Developing, Deploying, Publishing and Finding Your Own Web Service – Critical Analysis

Nicholas Cowling - 17072242

Contents

Introduction

This document aims to cover the sound Software Engineering techniques deployed within the project solutions that improve code quality – ultimately making the code more readable and maintainable. The solutions that will be covered are: FilmCloudWebService, FilmHTTPWebService, FilmRESTfulWebService, FilmSOAPWebService.

The advantages of using these techniques will be discussed, compared to other possible approaches, and evaluated; screenshots are provided throughout to demonstrate discussed techniques in action within the solutions.

Commentary

To start, a discussion on the overall technologies such as HTTP, REST, SOAP, Cloud, format types (JSON, XML, Text), Data-Centric vs Content Centric – and more will be covered in this section.

What is an API?

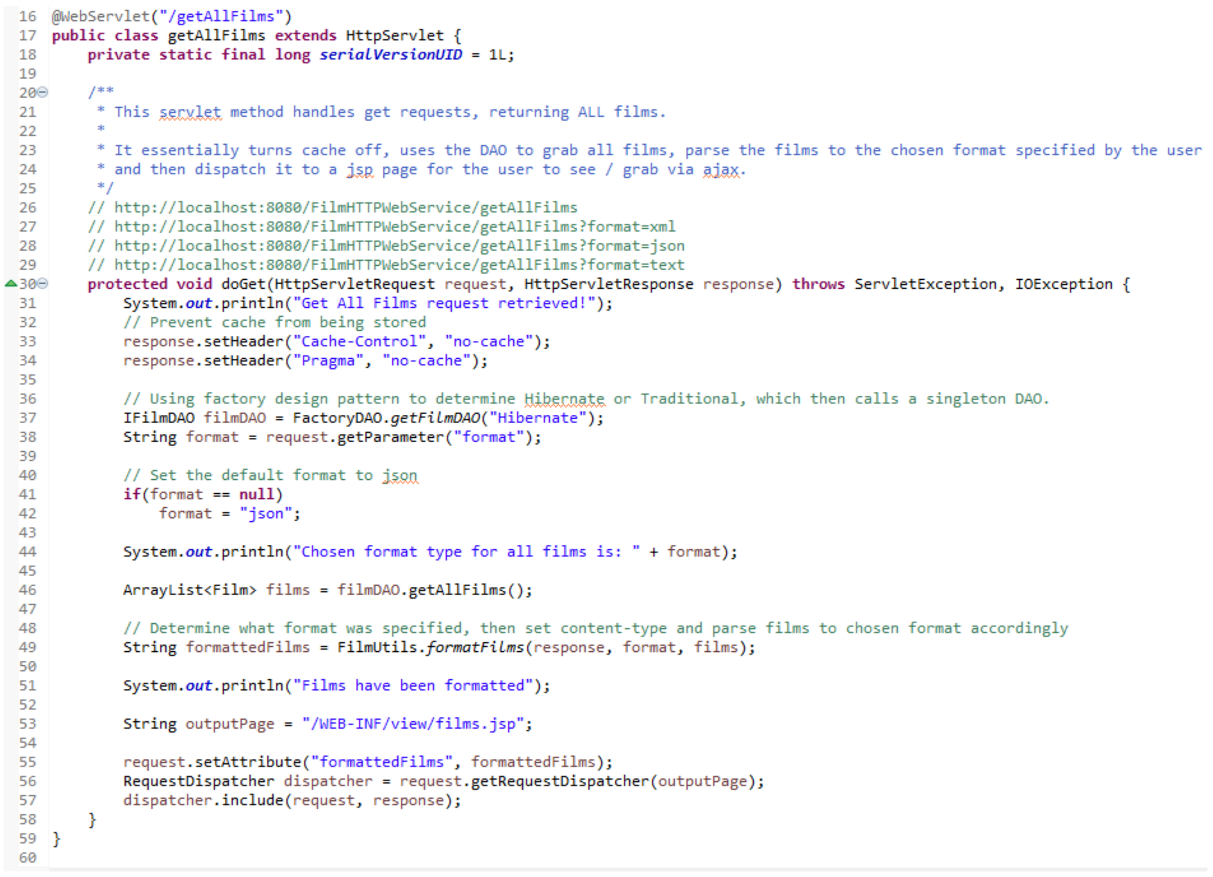
An API stands for "**A**pplication **P**rogramming **I**nterface" – it is a type of computing interface that defines the interactions between other software intermediaries. It specifies what requests can be made to it (and how they can be made), the different types of formats that can be used (XML, JSON etc..), conventions that should be followed, and more.

HTTP API

HTTP stands for "**H**yper**t**ext **T**ransfer **P**rotocol" – this is an application transfer protocol and is one way of transferring data on the World-Wide-Web. There are other protocols available other than HTTP, such as FTP, though they are not as widely used.

A HTTP API is therefore essentially any API that uses HTTP as the transfer protocol. Documentation will exist for these API's from the service provider, showing the available methods that can be invoked and the parameters needed.

The project "FilmHTTPWebService" is an example of a HTTP API that does not follow constraints unlike REST. An example of a Servlet from this solution can be seen below:



```java
@WebServlet("/getAllFilms")
public class getAllFilms extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * This servlet method handles get requests, returning ALL films.
     *
     * It essentially turns cache off, uses the DAO to grab all films, parse the films to the chosen format specified by the user
     * and then dispatch it to a jsp page for the user to see / grab via ajax.
     */
    // http://localhost:8080/FilmHTTPWebService/getAllFilms
    // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=xml
    // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=json
    // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=text
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Get All Films request retrieved!");
        // Prevent cache from being stored
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma", "no-cache");

        // Using factory design pattern to determine Hibernate or Traditional, which then calls a singleton DAO.
        IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
        String format = request.getParameter("format");

        // Set the default format to json
        if(format == null)
            format = "json";

        System.out.println("Chosen format type for all films is: " + format);

        ArrayList<Film> films = filmDAO.getAllFilms();

        // Determine what format was specified, then set content-type and parse films to chosen format accordingly
        String formattedFilms = FilmUtils.formatFilms(response, format, films);

        System.out.println("Films have been formatted");

        String outputPage = "/WEB-INF/view/films.jsp";

        request.setAttribute("formattedFilms", formattedFilms);
        RequestDispatcher dispatcher = request.getRequestDispatcher(outputPage);
        dispatcher.include(request, response);
    }
}
```

REST API

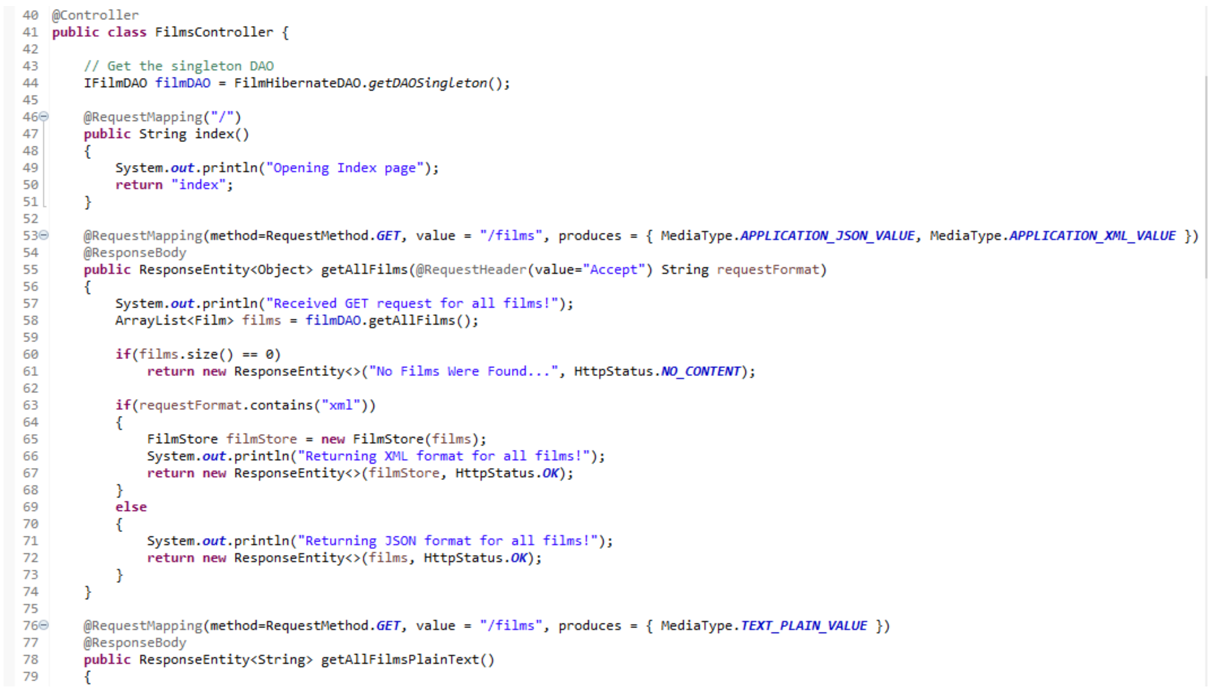
REST stands for "**Re**presentational **S**tate **T**ransfer" – it is a set of constraints laid out in Roy Fielding's dissertation, and aims to ensure a system that is scalable, easily extendable and fault-tolerant. The World-Wide-Web is an example a REST system. A RESTful API's transfer protocol is not defined within these constraints, so it is possible for RESTful API's to use FTP or other protocols; however, in reality, most RESTful API's do in-fact use the HTTP protocol because things such as infrastructure and libraries are already available for use.

RESTful services should ultimately:

- Expose directory structure-like URI's
- Use HTTP Methods explicitly (if using HTTP)
- Be stateless
- Transfer XML or JSON, or both

Documentation is required by the service provider to know the list of endpoints for this API, and for other information such as the type of data that is accepted.

The project "FilmRESTfulWebService" is an example of a RESTful API that follows the constraints of REST. A screenshot showing part of the controller for this project can be seen below (for retrieving all films), it essentially outputs the same result as the screenshot shown in the HTTP API section. To note, this project is using Spring.



```java
@Controller
public class FilmsController {

    // Get the singleton DAO
    IFilmDAO filmDAO = FilmHibernateDAO.getDAOSingleton();

    @RequestMapping("/")
    public String index()
    {
        System.out.println("Opening Index page");
        return "index";
    }

    @RequestMapping(method=RequestMethod.GET, value = "/films", produces = { MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
    @ResponseBody
    public ResponseEntity<Object> getAllFilms(@RequestHeader(value="Accept") String requestFormat)
    {
        System.out.println("Received GET request for all films!");
        ArrayList<Film> films = filmDAO.getAllFilms();

        if(films.size() == 0)
            return new ResponseEntity<>("No Films Were Found...", HttpStatus.NO_CONTENT);

        if(requestFormat.contains("xml"))
        {
            FilmStore filmStore = new FilmStore(films);
            System.out.println("Returning XML format for all films!");
            return new ResponseEntity<>(filmStore, HttpStatus.OK);
        }
        else
        {
            System.out.println("Returning JSON format for all films!");
            return new ResponseEntity<>(films, HttpStatus.OK);
        }
    }

    @RequestMapping(method=RequestMethod.GET, value = "/films", produces = { MediaType.TEXT_PLAIN_VALUE })
    @ResponseBody
    public ResponseEntity<String> getAllFilmsPlainText()
    {
```

SOAP API

SOAP stands for "**S**imple **O**bject **A**ccess **P**rotocol" – it is a messaging protocol that uses XML for both its request and response objects, and the way to use the methods is also stored in XML (known as WSDL's – Web-Service Description Language). These web services must comply with the SOAP specifications:

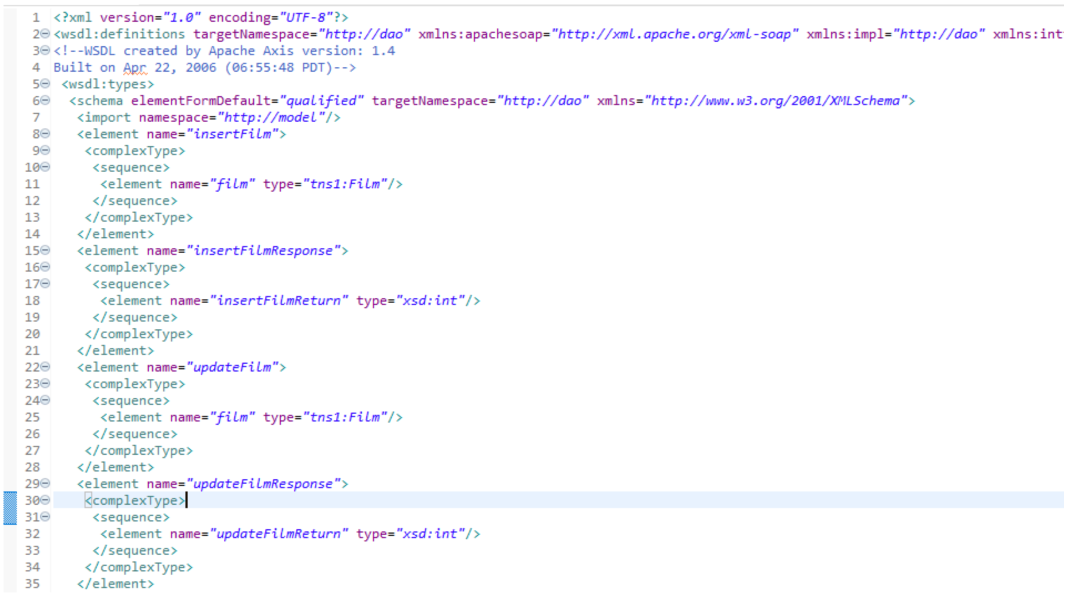
- Message Format – An XML-based document that follows the structure:
    - Envelope (the root element, wrapping all)
    - Header (an optional element containing information about the message, or for authentication)
    - Body (the data that is to be exchanged between both the client and the server).
- A Transport Protocol such as HTTP, but could also be SMTP, FTP…
- A WSDL file should be included
- UDDI (Universal Description, Discovery and Integration), for publishing and discovering information about a web service.

"FilmSOAPWebService" is an example of a SOAP server, and "FilmSOAPWebServiceClient" is an example of a SOAP client. These follow the constraints of SOAP specification.

WSDL's

WSDL's allow the service providers to list information on what methods are available, what inputs are required and what outputs are provided, as an XML document that is both self-describing and machine-readable. As a result, it allows the automatic invocation of a web service, and the clients can be designed independently. This document provides the descriptions of the: Service location; Communication protocol (usually HTTP); Functionality provided by the service; structure of XML required to make a service request; structure of XML for service response.

The below screenshot shows the WSDL file for a Data Accessor Object in my program:



```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://dao" xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://dao" xmlns:int
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://dao" xmlns="http://www.w3.org/2001/XMLSchema">
   <import namespace="http://model"/>
   <element name="insertFilm">
    <complexType>
     <sequence>
      <element name="film" type="tns1:Film"/>
     </sequence>
    </complexType>
   </element>
   <element name="insertFilmResponse">
    <complexType>
     <sequence>
      <element name="insertFilmReturn" type="xsd:int"/>
     </sequence>
    </complexType>
   </element>
   <element name="updateFilm">
    <complexType>
     <sequence>
      <element name="film" type="tns1:Film"/>
     </sequence>
    </complexType>
   </element>
   <element name="updateFilmResponse">
    <complexType>
     <sequence>
      <element name="updateFilmReturn" type="xsd:int"/>
     </sequence>
    </complexType>
   </element>
```

HTTP vs REST vs SOAP

A HTTP API is different to both REST and SOAP API's in that SOAP and REST have to rely on well-established rules that everyone has agreed to follow in order to exchange information; it may be the case that your HTTP API is close to actually being a true RESTful API. API's can be graded to the constraints of REST using the "Richardson Maturity Model".

Both SOAP and RESTful API's can be considered as HTTP API's if their transfer protocol used is HTTP; However, as previously discussed in their individual sections, SOAP and REST do not specify an exact transfer protocol, so it is possible to have a SOAP or RESTful service using FTP.

RESTful was created to address the problems of SOAP, and can support XML, JSON, YAML, or any other parser that the service provider wishes. This is beneficial over SOAP in terms of flexibility. Furthermore, REST is lightweight so is more likely to be used in high-rate applications where transaction times really matter.

Public facing applications prefer REST or standard HTTP calls over SOAP; For machine-to-machine communication at the enterprise level, SOAP is more likely to be used (for security reasons, must be parsed by software to understand – will be explained in later points).

REST makes full use of GET, POST, PUT, DELETE for the CRUD operations, whereas HTTP most commonly makes use of GET and POST (other requests such as PUT and DELETE are possible with HTTP, but HTML does not permit the use of these, AJAX must be used to utilize these requests.). On the other hand, SOAP only utilizes the POST request for all forms of communication.

Most new API's are RESTful interfaces, though there is still many legacy systems that use SOAP and so have to be maintained, and may not switch because it costs too much money to rebuild a new API, or because SOAP is better in regards to security.

SOAP is useful for places such finance or insurance sectors that require high security and complex transactions - like banks, as it has W3C XML encryption standards for encrypting XML content (toolkits available from vendors such as IBM). There are W3C digital signature standards for authentication on who is using the web service, and if they are authorized to do so. Whilst REST does have some of these that have been re-implemented for JSON, they are not widely used and the implementations are not as good. SOAP further provides reliable messaging (with WS-Reliable messaging). PayPal is an example of a public SOAP API, allowing you to accept PayPal and credit card payments, add PayPal buttons to your website, allow users to login with PayPal, and more.

Cloud Computing

Cloud computing can be defined as the delivery of computing services, such as: servers storage, databases, networking, software, intelligence and analytics over the Internet. Examples of this include Microsoft Azure, Google Cloud, Amazon AWS. There are many benefits that come with this technology:

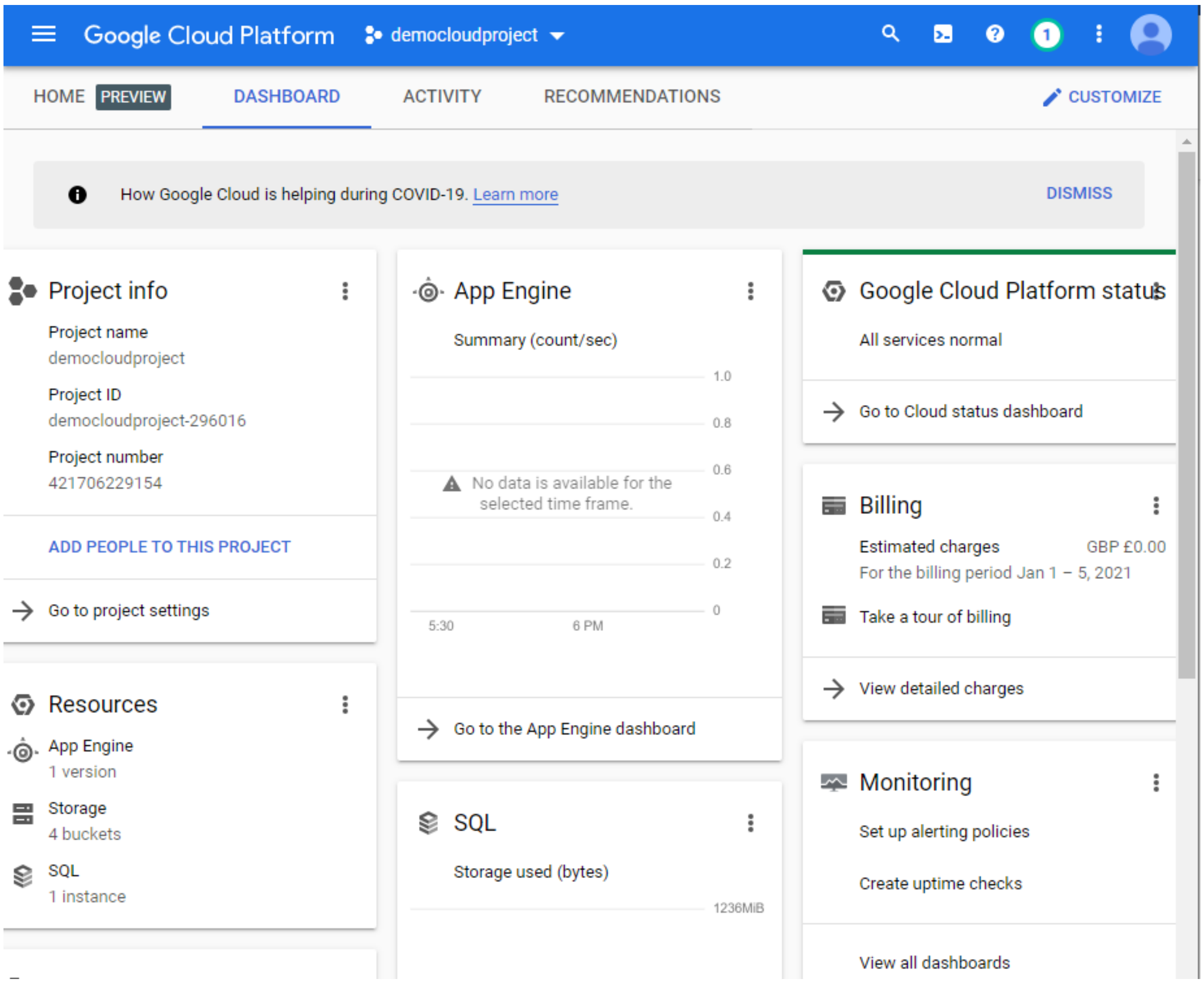
- Transparently scalable – Can handle large variations in load transparently, and the server capacity can vary based on the traffic without any need for human intervention (scalability)
- No need to buy IT resources – Hardware such as the servers, storage capacity, networks, routers and load balancers are provided with the Cloud. Furthermore, software applications, database servers, application servers and development software are provided depending on what services you select
- A seemingly infinite pool of resources – It is based on virtualization technology, allowing multiple virtual machines to run on a single physical machine; this means resources can be shared to improve utilization
- Self-service – it is possible to add new resources or remove them immediately. Additionally, there is no management and maintenance required, as the cloud provider does this for you.

There are three classifications for categorizing cloud computing services:

- Infrastructure as a Service (IaaS) – providing virtualized resources / machines that have operating systems and IPs, over the internet. These resources are delivered as a service, and includes servers, networks, memory, CPU, and so on.
    - Examples include Amazon Elastic Compute Cloud (EC2), Microsoft and Google have their version too.
    - The key feature of IaaS is the instant provisioning – the new resources are available within minutes to start using.
- Platform as a Service (PaaS) – providing a computing platform with a user interface over the internet. Software is already included for development, middleware (Web servers & messaging systems for example), and deployment. There is no access to the operating system or the IP.
    - This is beneficial as it is easy to use, allowing organizations to concentrate on the application development without having to maintain or buy further resources that are required to execute the software
    - It has automatic scalability built-in, is reliable, has good performance and security of its major providers (Amazon, Google, Microsoft etc..), and is overall cost-efficient
- Software as a Service (SaaS) – providing a business platform (i.e. a piece of software) over the internet. There is no need to install software on the client computer, and it is often accessed through web browsers.
    - This is beneficial as you receive the latest updates transparently and there is no in-house administration costs or installation fees.
    - Vendors such as Microsoft now offer traditional software as SaaS (such as Office 365, for Microsoft Word, Microsoft PowerPoint etc…)

The “FilmCloudWebService” project uses PaaS with Google App Engine, providing us with tools and a development stack for Java. Data storage services are also available via Cloud SQL. This project solution therefore benefits from the aforementioned points regarding PaaS just above.

The screenshot below shows the Google Cloud Console relating to the project set up for "FilmCloudWebService". It displays an App Engine currently in use, multiple storage buckets, and an SQL instance (the Cloud database).


Google Cloud Platform · democloudproject

HOME PREVIEW · DASHBOARD · ACTIVITY · RECOMMENDATIONS · CUSTOMIZE

How Google Cloud is helping during COVID-19. Learn more · DISMISS

Project info

Project name
democloudproject

Project ID
democloudproject-296016

Project number
421706229154

ADD PEOPLE TO THIS PROJECT

→ Go to project settings

Resources

App Engine
1 version

Storage
4 buckets

SQL
1 instance

App Engine

Summary (count/sec)

1.0 · 0.8 · 0.6 · 0.4 · 0.2 · 0

No data is available for the selected time frame.

5:30 · 6 PM

→ Go to the App Engine dashboard

SQL

Storage used (bytes)

1236MiB

Google Cloud Platform status

All services normal

→ Go to Cloud status dashboard

Billing

Estimated charges	GBP £0.00

For the billing period Jan 1 – 5, 2021

Take a tour of billing

→ View detailed charges

Monitoring

Set up alerting policies

Create uptime checks

View all dashboards

Drawbacks however do exist when it comes to using Public Clouds like this:

- Cannot have total control of the security, governance and the reliability. The provider must be trusted with this.
- Compromises might have to be made to meet needs

Private Clouds, Community Clouds, and Hybrid Clouds are alternative approaches to deploying on the Cloud.

In Private Clouds, the infrastructure and the services are dedicated to a single organization on a standalone network; it essentially emulates a public cloud on a private network – no resources are shared with any other organizations unlike with Public Clouds. These Private Clouds can be managed by either the organization itself, or a third-party. In comparison to Public Clouds:

- They minimize the potential problems one may encounter with Public Clouds (things such as data security, corporate governance and reliability)
- They are easier to comply with the corporate security standards, policies and regulatory compliances that an organization may be under.
- One drawback is that they do require experts to set up, maintain and administrate the cloud. However, some or all of this could be outsourced, so may not be much of an issue.

In Community Clouds, the infrastructure and services are shared by multiple organizations who have similar needs. This is commonly seen in organizations within a specific business sector, such as within Government agencies, travel agencies, and health-care professionals. These Community Clouds can be managed by either the organizations themselves, or by a third party. In comparison to Private Clouds:

- Resources and services can be used to their fullest potential by similar businesses
- Reduce costs by sharing the infrastructure and services (alternatively, each organization would otherwise have to repeat the same infrastructure internally, wasting money)
- Standards can be implemented much more easily across the organizations and business sectors.

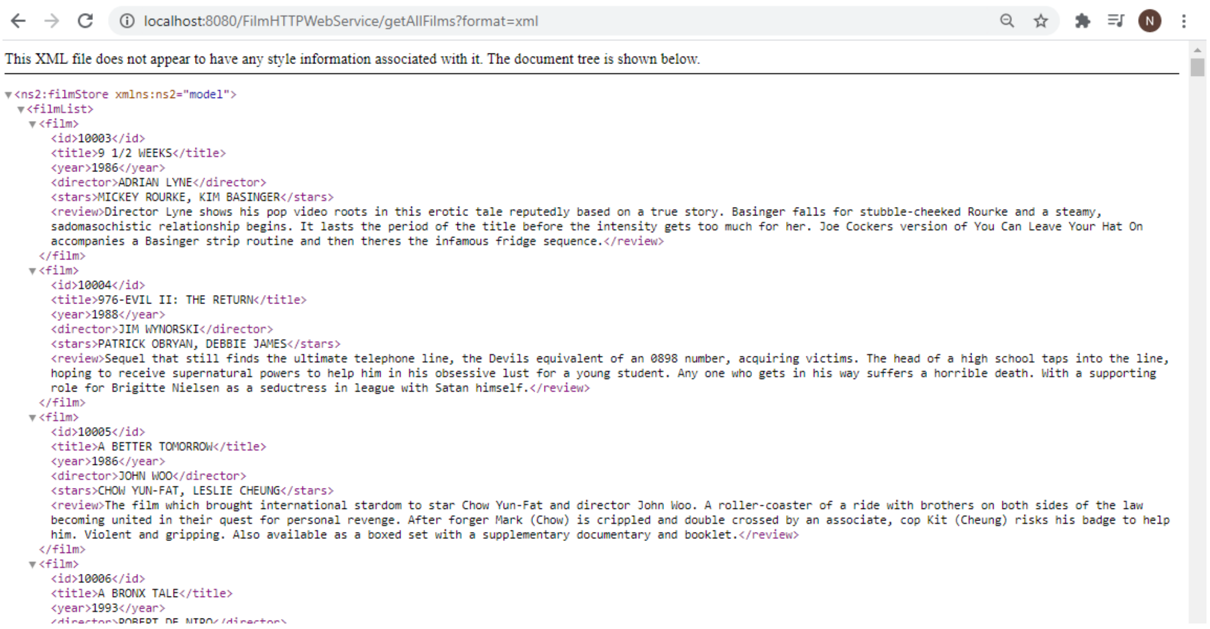
A Hybrid Cloud is essentially a mixture of two or more models (either Public, Private or Community). These Clouds are bound together, so Private Clouds can host their core applications and sensitive data in-house, and then utilize non-core applications in a Public or Community Cloud. Compared to other models:

- The features of each type of cloud (Public, Private, Community) can be exploited with Hybrid Clouds, enabling organizations to achieve their desired business needs.
- Public Clouds can be used when the computing capacity on the Private Cloud spikes. This is known as "Cloud Bursting".

Different Format Types (XML, JSON, Text)

XML stands for E**x**tensible **M**arkup **L**anguage. It is a markup language that defines the rules for encoding documents in a format that is both human-readable and machine-readable. It does not specify the tag set or the actual grammar of the language itself. It is often seen in use in configuration files (such as in Spring or Hibernate), transactions on the web such as SOAP for message exchanges and more. The reason XML was used in the data layer originally was because of its ability to use custom tags; it is more commonly seen in other areas such as UI development processes now due to it being a markup language – a perfect platform-agnostic choice for displaying and styling data.

The below screenshot shows a valid XML format returned by the server in "FilmHTTPWebService" project when retrieving all films and requesting XML:



localhost:8080/FilmHTTPWebService/getAllFilms?format=xml

This XML file does not appear to have any style information associated with it. The document tree is shown below.

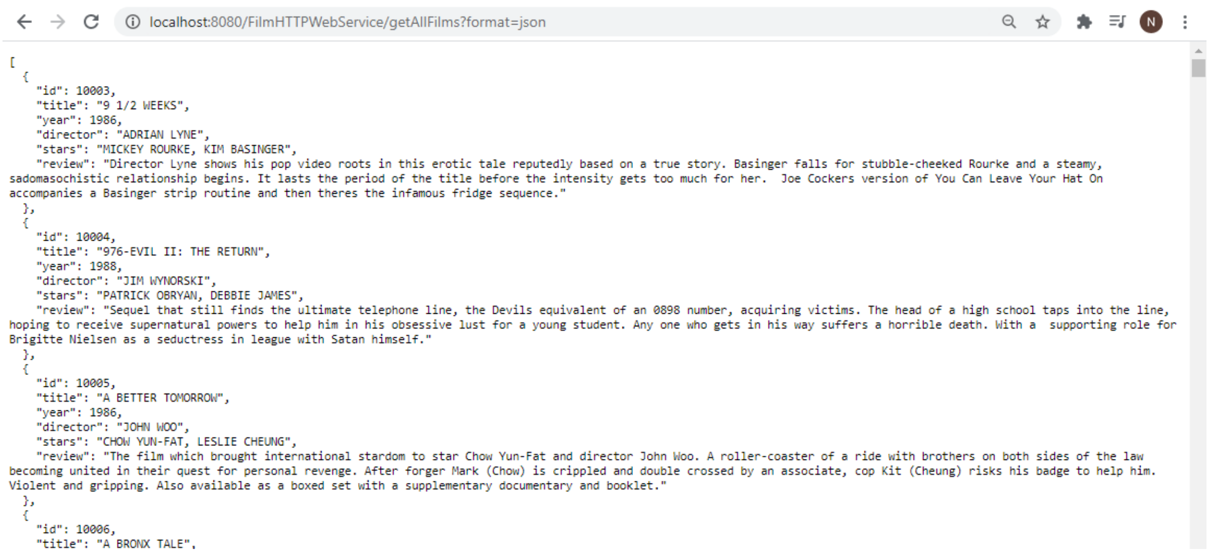
```
▼<ns2:filmStore xmlns:ns2="model">
  ▼<filmList>
    ▼<film>
        <id>10003</id>
        <title>9 1/2 WEEKS</title>
        <year>1986</year>
        <director>ADRIAN LYNE</director>
        <stars>MICKEY ROURKE, KIM BASINGER</stars>
        <review>Director Lyne shows his pop video roots in this erotic tale reputedly based on a true story. Basinger falls for stubble-cheeked Rourke and a steamy,
        sadomasochistic relationship begins. It lasts the period of the title before the intensity gets too much for her. Joe Cockers version of You Can Leave Your Hat On
        accompanies a Basinger strip routine and then theres the infamous fridge sequence.</review>
      </film>
    ▼<film>
        <id>10004</id>
        <title>976-EVIL II: THE RETURN</title>
        <year>1988</year>
        <director>JIM WYNORSKI</director>
        <stars>PATRICK OBRYAN, DEBBIE JAMES</stars>
        <review>Sequel that still finds the ultimate telephone line, the Devils equivalent of an 0898 number, acquiring victims. The head of a high school taps into the line,
        hoping to receive supernatural powers to help him in his obsessive lust for a young student. Any one who gets in his way suffers a horrible death. With a supporting
        role for Brigitte Nielsen as a seductress in league with Satan himself.</review>
      </film>
    ▼<film>
        <id>10005</id>
        <title>A BETTER TOMORROW</title>
        <year>1986</year>
        <director>JOHN WOO</director>
        <stars>CHOW YUN-FAT, LESLIE CHEUNG</stars>
        <review>The film which brought international stardom to star Chow Yun-Fat and director John Woo. A roller-coaster of a ride with brothers on both sides of the law
        becoming united in their quest for personal revenge. After forger Mark (Chow) is crippled and double crossed by an associate, cop Kit (Cheung) risks his badge to help
        him. Violent and gripping. Also available as a boxed set with a supplementary documentary and booklet.</review>
      </film>
    ▼<film>
        <id>10006</id>
        <title>A BRONX TALE</title>
        <year>1993</year>
        <director>ROBERT DE NIRO</director>
```

JSON stands for **J**ava**S**cript **O**bject **N**otation. It is a lightweight format designed for data and data only; because of this, it is widely used to transfer data between applications via API's. It is built on the two following structures:

- A collection of name / value pairs (can be seen as a hash-table for example in some languages)
- An ordered list of values (can be seen as an array in some languages)

This simple map format (key-value pairs) creates a high level of predictability and is less verbose than XML, resulting in faster transfer speeds. The main reason for JSON's popularity over XML is because of how much more lightweight it is, and because it is formatted in a way that is (arguably) more human readable.

The screenshot below shows a valid JSON format returned by the server in the "FilmHTTPWebService" project when retrieving all films with the format as JSON.



localhost:8080/FilmHTTPWebService/getAllFilms?format=json

```
[
  {
    "id": 10003,
    "title": "9 1/2 WEEKS",
    "year": 1986,
    "director": "ADRIAN LYNE",
    "stars": "MICKEY ROURKE, KIM BASINGER",
    "review": "Director Lyne shows his pop video roots in this erotic tale reputedly based on a true story. Basinger falls for stubble-cheeked Rourke and a steamy,
sadomasochistic relationship begins. It lasts the period of the title before the intensity gets too much for her.  Joe Cockers version of You Can Leave Your Hat On
accompanies a Basinger strip routine and then theres the infamous fridge sequence."
  },
  {
    "id": 10004,
    "title": "976-EVIL II: THE RETURN",
    "year": 1988,
    "director": "JIM WYNORSKI",
    "stars": "PATRICK OBRYAN, DEBBIE JAMES",
    "review": "Sequel that still finds the ultimate telephone line, the Devils equivalent of an 0898 number, acquiring victims. The head of a high school taps into the line,
hoping to receive supernatural powers to help him in his obsessive lust for a young student. Any one who gets in his way suffers a horrible death. With a  supporting role for
Brigitte Nielsen as a seductress in league with Satan himself."
  },
  {
    "id": 10005,
    "title": "A BETTER TOMORROW",
    "year": 1986,
    "director": "JOHN WOO",
    "stars": "CHOW YUN-FAT, LESLIE CHEUNG",
    "review": "The film which brought international stardom to star Chow Yun-Fat and director John Woo. A roller-coaster of a ride with brothers on both sides of the law
becoming united in their quest for personal revenge. After forger Mark (Chow) is crippled and double crossed by an associate, cop Kit (Cheung) risks his badge to help him.
Violent and gripping. Also available as a boxed set with a supplementary documentary and booklet."
  },
  {
    "id": 10006,
    "title": "A BRONX TALE",
```

Some benefits of XML are:

- It can handle mixed content better than JSON. Additionally, meta-data cannot be added to JSON without creating an additional key-value pair to hold it. XML on the other hand provides reliable tools for parsing and validating XML in an array of different platforms and languages
- XML can be transferred into other formats rather easily, using XLST (Extensible Stylesheet Language Transformations) for example
- It can be argued that a well-structured xml document is actually still simple for humans to read and modify – it isn't necessarily the case that JSON is better here
- Encoding is already handled
- The features XML provides can significantly reduce software risk which is partly why it is still used in enterprise

Some drawbacks of XML are:

- It's verbose syntax
- It will break if the document was not well formed (though this could also be a positive, ensuring data is structurally correct)
- XML is prone to redundancies in structure as a data layer, and so is more fit for purpose in the UI layer rather than in the data layer
- Using XML in the data layer can result in unnecessarily expensive costs due to size

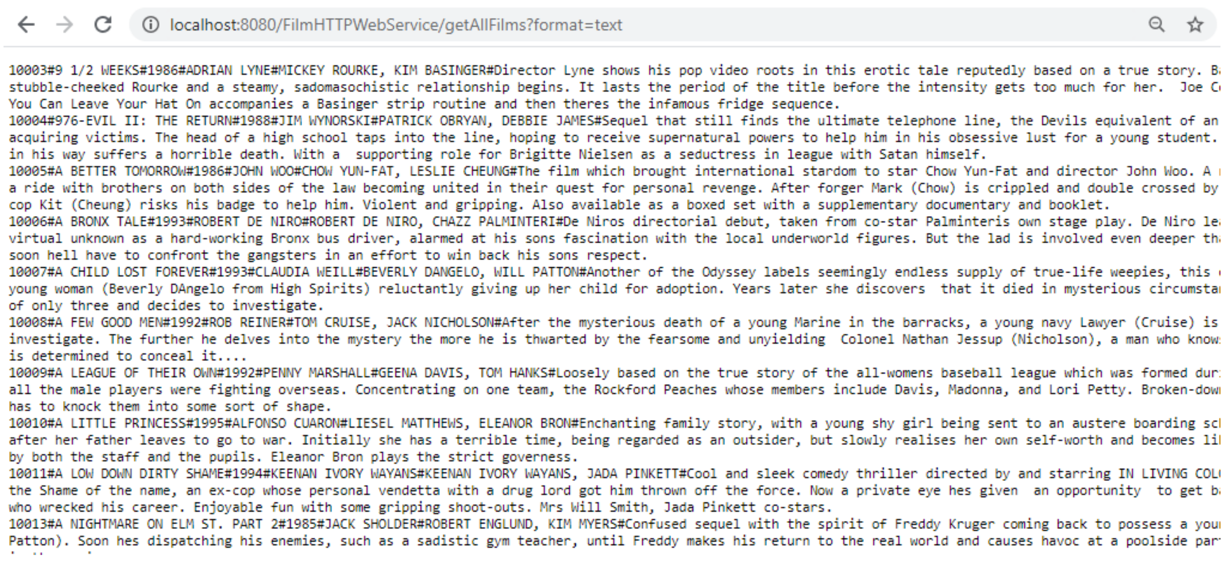
Some benefits of JSON are:

- It is (arguably) easier to understand by humans, and so is easier to work with
- The simple structures it employs (key-value pairs), being designed for data and data only, makes its data size smaller compared to XML (more lightweight) – resulting in much faster transmissions across the internet
- JSON is extensively supported in JavaScript. It can almost be considered "native" to the language, with the built-in libraries for it
- JSON distinguishes between data types (strings, numbers, Boolean), unlike XML that only represents data as plain text (data types can be defined in XML through the XML Schema, but can be complicated, whereas JSON's simple structure makes it easy to do so)

Some drawbacks of JSON are:

- The data can be more difficult to describe when compared to XML as it is one-dimensional (XML can use metadata to describe the data as required)
- There is a lack of formatting validation, meaning that the wrong data structure can be passed into an API, whereas XML would automatically detect such issues and throw errors

Formatting data into a text format with a simple delimiter is useful for applications that do not possess the ability to parse more complex formats such as XML or JSON – an example being spreadsheets. Having the ability to select this format opens the API up to more platforms. The below screenshot is an example of a simple text format returned by the server when retrieving all films in the "FilmHTTPWebService" project (delimiter used is a hashtag '#'):



```
10003#9 1/2 WEEKS#1986#ADRIAN LYNE#MICKEY ROURKE, KIM BASINGER#Director Lyne shows his pop video roots in this erotic tale reputedly based on a true story. B
stubble-cheeked Rourke and a steamy, sadomasochistic relationship begins. It lasts the period of the title before the intensity gets too much for her.  Joe C
You Can Leave Your Hat On accompanies a Basinger strip routine and then theres the infamous fridge sequence.
10004#976-EVIL II: THE RETURN#1988#JIM WYNORSKI#PATRICK OBRYAN, DEBBIE JAMES#Sequel that still finds the ultimate telephone line, the Devils equivalent of an
acquiring victims. The head of a high school taps into the line, hoping to receive supernatural powers to help him in his obsessive lust for a young student.
in his way suffers a horrible death. With a  supporting role for Brigitte Nielsen as a seductress in league with Satan himself.
10005#A BETTER TOMORROW#1986#JOHN WOO#CHOW YUN-FAT, LESLIE CHEUNG#The film which brought international stardom to star Chow Yun-Fat and director John Woo. A
a ride with brothers on both sides of the law becoming united in their quest for personal revenge. After forger Mark (Chow) is crippled and double crossed by
cop Kit (Cheung) risks his badge to help him. Violent and gripping. Also available as a boxed set with a supplementary documentary and booklet.
10006#A BRONX TALE#1993#ROBERT DE NIRO#ROBERT DE NIRO, CHAZZ PALMINTERI#De Niros directorial debut, taken from co-star Palminteris own stage play. De Niro le
virtual unknown as a hard-working Bronx bus driver, alarmed at his sons fascination with the local underworld figures. But the lad is involved even deeper th
soon hell have to confront the gangsters in an effort to win back his sons respect.
10007#A CHILD LOST FOREVER#1993#CLAUDIA WEILL#BEVERLY DANGELO, WILL PATTON#Another of the Odyssey labels seemingly endless supply of true-life weepies, this
young woman (Beverly DAngelo from High Spirits) reluctantly giving up her child for adoption. Years later she discovers  that it died in mysterious circumsta
of only three and decides to investigate.
10008#A FEW GOOD MEN#1992#ROB REINER#TOM CRUISE, JACK NICHOLSON#After the mysterious death of a young Marine in the barracks, a young navy Lawyer (Cruise) is
investigate. The further he delves into the mystery the more he is thwarted by the fearsome and unyielding  Colonel Nathan Jessup (Nicholson), a man who know
is determined to conceal it....
10009#A LEAGUE OF THEIR OWN#1992#PENNY MARSHALL#GEENA DAVIS, TOM HANKS#Loosely based on the true story of the all-womens baseball league which was formed dur
all the male players were fighting overseas. Concentrating on one team, the Rockford Peaches whose members include Davis, Madonna, and Lori Petty. Broken-dow
has to knock them into some sort of shape.
10010#A LITTLE PRINCESS#1995#ALFONSO CUARON#LIESEL MATTHEWS, ELEANOR BRON#Enchanting family story, with a young shy girl being sent to an austere boarding sc
after her father leaves to go to war. Initially she has a terrible time, being regarded as an outsider, but slowly realises her own self-worth and becomes li
by both the staff and the pupils. Eleanor Bron plays the strict governess.
10011#A LOW DOWN DIRTY SHAME#1994#KEENAN IVORY WAYANS#KEENAN IVORY WAYANS, JADA PINKETT#Cool and sleek comedy thriller directed by and starring IN LIVING COL
the Shame of the name, an ex-cop whose personal vendetta with a drug lord got him thrown off the force. Now a private eye hes given  an opportunity  to get b
who wrecked his career. Enjoyable fun with some gripping shoot-outs. Mrs Will Smith, Jada Pinkett co-stars.
10013#A NIGHTMARE ON ELM ST. PART 2#1985#JACK SHOLDER#ROBERT ENGLUND, KIM MYERS#Confused sequel with the spirit of Freddy Kruger coming back to possess a you
Patton). Soon hes dispatching his enemies, such as a sadistic gym teacher, until Freddy makes his return to the real world and causes havoc at a poolside par
```

Design Patterns

This section aims to cover the software design patterns used in my solutions, as well as some that were not used that are common in the working world and have experienced myself during placement.

Model-View-Controller (MVC)

MVC is a commonly used design pattern for separating an application into three main components, these are: the model, the view, and the controller. Its main purpose is to separate the view from the domain logic with these components, and by doing so we can avoid tying the domain logic to a particular visual representation. This is beneficial because the domain logic represents the theoretical model of a real-world business process, and any changes in the way that the user views it should not affect it.

There are two primary use cases that this architecture is used for:

- Traditional webservers that return HTML for each user request (the type of web server that may also use a 3-tier layered architecture)
    - Though this is still prevalent, it is becoming less popular
- Graphical user interfaces on both mobile, desktop and Web-Apps

The "Model" is the domain logic – our model for some sort of real-life business process. It is ultimately concerned with the business entities, not the appearance of the data itself.

The "View" displays the information to the user; everything related to the appearance (layout, style, text, time formatting) is controlled here. The view should consist of little logic. Also, it is possible to have multiple views of the same model.

The "Controller" is the intermediary between the views and models. It might update or receive updates from the model, or may update the view, or choose a different view to be displayed – this essentially ensures decoupling.

MVC benefits from:

- The separation of concerns (ensuring decoupling)

- Easier maintenance (updating, debugging) due to separation of concerns
- Supports teams; multiple developers can work at the same time on different aspects without any issues
- Development being faster

Some drawbacks of MVC are:

- The architecture itself has to be understood by the developers – it may be hard to understand
- Some criticisms of the Controller and View being tightly coupled (depending on the interpretation of MVC), resulting in unit testing being harder

Alternative approaches to MVC include: Model-View-Presenter (MVP) and Model-View-ViewModel(MVVM).

MVP emphasizes further decoupling – using a presenter instead of a controller. A presenter defines an interface that the views can then implement, this is so it does not know anything about the views. Additionally, MVP emphasizes that the views should not see the model directly – the raw data should be filtered.

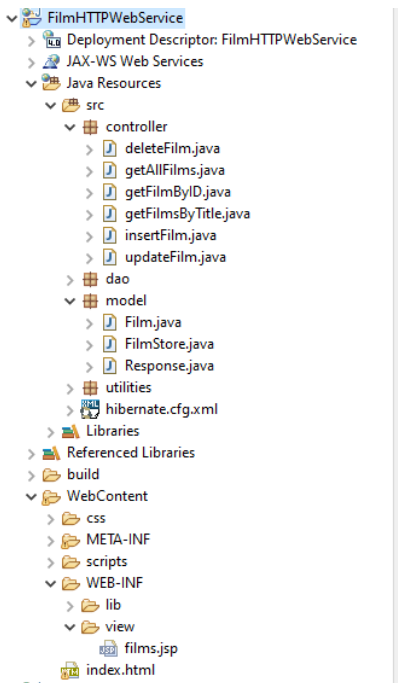
MVVM focuses on the separation of views from the models just like MVP, but approaches this differently by creating a new model called the "ViewModel". This ViewModel should only contain parts of the model that are actually needed by the views. They can also update and be updated by the full model.

Example of MVC Used in Solutions

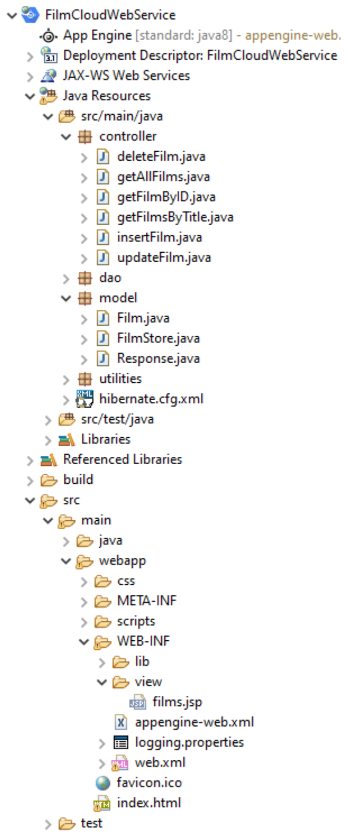
The projects "FilmHTTPWebService" and "FilmCloudWebService" have used this pattern for the retrieval of films. "FilmRESTfulWebService" separates the application with the same structure, but does not use a view itself for displaying all data, rather just returns raw data straight from the controller (there is no point using the view in this case for all this raw data – it is another thing to maintain, and for a RESTful API, is not necessary). These projects contain (listing what's related to MVC):

- A "Controller" package – containing all the Servlets required, such as "getAllFilms.java"
- A "Model" package – containing the domain logic, namely "Film.java" and "FilmStore.Java"
- A "View" folder located in "WEB-INF"
- A "dao" package – containing our data accessor objects (will be discussed in the DAO section)
- A "utilities" package – containing all of our utility methods (will be discussed in the refactoring / code-reuse section)

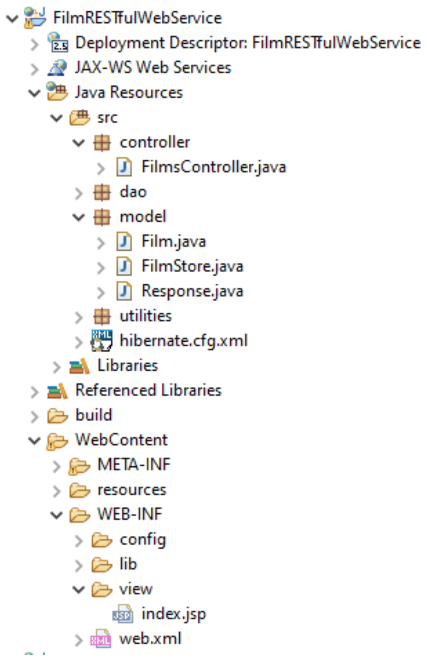
Below shows screenshots of the layout for these projects:



```
FilmHTTPWebService
  Deployment Descriptor: FilmHTTPWebService
  JAX-WS Web Services
  Java Resources
    src
      controller
        deleteFilm.java
        getAllFilms.java
        getFilmByID.java
        getFilmsByTitle.java
        insertFilm.java
        updateFilm.java
      dao
      model
        Film.java
        FilmStore.java
        Response.java
      utilities
      hibernate.cfg.xml
    Libraries
  Referenced Libraries
  build
  WebContent
    css
    META-INF
    scripts
    WEB-INF
      lib
      view
        films.jsp
    index.html
```

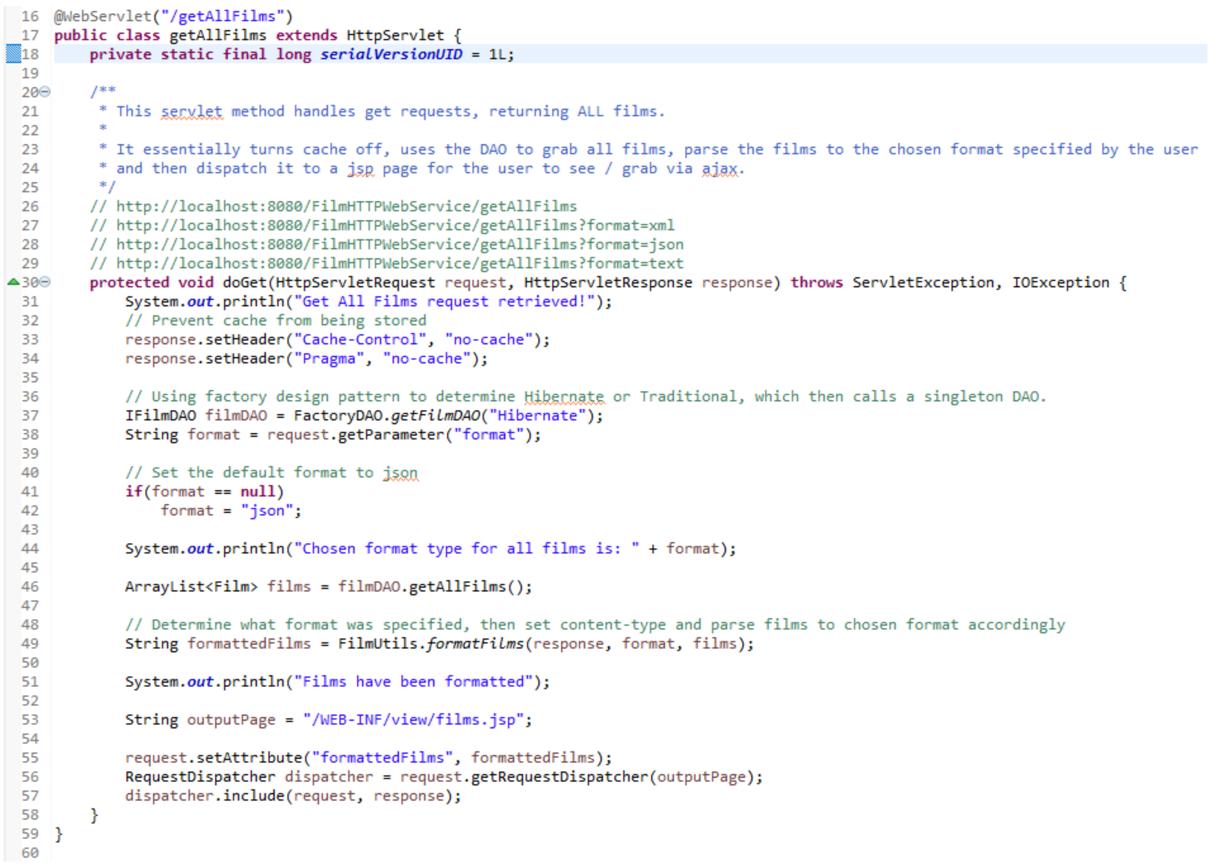


```
FilmCloudWebService
  App Engine [standard: java8] - appengine-web.
  Deployment Descriptor: FilmCloudWebService
  JAX-WS Web Services
  Java Resources
    src/main/java
      controller
        deleteFilm.java
        getAllFilms.java
        getFilmByID.java
        getFilmsByTitle.java
        insertFilm.java
        updateFilm.java
      dao
      model
        Film.java
        FilmStore.java
        Response.java
      utilities
      hibernate.cfg.xml
    src/test/java
    Libraries
  Referenced Libraries
  build
  src
    main
      java
      webapp
        css
        META-INF
        scripts
        WEB-INF
          lib
          view
            films.jsp
          appengine-web.xml
          logging.properties
          web.xml
        favicon.ico
        index.html
    test
```



```
FilmRESTfulWebService
  Deployment Descriptor: FilmRESTfulWebService
  JAX-WS Web Services
  Java Resources
    src
      controller
        FilmsController.java
      dao
      model
        Film.java
        FilmStore.java
        Response.java
      utilities
      hibernate.cfg.xml
    Libraries
  Referenced Libraries
  build
  WebContent
    META-INF
    resources
    WEB-INF
      config
      lib
      view
        index.jsp
      web.xml
```

The below screenshot from the “FilmHTTPWebService” project shows the “getAllFilms” Servlet (a Controller) creating an ArrayList of the Films (a Model), dispatching to a .jsp page “films” (a View).



```java
@WebServlet("/getAllFilms")
public class getAllFilms extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * This servlet method handles get requests, returning ALL films.
     *
     * It essentially turns cache off, uses the DAO to grab all films, parse the films to the chosen format specified by the user
     * and then dispatch it to a jsp page for the user to see / grab via ajax.
     */
    // http://localhost:8080/FilmHTTPWebService/getAllFilms
    // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=xml
    // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=json
    // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=text
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Get All Films request retrieved!");
        // Prevent cache from being stored
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma", "no-cache");

        // Using factory design pattern to determine Hibernate or Traditional, which then calls a singleton DAO.
        IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
        String format = request.getParameter("format");

        // Set the default format to json
        if(format == null)
            format = "json";

        System.out.println("Chosen format type for all films is: " + format);

        ArrayList<Film> films = filmDAO.getAllFilms();

        // Determine what format was specified, then set content-type and parse films to chosen format accordingly
        String formattedFilms = FilmUtils.formatFilms(response, format, films);

        System.out.println("Films have been formatted");

        String outputPage = "/WEB-INF/view/films.jsp";

        request.setAttribute("formattedFilms", formattedFilms);
        RequestDispatcher dispatcher = request.getRequestDispatcher(outputPage);
        dispatcher.include(request, response);
    }
}
```

Data Access Object (DAO)

The Data Access Object pattern provides an abstract interface to some type of persistence mechanism such as a database. This essentially separates the low-level code that is accessing the API / operations, from the high-level (business-logic) code, hiding the implementation of the persistence with an interface (known as programming to an interface).

Some benefits of using a DAO are:

- It promotes loose coupling as you are programming to an interface, promoting the OOP design principles: Encapsulation (having it in a separate class with methods), Abstraction (Using an interface to communicate with it), Polymorphism for switching the implementation out for another DAO
- Any changes required to the persistence mechanism means only the DAO layer must be changed; the higher-level code where the DAO interface is used does not have to be changed
- Unit tests are easier as a Mock DAO can be created to avoid the actual connection to a database to run unit tests
- The general interface used can be applied to other persistence mechanisms and the DAO's can then be switched if required without effecting the higher-level code

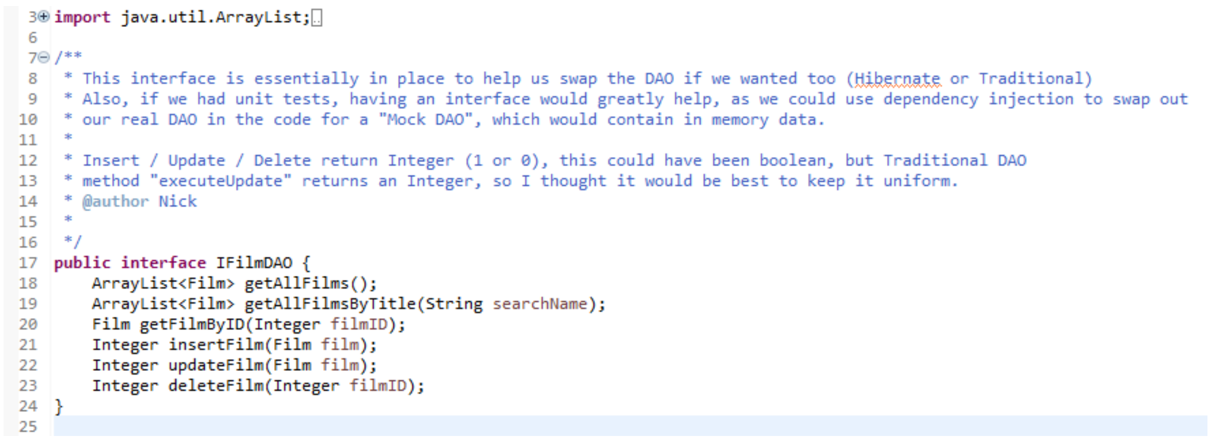
Some drawbacks of using a DAO are:

- Having to maintain another layer of abstraction (knit-picking here, not really many drawbacks, as all the benefits ultimately help maintenance)

Example of DAO Used in Solutions

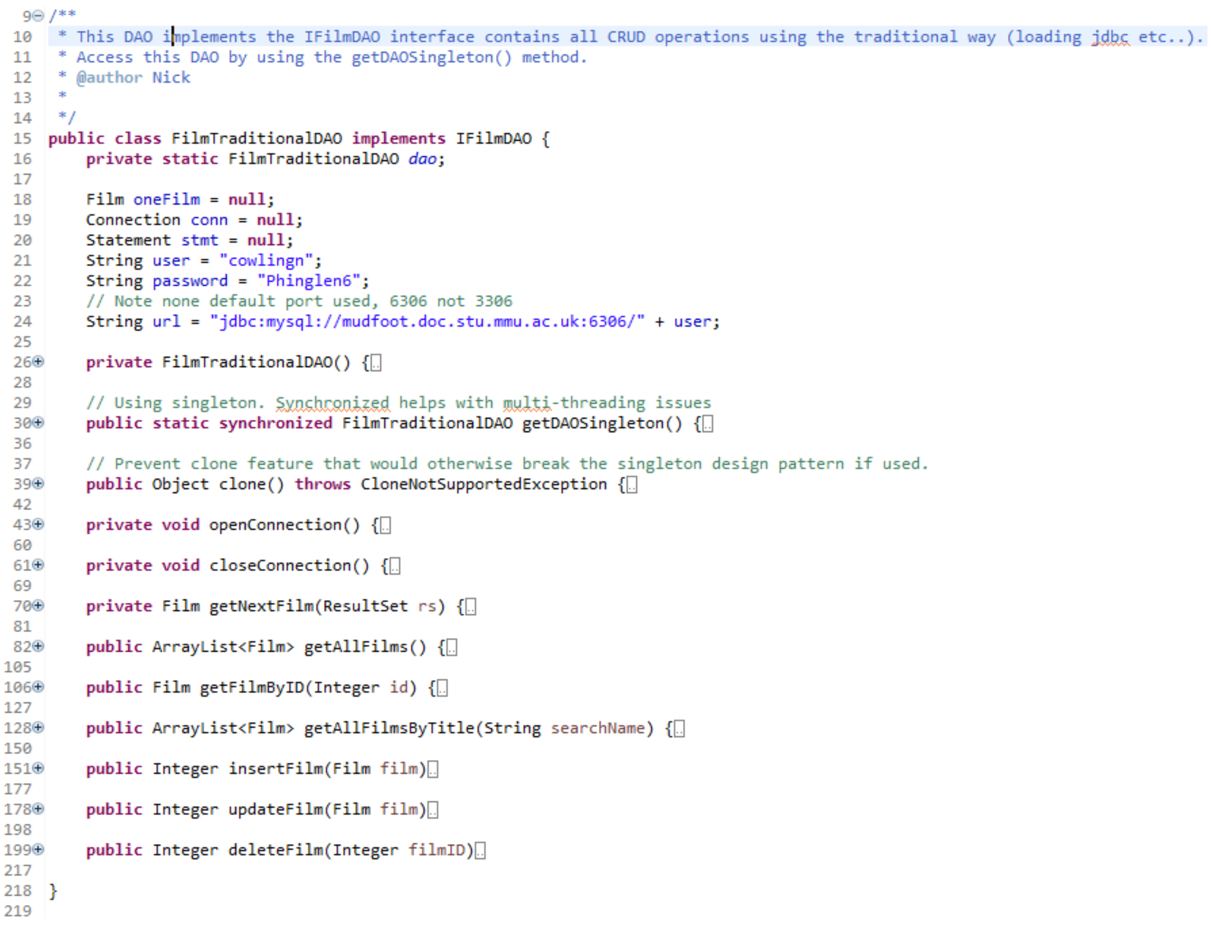
The DAO pattern can be seen in use in the “FilmCloudWebService”, “FilmHTTPWebService”, “FilmRESTfulWebService”.

The interface for the DAO can be seen in the below screenshot:



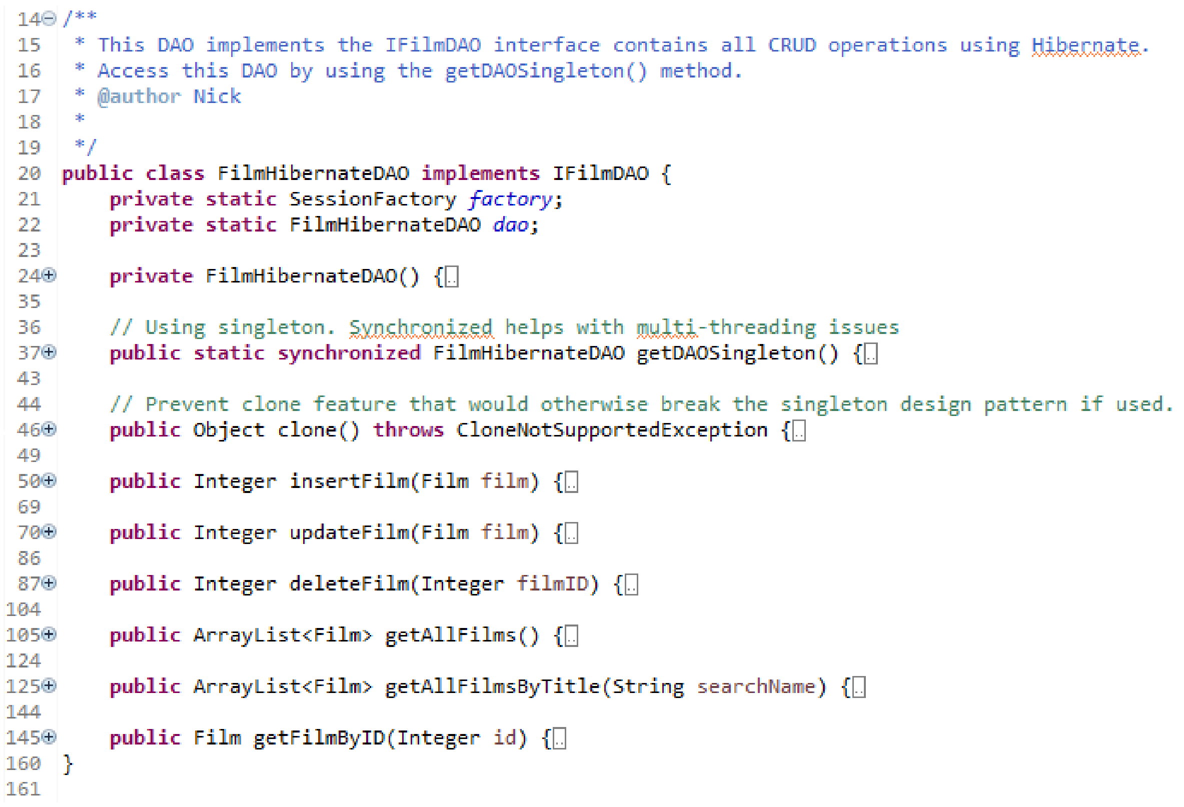
```
import java.util.ArrayList;

/**
 * This interface is essentially in place to help us swap the DAO if we wanted too (Hibernate or Traditional)
 * Also, if we had unit tests, having an interface would greatly help, as we could use dependency injection to swap out
 * our real DAO in the code for a "Mock DAO", which would contain in memory data.
 *
 * Insert / Update / Delete return Integer (1 or 0), this could have been boolean, but Traditional DAO
 * method "executeUpdate" returns an Integer, so I thought it would be best to keep it uniform.
 * @author Nick
 *
 */
public interface IFilmDAO {
    ArrayList<Film> getAllFilms();
    ArrayList<Film> getAllFilmsByTitle(String searchName);
    Film getFilmByID(Integer filmID);
    Integer insertFilm(Film film);
    Integer updateFilm(Film film);
    Integer deleteFilm(Integer filmID);
}
```

The implementation of the interface “IFilmDAO” can be seen below, named “FilmTraditionalDAO” (uses only jdbc drivers to connect, with no libraries or frameworks):



```
/**
 * This DAO implements the IFilmDAO interface contains all CRUD operations using the traditional way (loading jdbc etc..).
 * Access this DAO by using the getDAOSingleton() method.
 * @author Nick
 *
 */
public class FilmTraditionalDAO implements IFilmDAO {
    private static FilmTraditionalDAO dao;

    Film oneFilm = null;
    Connection conn = null;
    Statement stmt = null;
    String user = "cowlingn";
    String password = "Phinglen6";
    // Note none default port used, 6306 not 3306
    String url = "jdbc:mysql://mudfoot.doc.stu.mmu.ac.uk:6306/" + user;

    private FilmTraditionalDAO() {

    // Using singleton. Synchronized helps with multi-threading issues
    public static synchronized FilmTraditionalDAO getDAOSingleton() {

    // Prevent clone feature that would otherwise break the singleton design pattern if used.
    public Object clone() throws CloneNotSupportedException {

    private void openConnection() {

    private void closeConnection() {

    private Film getNextFilm(ResultSet rs) {

    public ArrayList<Film> getAllFilms() {

    public Film getFilmByID(Integer id) {

    public ArrayList<Film> getAllFilmsByTitle(String searchName) {

    public Integer insertFilm(Film film)

    public Integer updateFilm(Film film)

    public Integer deleteFilm(Integer filmID)

}
```

Similarly, the “IFilmDAO” interface is implemented below, named “FilmHibernateDAO” (uses Hibernate):



```java
/**
 * This DAO implements the IFilmDAO interface contains all CRUD operations using Hibernate.
 * Access this DAO by using the getDAOSingleton() method.
 * @author Nick
 *
 */
public class FilmHibernateDAO implements IFilmDAO {
    private static SessionFactory factory;
    private static FilmHibernateDAO dao;

    private FilmHibernateDAO() {□

    // Using singleton. Synchronized helps with multi-threading issues
    public static synchronized FilmHibernateDAO getDAOSingleton() {□

    // Prevent clone feature that would otherwise break the singleton design pattern if used.
    public Object clone() throws CloneNotSupportedException {□

    public Integer insertFilm(Film film) {□

    public Integer updateFilm(Film film) {□

    public Integer deleteFilm(Integer filmID) {□

    public ArrayList<Film> getAllFilms() {□

    public ArrayList<Film> getAllFilmsByTitle(String searchName) {□

    public Film getFilmByID(Integer id) {□
}

```

Singleton Pattern

The singleton pattern is classed as a “creational” design pattern and is used to restrict the instantiation of a class to one “single” instance. This is beneficial for when only one object is needed to coordinate actions across the entire system, and having multiple instances would otherwise cause a problem. Examples of valid Singletons include: event managers, network interfaces, application configuration, and hardware access (such as graphics and sound).

When the object has been requested for the first time, a new instance is created. All subsequent requests returns the same object.

It is created by using a private constructor (so that no other classes can create an instance), a private instance of the class, and a method that contains checks whether the instance is null. If the instance is null, then it has not been instantiated yet, and so the instance is instantiated. If the instance is not null, then it has already been instantiated, so the existing object is returned.

The benefits of using a Singleton are:

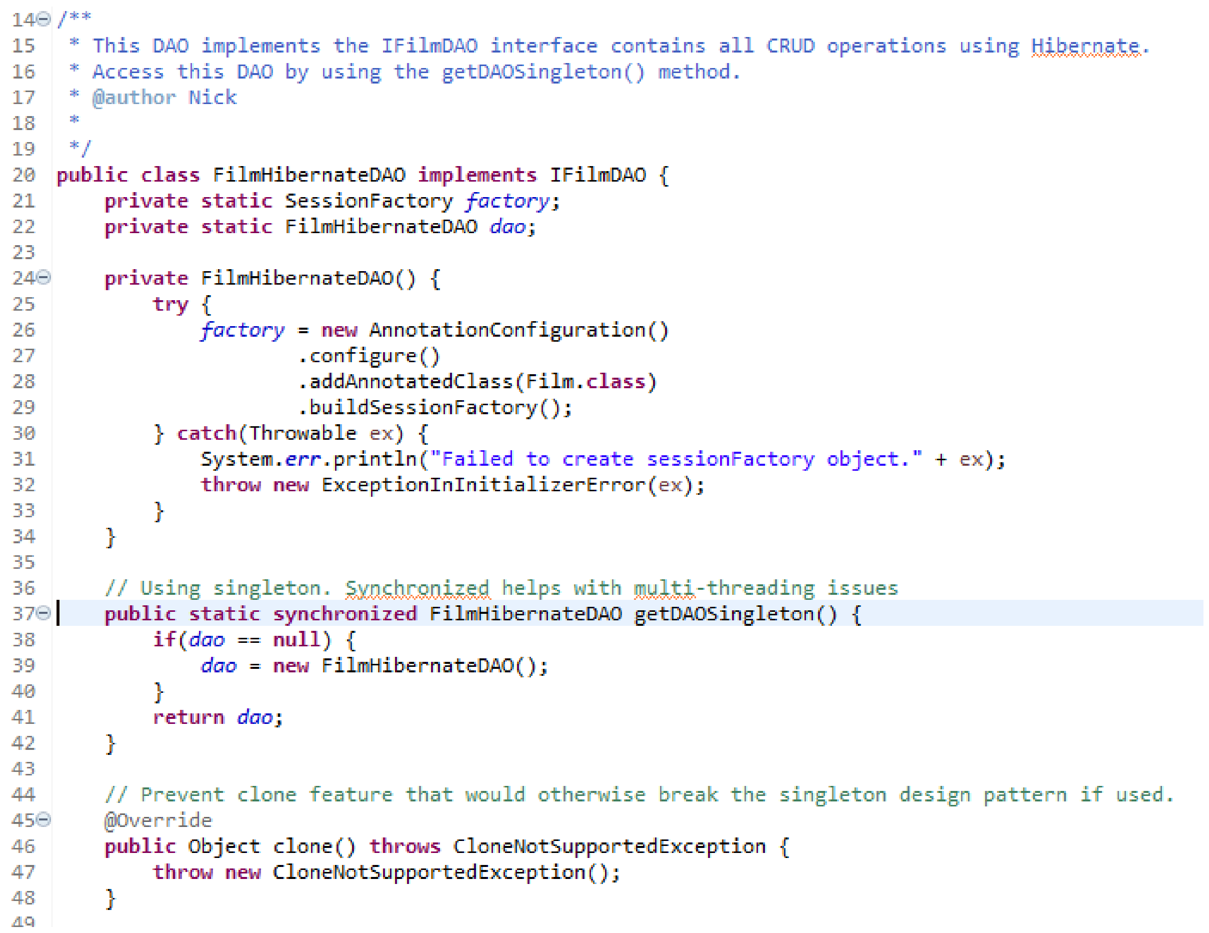
- The instance is globally accessible as read only, whilst still adhering to the Object-Oriented Programming principles
- Using a Singleton means you know how many number of instances there are and they can be managed, unlike with global variables

The drawbacks of using a Singleton are:

- Some recognize it as an “anti-pattern” rather than a design pattern. This is because it is commonly used in scenarios where it does not provide any benefit, but rather introduces unnecessary restrictions (is a single instance of that class actually required?)
- Unit testing may become harder due to tight coupling
- Hidden dependencies are created as it can be over-used throughout the codebase. Additionally, they can become difficult to track when stepping through the code
- Should be used with care in a multi-thread context, since a race condition may result in an instance to be created multiple time (use “synchronized” keyword in Java to avoid this)

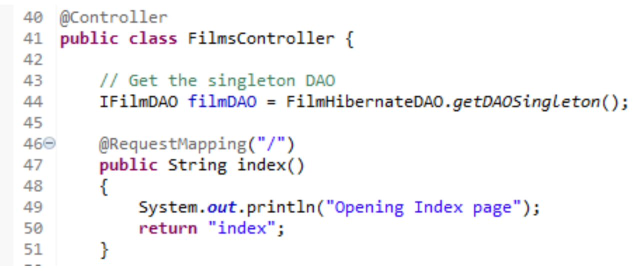
Example of Singleton Used in Solutions

The below screenshot shows the “FilmHibernateDAO.java” class using the Singleton pattern, note the private constructor, the private variable of the class “FilmHibernateDAO”, and the method “getDAOSingleton”. Additionally, I have overridden the clone method in the class so that it is not possible to break the pattern.



```java
/**
 * This DAO implements the IFilmDAO interface contains all CRUD operations using Hibernate.
 * Access this DAO by using the getDAOSingleton() method.
 * @author Nick
 *
 */
public class FilmHibernateDAO implements IFilmDAO {
    private static SessionFactory factory;
    private static FilmHibernateDAO dao;

    private FilmHibernateDAO() {
        try {
            factory = new AnnotationConfiguration()
                    .configure()
                    .addAnnotatedClass(Film.class)
                    .buildSessionFactory();
        } catch(Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    // Using singleton. Synchronized helps with multi-threading issues
    public static synchronized FilmHibernateDAO getDAOSingleton() {
        if(dao == null) {
            dao = new FilmHibernateDAO();
        }
        return dao;
    }

    // Prevent clone feature that would otherwise break the singleton design pattern if used.
    @Override
    public Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }
```

The below screenshot shows the method “getDAOSingleton” being used in the controller (project: “FilmRESTfulWebService”):



```java
@Controller
public class FilmsController {

    // Get the singleton DAO
    IFilmDAO filmDAO = FilmHibernateDAO.getDAOSingleton();

    @RequestMapping("/")
    public String index()
    {
        System.out.println("Opening Index page");
        return "index";
    }
```

Factory Method Pattern

The Factory Method pattern is a creational pattern used when some logic is required to decide what subclass an instance should be; the subclass that is required is decided at runtime based on some sort of input.

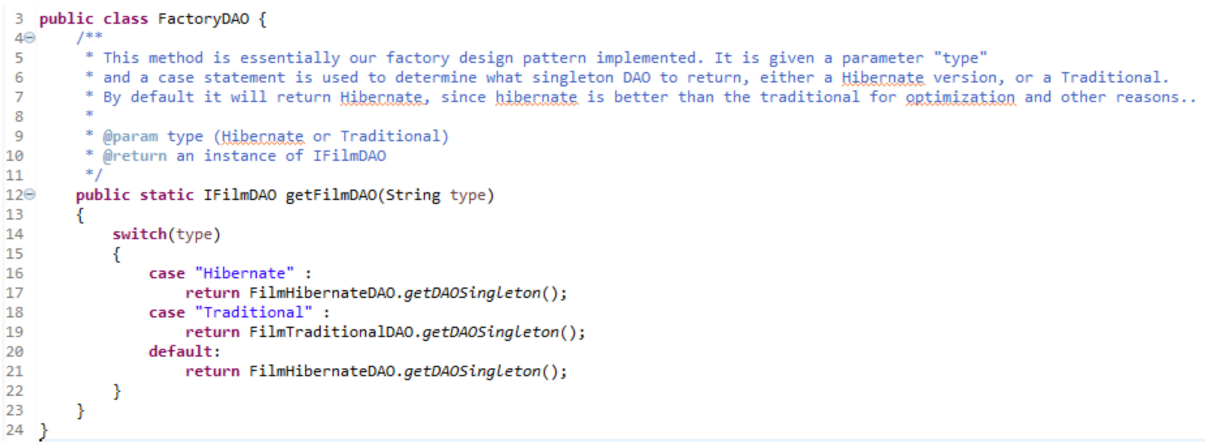
To implement this pattern, a factory method is used to create an object without having to specify the exact class of the object that is to be created – this is achieved by implementing an interface or abstract class for all of the objects the factory method uses, along with the return type of that method being the interface / abstract class.

The benefits of using a Factory method are:

- It helps avoid tight coupling – refactoring and renaming the subclasses would not affect any of the code that uses the factory method (as long as functionality remains the same)
    - This is because of "programming to an interface" rather than a concrete implementation
    - It provides a layer of abstraction between the implementation and the client classes
- Supports the separation of concerns – logic for deciding what subclass to use is within the superclass, not the client code
- Supports teamwork – developers can create / modify different subclasses at the same time.

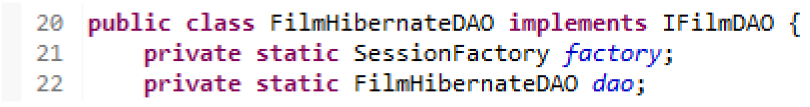
Example of Factory Method Used in Solutions

The below screenshot is from the project "FilmHTTPWebService" and shows the factory method being used, note the return type of the method and the parameter it takes in:

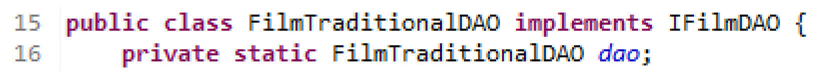


```java
public class FactoryDAO {
    /**
     * This method is essentially our factory design pattern implemented. It is given a parameter "type"
     * and a case statement is used to determine what singleton DAO to return, either a Hibernate version, or a Traditional.
     * By default it will return Hibernate, since hibernate is better than the traditional for optimization and other reasons..
     *
     * @param type (Hibernate or Traditional)
     * @return an instance of IFilmDAO
     */
    public static IFilmDAO getFilmDAO(String type)
    {
        switch(type)
        {
            case "Hibernate" :
                return FilmHibernateDAO.getDAOSingleton();
            case "Traditional" :
                return FilmTraditionalDAO.getDAOSingleton();
            default:
                return FilmHibernateDAO.getDAOSingleton();
        }
    }
}
```

The below screenshots show the two DAO classes implementing the "IFilmDAO" interface:

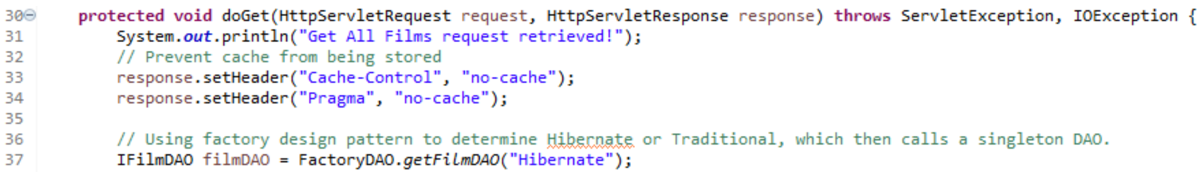


```java
public class FilmHibernateDAO implements IFilmDAO {
    private static SessionFactory factory;
    private static FilmHibernateDAO dao;
```



```java
public class FilmTraditionalDAO implements IFilmDAO {
    private static FilmTraditionalDAO dao;
```

The below screenshot shows the servlet "getAllFilms" using the factory method to determine what DAO to use:



```
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Get All Films request retrieved!");
        // Prevent cache from being stored
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma", "no-cache");

        // Using factory design pattern to determine Hibernate or Traditional, which then calls a singleton DAO.
        IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
```

Object Pool / Connection Pooling

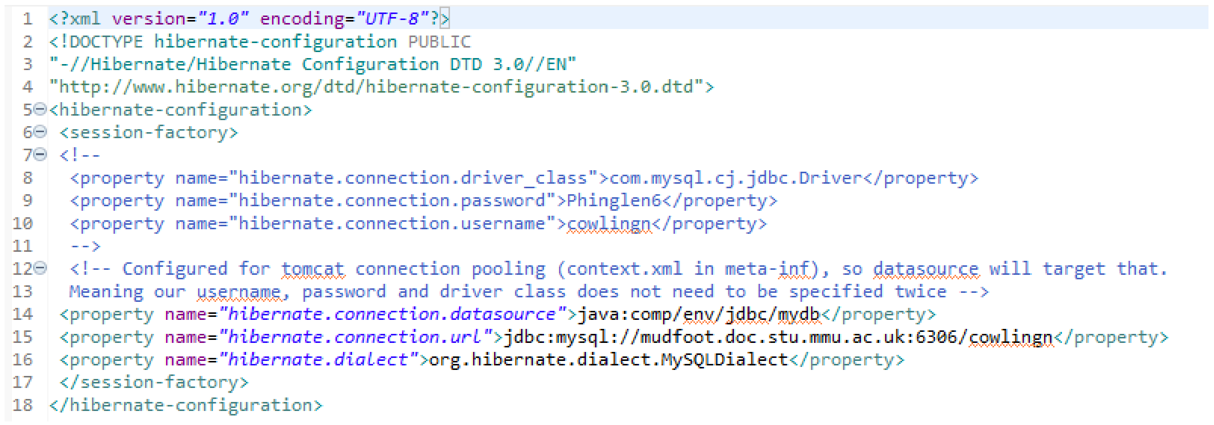
The object pool is a creational pattern that allows you to reuse objects that are expensive to create; when applied to database connections, this is known as connection pooling. These database connections are maintained once created (by being placed in the "pool"), so that they can be reused when future requests to the database are made – meaning a new connection does not have to be established – a costly and time consuming operation. Some examples of libraries for connection pooling include: c3p0, JDBC Connection Pool (from Tomcat), and HikariCP.

The benefits of connection pooling are:

- Enhances the performance of requests to the database, as connections already exist and so do not have to be created first every time (helps a lot when scaling up, and when there are more requests made by multiple clients)
- Reduces the amount of times new connection objects are created
- Minimizes the number of stale connections
- Using libraries, it is not hard to implement –it requires little effort compared to manually managing these connection objects

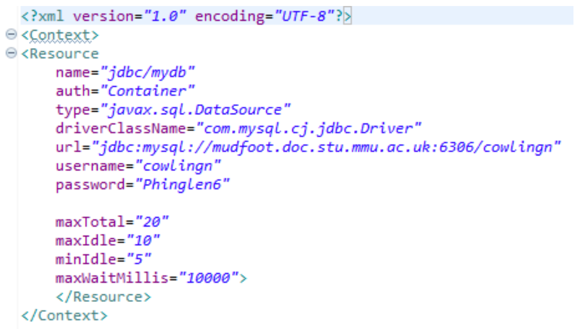
Example of Connection Pooling Used in Solutions (Tomcat)

The below screenshot shows the hibernate configuration used (hibernate.cfg.xml), note the property "hibernate.connection.datasource", and how it targets an environment variable called "jdbc/mydb":



```
<?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.cj.jdbc.Driver</property>
  <property name="hibernate.connection.password">Phinglen6</property>
  <property name="hibernate.connection.username">cowlingn</property>
  -->
  <!-- Configured for tomcat connection pooling (context.xml in meta-inf), so datasource will target that.
  Meaning our username, password and driver class does not need to be specified twice -->
 <property name="hibernate.connection.datasource">java:comp/env/jdbc/mydb</property>
 <property name="hibernate.connection.url">jdbc:mysql://mudfoot.doc.stu.mmu.ac.uk:6306/cowlingn</property>
 <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
 </session-factory>
</hibernate-configuration>
```

The below screenshot shows the "context.xml" located in "WebContent/META-INF", note the "name" attribute "jdbc/mydb" – links to the hibernate config:



```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
<Resource
    name="jdbc/mydb"
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://mudfoot.doc.stu.mmu.ac.uk:6306/cowlingn"
    username="cowlingn"
    password="Phinglen6"

    maxTotal="20"
    maxIdle="10"
    minIdle="5"
    maxWaitMillis="10000">
    </Resource>
</Context>
```

Data Transfer Object (DTO) pattern

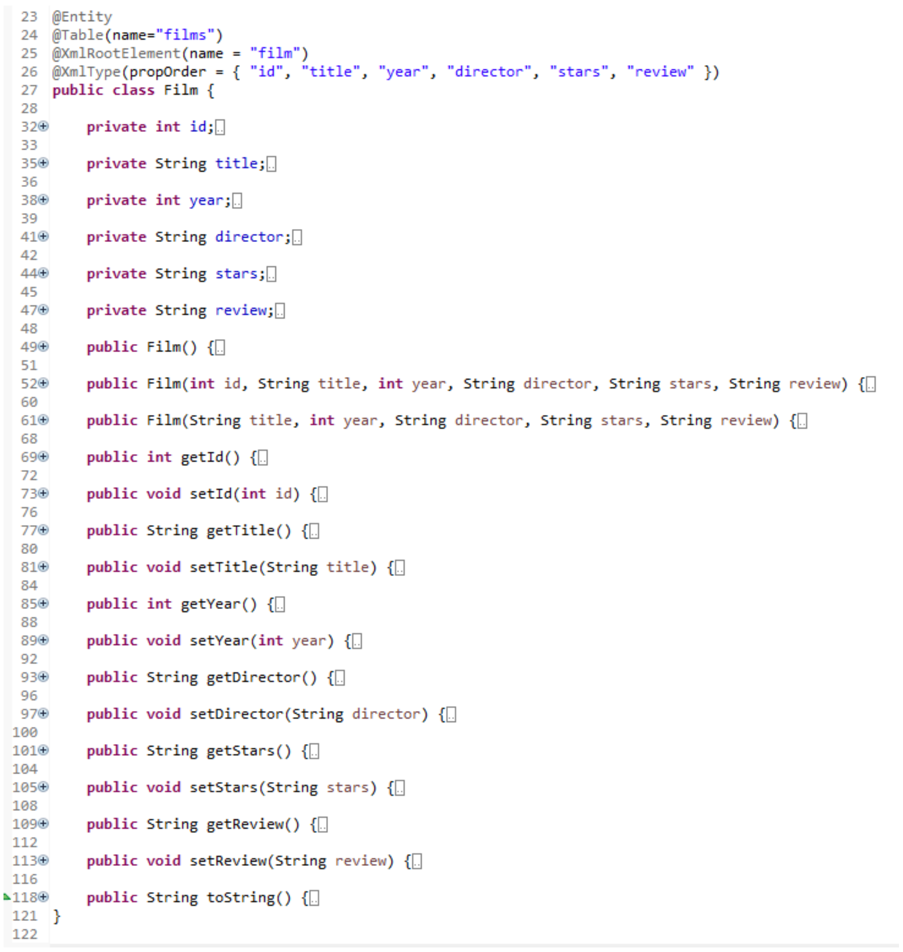
A DTO is an object that carries data between processes. The reason for using this is because calls to any remote interface (such as a web service) is an expensive operation, so combining the data together into a DTO means only one call is needed, rather than several calls. DTO's are different to business objects and DAO's in that a DTO only contains state, the setting / retrieval of that state, along with serialization / deserialization methods if required (for transferring data) – there is no behavior – they are simple objects that should not contain business logic. A DTO is typically passed to the DAO layer that will use this object for CRUD operations.

The benefits of using a DTO are:

- Less calls between the client and server are made (expensive, time-consuming)
- The encapsulation of data

Example of DTO Used in Solutions

The below screenshot shows the model "Film.java" – this can be considered a DTO:



```java
@Entity
@Table(name="films")
@XmlRootElement(name = "film")
@XmlType(propOrder = { "id", "title", "year", "director", "stars", "review" })
public class Film {

    private int id;

    private String title;

    private int year;

    private String director;

    private String stars;

    private String review;

    public Film() {

    public Film(int id, String title, int year, String director, String stars, String review) {

    public Film(String title, int year, String director, String stars, String review) {

    public int getId() {

    public void setId(int 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) {

    public String toString() {
}
```

Dependency Injection

The Dependency Injection (DI) is a technique that is used to implement an inversion of control (IoC). This is done by creating dependent objects outside of the class, and providing these objects to the class through alternative ways. In other words, the creation and binding of the dependent object is moved outside of the class that depends on them. There are three types of classes involved in this process:

- The Client Class – the dependent class, depending on the service class
- The Service Class – the dependency, providing a service to the client class
- The Injector Class – the class that injects the service class object into the client class

There are a few ways to inject dependencies into the client:

- Constructor Injection – the injector sends the dependency (the service in this case) through the client class's constructor
- Property Injection – the injector sends the dependency through a public property of the client class
- Method Injection – the client class implements an interface that declares the methods to supply the dependency. The injector uses the interface in order to supply dependencies to the class

Using DI is beneficial as:

- It decouples the usage of an object from its creation (helping to follow SOLID, with dependency inversion and the single responsibility)
- Reduced dependencies
- Promotes reusable code
- Promotes more testable code
- Promotes more readable code
- Dependency Injection can be used over the Singleton pattern to inject a single instance of the dependency – this makes this pattern more useful than the Singleton pattern, as unit testing can then be done via mocking

If unit tests were required for this assignment and I had time, I would have implemented DI, as it would have enabled me to mock out the DAO for example.

Observer pattern

The Observer is a behavioral pattern used in one-to-many relationships between objects, where one object is modified, the other objects dependent on it need to be notified automatically. The single object - the subject - keeps a list of all the dependents - the observers - and notifies these observers automatically when any state changes; this is typically done through one of their methods. This pattern is commonly seen in event-handling systems.

The benefits using an Observer pattern are:

- Supports loose coupling between the objects that interact with each other
- Observers can be added or removed from the list when required
- Data can be sent to other objects effectively without any change in the subject or observer classes

Façade pattern

The Façade pattern is a structural pattern used to hide the complexities of the system by providing an interface to the client that they can instead use to access the system.

The benefits of using a Façade pattern are:

- The readability and usability is improved, as complex components are hidden away behind a simple interface
- Reduces dependencies and so helps with creating loosely coupled code

An example where this could have potentially been used in my solutions was for the parsing from/to films via XML, JSON or Text. I decided against doing this, as it would have ultimately meant creating more classes that I would have to maintain, just to encapsulate simple code (Gson can convert to / from JSON in one line of code) that I have instead just placed into a utility class. If my conversion logic was more complex and it effected the readability and usability of my utility classes and business-level code, I would use this pattern.

Refactoring

Refactoring is the process of restructuring existing code without changing its external behavior, helping to clean up the code, reduce the chance of bugs, and overall improve the design of the code. Refactoring contributes to the removal of "Technical Debt" – a backlog of work required to improve the software quality. Developing software is a constant learning process – developers learn how to implement features mor effectively and efficiently over time, which is why refactoring is required.

The benefits of refactoring code are:

- Improves maintainability of the code-base; fixing bugs are easier as the source code becomes easier to read and understand by other developers. This reduces the cost of bug fixing and maintenance in the future
- Improves extensibility – it is easier to extend the capabilities of the application if it uses recognizable design patterns

The drawbacks of refactoring are:

- From the managements point of view, refactoring could be viewed as wasteful as they do not see any new features being created
- Costs time and money (though it should pay off with the aid to maintainability)

To refactor, you must follow these following steps:

1. Ensure the code is working correctly beforehand
    a. This can be done either manually, or with automated unit tests
2. Back-up the code via version-control – commit both before and after refactoring
    a. Doing so means the code can always be rolled back if anything breaks
3. Do not change the functionality during refactoring; the external behavior should not be altered
4. Break each refactoring into small, safe steps; it should be done in iterations, do not refactor the whole codebase or a feature in one go

Throughout all my projects, refactoring took place. It was a constant process in which I would develop my features such as the AJAX for the front-end, and then tidy up the code by moving repeated code into methods and searching for and using libraries such as jQuery that provides the same functionality but in a more concise way.

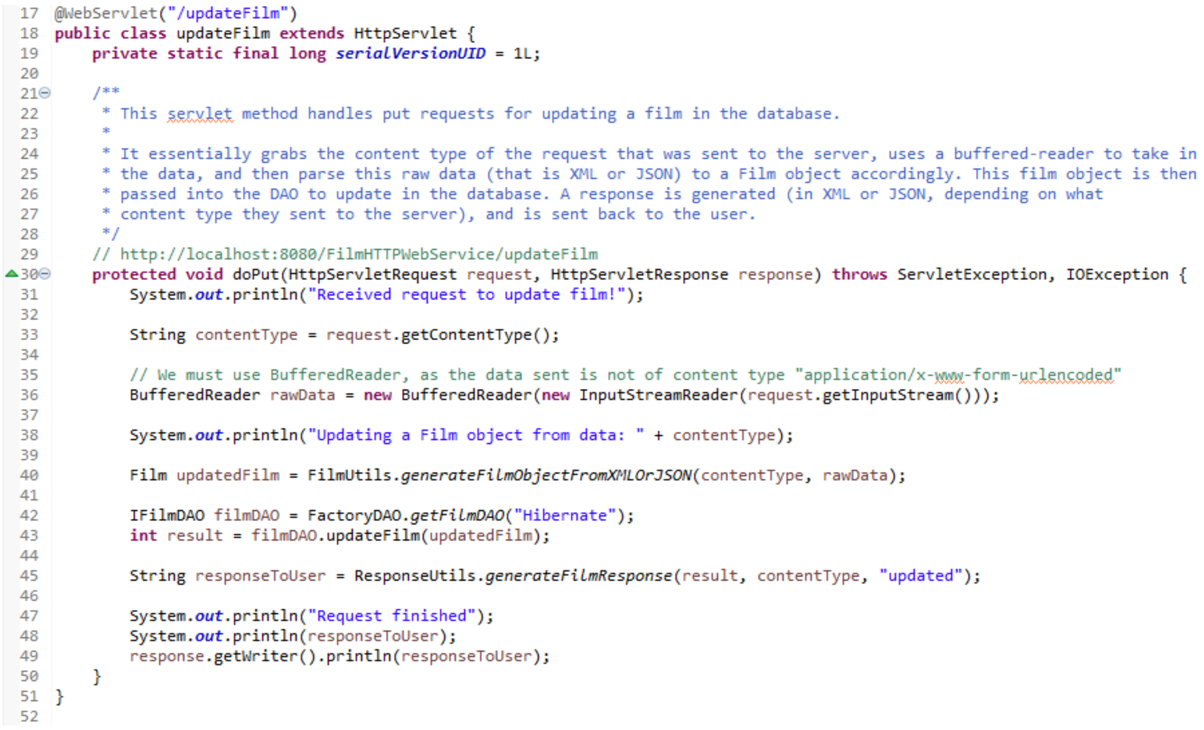
Don't Repeat Yourself (DRY)

DRY is a basic principle of software development that aims to reduce the repetition of information. This principle states "every piece of knowledge or logic must have a single, unambiguous representation within a system". To abide by this principle, the system must be divided into pieces; the code and logic should be small, reusable units that are used where needed. Additionally, the methods themselves should not be lengthy. This ultimately saves time and effort, makes the code easier to maintain, and reduces the chances of bugs.

Examples of Refactoring – Backend (Servlets, Controllers, Utilities)

This section aims to show examples of reusable, maintainable code that was achieved through refactoring the code on the backend. A utilities package was created for this, and any reused code was placed in the appropriate class (FilmUtils or ResponseUtils).

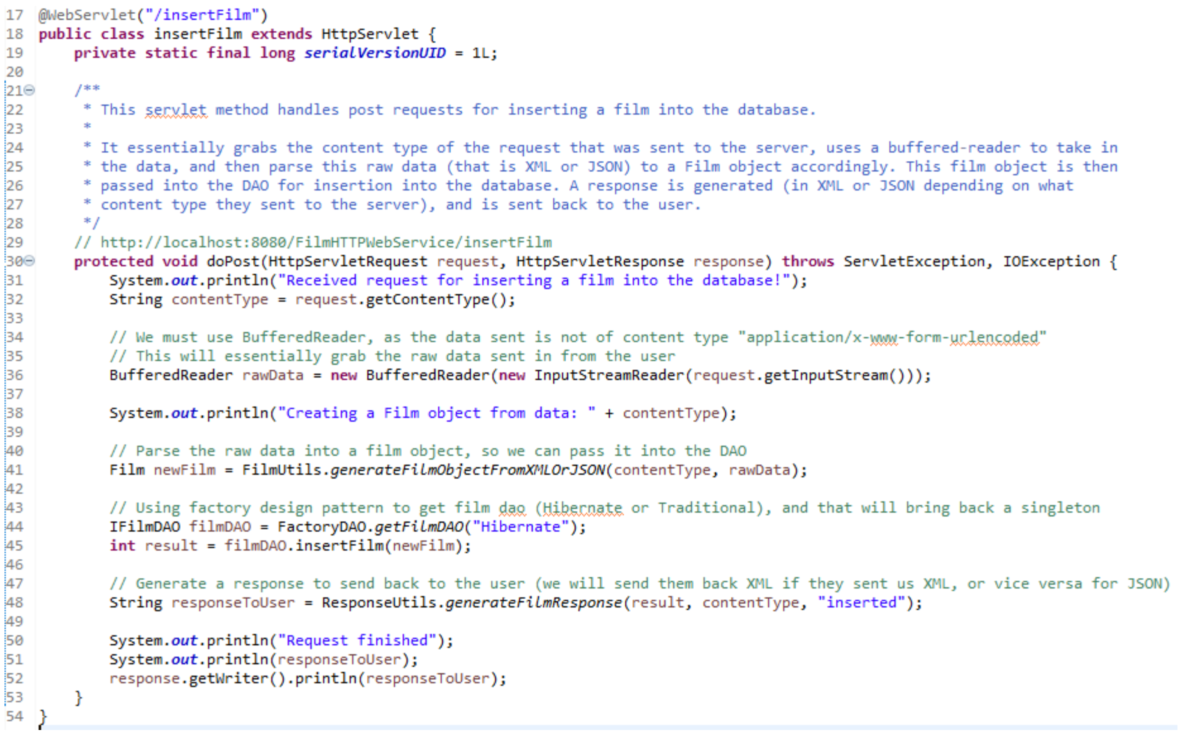
The below screenshots are from “FilmHTTPWebService” (almost identical to “FilmCloudWebService”) and shows the “updateFilm.java” class:



```java
17  @WebServlet("/updateFilm")
18  public class updateFilm extends HttpServlet {
19      private static final long serialVersionUID = 1L;
20
21      /**
22       * This servlet method handles put requests for updating a film in the database.
23       *
24       * It essentially grabs the content type of the request that was sent to the server, uses a buffered-reader to take in
25       * the data, and then parse this raw data (that is XML or JSON) to a Film object accordingly. This film object is then
26       * passed into the DAO to update in the database. A response is generated (in XML or JSON, depending on what
27       * content type they sent to the server), and is sent back to the user.
28       */
29      // http://localhost:8080/FilmHTTPWebService/updateFilm
30      protected void doPut(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
31          System.out.println("Received request to update film!");
32
33          String contentType = request.getContentType();
34
35          // We must use BufferedReader, as the data sent is not of content type "application/x-www-form-urlencoded"
36          BufferedReader rawData = new BufferedReader(new InputStreamReader(request.getInputStream()));
37
38          System.out.println("Updating a Film object from data: " + contentType);
39
40          Film updatedFilm = FilmUtils.generateFilmObjectFromXMLOrJSON(contentType, rawData);
41
42          IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
43          int result = filmDAO.updateFilm(updatedFilm);
44
45          String responseToUser = ResponseUtils.generateFilmResponse(result, contentType, "updated");
46
47          System.out.println("Request finished");
48          System.out.println(responseToUser);
49          response.getWriter().println(responseToUser);
50      }
51  }
52
```

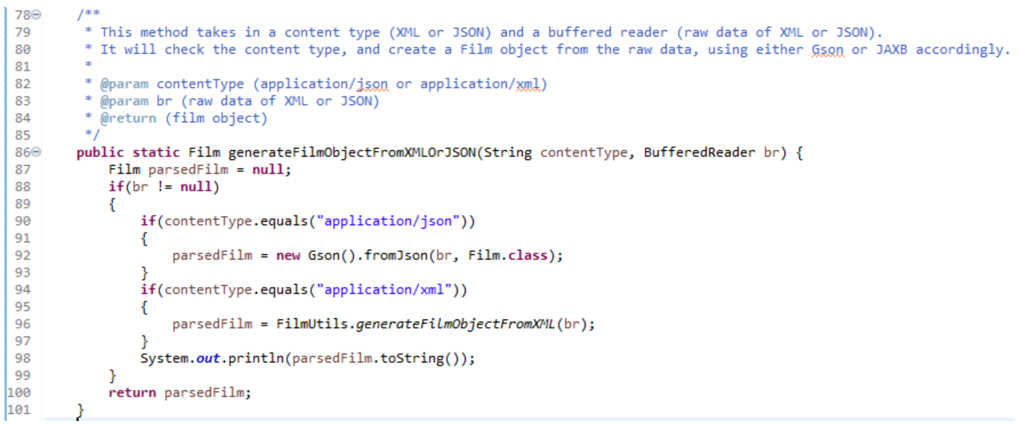
On line 40 you can see a method from FilmUtils being called, “generateFilmObjectFromXMLOrJSON”, and on line 45 you can see a method from ResponseUtils being called, “generateFilmResponse”. These methods are used in both the “updateFilm.java” class, and the “insertFilm.java” class – originally, the code from these methods used to be repeated in each Servlet. Refactoring this into its own method has promoted code-reuse and made my code more maintainable. Additionally, on line 42, a FactoryDAO was introduced into the code, that calls a singleton – this refactoring essentially introduced two design patterns into my codebase (refer to the appropriate sections in this document for the benefits and potential drawbacks)

Below shows insertFilm using the same methods:

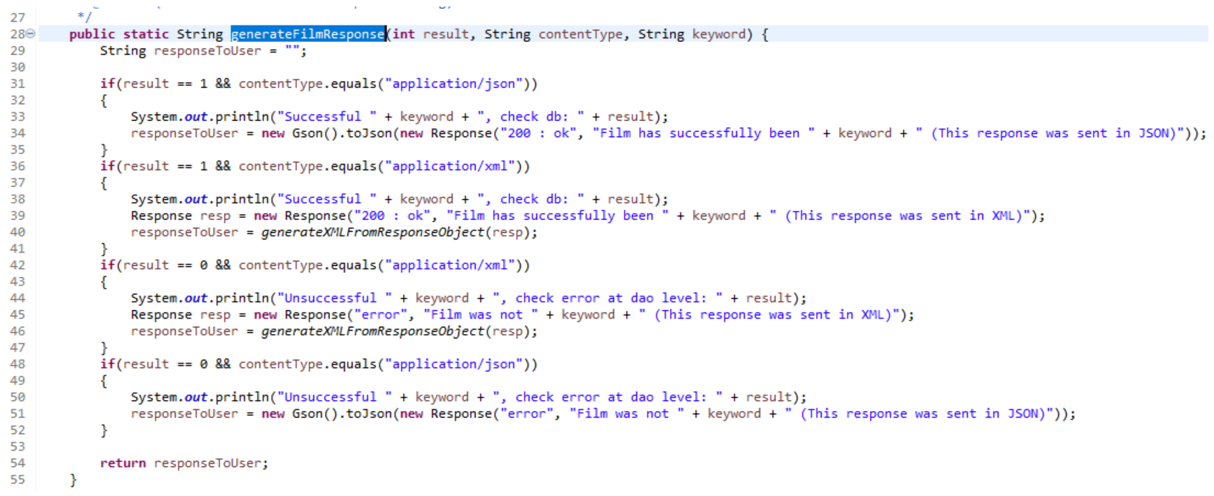


```java
@WebServlet("/insertFilm")
public class insertFilm extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * This servlet method handles post requests for inserting a film into the database.
     *
     * It essentially grabs the content type of the request that was sent to the server, uses a buffered-reader to take in
     * the data, and then parse this raw data (that is XML or JSON) to a Film object accordingly. This film object is then
     * passed into the DAO for insertion into the database. A response is generated (in XML or JSON depending on what
     * content type they sent to the server), and is sent back to the user.
     */
    // http://localhost:8080/FilmHTTPWebService/insertFilm
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Received request for inserting a film into the database!");
        String contentType = request.getContentType();

        // We must use BufferedReader, as the data sent is not of content type "application/x-www-form-urlencoded"
        // This will essentially grab the raw data sent in from the user
        BufferedReader rawData = new BufferedReader(new InputStreamReader(request.getInputStream()));

        System.out.println("Creating a Film object from data: " + contentType);

        // Parse the raw data into a film object, so we can pass it into the DAO
        Film newFilm = FilmUtils.generateFilmObjectFromXMLOrJSON(contentType, rawData);

        // Using factory design pattern to get film dao (Hibernate or Traditional), and that will bring back a singleton
        IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
        int result = filmDAO.insertFilm(newFilm);

        // Generate a response to send back to the user (we will send them back XML if they sent us XML, or vice versa for JSON)
        String responseToUser = ResponseUtils.generateFilmResponse(result, contentType, "inserted");

        System.out.println("Request finished");
        System.out.println(responseToUser);
        response.getWriter().println(responseToUser);
    }
}
```

Below shows the code of these reusable methods (located in FilmUtils, and ResponseUtils):

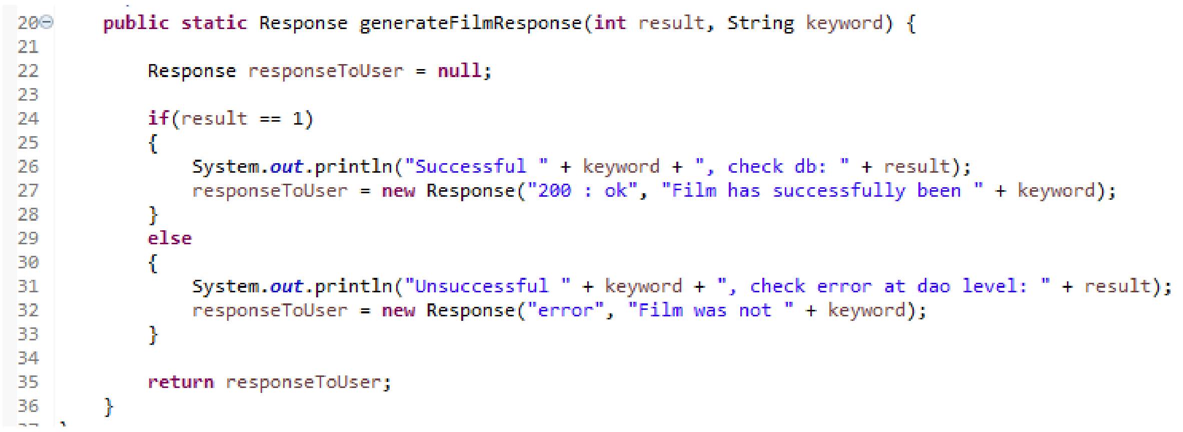


```java
    /**
     * This method takes in a content type (XML or JSON) and a buffered reader (raw data of XML or JSON).
     * It will check the content type, and create a Film object from the raw data, using either Gson or JAXB accordingly.
     *
     * @param contentType (application/json or application/xml)
     * @param br (raw data of XML or JSON)
     * @return (film object)
     */
    public static Film generateFilmObjectFromXMLOrJSON(String contentType, BufferedReader br) {
        Film parsedFilm = null;
        if(br != null)
        {
            if(contentType.equals("application/json"))
            {
                parsedFilm = new Gson().fromJson(br, Film.class);
            }
            if(contentType.equals("application/xml"))
            {
                parsedFilm = FilmUtils.generateFilmObjectFromXML(br);
            }
            System.out.println(parsedFilm.toString());
        }
        return parsedFilm;
    }
```



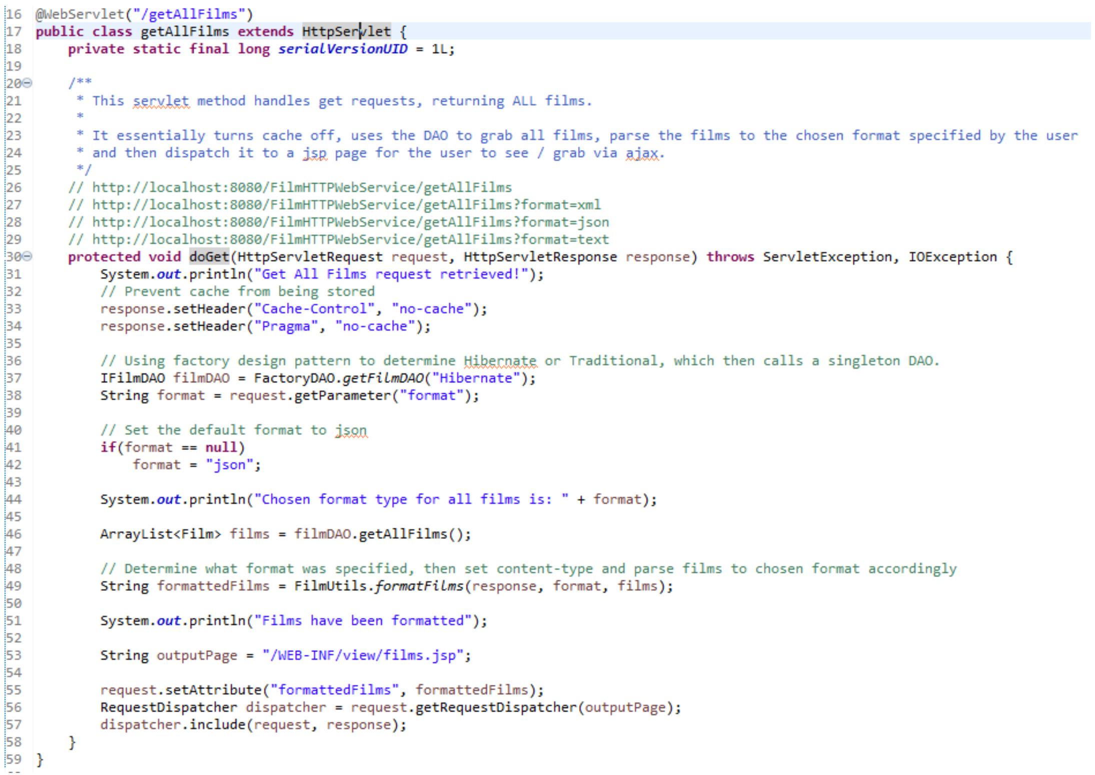
```java
     */
    public static String generateFilmResponse(int result, String contentType, String keyword) {
        String responseToUser = "";

        if(result == 1 && contentType.equals("application/json"))
        {
            System.out.println("Successful " + keyword + ", check db: " + result);
            responseToUser = new Gson().toJson(new Response("200 : ok", "Film has successfully been " + keyword + " (This response was sent in JSON)"));
        }
        if(result == 1 && contentType.equals("application/xml"))
        {
            System.out.println("Successful " + keyword + ", check db: " + result);
            Response resp = new Response("200 : ok", "Film has successfully been " + keyword + " (This response was sent in XML)");
            responseToUser = generateXMLFromResponseObject(resp);
        }
        if(result == 0 && contentType.equals("application/xml"))
        {
            System.out.println("Unsuccessful " + keyword + ", check error at dao level: " + result);
            Response resp = new Response("error", "Film was not " + keyword + " (This response was sent in XML)");
            responseToUser = generateXMLFromResponseObject(resp);
        }
        if(result == 0 && contentType.equals("application/json"))
        {
            System.out.println("Unsuccessful " + keyword + ", check error at dao level: " + result);
            responseToUser = new Gson().toJson(new Response("error", "Film was not " + keyword + " (This response was sent in JSON)"));
        }

        return responseToUser;
    }
```

"generateFilmResponse" could have been two if statements here, but I wanted to specify to the user precisely what format the response was sent in. Below, you can see an alternative version of this method, located in the project "FilmRESTfulWebService" project:



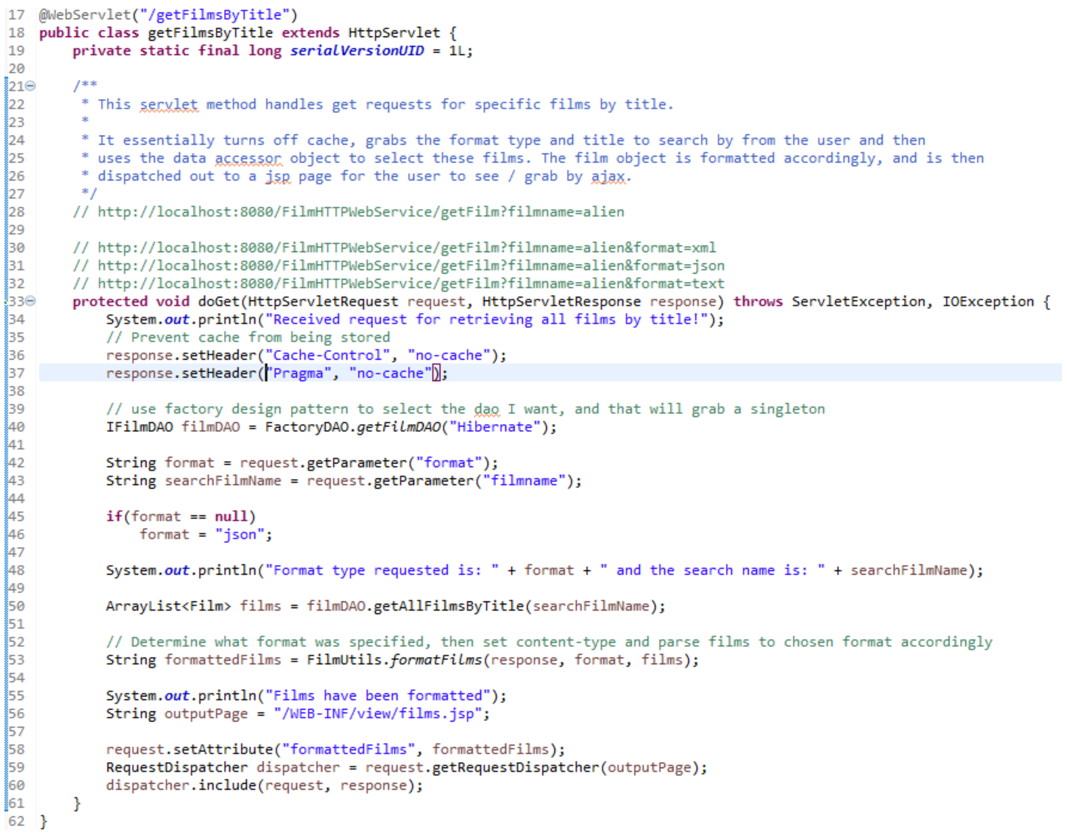
```java
    public static Response generateFilmResponse(int result, String keyword) {

        Response responseToUser = null;

        if(result == 1)
        {
            System.out.println("Successful " + keyword + ", check db: " + result);
            responseToUser = new Response("200 : ok", "Film has successfully been " + keyword);
        }
        else
        {
            System.out.println("Unsuccessful " + keyword + ", check error at dao level: " + result);
            responseToUser = new Response("error", "Film was not " + keyword);
        }

        return responseToUser;
    }
```

Similarly, in the class "getAllFilms.java" and "getFilmsByTitle.java", you can see the same method "formatFilms" from the film utility class being used:



```java
16 @WebServlet("/getAllFilms")
17 public class getAllFilms extends HttpServlet {
18     private static final long serialVersionUID = 1L;
19 
20     /**
21      * This servlet method handles get requests, returning ALL films.
22      *
23      * It essentially turns cache off, uses the DAO to grab all films, parse the films to the chosen format specified by the user
24      * and then dispatch it to a jsp page for the user to see / grab via ajax.
25      */
26     // http://localhost:8080/FilmHTTPWebService/getAllFilms
27     // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=xml
28     // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=json
29     // http://localhost:8080/FilmHTTPWebService/getAllFilms?format=text
30     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
31         System.out.println("Get All Films request retrieved!");
32         // Prevent cache from being stored
33         response.setHeader("Cache-Control", "no-cache");
34         response.setHeader("Pragma", "no-cache");
35 
36         // Using factory design pattern to determine Hibernate or Traditional, which then calls a singleton DAO.
37         IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
38         String format = request.getParameter("format");
39 
40         // Set the default format to json
41         if(format == null)
42             format = "json";
43 
44         System.out.println("Chosen format type for all films is: " + format);
45 
46         ArrayList<Film> films = filmDAO.getAllFilms();
47 
48         // Determine what format was specified, then set content-type and parse films to chosen format accordingly
49         String formattedFilms = FilmUtils.formatFilms(response, format, films);
50 
51         System.out.println("Films have been formatted");
52 
53         String outputPage = "/WEB-INF/view/films.jsp";
54 
55         request.setAttribute("formattedFilms", formattedFilms);
56         RequestDispatcher dispatcher = request.getRequestDispatcher(outputPage);
57         dispatcher.include(request, response);
58     }
59 }
```

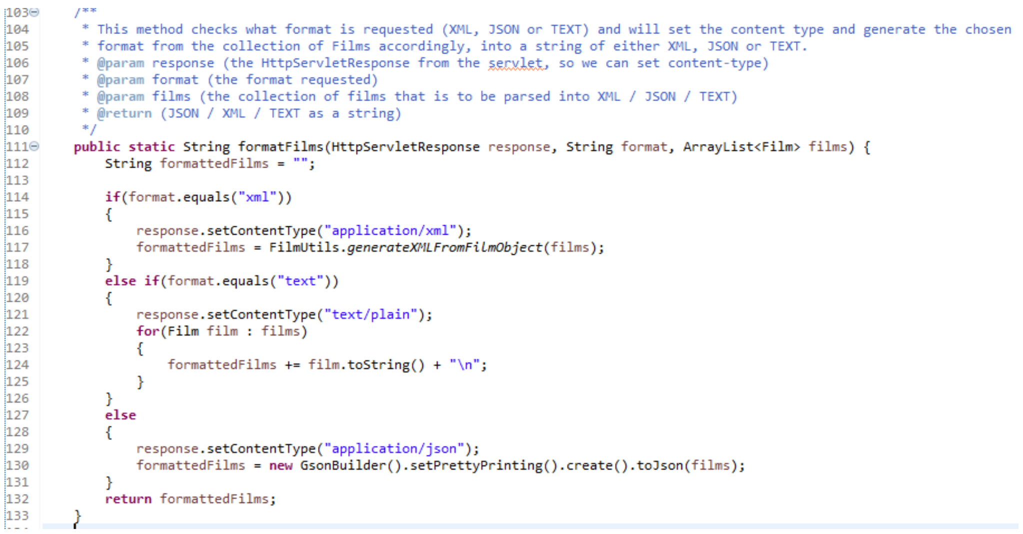
Above on line 49 shows the "formatFilms" method, and line 37 shows the Factory method pattern I implemented.



```java
17  @WebServlet("/getFilmsByTitle")
18  public class getFilmsByTitle extends HttpServlet {
19      private static final long serialVersionUID = 1L;
20
21      /**
22       * This servlet method handles get requests for specific films by title.
23       *
24       * It essentially turns off cache, grabs the format type and title to search by from the user and then
25       * uses the data accessor object to select these films. The film object is formatted accordingly, and is then
26       * dispatched out to a jsp page for the user to see / grab by ajax.
27       */
28      // http://localhost:8080/FilmHTTPWebService/getFilm?filmname=alien
29
30      // http://localhost:8080/FilmHTTPWebService/getFilm?filmname=alien&format=xml
31      // http://localhost:8080/FilmHTTPWebService/getFilm?filmname=alien&format=json
32      // http://localhost:8080/FilmHTTPWebService/getFilm?filmname=alien&format=text
33      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
34          System.out.println("Received request for retrieving all films by title!");
35          // Prevent cache from being stored
36          response.setHeader("Cache-Control", "no-cache");
37          response.setHeader("Pragma", "no-cache");
38
39          // use factory design pattern to select the dao I want, and that will grab a singleton
40          IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
41
42          String format = request.getParameter("format");
43          String searchFilmName = request.getParameter("filmname");
44
45          if(format == null)
46              format = "json";
47
48          System.out.println("Format type requested is: " + format + " and the search name is: " + searchFilmName);
49
50          ArrayList<Film> films = filmDAO.getAllFilmsByTitle(searchFilmName);
51
52          // Determine what format was specified, then set content-type and parse films to chosen format accordingly
53          String formattedFilms = FilmUtils.formatFilms(response, format, films);
54
55          System.out.println("Films have been formatted");
56          String outputPage = "/WEB-INF/view/films.jsp";
57
58          request.setAttribute("formattedFilms", formattedFilms);
59          RequestDispatcher dispatcher = request.getRequestDispatcher(outputPage);
60          dispatcher.include(request, response);
61      }
62  }
```

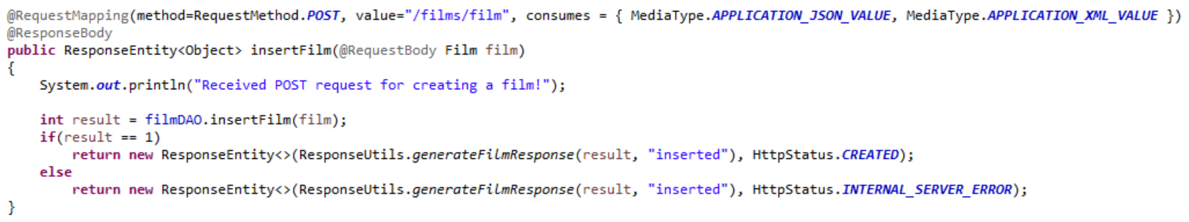
Above you can see the same method “formatFilms” being used on line 53, and the FactoryDAO used on line 40.

Below is a screenshot of this method, promoting code-reuse, note line 117, where I created another method that actually handles the JAXB marshalling:



```java
103     /**
104      * This method checks what format is requested (XML, JSON or TEXT) and will set the content type and generate the chosen
105      * format from the collection of Films accordingly, into a string of either XML, JSON or TEXT.
106      * @param response (the HttpServletResponse from the servlet, so we can set content-type)
107      * @param format (the format requested)
108      * @param films (the collection of films that is to be parsed into XML / JSON / TEXT)
109      * @return (JSON / XML / TEXT as a string)
110      */
111     public static String formatFilms(HttpServletResponse response, String format, ArrayList<Film> films) {
112         String formattedFilms = "";
113
114         if(format.equals("xml"))
115         {
116             response.setContentType("application/xml");
117             formattedFilms = FilmUtils.generateXMLFromFilmObject(films);
118         }
119         else if(format.equals("text"))
120         {
121             response.setContentType("text/plain");
122             for(Film film : films)
123             {
124                 formattedFilms += film.toString() + "\n";
125             }
126         }
127         else
128         {
129             response.setContentType("application/json");
130             formattedFilms = new GsonBuilder().setPrettyPrinting().create().toJson(films);
131         }
132         return formattedFilms;
133     }
```

Below is a screenshot from the controller of the project “FilmRESTfulWebService”, showing how it inserts a film. This was refactored by creating a “ResponseUtils” class with a method “generateFilmResponse”:



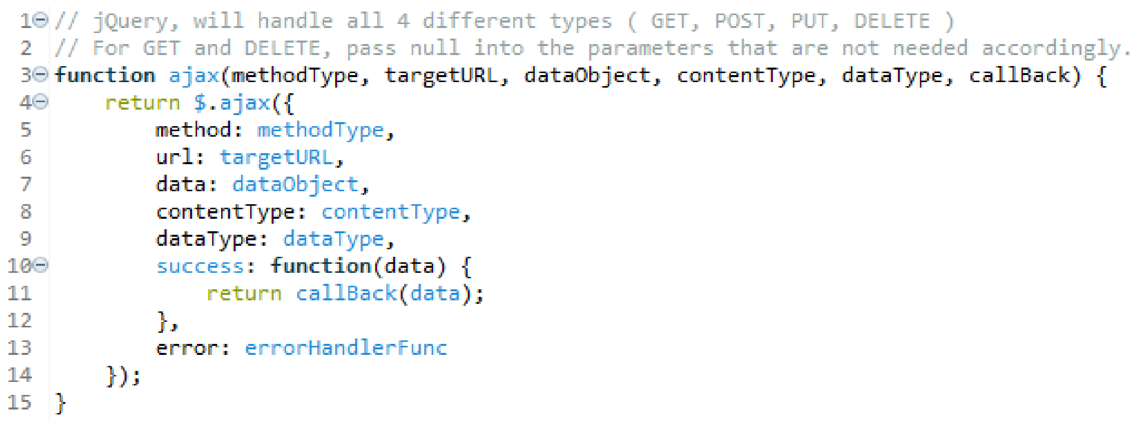
```
@RequestMapping(method=RequestMethod.POST, value="/films/film", consumes = { MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
@ResponseBody
public ResponseEntity<Object> insertFilm(@RequestBody Film film)
{
    System.out.println("Received POST request for creating a film!");

    int result = filmDAO.insertFilm(film);
    if(result == 1)
        return new ResponseEntity<>(ResponseUtils.generateFilmResponse(result, "inserted"), HttpStatus.CREATED);
    else
        return new ResponseEntity<>(ResponseUtils.generateFilmResponse(result, "inserted"), HttpStatus.INTERNAL_SERVER_ERROR);
}
```

This could have been further refactored by calling the utility method right after the insertFilm method was called, and this result could have been placed into the ResponseEntity for both return conditions, rather than having to call the method "genereteFilmResponse" twice.

Examples of Refactoring – Frontend JavaScript/jQuery

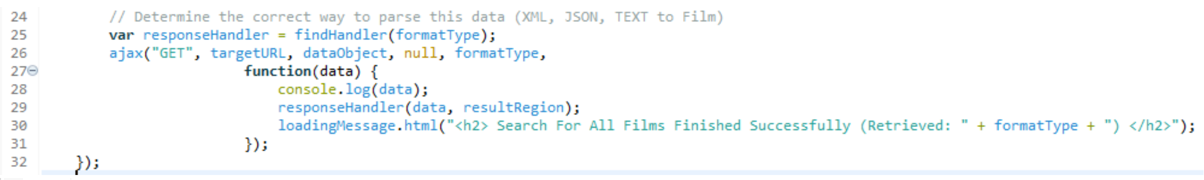
I spent a lot of time refactoring my scripts for the front-end. This can be seen in the "FilmCloudWebService", "FilmHTTPWebService", and "FilmRESTfulWebService" projects, as they all have the same front-end and scripts (except for the URL's that target the API differently).

Originally, I had four AJAX methods – one for each of the HTTP requests (GET, POST, PUT, DELETE). All of these methods repeated the same code except for the method parameter and URL. After refactoring, these four methods were merged into one:

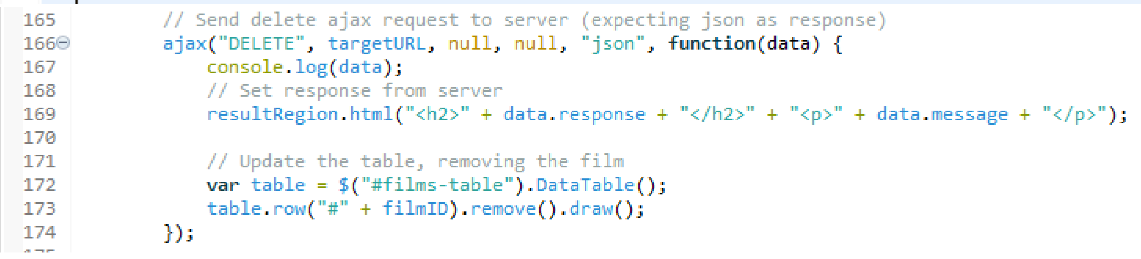


```
// jQuery, will handle all 4 different types ( GET, POST, PUT, DELETE )
// For GET and DELETE, pass null into the parameters that are not needed accordingly.
function ajax(methodType, targetURL, dataObject, contentType, dataType, callBack) {
    return $.ajax({
        method: methodType,
        url: targetURL,
        data: dataObject,
        contentType: contentType,
        dataType: dataType,
        success: function(data) {
            return callBack(data);
        },
        error: errorHandlerFunc
    });
}
```

In the calling code, this method is used like so:

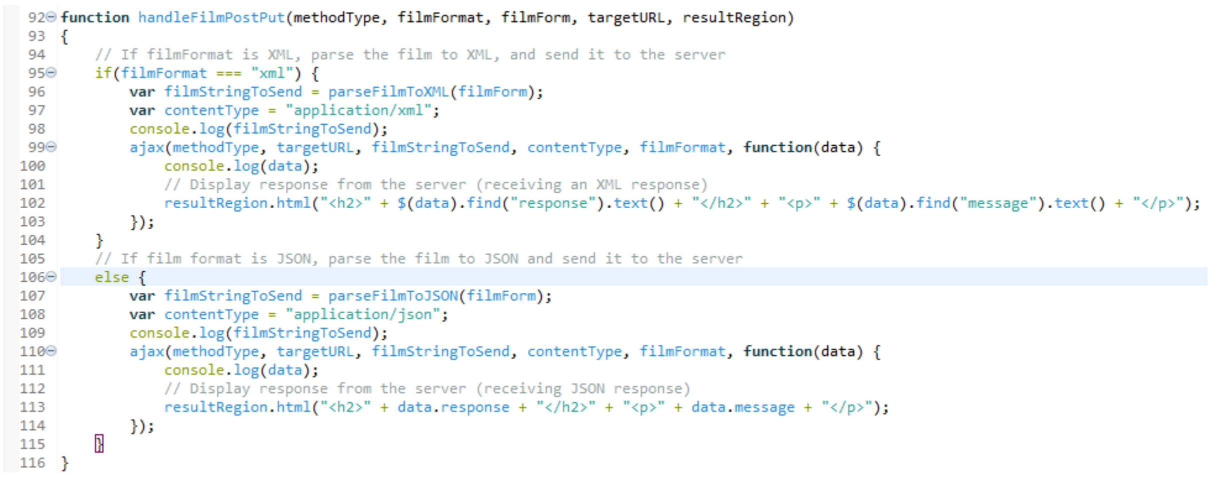


```
        // Determine the correct way to parse this data (XML, JSON, TEXT to Film)
        var responseHandler = findHandler(formatType);
        ajax("GET", targetURL, dataObject, null, formatType,
                    function(data) {
                        console.log(data);
                        responseHandler(data, resultRegion);
                        loadingMessage.html("<h2> Search For All Films Finished Successfully (Retrieved: " + formatType + ") </h2>");
                    });
    });
```



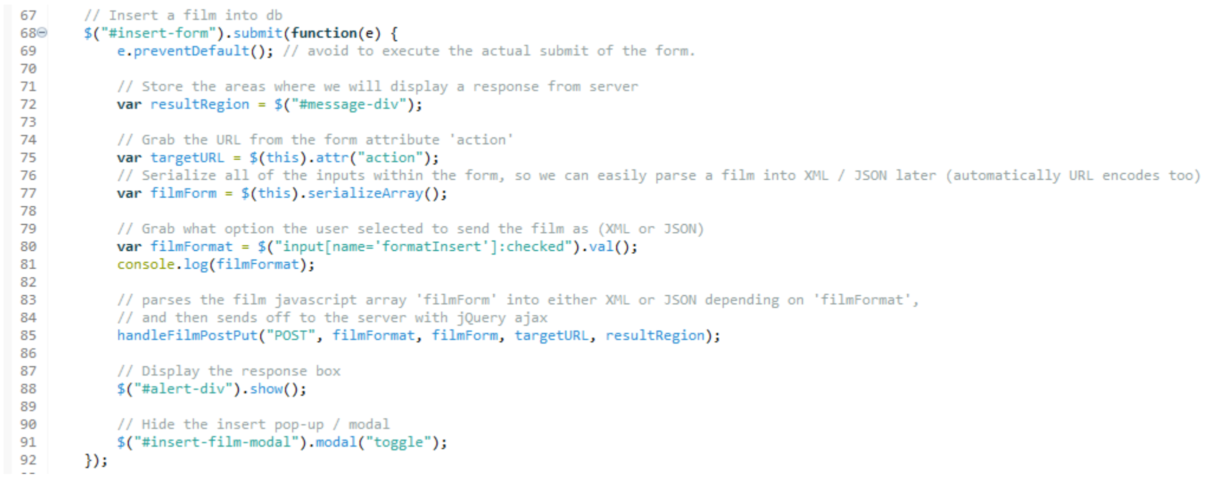
```
        // Send delete ajax request to server (expecting json as response)
        ajax("DELETE", targetURL, null, null, "json", function(data) {
            console.log(data);
            // Set response from server
            resultRegion.html("<h2>" + data.response + "</h2>" + "<p>" + data.message + "</p>");

            // Update the table, removing the film
            var table = $("#films-table").DataTable();
            table.row("#" + filmID).remove().draw();
        });
```

For POST and PUT requests, the scripts for inserting a film and updating a film repeated the same code for: parsing a website form into the correct format (XML or JSON), using AJAX, and then handling the response. Because of this, I moved the repeated code into a separate method, promoting code-reuse, this can be seen below:

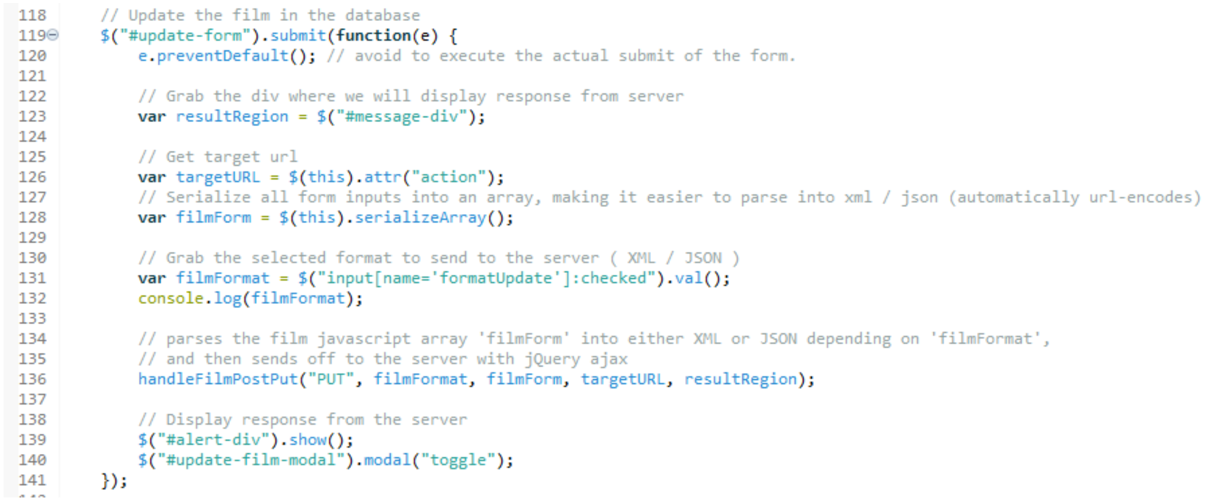


```javascript
function handleFilmPostPut(methodType, filmFormat, filmForm, targetURL, resultRegion)
{
    // If filmFormat is XML, parse the film to XML, and send it to the server
    if(filmFormat === "xml") {
        var filmStringToSend = parseFilmToXML(filmForm);
        var contentType = "application/xml";
        console.log(filmStringToSend);
        ajax(methodType, targetURL, filmStringToSend, contentType, filmFormat, function(data) {
            console.log(data);
            // Display response from the server (receiving an XML response)
            resultRegion.html("<h2>" + $(data).find("response").text() + "</h2>" + "<p>" + $(data).find("message").text() + "</p>");
        });
    }
    // If film format is JSON, parse the film to JSON and send it to the server
    else {
        var filmStringToSend = parseFilmToJSON(filmForm);
        var contentType = "application/json";
        console.log(filmStringToSend);
        ajax(methodType, targetURL, filmStringToSend, contentType, filmFormat, function(data) {
            console.log(data);
            // Display response from the server (receiving JSON response)
            resultRegion.html("<h2>" + data.response + "</h2>" + "<p>" + data.message + "</p>");
        });
    }
}
```

This method served as the main functionality for handling an insert-form and an update-form. See the screenshots below, notably line 85 for the first screenshot, and line 136 for the second:



```javascript
67      // Insert a film into db
68      $("#insert-form").submit(function(e) {
69          e.preventDefault(); // avoid to execute the actual submit of the form.
70
71          // Store the areas where we will display a response from server
72          var resultRegion = $("#message-div");
73
74          // Grab the URL from the form attribute 'action'
75          var targetURL = $(this).attr("action");
76          // Serialize all of the inputs within the form, so we can easily parse a film into XML / JSON later (automatically URL encodes too)
77          var filmForm = $(this).serializeArray();
78
79          // Grab what option the user selected to send the film as (XML or JSON)
80          var filmFormat = $("input[name='formatInsert']:checked").val();
81          console.log(filmFormat);
82
83          // parses the film javascript array 'filmForm' into either XML or JSON depending on 'filmFormat',
84          // and then sends off to the server with jQuery ajax
85          handleFilmPostPut("POST", filmFormat, filmForm, targetURL, resultRegion);
86
87          // Display the response box
88          $("#alert-div").show();
89
90          // Hide the insert pop-up / modal
91          $("#insert-film-modal").modal("toggle");
92      });
```



```javascript
118     // Update the film in the database
119     $("#update-form").submit(function(e) {
120         e.preventDefault(); // avoid to execute the actual submit of the form.
121
122         // Grab the div where we will display response from server
123         var resultRegion = $("#message-div");
124
125         // Get target url
126         var targetURL = $(this).attr("action");
127         // Serialize all form inputs into an array, making it easier to parse into xml / json (automatically url-encodes)
128         var filmForm = $(this).serializeArray();
129
130         // Grab the selected format to send to the server ( XML / JSON )
131         var filmFormat = $("input[name='formatUpdate']:checked").val();
132         console.log(filmFormat);
133
134         // parses the film javascript array 'filmForm' into either XML or JSON depending on 'filmFormat',
135         // and then sends off to the server with jQuery ajax
136         handleFilmPostPut("PUT", filmFormat, filmForm, targetURL, resultRegion);
137
138         // Display response from the server
139         $("#alert-div").show();
140         $("#update-film-modal").modal("toggle");
141     });
```

The code is much more concise and easier to understand after this refactoring took place, making it easier to maintain, debug, and extend when required.

Code Quality

Refactoring is one way to improve code quality – this was previously discussed in the previous section. That section also discussed code-reuse through the use of methods, rather than repeating code (the DRY principle).

This section aims to cover specific topics that I feel simply comes under "Code Quality", such as comments, SOLID principles, and more, and where they are used in my solutions.

Comments and Javadoc

Javadoc is a tool that comes with JDK, it is used to generate Java code documentation in HTML format from the Java Source code, requiring documentation in a predefined format. This is useful for users of a class, as it provides information on that class and its methods to them.

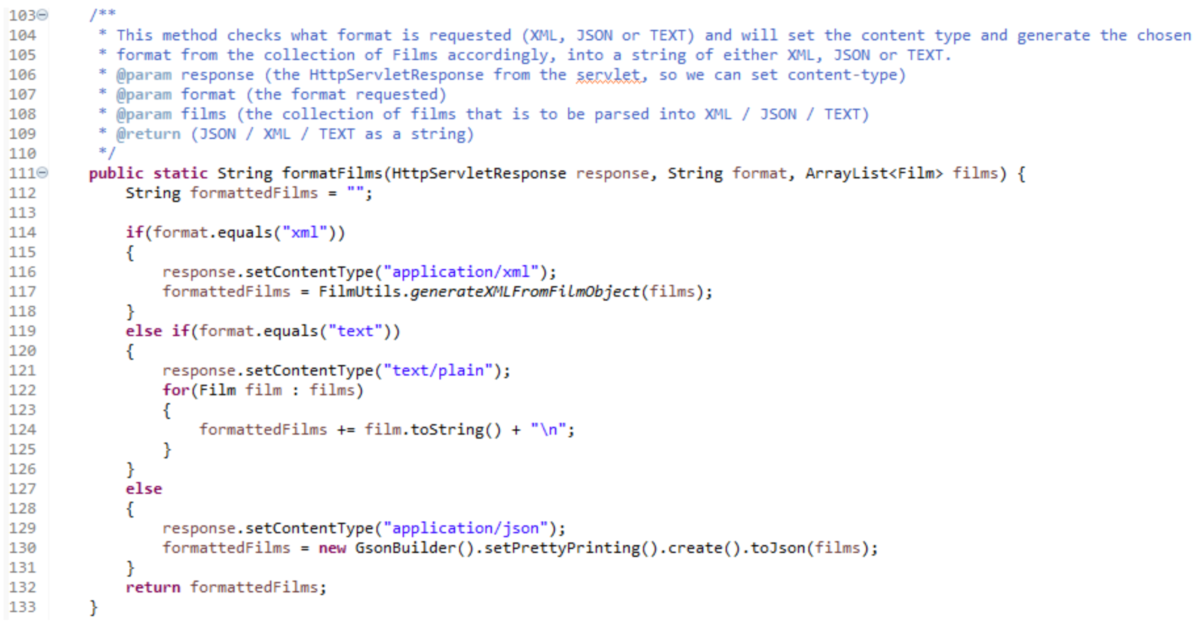
Comments are a readable explanation, or an annotation in the code, and are aimed to provide information to the developer who is modifying that class or method.

It is generally best practice to write code that is self-documenting through practices such as: good variable names, method names, class names – rather than writing too many comments. This is because comments can be viewed as a code-smell – they take up space on the screen, can become outdated quickly unless maintained (confusing developers), and they should be unnecessary to use to begin with, as your code should be well written to the point where they are not required.

Javadoc was used throughout my solutions for the methods and classes where appropriate, to explain the intention of the method (why it exists, what is its goal?). Comments were used too, but not extensively.

Examples of Javadoc and Comments:

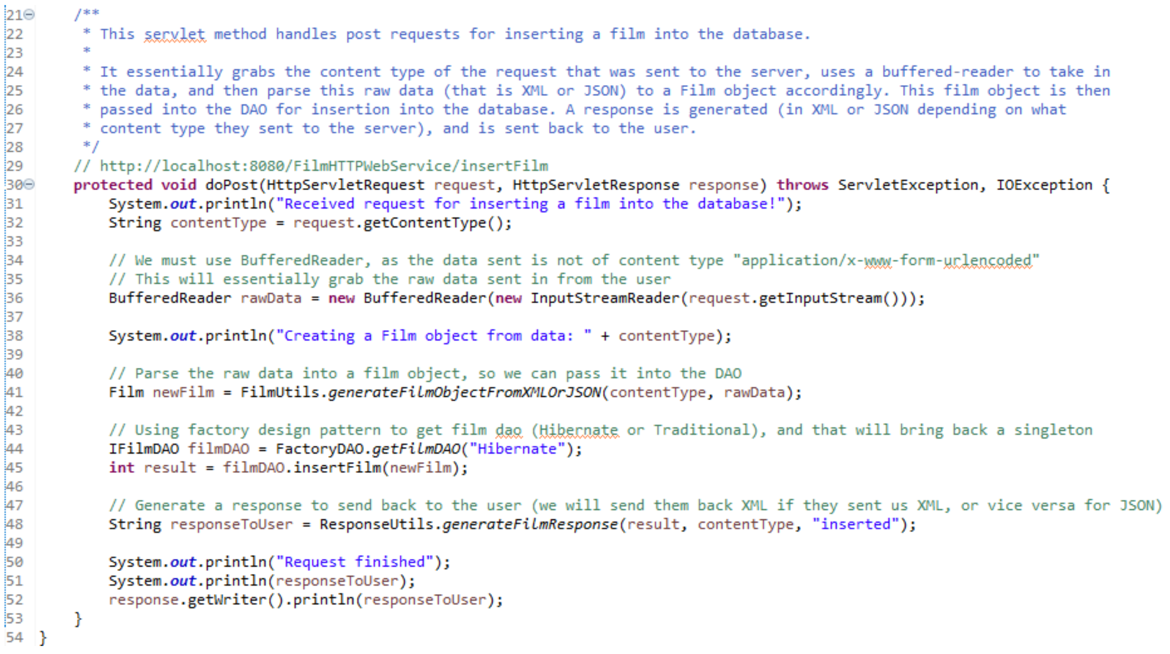
The below screenshots are taken from the "FilmHTTPWebService" project, just to simply demonstrate how I have applied Javadoc. This same structure can be seen throughout all solutions.



```java
    /**
     * This method checks what format is requested (XML, JSON or TEXT) and will set the content type and generate the chosen
     * format from the collection of Films accordingly, into a string of either XML, JSON or TEXT.
     * @param response (the HttpServletResponse from the servlet, so we can set content-type)
     * @param format (the format requested)
     * @param films (the collection of films that is to be parsed into XML / JSON / TEXT)
     * @return (JSON / XML / TEXT as a string)
     */
    public static String formatFilms(HttpServletResponse response, String format, ArrayList<Film> films) {
        String formattedFilms = "";

        if(format.equals("xml"))
        {
            response.setContentType("application/xml");
            formattedFilms = FilmUtils.generateXMLFromFilmObject(films);
        }
        else if(format.equals("text"))
        {
            response.setContentType("text/plain");
            for(Film film : films)
            {
                formattedFilms += film.toString() + "\n";
            }
        }
        else
        {
            response.setContentType("application/json");
            formattedFilms = new GsonBuilder().setPrettyPrinting().create().toJson(films);
        }
        return formattedFilms;
    }
```

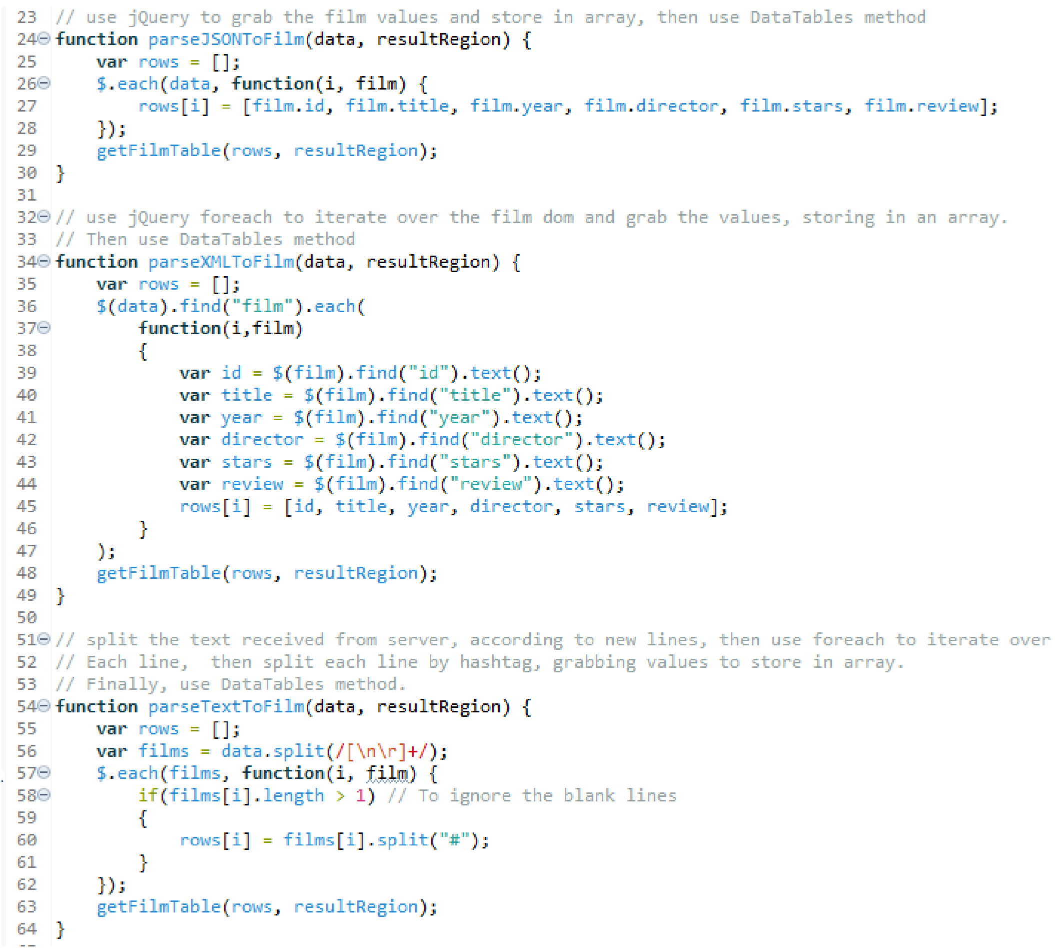
The above screenshot is a utility method in "FilmUtils.java", note the lack of comments – only Javadoc. This is because I felt commenting the individual lines themselves was unnecessary here.

The below screenshot shows “insertFilm.java”, using both Javadoc and comments. In the real-world, I would omit these normal comments, as I feel they are unnecessary, but for the purposes of the assignment and proving I understand parts of the code, I have added them here:



```java
    /**
     * This servlet method handles post requests for inserting a film into the database.
     *
     * It essentially grabs the content type of the request that was sent to the server, uses a buffered-reader to take in
     * the data, and then parse this raw data (that is XML or JSON) to a Film object accordingly. This film object is then
     * passed into the DAO for insertion into the database. A response is generated (in XML or JSON depending on what
     * content type they sent to the server), and is sent back to the user.
     */
    // http://localhost:8080/FilmHTTPWebService/insertFilm
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Received request for inserting a film into the database!");
        String contentType = request.getContentType();

        // We must use BufferedReader, as the data sent is not of content type "application/x-www-form-urlencoded"
        // This will essentially grab the raw data sent in from the user
        BufferedReader rawData = new BufferedReader(new InputStreamReader(request.getInputStream()));

        System.out.println("Creating a Film object from data: " + contentType);

        // Parse the raw data into a film object, so we can pass it into the DAO
        Film newFilm = FilmUtils.generateFilmObjectFromXMLOrJSON(contentType, rawData);

        // Using factory design pattern to get film dao (Hibernate or Traditional), and that will bring back a singleton
        IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
        int result = filmDAO.insertFilm(newFilm);

        // Generate a response to send back to the user (we will send them back XML if they sent us XML, or vice versa for JSON)
        String responseToUser = ResponseUtils.generateFilmResponse(result, contentType, "inserted");

        System.out.println("Request finished");
        System.out.println(responseToUser);
        response.getWriter().println(responseToUser);
    }
}
```

Below is a screenshot of JavaScript, showing how I have applied comments there:



```javascript
// use jQuery to grab the film values and store in array, then use DataTables method
function parseJSONToFilm(data, resultRegion) {
    var rows = [];
    $.each(data, function(i, film) {
        rows[i] = [film.id, film.title, film.year, film.director, film.stars, film.review];
    });
    getFilmTable(rows, resultRegion);
}

// use jQuery foreach to iterate over the film dom and grab the values, storing in an array.
// Then use DataTables method
function parseXMLToFilm(data, resultRegion) {
    var rows = [];
    $(data).find("film").each(
        function(i,film)
        {
            var id = $(film).find("id").text();
            var title = $(film).find("title").text();
            var year = $(film).find("year").text();
            var director = $(film).find("director").text();
            var stars = $(film).find("stars").text();
            var review = $(film).find("review").text();
            rows[i] = [id, title, year, director, stars, review];
        }
    );
    getFilmTable(rows, resultRegion);
}

// split the text received from server, according to new lines, then use foreach to iterate over
// Each line,  then split each line by hashtag, grabbing values to store in array.
// Finally, use DataTables method.
function parseTextToFilm(data, resultRegion) {
    var rows = [];
    var films = data.split(/[\n\r]+/);
    $.each(films, function(i, film) {
        if(films[i].length > 1) // To ignore the blank lines
        {
            rows[i] = films[i].split("#");
        }
    });
    getFilmTable(rows, resultRegion);
}
```

Naming conventions

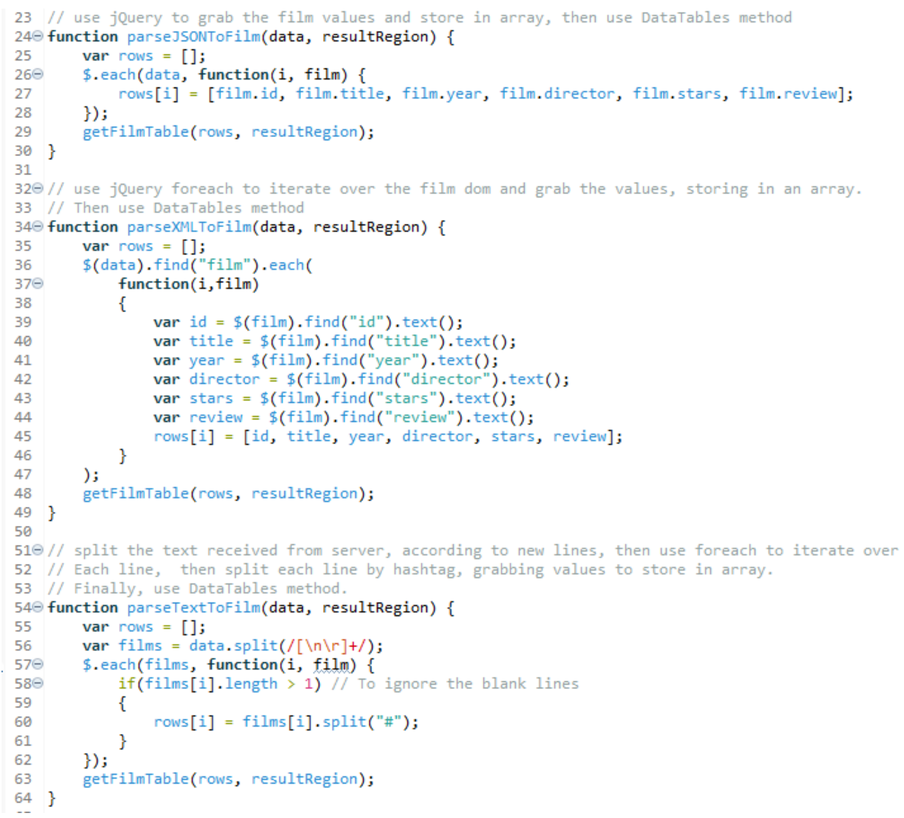
The way in which variables, methods and classes are named help make the code become self-documenting. It is good practice to give everything a clear and understandable name. Naming a variable “x” for example, does not provide any information on what this variable actually holds or its intentions. The better the names are, the easier it is for the maintainability of the codebase; additionally, it also means you should not need many comments – or even none at all.

Each programming language has its own preferred way to name its structures too – examples include camel-case and kebab-case, that may be a convention to use in certain parts. Another example is how interfaces in some languages should start with the letter ‘I’.

Example of Good Naming:

Throughout the project, you should see clearly named variables, methods and classes. This section aims to show some of these.

The below screenshot is JavaScript from the front-end, it is the same for all projects. Note the naming of each of these functions, their variables and what they do:



```
// use jQuery to grab the film values and store in array, then use DataTables method
function parseJSONToFilm(data, resultRegion) {
    var rows = [];
    $.each(data, function(i, film) {
        rows[i] = [film.id, film.title, film.year, film.director, film.stars, film.review];
    });
    getFilmTable(rows, resultRegion);
}

// use jQuery foreach to iterate over the film dom and grab the values, storing in an array.
// Then use DataTables method
function parseXMLToFilm(data, resultRegion) {
    var rows = [];
    $(data).find("film").each(
        function(i,film)
        {
            var id = $(film).find("id").text();
            var title = $(film).find("title").text();
            var year = $(film).find("year").text();
            var director = $(film).find("director").text();
            var stars = $(film).find("stars").text();
            var review = $(film).find("review").text();
            rows[i] = [id, title, year, director, stars, review];
        }
    );
    getFilmTable(rows, resultRegion);
}

// split the text received from server, according to new lines, then use foreach to iterate over
// Each line,  then split each line by hashtag, grabbing values to store in array.
// Finally, use DataTables method.
function parseTextToFilm(data, resultRegion) {
    var rows = [];
    var films = data.split(/[\n\r]+/);
    $.each(films, function(i, film) {
        if(films[i].length > 1) // To ignore the blank lines
        {
            rows[i] = films[i].split("#");
        }
    });
    getFilmTable(rows, resultRegion);
}
```

The above functions convert a given format (XML, JSON, Text) to a film object to display in a table. “parseJSONToFilm”, “parseXMLToFilm”, “parseTextToFilm”, gives a clear explanation on its purpose, and it is consistent for each format.

Other examples of this can be seen throughout the whole codebase.

Loose Coupling

A loosely coupled system is one where each component depends on another to the least possible extent – they should have the as little direct knowledge on an element as possible. It is essentially the opposite of tight coupling; these points kept coming up in the design patterns section.

It is beneficial to have a loosely coupled system, some of these benefits include:

- Being able to replace components with an alternative implementation that provides the same services (as seen in my project solutions, with the DAO)
- Easier to test (as the code is not dependent on other components)
- Extensibility / scalability – it is easier to add new features to the system
- Easier to read and understand

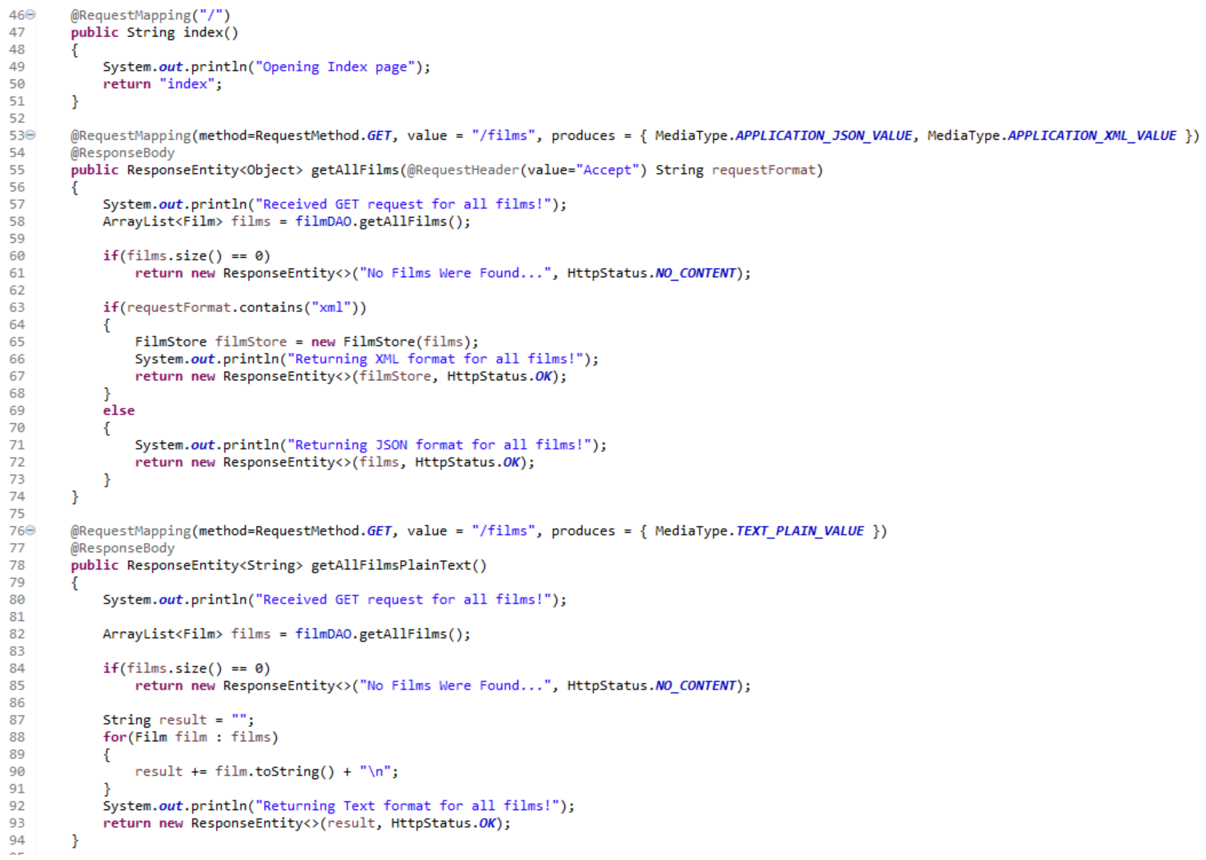
To see examples of where I have implemented loose coupling in my code, refer to the sections "Data Access Object (DAO)", and "Factory Method Pattern" within this documentation, as the screenshots have already been provided there.

Consistent Indentation and brackets

Keeping consistent with indentation and brackets is a simple and easy way to improve code quality instantly. There is no right or wrong way to indent code – what matters is that you stay consistent with one style.

Example of Consistent Indentation

The below screenshot is from the controller of "FilmRESTfulWebService", notice how each line of code is indented the same amount – with tabs, and that the brackets are consistently on a new line:



```java
    @RequestMapping("/")
    public String index()
    {
        System.out.println("Opening Index page");
        return "index";
    }

    @RequestMapping(method=RequestMethod.GET, value = "/films", produces = { MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
    @ResponseBody
    public ResponseEntity<Object> getAllFilms(@RequestHeader(value="Accept") String requestFormat)
    {
        System.out.println("Received GET request for all films!");
        ArrayList<Film> films = filmDAO.getAllFilms();

        if(films.size() == 0)
            return new ResponseEntity<>("No Films Were Found...", HttpStatus.NO_CONTENT);

        if(requestFormat.contains("xml"))
        {
            FilmStore filmStore = new FilmStore(films);
            System.out.println("Returning XML format for all films!");
            return new ResponseEntity<>(filmStore, HttpStatus.OK);
        }
        else
        {
            System.out.println("Returning JSON format for all films!");
            return new ResponseEntity<>(films, HttpStatus.OK);
        }
    }

    @RequestMapping(method=RequestMethod.GET, value = "/films", produces = { MediaType.TEXT_PLAIN_VALUE })
    @ResponseBody
    public ResponseEntity<String> getAllFilmsPlainText()
    {
        System.out.println("Received GET request for all films!");

        ArrayList<Film> films = filmDAO.getAllFilms();

        if(films.size() == 0)
            return new ResponseEntity<>("No Films Were Found...", HttpStatus.NO_CONTENT);

        String result = "";
        for(Film film : films)
        {
            result += film.toString() + "\n";
        }
        System.out.println("Returning Text format for all films!");
        return new ResponseEntity<>(result, HttpStatus.OK);
    }
```

Folder Structure & Packages

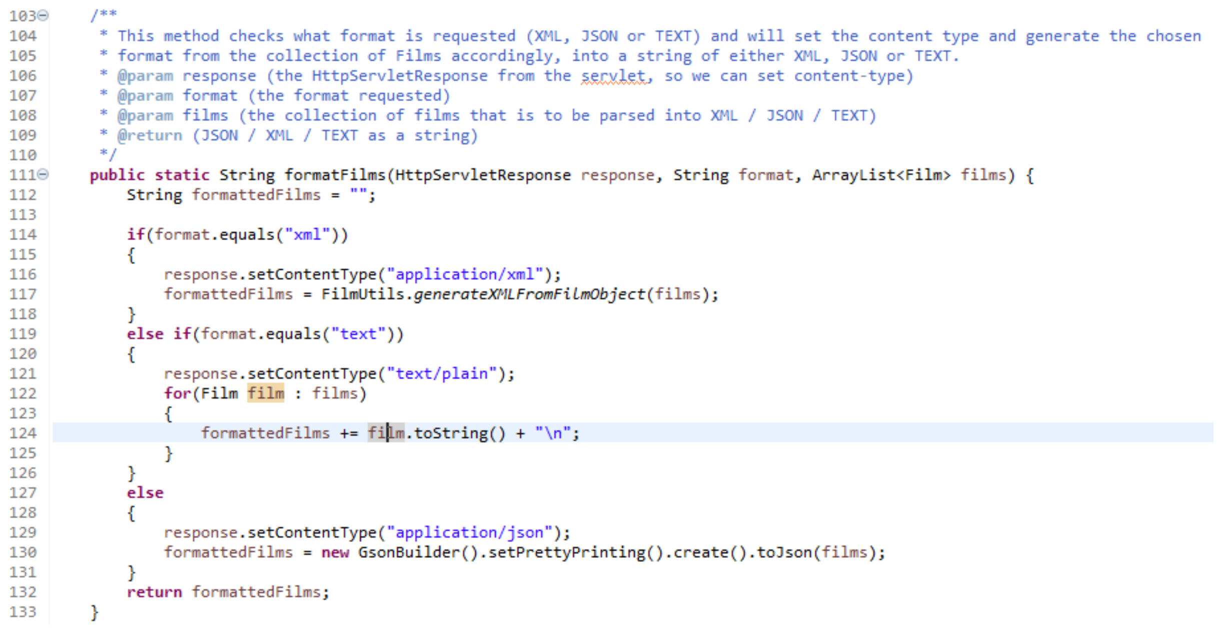
The layout of the project makes a big impact on quality – this affects maintenance and readability of the project in general. Having a clear, consistent layout that can be associated with other styles of layouts commonly seen before (perhaps by following a design pattern), greatly aids the maintainability of a project. Examples of this can be seen in my project solutions, and screenshots for this are located in the "Model-View-Controller (MVC)" section of this document.

General Limit Line Length

Having a general limit line for each line of code or comment in your application is one of the easiest ways to improve code quality. Keeping consistent with this line length – and picking a length that doesn't require you to scroll side ways to begin with – makes the code much easier to read. A clear example of this being followed in my project solutions are with the extensve Javadoc comments.

Example of General Limit Line Length

Below is a screenshot of the "FilmUtils.java" class, regarding the method "formatFilms". Note how a new line is entered on line 105 during the same sentence. This is to avoid having to potentially scroll sideways, or look too far away from the rest of the code:



```
103      /**
104       * This method checks what format is requested (XML, JSON or TEXT) and will set the content type and generate the chosen
105       * format from the collection of Films accordingly, into a string of either XML, JSON or TEXT.
106       * @param response (the HttpServletResponse from the servlet, so we can set content-type)
107       * @param format (the format requested)
108       * @param films (the collection of films that is to be parsed into XML / JSON / TEXT)
109       * @return (JSON / XML / TEXT as a string)
110       */
111      public static String formatFilms(HttpServletResponse response, String format, ArrayList<Film> films) {
112          String formattedFilms = "";
113  
114          if(format.equals("xml"))
115          {
116              response.setContentType("application/xml");
117              formattedFilms = FilmUtils.generateXMLFromFilmObject(films);
118          }
119          else if(format.equals("text"))
120          {
121              response.setContentType("text/plain");
122              for(Film film : films)
123              {
124                  formattedFilms += film.toString() + "\n";
125              }
126          }
127          else
128          {
129              response.setContentType("application/json");
130              formattedFilms = new GsonBuilder().setPrettyPrinting().create().toJson(films);
131          }
132          return formattedFilms;
133      }
```

Keep it Simple, Stupid (KISS)

This acronym is well known as it is a simple but descriptive way of remembering to keep the code clean and simple. This helps with the maintainability of the codebase, so it is best to follow from the beginning. Part of following this involves not over-complicating the code – the simpler the code is to understand, the better – using complex programming techniques for a simple feature is overkill – it wastes time and will be harder to understand by other developers. Another way to follow this acronym is to keep your methods short – they should not be longer than the screen preferably; the number of lines on average changes from developer to developer, but it can be anywhere from 20-50 lines of code.

Having clear and simple code will make it easier for future developers to understand and to modify.

Following what was discussed in this section, as well as other ones above, will put you on the right tracks of following KISS.

SOLID Principles

Following the SOLID principles is good practice when designing code. These principles are:

- The Single Responsibility Principle – each class should have a single responsibility. This closely relates to the notion of the separation of concerns principle
- The Open-Closed Principle – classes can be extended by other classes, but not modified. This relates to data-hiding

- The Liskov-Substitution Principle – objects can be replaced by instances of their subtypes and still work correctly (an example can be seen with the IFilmDAO in the “FilmHTTPWebService” project)
- The Interface Segregation Principle – it is better to have many, smaller interfaces over one general purpose interface
- Dependency Inversion Principle – it is better to depend upon abstraction such as interfaces, rather than concrete classes; this avoids tight coupling

Libraries / Frameworks

This section aims to cover the libraries and frameworks used in my project solutions.

jQuery

jQuery is a lightweight JavaScript library designed to make it much easier to use JavaScript on the front-end. The library essentially takes many of the common tasks performed in JavaScript (that require many lines of code to perform) and wraps them into methods that can be executed with one single line of code.

jQuery was used throughout my front-end scripts for the projects “FilmHTTPWebService”, “FilmCloudWebService” and “FilmRESTfulWebService”. The scripts for these projects are the exact same, except for the URL’s (as they are different API’s).

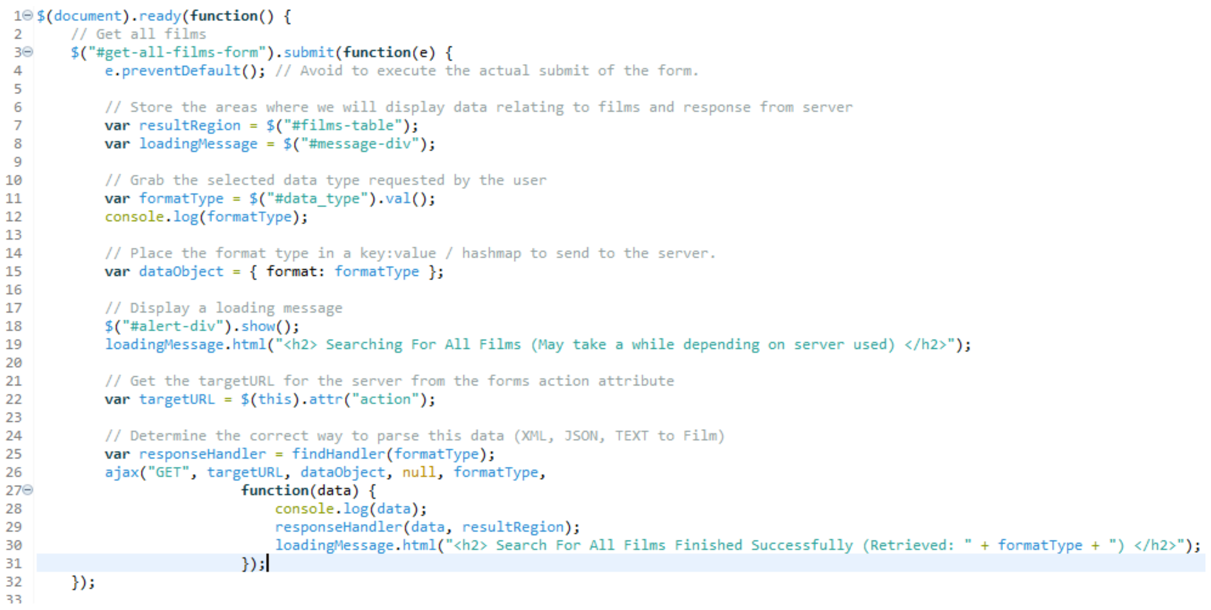
What is AJAX?

AJAX stands for **A**synchronous **J**avaScript **A**nd **X**ML. It is a technique commonly used to create fast and dynamic web pages; these webpages can be updated asynchronously (without the need to reload the webpage) by making requests to the server behind the scenes with JavaScript. This makes it possible to update sections of a webpage without having to reload it. Despite the name, it is possible to send and retrieve information in formats other than XML, such as JSON.

AJAX has been used throughout the front-end of my applications for all CRUD features.

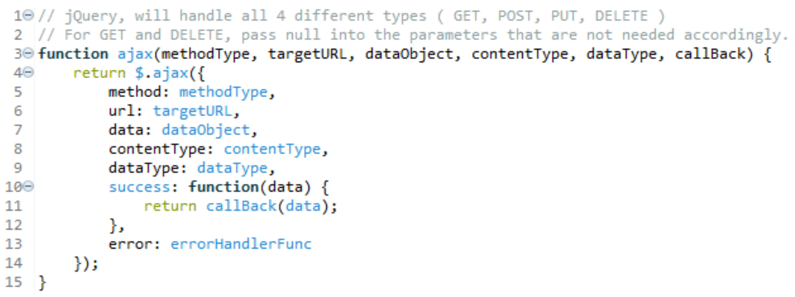
Example of AJAX implementation using jQuery in Solutions:

The below screenshot shows a jQuery event handler function for when the form with the id “get-all-films-form” is submitted:



```
$(document).ready(function() {
    // Get all films
    $("#get-all-films-form").submit(function(e) {
        e.preventDefault(); // Avoid to execute the actual submit of the form.

        // Store the areas where we will display data relating to films and response from server
        var resultRegion = $("#films-table");
        var loadingMessage = $("#message-div");

        // Grab the selected data type requested by the user
        var formatType = $("#data_type").val();
        console.log(formatType);

        // Place the format type in a key:value / hashmap to send to the server.
        var dataObject = { format: formatType };

        // Display a loading message
        $("#alert-div").show();
        loadingMessage.html("<h2> Searching For All Films (May take a while depending on server used) </h2>");

        // Get the targetURL for the server from the forms action attribute
        var targetURL = $(this).attr("action");

        // Determine the correct way to parse this data (XML, JSON, TEXT to Film)
        var responseHandler = findHandler(formatType);
        ajax("GET", targetURL, dataObject, null, formatType,
                    function(data) {
                        console.log(data);
                        responseHandler(data, resultRegion);
                        loadingMessage.html("<h2> Search For All Films Finished Successfully (Retrieved: " + formatType + ") </h2>");
                    });
    });
```

On line 3 you can see the beginning of the function – it starts with a '$' sign, indicating that its jQuery. Throughout the method (line 7, 8, 11, 18, 19, 22 etc..) jQuery is used to target elements on the webpage. On line 26 you can see a JavaScript function being called – the code inside this function is jQuery AJAX:



```
// jQuery, will handle all 4 different types ( GET, POST, PUT, DELETE )
// For GET and DELETE, pass null into the parameters that are not needed accordingly.
function ajax(methodType, targetURL, dataObject, contentType, dataType, callBack) {
    return $.ajax({
        method: methodType,
        url: targetURL,
        data: dataObject,
        contentType: contentType,
        dataType: dataType,
        success: function(data) {
            return callBack(data);
        },
        error: errorHandlerFunc
    });
}
```

The jQuery method "$.ajax" is the jQuery function that performs an asynchronous HTTP request, the setting:

- "method" determines whether it is a GET, POST, PUT or DELETE request
- "url" determines the URL to send the request too
- "data" determines the data to be sent to the server
- "contentType" determines what type of data is being sent over, such as application/json. By default it's "application/x-www-form-urlencoded;charset=UTF-8".
- "dataType" determines the type of data we are expecting back from the server, such as "xml" or "json".
- "success" is the function that is called if the request succeeds
- "error" is the function that is called if the request fails

Refer to the section Refactoring, namely "Examples of Refactoring – Frontend JavaScript/jQuery" to see more examples of jQuery and how it was refactored. The rest of the event handlers for the insert, update and delete forms follow a similar structure.

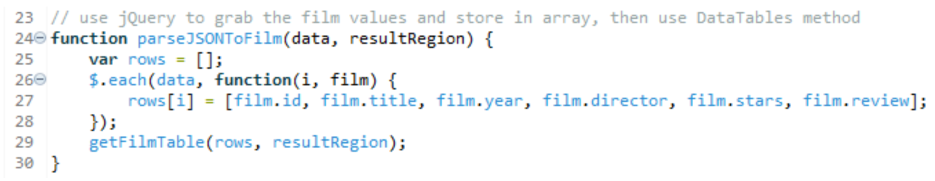
DataTables with Bootstrap (UI library)

DataTables is a plugin for jQuery that enables you to create a HTML table (with advanced features, such as pagination, instant search, multi-column ordering, and more) dynamically. You can use multiple data sources for creating this table: JavaScript arrays, AJAX sourced data (from servers that return JSON), and server-side processing.

Using a library like this:

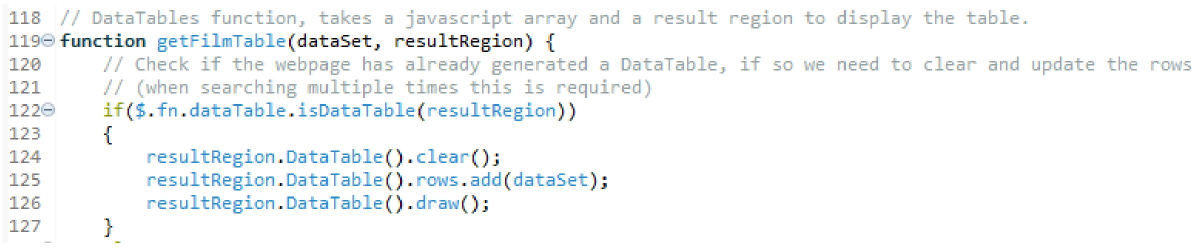
- Saves time as you do not have to write these functions yourself
- Means you are using code that has been tested already
- Reduces lines of code in your scripts and HTML, helping with reusability and maintainability

My project solutions used a JavaScript array as a data source for DataTables. My methods parse the result that comes back from the server (JSON, XML or Text), into a JavaScript array that is then passed into the "getFilmTable" function (a function that wraps DataTables):



```
// use jQuery to grab the film values and store in array, then use DataTables method
function parseJSONToFilm(data, resultRegion) {
    var rows = [];
    $.each(data, function(i, film) {
        rows[i] = [film.id, film.title, film.year, film.director, film.stars, film.review];
    });
    getFilmTable(rows, resultRegion);
}
```

The below screenshot shows the beginning of the “getFilmTable” method:

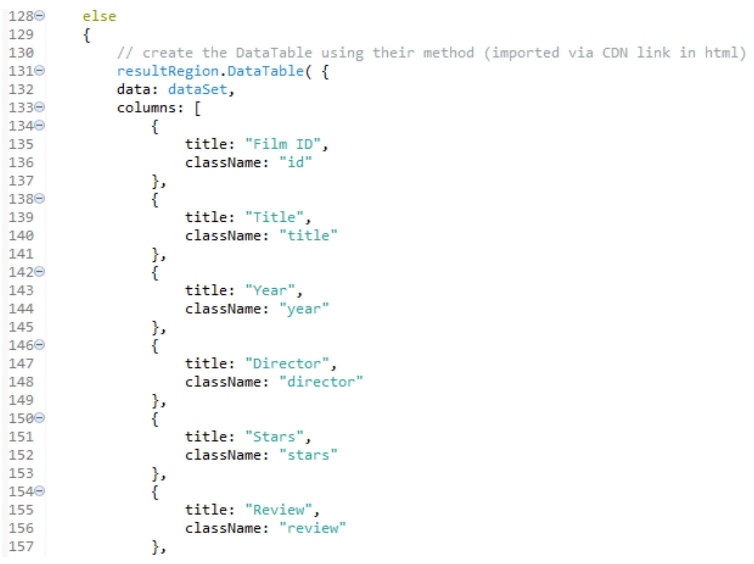


```
118 // DataTables function, takes a javascript array and a result region to display the table.
119 function getFilmTable(dataSet, resultRegion) {
120     // Check if the webpage has already generated a DataTable, if so we need to clear and update the rows
121     // (when searching multiple times this is required)
122     if($.fn.dataTable.isDataTable(resultRegion))
123     {
124         resultRegion.DataTable().clear();
125         resultRegion.DataTable().rows.add(dataSet);
126         resultRegion.DataTable().draw();
127     }
```

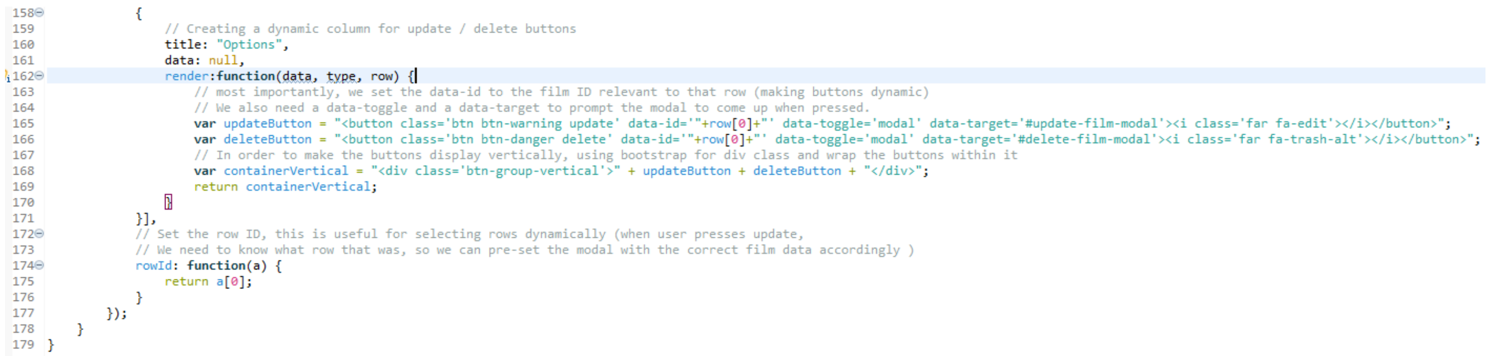
The “dataset” parameter is the JavaScript array that we have sent in (originally was JSON, XML or Text returned from the server). The “resultRegion” parameter is a table element, in these solutions it is a table with the id “films-table”.

The if statement on line 122 checks if the element “resultRegion” has already been instantiated as a DataTable – if it has: the DataTable is cleared, the new rows are added from the “dataSet”, and then the new DataTable is drawn.

The below screenshot shows the rest of the “getFilmTable” method that ultimately instantiates a DataTable if it has not yet already been created:



```
128     else
129     {
130         // create the DataTable using their method (imported via CDN link in html)
131         resultRegion.DataTable( {
132         data: dataSet,
133         columns: [
134             {
135                 title: "Film ID",
136                 className: "id"
137             },
138             {
139                 title: "Title",
140                 className: "title"
141             },
142             {
143                 title: "Year",
144                 className: "year"
145             },
146             {
147                 title: "Director",
148                 className: "director"
149             },
150             {
151                 title: "Stars",
152                 className: "stars"
153             },
154             {
155                 title: "Review",
156                 className: "review"
157             },
```



```
158             {
159                 // Creating a dynamic column for update / delete buttons
160                 title: "Options",
161                 data: null,
162                 render:function(data, type, row) {
163                     // most importantly, we set the data-id to the film ID relevant to that row (making buttons dynamic)
164                     // We also need a data-toggle and a data-target to prompt the modal to come up when pressed.
165                     var updateButton = "<button class='btn btn-warning update' data-id='"+row[0]+"' data-toggle='modal' data-target='#update-film-modal'><i class='far fa-edit'></i></button>";
166                     var deleteButton = "<button class='btn btn-danger delete' data-id='"+row[0]+"' data-toggle='modal' data-target='#delete-film-modal'><i class='far fa-trash-alt'></i></button>";
167                     // In order to make the buttons display vertically, using bootstrap for div class and wrap the buttons within it
168                     var containerVertical = "<div class='btn-group-vertical'>" + updateButton + deleteButton + "</div>";
169                     return containerVertical;
170                 }
171             }],
172             // Set the row ID, this is useful for selecting rows dynamically (when user presses update,
173             // We need to know what row that was, so we can pre-set the modal with the correct film data accordingly )
174             rowId: function(a) {
175                 return a[0];
176             }
177         });
178     }
179 }
```

On line 131 you can see the “DataTable” method being used. The “data” parameter is the data source (in this case, a JavaScript array) you wish to put into the HTML table. The “columns” parameter is used to set the table headers (the “title” parameter is the value inside the “th” tag it creates, and the “className” parameter adds a class to each of the headers.) On line 158, a dynamic column is created with the table header of “Options”. The render function on line 162 dynamically adds dynamic update and delete buttons to the table, with the appropriate attributes (most importantly, the film-id for that specific row is stored in each button as the attribute “data-id”). On line 174, the row id for each row is dynamically added – setting the film ID as the unique row id.

Data-Centric AJAX vs Content-Centric AJAX vs Script-Centric AJAX

There are a few different approaches to sending data from the server to an AJAX client – these are: Data-Centric, Content-Centric, and Script-Centric.

Data-Centric is where the server sends the raw data to the client so the client can parse the data and then build the HTML themselves. This was the approach used in the HTTP, REST and Cloud project solutions. With jQuery, using more complex methods other than “$.load” would be required, as the data must be parsed before being displayed, so methods such as “$.ajax” are required.

Content-Centric is where the server builds the HTML from the data and sends it to the client ready to be displayed straight away. The only step the client is required to do in this approach is to insert the content into the page; style-sheets can be used by the client to customize the look. With jQuery, the “$.load” function would suffice for this, as the data itself does not need to be parsed.

Script-Centric is where the server sends the JavaScript functions and the data to the client, and the client executes these functions. Many web applications do use this, such as Office365 online. One drawback to this approach is that it requires the server to know to much about the client.

Spring Framework

The Spring Framework is both an application framework and IoC container for Java. This framework addresses many of the common problems that Java developers face – providing tried and tested code ready to use.

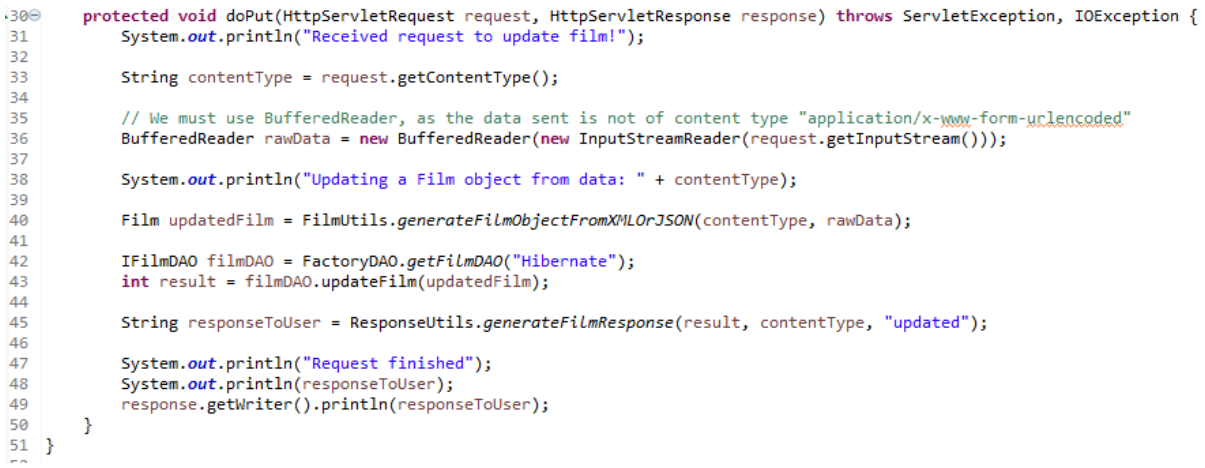
Using the Spring Framework provides many benefits over using no framework at all:

- Many problems are already solved for us – making code much more concise and even optimized, improving productivity and reducing errors within the code
- It facilitates good programming practices – programming to an interface
- It is a modular framework – you only have to use parts that you actually need
- XML and annotation based configuration is supported
- Supports Inversion of Control and Dependency Injection – helping with loose coupling

I used Spring in the project “FilmRESTfulWebService” in order to create a RESTful API. When comparing this project to “FilmHTTPWebService” you can see the difference in how much more concise and cleaner my code is using Spring – many utility methods for things such as JAXB are no longer required, as this is done automatically in the controller with Spring annotations (namely @RequestMapping).

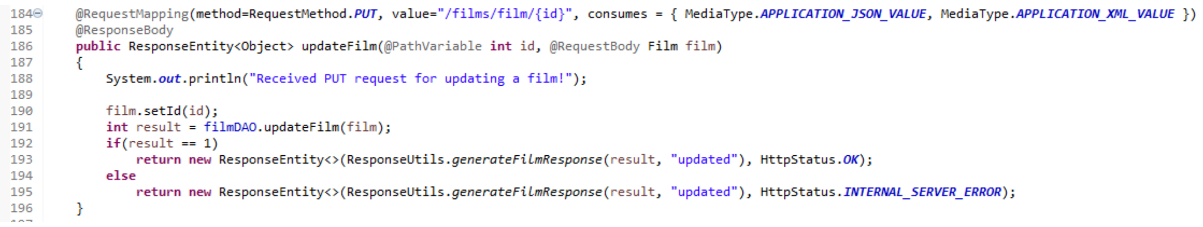
No Spring (FilmHTTPWebService) vs Spring (FilmRESTfulWebService)

Below is a screenshot of the “updateFilm” Servlet in “FilmHTTPWebService”:



```java
    protected void doPut(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Received request to update film!");

        String contentType = request.getContentType();

        // We must use BufferedReader, as the data sent is not of content type "application/x-www-form-urlencoded"
        BufferedReader rawData = new BufferedReader(new InputStreamReader(request.getInputStream()));

        System.out.println("Updating a Film object from data: " + contentType);

        Film updatedFilm = FilmUtils.generateFilmObjectFromXMLOrJSON(contentType, rawData);

        IFilmDAO filmDAO = FactoryDAO.getFilmDAO("Hibernate");
        int result = filmDAO.updateFilm(updatedFilm);

        String responseToUser = ResponseUtils.generateFilmResponse(result, contentType, "updated");

        System.out.println("Request finished");
        System.out.println(responseToUser);
        response.getWriter().println(responseToUser);
    }
}
```

Below is the method for updating a film in the controller for the project “FilmRESTfulWebService”:



```java
    @RequestMapping(method=RequestMethod.PUT, value="/films/film/{id}", consumes = { MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
    @ResponseBody
    public ResponseEntity<Object> updateFilm(@PathVariable int id, @RequestBody Film film)
    {
        System.out.println("Received PUT request for updating a film!");

        film.setId(id);
        int result = filmDAO.updateFilm(film);
        if(result == 1)
            return new ResponseEntity<>(ResponseUtils.generateFilmResponse(result, "updated"), HttpStatus.OK);
        else
            return new ResponseEntity<>(ResponseUtils.generateFilmResponse(result, "updated"), HttpStatus.INTERNAL_SERVER_ERROR);
    }
```

When comparing the two, straight away you notice the difference in lines of code – the second screenshot is much more concise – the data is parsed from an XML or JSON object to a Film object for us automatically. This means we do not need to have utility methods that use JAXB explicitly, or GSON, to convert to and from a film object.

Hibernate

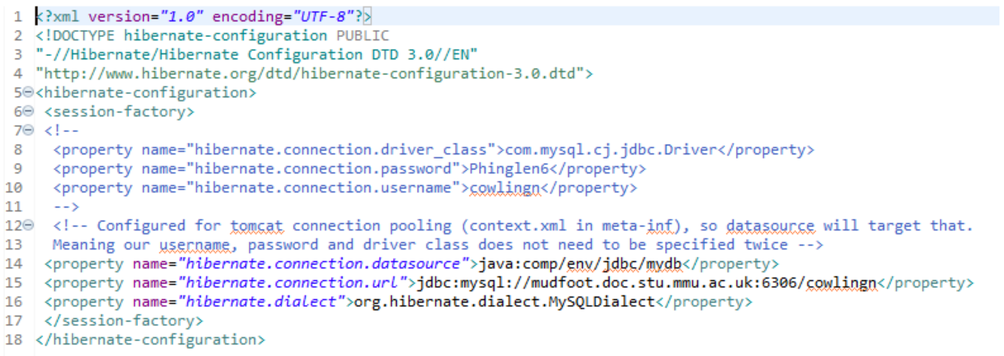
Hibernate is a framework designed to simplify the development of an application when interacting with databases; it's a lightweight, Object-Relational-Mapping (ORM) tool that implements the Java-Persistence-API (JPA) specifications for data persistence.

Using Hibernate simplifies all CRUD features – data creation, manipulation and access. The benefits of using it are:

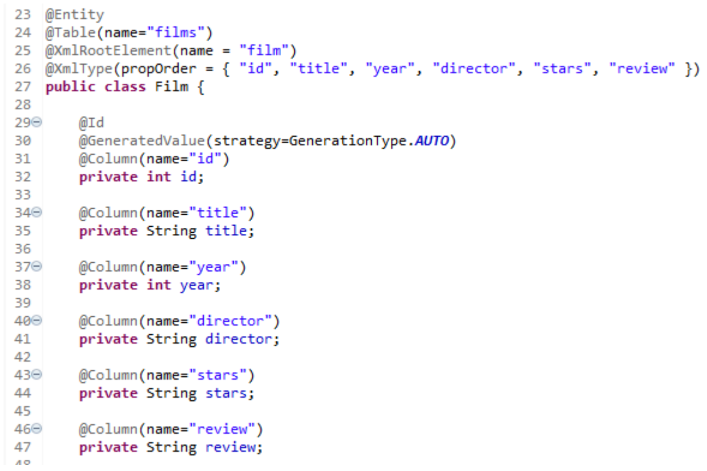
- Its lightweight and open-source
- Provides fast performance due to cache being used
- Allows us to perform database independent queries – our queries do not have to be tailored to the database specifically (using HQL – Hibernate Query Language)
- We can automatically create tables if required
- Complex queries such as joining multiple tables is much easier using Hibernate

Setting up Hibernate requires the jar files (or use Gradle / Maven), a hibernate.cfg.xml file, and either annotations on your model or an XML configuration that maps this.

Below you can see the hibernate.cfg.xml file used in my projects:


```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.cj.jdbc.Driver</property>
  <property name="hibernate.connection.password">Phinglen6</property>
  <property name="hibernate.connection.username">cowlingn</property>
  -->
  <!-- Configured for tomcat connection pooling (context.xml in meta-inf), so datasource will target that.
  Meaning our username, password and driver class does not need to be specified twice -->
 <property name="hibernate.connection.datasource">java:comp/env/jdbc/mydb</property>
 <property name="hibernate.connection.url">jdbc:mysql://mudfoot.doc.stu.mmu.ac.uk:6306/cowlingn</property>
 <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
 </session-factory>
</hibernate-configuration>
```

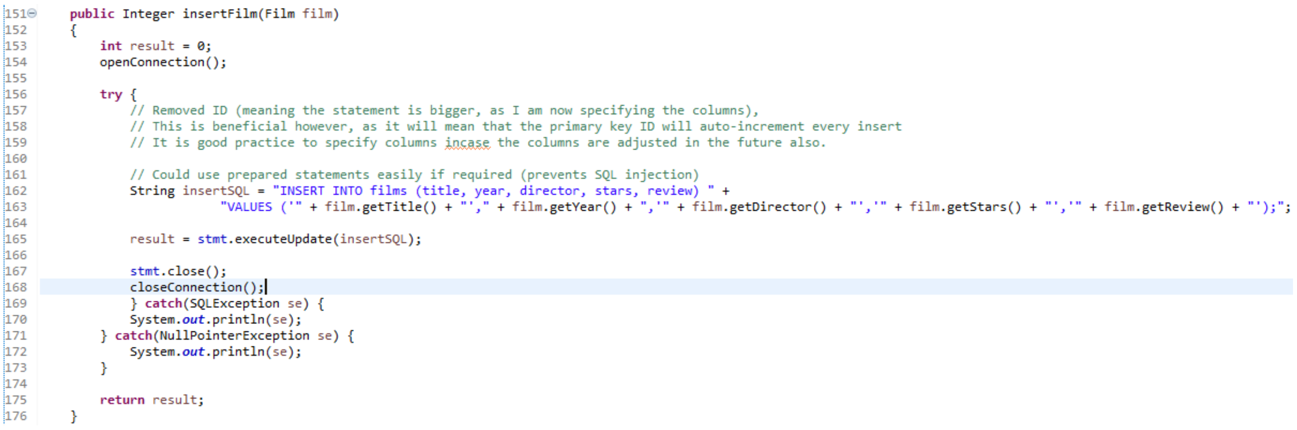
Below is a screenshot of the annotations used to map my model to the database:


```java
@Entity
@Table(name="films")
@XmlRootElement(name = "film")
@XmlType(propOrder = { "id", "title", "year", "director", "stars", "review" })
public class Film {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="id")
    private int id;

    @Column(name="title")
    private String title;

    @Column(name="year")
    private int year;

    @Column(name="director")
    private String director;

    @Column(name="stars")
    private String stars;

    @Column(name="review")
    private String review;
```

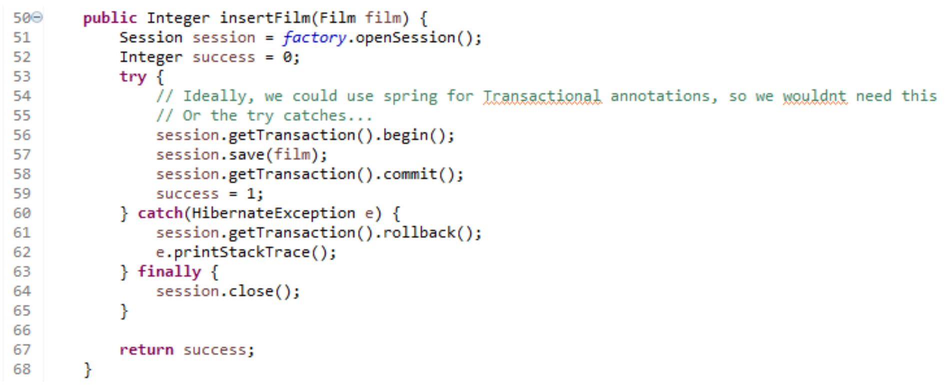
The @Entity annotation lets Hibernate know that this model is an entity in the database. The @Table annotation specifies what the name of the entity is in the database. The @Id annotation is used to specify what the primary key is. The @GeneratedValue annotation lets Hibernate know to auto-increment the ID upon the creation of new films. The @Column attribute is used to map each variable to a column in the database.

No Hibernate (FilmTraditionalDAO.java) vs Hibernate (FilmHibernateDAO.java)

The below screenshot shows the functionality to insert a film when we are not using Hibernate:


```java
    public Integer insertFilm(Film film)
    {
        int result = 0;
        openConnection();

        try {
            // Removed ID (meaning the statement is bigger, as I am now specifying the columns),
            // This is beneficial however, as it will mean that the primary key ID will auto-increment every insert
            // It is good practice to specify columns incase the columns are adjusted in the future also.

            // Could use prepared statements easily if required (prevents SQL injection)
            String insertSQL = "INSERT INTO films (title, year, director, stars, review) " +
                        "VALUES ('" + film.getTitle() + "'," + film.getYear() + ",'" + film.getDirector() + "','" + film.getStars() + "','" + film.getReview() + "');";

            result = stmt.executeUpdate(insertSQL);

            stmt.close();
            closeConnection();
            } catch(SQLException se) {
            System.out.println(se);
        } catch(NullPointerException se) {
            System.out.println(se);
        }

        return result;
    }
```

The below screenshot is the provides the same functionality for inserting a film, but uses hibernate:



```
    public Integer insertFilm(Film film) {
        Session session = factory.openSession();
        Integer success = 0;
        try {
            // Ideally, we could use spring for Transactional annotations, so we wouldnt need this
            // Or the try catches...
            session.getTransaction().begin();
            session.save(film);
            session.getTransaction().commit();
            success = 1;
        } catch(HibernateException e) {
            session.getTransaction().rollback();
            e.printStackTrace();
        } finally {
            session.close();
        }

        return success;
    }
```

When comparing the two, the most notable difference is the query that had to be written manually in the first screenshot, compared to not having to write any query at all in the second – rather, we just use the method "save". This is beneficial as our code is less likely to contain any bugs from writing these ourselves, and it makes the code easier to read and understand. As stated before, hibernate is optimized, so using it will also mean faster operations.

Conclusion

Following sound software engineering techniques and design patterns (such as the ones described in this document) produces code that is: reusable, maintainable, expandable / scalable, readable, teamwork friendly, and more! They are constantly utilized throughout the working world, especially in enterprise environments where you are dealing with mammoth-sized codebases that would otherwise be unmanageable if it was not for these techniques and patterns.

Using frameworks such as Spring or Hibernate where possible is greatly recommended as it will ultimately improve your productivity since the majority of the main problems are solved for you and the code is ready to use (less chance of bugs, do not have to write this functionality yourself, it is optimized etc..). You do not want to waste time reinventing the wheel in this industry – it is fast-paced, and time is precious!

This document has also aimed to link these sound SE techniques and design patterns to the project solutions I created – demonstrating them in action. The definitions of each of these techniques and patterns can be seen in the sections throughout this document, along with the benefits, possible drawbacks, alternative approaches, and examples of it being used in my assignment (if it was at all). This document satisfies Criteria 7 in the assignment specification.

The proof of working code is a document separate to this, demonstrating that Criteria 1-6 has been satisfied.

The projects "FilmHTTPWebService", "FilmCloudWebService", "FilmRESTfulWebService", "FilmSOAPWebService" and FilmSOAPWebServiceClient" are provided with a README file, so that you can open these up and give them a test.