Critical Analysis

# Introduction

# In my report, I demonstrate my proficiency in creating, deploying, and utilizing web services. I have carefully examined design patterns and investigated the use of advanced technologies. The project objective was to create a Film Database Management System that enables the software administrator to manage the database and provide features such as adding new films, deleting, updating, and searching for films by title, year, and actor. Throughout the report, I have included screenshots that display the code and specific functionality implemented, and I have also provided clear evaluations and conclusions about my choices for implementing certain features and techniques.

# HTTP/MVC Application

Graphical user interface, text, application

Description automatically generatedThe MVC part of the project entails the use of Packages. Using packages allows organization throughout my project, this aids in design decisions such as where to place a class. The next developer will be able to clearly pick up the project scope and understand the application because of the well-organized nature of the MVC.

In my project I injected JSP’s which ensures a dynamic web project, this separates business logic from presentation to add structure and organize the code.

## Model

### Film

The Model layer contains Film.java a class which defines a Film with properties and getter and setter functions.

When Films are initialized the constructor method requires id, title, year, director, stars, and review are all parameters.

The Film Getters are used within the View.jsp to collect the specified properties to be displayed within the presentation layer.

Graphical user interface, application

Description automatically generated

### FilmDAO

The Model layer is also home to FilmDAO.java, this is an example of the Design pattern : Data Access Object. Using a DAO allows you to test the databased operations separate to the rest of the code .It opens a connection between the application and the database, which is then used to handle all the CRUD operations. I use PreparedStatement within these operations, in order to take advantage of cache which will reduce the number of calls to the database and improve performance of the application.

Create Operation:

Graphical user interface, text, application

Description automatically generated

Read Operations:

You will notice all of these functions use the getNextFilm function which converts the result into the Film object.

Text

Description automatically generated

We have the getAllFilms function which collects all films from the database:

Graphical user interface, text

Description automatically generated

We also have the searchFilms function which returns an ArrayList of films which match the search result. The search checks a concatenated column of title, year , director , stars, and review for an occurrence of the search string.

Graphical user interface, text, application

Description automatically generated

Next, we have the getFilmById function, which returns the film which matches the searched ID.

Graphical user interface, text, application

Description automatically generated

Finally, I made a function to return the last film in the database to determine the correct ID when creating a new entry.

Text

Description automatically generated

Update Operation:

Text

Description automatically generated

Delete Operation:

Graphical user interface, text, application

Description automatically generated

## View

The View layer , otherwise known as the presentation layer , displays information to the user. Within my application this comes in the form of three JSP’s ; filmform.jsp, filmlist.jsp, view.jsp.

JSP’s dynamically update the page to show a new interface to the user. For Example, the filmlist.jsp shows the base layout with the search functionalities, with links to ajax scripts to update the page dynamically and request/receive data from the server :

Graphical user interface, text, application

Description automatically generated

Another JSP we use is View.jsp. The DAO collects the film/films from the database and returns them to the controller, where the controller will set films as an attribute to be read by the JSP. We use the JSTL library to simplify our JSP and avoid scriptlet tags. In particular, the View.jsp uses <c:out> to display results from expressions such as <c:out value=*"*${f.getTitle()}*"* to output the current films title. I have also used <c:forEach items=*"*${films}*"* var=*"f"* > , this is used to retrieve the films attribute and loop through each item within the array. In conjunction, these tags are used to create the table, which will be returned within the result region passed through AJAX. You can see this code below:

Graphical user interface, application

Description automatically generated

The final JSP used is Filmform.jsp. This JSP is used to output a new page where the user will input the required fields to create a new film entry. We use AJAX functionality to collect the films information and insert it into the database. We call the CreateNewFilm() function, which will get the values from the inputs to be used by the database via AJAX functionality .Here is the code:

A picture containing logo

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

## Controller

The controller component receives input from the user, processes it and updates the model accordingly. It also updates the view to reflect the changes in the model. The controller acts as a bridge between the model and view, and coordinates the interaction between them.

Within the application there is multiple controllers, which have their own utility. For example, the DeleteFilmController is responsible for deleting the requested Film. It collects the ID as a parameter passed through AJAX, collects a film object from the ID, enters the DAO passing the Film object as a parameter, and finally directs the page back. Here is the code:

Graphical user interface, text, application

Description automatically generated

The GetAllFilmsControlller is responsible for entering the DAO in order to make a request for all the films in the database, then the controller will use RequestDispatcher rd = request.getRequestDispatcher("view.jsp"); to display the data. Here is the code:

Graphical user interface, text, application

Description automatically generated

The InserFilmContoller creates a film object using by assigning an ID and collecting the required parameters passed from AJAX, the film object is then passed to the DAO to be inserted into the database. Here is the code:

Text

Description automatically generated

The SearchFilmController and the SearchIdController operates almost identical to the GetAllFilmsController, except it collects the search string/id as a parameter then enters the corresponding DAO. Here is the code for SearchFilmController:

Text

Description automatically generated

Here is the code for SearchIdController:

Text, application

Description automatically generated

Finally, UpdateFilmController retrieves the required parameters sent by AJAX to create a film object, which is then passed as a parameter to the DAO to update the database record. Here is the code:

Text

Description automatically generated

## AJAX Scripts

AJAX allows the application to be dynamic, to update the page without reloading it. It does this by allowing data to be requested and received after the page is loaded, it sends data in the background.

## Overview Of MVC

## The Model-View-Controller (MVC) design pattern is a widely used architectural pattern that helps to divide an application into three distinct components: the model, which represents the data and business logic; the view, which represents the user interface; and the controller, which acts as a mediator between the model and the view. Utilizing the MVC pattern has several advantages, such as promoting separation of concerns, making it easier to maintain and extend the codebase. It also allows for greater flexibility in changing the appearance of the user interface without affecting the underlying business logic. Furthermore, it enables multiple views of the same data, allowing for the creation of different interfaces for different devices or platforms.

## Future Improvements

These are the improvements I would make with more time and resources:

* I would code the Filmform.jsp to be a modal using CSS, currently I navigate to another page to view the JSP. I believe it is preferred on the same page as it would improve performance by reducing redirects. It would also be easily closed and bring the user to the same screen as before.
* I would also make the application secure by adding client and server validation to stop potential SQL injections.

# Film RESTful API (Server)

The REST API retrieves requests from the client as data. The server uses this client input to start internal functions and returns output data back to the client.

We use REST API’s because the separation of the client and server allows developers to scale apps with ease. We can create a new client which interacts with the same REST API, for example, within the Film REST API a developer can create a client which filters by genre by collecting keywords from the database. REST API’s have platform- and language-independence which makes creating new clients possible and unrestricted.

### doGet()

The doGet Method determines what search is requested from the client, then enters the corresponding method.

#### searchFilmById()

Below is the method to search the Film database by film ID

Graphical user interface, text, application

Description automatically generated

Here is searchFilmById() getting a JSON response:

Graphical user interface, text, application, email

Description automatically generated

Here is searchFilmById() getting an XML response:

Graphical user interface, text, application, email

Description automatically generated

Here is searchFilmById() getting a String response:

Graphical user interface, text, application, email

Description automatically generated

#### getAllFilms()

Below is the method to get all films in each datatype from the Film database.

Text

Description automatically generated with medium confidence

Here is getAllFilms() creating a JSON response: Graphical user interface, text, application, email

Description automatically generated

Here is getAllFilms() creating an XML response:

Graphical user interface, text, application, email

Description automatically generated

Here is getAllFilms() creating a String response: Text

Description automatically generated

#### searchFilm()

Here is the searchFilm() creating a JSON response:

Graphical user interface, text, application, email

Description automatically generated

Here is the searchFilm() creating an XML response:

Graphical user interface, text, application, email

Description automatically generated

Here is the searchFilm() creating a String response:

­­­ Graphical user interface, text

Description automatically generated

## POST

The doPost Method uses the DAO instance to collect the last films ID to determine the new films ID, it then collects the format as a header called data-type, this is used in a switch statement to decide how to parse the request.

To parse JSON we read the body and use gson.fromJson(body, Film.**class**) to convert JSON into Film object, which is then used by passing the film as a parameter into the DAO ready to be inserted to the database. Here is the code:

Text, letter

Description automatically generated

To parse XML we read the InputStream using the JAVA.io library, we then use the JAXB library to unmarshal the input into a Film object ready to be passed and inserted with the DAO. Here is the code:

Text, letter

Description automatically generated

To parse String we read from the body, we then create a string array by splitting the string by its delimiter : String[] tokens = body.split("#"); , each item of the array is then looped through to collect the required properties, which are then used to create a Film object ready to be passed and inserted by the DAO. Here is the code :

Graphical user interface, text, application

Description automatically generated

The API will then attempt to enter the DAO with the new Film object, if successful it sends the response “Film Inserted” or in case of an SQL exception it prints the error. Here is the code :

Text, letter

Description automatically generated

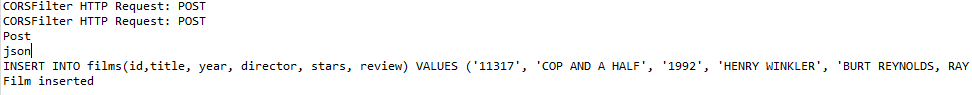
Now to test the FilmAPI works as expected we will run each format through Postman, the correct response will be “Film Inserted”.

Here is the JSON test:

Graphical user interface, text, application, email

Description automatically generated

This screenshot shows the console.



Here is the XML test:

Graphical user interface, text, application, email

Description automatically generated

This screenshot shows the console:



Here is the String test:

Graphical user interface, text, application, email

Description automatically generated

This screenshot shows the console:

Logo

Description automatically generated

## DELETE

The doDelete method collects the data-type from the headers to be used when determining how to parse the request, the parsing is the same as the POST method. Finally, the parsed film will be sent to the DAO as a parameter to be deleted.

Here is the JSON test:

Graphical user interface, text, application, email

Description automatically generated

Here is the XML test:

Graphical user interface, text, application, email

Description automatically generated

Here is the String test:

Graphical user interface, text, application, email

Description automatically generated

## PUT

The doPut method collects the data-type from the headers to be used when determining how to parse the request, the parsing is the same as the POST method. Finally, the parsed film will be sent to the DAO as a parameter to be updated.

Here is the JSON test:

Graphical user interface, text, application, email

Description automatically generated

Here is a screenshot of the console :

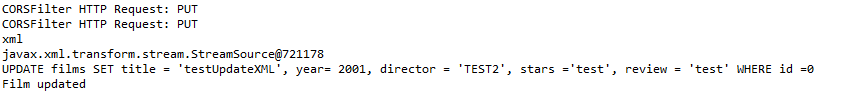


Here is the XML test:

Graphical user interface, text, application, email

Description automatically generated

Here is a screenshot of the console:

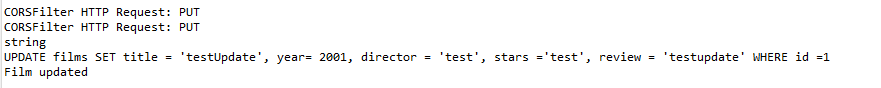


Here is the String test:

Graphical user interface, text, application, email

Description automatically generated

Here is a screenshot of the console:

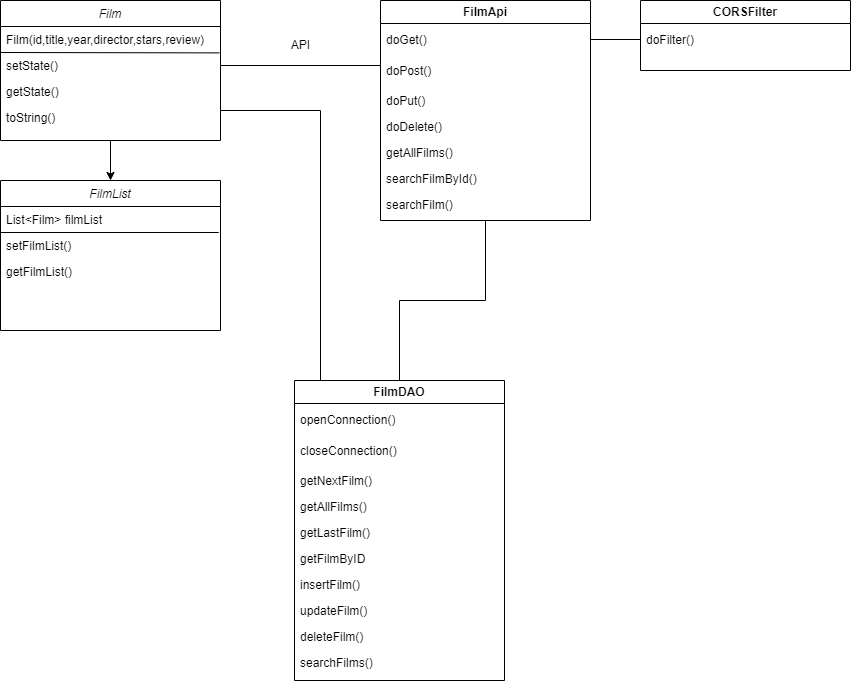


## Design Pattern

I as seen below; I have implemented a Film Class with a constructor for the necessary properties. I also have created a FilmList class which implements the Film class to create a list of Films.

These models are then used within the FilmApi and FilmDAO Class to store and pass Film data between each other.

I also have a CORSFilter which authorizes all domains to consume the content and regulates the allowed methods and headers.



A singleton is a creational design pattern that ensures that a class has only one instance, while providing global access to this instance. I used this design pattern (DP) within the FilmDAO class as it is needed to be called within FilmApi once, therefore a singleton, which allows global access and ensures only one instance is fitting.

## Refactoring Code

When developing the Client Side I found a few issues with the RESTFul Service code. The first problem occurred around the way I was receiving the request data, initially I was accessing the data as parameters with POST, DELETE, and PUT however I should have been sending my data as headers for these methods. Therefore, I updated the code to getHeaders over getParameters, which I kept using for GET.

## Overview

RESTful API design pattern is a good choice for building web services because it is easy to scale, use on different platforms, modular, easy to test, and flexible. It uses standard methods and is not tied to any specific language which allows for easy integration with other systems.

# JavaScript Front End Client Consuming RESTful Service

I chose to utilize a combination of React, AXIOS, Bootstrap, and Sass when developing the front-end of my application. React is a powerful JavaScript library that allows for the creation of reusable UI components and efficient management of application state. It utilizes a virtual DOM to optimize performance and make updates more efficient. AXIOS allows for easy handling of HTTP requests and responses, which is crucial when fetching data from a server or API. Bootstrap provides pre-designed UI components and a responsive grid system, which streamlines the design process and ensures a mobile-friendly layout. Lastly, Sass is a CSS preprocessor that enables the use of variables and functions, making the maintenance and organization of CSS more manageable. All of these frameworks were managed through NPM, which made it simple to install, update, and keep track of dependencies.

The following screenshots show the client reading films from the database.

Here is a screenshot showing the search functionality:

Text

Description automatically generated with medium confidence

Here is a screenshot getting all the films in the database:

Graphical user interface

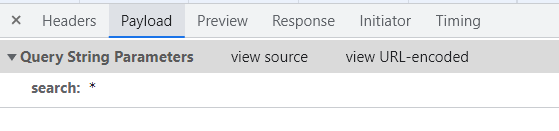
Description automatically generated with medium confidence

The following screenshot shows the JSON response from the API:

Text

Description automatically generated

The following screenshot shows the payload from the API:

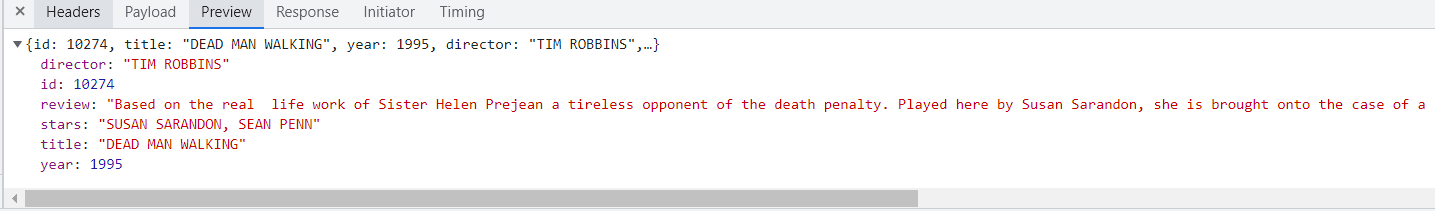


Here this a screenshot reading a film from id, you can also see the payload:

Graphical user interface, application, Word

Description automatically generated

Here is a screenshot showing the response from searching by id:



This a screenshot updating the film :

Graphical user interface, text, application

Description automatically generated

Here is a screenshot deleting a film :

Graphical user interface, text, application

Description automatically generated

# Cloud

# SOAP/WSDL

SOAP and WSDL are technical tools that allow different computer systems to exchange information and communicate with one another via the internet. SOAP is a method for formatting messages and WSDL is a language that describes the functions of a web service. Together, they make it easy for systems to connect and share data.

This screenshot shows the SOAP Client entering a new film into the database:

Graphical user interface, application

Description automatically generated

This screenshot shows the SOAP Client deleting a film from the database:

Graphical user interface, application, Word

Description automatically generated

These screenshots show the SOAP Client getting all films from the database for all data formats:

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

These screenshots show the SOAP Client searching films from the database for each data format:

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

These screenshots show the SOAP Client searching film by id from the database for each data

format:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application, Word

Description automatically generated

This screenshot shows the SOAP Client updating a film from the database:

Graphical user interface, application, Word

Description automatically generated