Critical analysis report– Enterprise programming

# **Design aspects**

I’ve done a lot of work on this project and will undoubtedly miss some of it off when discussing in the critical report – so please also refer to any notation in the code.

The best place to start will be discussing any key design patterns applied. In this project there are number that have been utilised:

* Singleton
* DAO
* MVC
* Three Tier architecture
* Single page application

In three tier architecture, as the name states, the application is split into three tiers:

1. Presentation tier
2. Application tier
3. Data tier

The presentation tier is the quite literally the end users computer, or where the web-app will be seen. The application tier is all of the HTML, java, Javascript etc within the project file. The data tier is the database itself, so the university mudfoot server in this case.

With respect to the presentation layer, no key comments will be made as there is no control over that. However, its important to talk about the data tier and the application tier in more detail.

# **Structure:**

## Data tier

For the database, the script from the assignment starter files was utilised to create the table in the university mudfoot server. Consideration of the structure of the table was made in order to determine if any changes could be made here that would make the project easier later. The key thing that was noted was there were no constraints of any kind, including primary key.

It was decided that, in order to avoid duplication of the id of the key, which should be unique, that the table would be altered in two ways:

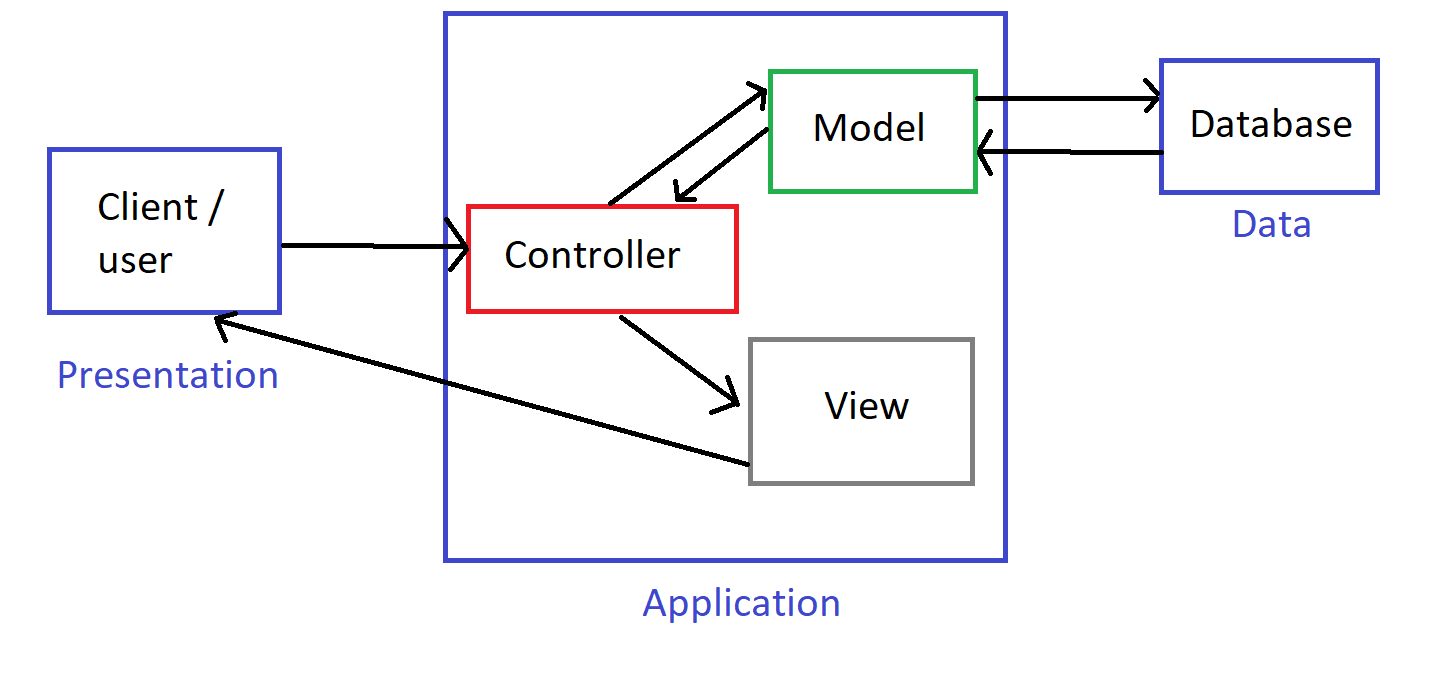
* Addition of a primary key constraint on the id column
* Addition of auto-incrementation of the id, from the value of the last id input

Adding auto incrementation of id means that the user will never have to search up id’s to find a value that’s free, but also also allows for practical implementation of all of the uses required by the primary key – such as updating and deleting specific records.

## Application tier

The remainder of this report will discuss the application tier including discussion about how the other three design patterns (singleton, DAO, MVC and single page application) have been implemented. This report will be broken down into the three areas of the MVC – Model, View and Controller – from here on out, and some discussion on how the work within the application tier will be considered.

An overall depiction of the utilisation of MVC and three tier in this project can be seen in this (crude) diagram:



# **Model:**

The purpose of the model is to interact with the database and should contain all of the business logic in which to do so – including the structure of the required beans (Film and FilmList), to that end the Model for this project is made up of the Film and FilmList classes as well as the FilmDAO class.

## Film and FilmList classes

The film class is fairly standard and utilises the same variables and variable naming as the films stored in the database for ease of access. The root element tag for XML has been added so for serialization and deserialization purposes with JAXB later on.

Next, the Filmlist class was generated when looking at how XML serializes, especially when using using collections – in this case, ArrayLists. I don’t know if JAXB can create context for an ArrayList, but a simple way around this was to create a class that stored a collection as a variable, hence FilmList.

Later on in the development of the webapp, when considering paginating larger data returns, I decided to add an integer variable into FilmList called numFilms, which stores the maximum number of films available to a specific get request – it’s important to know how this differs from FilmList.ArrayList.size().

The ArrayList size would only return the value representing that specific search, which can be limited due to pagination. For example, each page on the web application only shows 10 films per page – so the ArrayList in the FilmClass can contain a maximum of 10 films (see reasoning later on). The numFilms variable is the total possible from that search query – e.g. all films may return > 1000 for numFims.

Finally, XML tags were added for the FilmList wrapper and Film element, and numFilms element, to support serialization with XML.

## FilmDAO class

As suggested, the DAO class implements the DAO design pattern. The dao provides an interface to the Film and FilmList classes, separating out the required CRUD operations.

I used the base code provided for the **filmDAO** class but made a number of changes.

Firstly, I added a singleton design pattern to this, to ensure that multiple instances of the DAO weren’t being generated, to minimise the number of open connections to the server per user.

Next, I moved the close connection out of the try{} in the try catch, and put at the end of the function – the thinking being that if the try failed, it would leave instances of the open connection, unclosed, had I not done so.

Searches that are for all films or by name, an integer is passed into the argument as the current page, which is the page being called upon by the web application. This informs the select query as it will be limited using “LIMIT offset, limit” in mySQL, to only return values relevant to that page. I debated on how to do this for a long time, as collected an arraylist with the entirety of the search term, then filtering down by the required number would have been just as easy – but when you have calls onto large databases with 10s of thousands, or even millions of responses, this doesn’t seem like such a good idea then. So I thought in terms of scalability and the query only asks for the specific data that it needs at the time – nothing more.

I added a function to count the number of films when searched by “all films” and by title. This wasn’t necessary when by id, as id would either be 1, or no films found. This was to support the pagination in webapp – and an attempt was made here to ensure that this was only collected upon query to the database and never stored, supporting the stateless requirement of a rest API. The count of films also is irrespective of the page number, so I will return the total count, not limited, which is used to inform the web app with the required total number of pages.

Update and deletefilm are standard functions which don’t require much explaining other than changing them from void functions to Boolean to provide the output with some confirmation that it was successful or not. Add was also changed to a boolean and was given one more slight difference to it – where in the SQL insert, we don’t include the ID as I set the id column in the database to a primary key and to auto increment from the last provided value.

# **Controller:**

The purpose of the controller is to take user inputs (request parameters) and interact with the model and the view as necessary. In this project this is the role of the REST API.

A REST API should utilise the HTTP methods explicitly, which has been implemented. It should also be stateless, which has also been implemented. It also has the ability to transfer data in any of the three required formats.

Functionally, this is contained in a single servlet - FilmAPI with the url “filmapi” which all the web all calls will be made to. However, there are a number of supporting classes containing business code within the Utils package, each of which are named to represent their respective functions.

When considering the requirements of the REST API in this project, it was broken down as follows:

* GET – 3 possible return types
  + All films
  + Searched by title
  + Searched by ID
* POST – 1 insert type
* PUT – 1 update type
* DELETE – 1 delete type

This was made more complicated when considering that each of the above required the API to recognize the possibility of 3 different data types – JSON, XML and Text.

## GET

By and large the most complicated of the requests as there are three different search types with three different data types – which the start of the get requests needs to find both.

The data type is obtained from the accept header, whereas the search type is from the data appended to the url – for example, if it’s a search by id it will be filmapi?id=(n) – if id isn’t blank, the search type is id, same for title. If the search type is for all films, it is q=af or query=allfilms.

Functions to collect this information were added to the GetUtils class to keep the FilmAPI clear.

Next, the page number is obtained, also from the url.

Early on, the decision was made to not only use FilmList class for XML, but also to use it for JSON and Text to keep the code simple and it is generated by building the query based on the request type, page number and the request body. In GetUtils, a number of if clauses check the request type and create the ArrayList of films by calling on the relevant functions in the filmDAO class.

Lastly, the FilmList is then serialized by the relevant data type using GSON for JSON, JAXB for XML and for text, I used regular expressions that I knew would not be common in any of the returned text, such as %& and £%. The function that serializes the FilmList in GetUtils also sets the relevant response headers.

After that a printwriter is created and the serialized data is printed out, with the print writer bring closed afterwards.

## PUT & POST & DELETE

Similar to above, the data type is collected from the request header, but with POST and PUT, no data is appended in the url. Instead it is in the body of the request and is accessed using a stringbuilder and a buffered reader to build the request body as a string.

Next, the information is deserialized into a single film using the relevant methods of GSON, JAXB or regex. Notably, only a single film as updating, deleting and inserting can only be done as a single film at a time.

After this, the resultant operation is performed utilising the FilmDAO class. For each operation a true or false is returned, corresponding to wether the operation could occur or not. This true or false is then converted into string message, which is implemented into a class (for the sake of XML serialization). This class is then serialized and returned back via a printwriter to the web application informing them if it was successful or not.

# **View:**

The view is responsible for the code that generates what the user sees on their screen. In this project, the Index.Html file contains the main code for the page. It’s important to point out here that this is our only page and the single page application design pattern has been implemented in this project to keep it that way.

Supporting code to generate elements for the page is contained in the JQuery-Ajax script as well as the Utils script. Jquery was utilised as a library for generate the javascript files, and bootstrap and css we’re utilised to assist with specific design elements.

When the web app is loaded up, the user is greeted with a basic “welcome” card, along with direction to the navigation bar at the top, as follows:

Text

Description automatically generated

The decision was made not to launch all films on the page at launch – because there has to be the option for the user to select each data type they wish data to be return in, so the welcome card serves as a placeholder.

There is a lot of code in the JQuery-Ajax script and I wont be able to go over it, but I will make some comments at the end.

The navigation bar serves to allow the user the first of two possible actions – searching for films or adding a new film.

## Adding a film:

When the user clicks to add a new film, a card will appear requesting the film information. The div for the welcome card is cleared, and the HTML code for the new card is added using $(#element).append(); in JQuery.

Functionality has been added in here to check the following about certain inputs and selects:

* Fields aren’t blank
* Year is a number
* Data Type selection in the nav bar isn’t blank

Because there’s no instruction otherwise, or any examples where data fields are blank in the database, it made sense that the information for each field should be provided, so they cant be blank.

Year has to be a number as per the database and the film class in the model, therefore a test using isNaN was utilised to see if the input could be a number.

Data Type is required as the resultant data is then serialized via javascript, into their relevant data type objects (Json, XML and text) before being sent to the controller.

If any of these error, some css functionality, including animation, has been added to highlight the error and the AJAX request does not fire.

If there are no errors, a POST AJAX request is sent to the controller, with the film data in the body as plain/text type. If the Ajax request is successful – div containing the add film card is clear and a get request is made to the controller, where a card with the newly added film is generated and displayed.

If there is a problem with the Ajax request – a console message is parsed which can be seen in developer tools.

Finally, if the user decides not to add a new film and clicks X – it cleas the div with the add film card and presents the welcome card again.

## Searching a film:

When searching for a film there are three criteria that is required:

1. The type of search – all films, by id, by title
2. The search term – name or id (where applicable)
3. The Data type – JSON/XML/TEXT

As all films doesn’t require an input, the decision was made to hide the input field initially. Next, the ID field is required to be a number, so some validation was required there. Finally, title can contain anything, so separation of the ID and Title fields felt sensible.

Using JQuery, the nav bar was made so that an input for ID was shown when an ID was required and an input for title was shown when one for title was needed, but nothing was shown otherwise. For example:

Text

Description automatically generated

Next, an error component was added to the ID input when it wasn’t input as a number, much like with year above.

Finally, error elements were added to both the search type or the data type selectors, if nothing was selected (i.e. left default).

When all this required information is correct, and AJAX GET request is made to the server, with the following url appendixes:

* id=(user input id)&pg=1
* title=(user input title)&pg=1
* q=af&pg=1

The first key value pair is the search type and data, the second is defaulted to the first page being returned.

Some notes about pagination are made later.

If the AJAX request is successful, the resultant data is received, deserialized from its data type, and added to the screen via cards.

The decision was made not to do this as a table from a design element. Tables are great for displaying lots of data, but aren’t always user friendly. There was an aim to have these cards similar to the way that facebook and twitter operate with their posts and tweets – losts of functionality / pictures, etc. This wouldn’t have looked as nice in a table. Unfortunately I ran out of time to implement the film posters from IMDB API, but I still think the resultant cards look much better than a table view.

Each displayed hard has the film title and year in the card title, followed by the remaining information in the card body. Next, there is an edit and delete button on each card. Each button and each div containing the card, when generated, caches the id of the film in the card, as the data-id field of the button. This is important later on, so the view can inform the controller the identity of the film to update or delete.

## Delete:

As a function, delete is very simple. The SQL for delete in the film DAO only requires the film id to delete. As there are no inputs to read for this, a mock film is generate in the javascript with “undefined” and 9999 in each string field and the year respectively. The id is take from the data ID of the button and parsed to int, and the film object is serialized based on the data type selected. If no data type is selected it will error and inform the user to select a data type and not fire the AJAX request.

If everything is okay, an AJAX DELETE request fires and if successful, it then clears the div containing the card that the film to delete is on.

If there is an error, it prints to the console in developer tools.

## Edit:

If the edit button is pressed, a script to hide the current text and unhide a number of input boxes is triggered on the card itself. This is done by the id number cached in the button being the same as that of the respective card div.

As with add, if any of these are left blank, or the year is not a number, the AJAX request wont fire. If no data type is selected, it also wont fire.

When the data is correct and the user presses confirm, this triggers an AJAX put request to the server, and the text in the original card is updated to reflect the change.

This could have been done with a GET request for the update bool, but to avoid too many requests to the server each time, it made sense to do it this way, but that’s possibly not the right decision.

## Other considerations:

### Pagination:

Probably the most difficult thing to do in the project. Coming in with no knowledge of HTML, JS, JQUERY, CSS, Bootstrap etc, led to this being a challenge.

The decision was made that only 10 books per page would be displayed – its possible to let the user select this, which would then define the LIMIT component of the SQL queries, but due to time constraints, it was left to 10.

Next, it made sense to add first and last at the end of the pagination, then have the current page as the centre, where possible, allowing the user to move up to 2 pages a time for a total of 5 pages displayed.

There were a number of scenarios to consider in this –

1. What if there weren’t 5 pages?
2. What if they were on the first/second page?
3. What if they were on the second to last / last page?

So code was drawn up to help these situations.

Next, the pages couldn’t use the standard search function as that looked at values in input boxes and select boxes to determine what to search for. This was problematic for a few reasons:

* What if the user had come here from a bookmark and had no values cached in the input?
* What if the user changed the select box in the middle of changing pages?

The decision was made to cache the query string used to generate the original search, in the data-id of the button, but build the queryString, with the different page numbers.

# **Critical Review and Thoughts**

Overall, I think the model and controller are done well. Having more experience with Java and SQL going into this module, I think that shows. I made the decision to write my own API rather than using Jersey as we were shown in the labs. I wanted to fully understand what was happening in the background and I’m glad I did because I was able to learn a lot from it.

With respect to the rest API – I think the modularity of the Utils package and the classes within, help keep the code in the API itself clear and understandable.

With respect to the Utils package classes, I focused on breaking them down into respective requirements, while adding some new things I learnt in there too. For example, the serialization functions for XML and JSON can take in generic arguments using public static <T> String serializeJson(T t). This meant I could serialize both Film and Message with the same function.

Next, for the serialization in text, I used reflection to view the Film class and collect the fields from it. This made it much easier to build the text string and, while it wont future proof if the film class is changed, it will make it easier to modify later.

The FilmDAO class itself is quite big, however functions within are broken down well and are very modular.

When moving over to the view – I don’t have the same level of confidence as I do with the model and controller.

Coming into the project I had next to no experience of HTML, Javascript, CSS, any front end libraries such as React, Vue, Angular or Jquery, or any of the front end design libs, such as bootstrap – so I essentially had to learn all of this as I went along.

There is an expectation to explain the choices we made with respect to libraries used and simply put – I utilised JQuery because it seemed to be the most approachable for what I needed from it. I spent some time reading about React and Vue, but there felt like there was a lot I needed to understand and do, and with relatively no experience working with JavaScript at all, I felt the best approach was to learn JavaScript and then apply JQuery where needed – this was used in things like selecting elements, inserting HTML, doing AJAX calls, etc etc.

Im not 100% sure how to structure Javascript files, and how they interact with each other, like classes and packages in java, so the majority of my script is in a singular script called JQuery-Ajax – with some more understanding its possible that I could have made this more modular, but time constraints meant that wasn’t possible.

Where all the functions work – I’m not 100% confident that they’re the best way to approach them as this is the first web app, or even web page, that I’ve ever built, so a lot of what I have done was try it and see if it works.

I know there’s three key things that I didn’t implement that I really would have given time –

1. “0 results found for your search” card. At the moment if there isn’t anything that turns up on the search, nothing appears other than pagination for a single page.  
     
   This would be *fairly* simple to fix, but would require a bit of a reorder of the search function, obtaining the numFilms varible first, testing if it’s 0, and if it is, creating the card for no films found without any pagintion. If it isn’t, it carry’s as it would, building the film cards and pagination.
2. Incorporating the return message from the REST API somewhere. For PUT POST and DELETE, the return message is never incorporated or even tested. I’ve not ran into a point where this is a problem *YET*, **but** I’m assuming that’s just luck. I think for best practice, what I should do in the success section of the respective AJAX call, is have a function that checks the incoming message to check if it matches the success or failure message. Success would carry on as it is now, but failure would produce some kind of error message for the user in the web app.   
     
   Again, this wouldn’t be too difficult to do, but time constraints mean it wasn’t possible to do.
3. A call to the IMDB API for film posters and ratings – I looked into doing this and saw that there was a max number of calls you can do on a free plan, roughly 1000 a day, so started thinking about ways in which to avoid capping out on it.   
     
   Consider there are over 1000 films in the database – it doesn’t make sense on this plan to have it so that each time a get request is called, it queries the IMDB database for each film title and year, and collects the relevant data, because a single GET request iterating through the FilmList would result in over 1000 calls.   
     
   Instead, I planned to write an SQL statement that would call every single film in the database in one call. Using a statement something like (not sure on IMDB table names and column headers so this is speculation):

SELECT title, year, poster, rating FROM films WHERE (title LIKE (film.getTitle()) AND year = film.getYear()) AND

The text highlighted in red would need to be generated by iterating through the FilmList for all films, recognising when the last film had been reached and not adding an AND on the end.

Next, the data collected would need to be inserted into my database by adding a new column called poster and rating, then iterating through the returned data and inserting the poster and the rating where the title and year are the same.

This would minimise the number of future calls required for searches.

For PUT and POST – calls would need to be added to the IMDB database to do the same as above, but for a single film a time.

1. Security aspects: My knowledge of security is low –but Im aware that my project may be weak to SQL injection. Given more time I would have looked to methods to reduce the risk of that.
2. Data validation – I mentioned this briefly before, but at the moment the only stop on incorrect data going to the controller is the front end javascript validation. To make this more robust, I would also have this in the controller, but time constraints limited it.

Overall, I’d say Im happy with what I’ve produced for this project, and I look forward to any feedback in marking.