to the fact that I had done all my labs with servlets and had come to appreciate the modularity of having each one within their own class, having their own path and having the ability to have a get and a post request on the same servlet. This function was never used however an example of where I could have used it would be the search function for id where a get request could retrieve the film where as a post request could have been used to delete that film, or film where any variable had been passed. I ultimately chose not to do this however as I decided to use Mustache for the button and was happy with a singular servlet for the delete function and search function as it allowed for greater modularity between the two functions that are ultimately different.

Jersey

The Jersey library was used when creating the RESTful interface, this was because of the annotation it allowed to specify certain functions as rest operations. This can be seen in Figure 26 Jersey Get method where I am annotating the function as a get method and specifying what kind of data formats it can return being XML or JSON.

```
43      @GET
44      @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
45      public Film getFilmID() {
46          int ID = Integer.parseInt(id);
47          Film film = fd.getFilmByID(ID);
48          if (film == null)
49              throw new RuntimeException("Get: film with ID " + id + " not found");
50          return film;
51      }
```

Figure 26 Jersey Get method

With Jersey annotations I was also able to annotate the path that was needed to reach these functions as seen bellow in Figure 27 Jersey Path. This sets the URL of the file after the files location with it being http://localhost:8080/rest.18028669.services/rest/films in this case.

```
26  //Will map the resource to the URL films
27  @Path("/films")
```

Figure 27 Jersey Path

Conclusion

After reviewing my code, the techniques used, and the libraries used I can happily say that this is project is the combination of aspects I have learnt throughout my programming modules at MMU. From Software engineering techniques of waterfall vs agile in first year. To Data Accessing Objects and MVC within advanced programming in year 2 to now using such patterns as Singleton in this module. My use of libraries has expanded to rely more on common industry standard apis such as jQuery and JAXB for tasks that would have required a lot of work on my end if not for their prebuild functionality. My code is commented through out explaining what is going on and allows for easy understanding of modules and classes. Overall, I am more then happy with how most of this assignment came out, especially the HTTP/Cloud interface, the only aspect I fell short on was the SOAP interface.