Critical Analysis

Preliminary **1**

Backend & Database **1**

- Film Model 1
- Film DAO 3
- SQL Operations Interface 4
  - Prepared Statements 5
  - Connection Pool 6
- Converting to data exchange formats 8
  - Strategy pattern 8
  - Formatting libraries 9
- Example servlets 12
  - RESTful servlet 12

Frontend **14**

- Design decisions 14
  - Frontend library 14
  - Formatting and Linting 15
- Styling components 15
- React App 16
- Home page 17
  - Example - Sending data 18
  - Example - Retrieving data 21

Preliminary

I have divided up evidence of my code working and evidence of completed sections into two separate PDFs which can be found at the root of the project. Since having taken the screenshots of my code, I have made some alterations to my code including converting the frontend to TypeScript from JavaScript; aliasing imports, renaming and moving functions comprising the frontend for consistency and readability; adding JavaDocs; refactoring and renaming some methods; converting some methods to reflect the Strategy pattern. No new functionality has been added since these PDFs were created.

I was unable to get my frontend functioning using Google Cloud Platform so I have it deployed on Amazon Web Services whilst the backend and database are hosted on Google's GCP SQL and App Engine.

In the screenshots of the code I have taken, I have purposefully left in the file path at the top of the screenshot, the method/function names and line numbers to indicate where the code examples can be found in the project's source files.

Backend & Database

Film Model

The foundation of the project is the Model that comprises the Film object, this is composed of the attributes: id, title, year, director, stars, review. For this model, I decided to use the "Builder" creational pattern. The Builder pattern aims to simplify the construction of objects and encapsulate the construction of objects, it also delegates the construction of objects to the object builder class rather than creating the object manually via setters. For example:

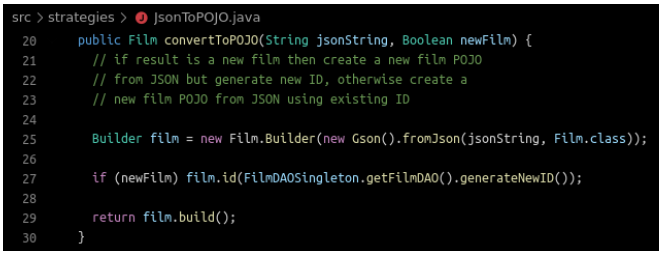
```
Film film = new Film();
film.setID(id);
film.setTitle(title);
film.setYear(year);
…
```

vs

```
Film film = new
Film.Builder().id(id).title(title).year(year) …
.build();
```

By removing the need for one line of code per object attribute being set, and having a simplistic, chainable instantiation method, the builder pattern promotes team collaboration by creating cleaner, simpler and more legible code. Although my implementation does not do so, the Builder pattern can also be modified to throw errors if certain object attributes have not been set and validation can be added inside the builder's setter-equivalent methods.

One issue I found with the Builder pattern is that the objects created using the Builder pattern are immutable (unable to have their attributes modified) as the Builder functionality replaces setters. A solution to the immutability of created objects was to pass an existing object into the Builder's constructor, allowing the instantiated Builder's properties to be set to the passed film's properties, which can then be overwritten by chaining the methods that set properties:



Implementation of Builder pattern:

src > models > Film.java

```
2   public class Film {
3     private final int id;
4     private final String title;
5     private final int year;
6     private final String director;
7     private final String stars;
8   > private final String review;  /** ...
13    private Film(Builder builder) {
14      this.id = builder.id;
15      this.title = builder.title;
16      this.year = builder.year;
17      this.director = builder.director;
18      this.stars = builder.stars;
19      this.review = builder.review;
20  >   } /** ...
25    public int getId() {
26      return id;
27  >   } /** ...
32    public String getTitle() {
33      return title;
34  >   } /** ...
39    public int getYear() {
40      return year;
41  >   } /** ...
46    public String getDirector() {
47      return director;
48  >   } /** ...
53    public String getStars() {
54      return stars;
55  >   } /** ...
```

src > models > Film.java

```
60    public String getReview() {
61      return review;
62  >   } /** ...
65    public static class Builder {
66      private int id;
67      private String title;
68      private int year;
69      private String director;
70      private String stars;
71  >   private String review;    /** ...
79      public Builder(Film film) {
80        if (film != null) {
81          this.id = film.getId();
82          this.title = film.getTitle();
83          this.year = film.getYear();
84          this.director = film.getDirector();
85          this.stars = film.getStars();
86          this.review = film.getReview();
87        }
88  >   }   /** ...
94      public Builder id(int id) {
95        this.id = id;
96        return this;
97  >   }   /** ...
103     public Builder title(String title) {
104       this.title = title;
105       return this;
106 >   }   /** ...
112     public Builder year(int year) {
113       this.year = year;
114       return this;
```

src > models > Film.java

```
121     public Builder director(String director) {
122       this.director = director;
123       return this;
124 >   }   /** ...
130     public Builder stars(String stars) {
131       this.stars = stars;
132       return this;       You, a month ago • Use
133 >   }   /** ...
137     public Builder review(String review) {
138       this.review = review;
139       return this;
140 >   }   /** ...
145     public Film build() {
146       return new Film(this);
147     }
148 > } /** ...
```

Film DAO

The Data Access Object is an architectural pattern that separates the main business logic from the logic that deals with the database's implementation. This follows the Single-responsibility principle (each module or class should be responsible for one piece of functionality) of the SOLID guideline for writing flexible and maintainable code.

A DAO also acts as a form of refactoring as code can be restructured so that calls to the database are no longer required inside of the class where the code that calls the database would be. Through separating the implementation of the DAO from the business logic, any changes to the database's mechanisms can be addressed through modifying the DAO rather than other parts of the application; using a DAO rather than implementing the database interaction code avoids the need to repeat code for each implementation of the business logic that requires it. As a group of developers work and alter code across an application, the DAO is beneficial as it can allow for alterations without necessarily breaking other sections of code.

An alternative to a DAO would be to implement the code responsible for communicating with the database in the form of an interface, by doing so, this would effectively stop the separation of business logic from the persistence (database) layer as the classes containing the business logic would have direct access to the database.

Example DAO function:

src > dao > FilmDAOSingleton.java

```
56  public ArrayList<Film> getFilmsByTitle(String title) {
57    // select all films and attributes where film's title contains
58    // value of title parameter
59    String SQL = "SELECT * FROM films WHERE title LIKE ?";
60
61    try {
62      // percentage symbols acts as wild card allowing
63      // for any string before and after the value of title
64      ArrayList<Object> paramVals = new ArrayList<>(Arrays.asList("%" + title + "%"));
65
66      // execute sql with title as SQL parameter,
67      // convert results to usable list, then return list
68      return ISQLOperations.sqlSelect(SQL, paramVals);
69    } catch (Exception e) {
70      e.printStackTrace();
71    }
72
73    return null;
74  }
```

Call to DAO:

src > coreservlets > GetAllFilms.java

```
28    @Override
29    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
30      // set relevant headers
31      response = IRequestHelpers.setHeaders(response, "GET");
32      // get all films from data access object
33      ArrayList<Film> films = FilmDAOSingleton.getFilmDAO().getAllFilms();
```

The DAO is also implemented with the creational Singleton pattern. The Singleton pattern restricts the instantiation class so that only one object of the class can exist during runtime. By only allowing one instance of the object, reduces memory usage and unnecessary instantiations.

The Singleton pattern can be considered to have the drawback of providing global access to the object wrapped in the Singleton, as global access is considered a code-smell in lieu of passing object/variables as parameters.

Creation of DAO Singleton:

src > dao > FilmDAOSingleton.java

```
12  public class FilmDAOSingleton {
13    private static FilmDAOSingleton filmDAO;
14
15    /**
16     * Prevents other classes from instantiating a new film DAO singleton.
17     */
18    private FilmDAOSingleton() {
19    }
20
21  > /**...
26    public static synchronized FilmDAOSingleton getFilmDAO() {
27      if (filmDAO == null) filmDAO = new FilmDAOSingleton();
28      return filmDAO;
29    }       You, a month ago • Convert to singleton
```

SQL Operations Interface

Following the single-responsibility principle, I decided to refactor the code responsible for executing SQL statements from the DAO to an interface called “ISQLOperations”; this follows the conventions of interfaces being prefixed with “I”. Ignoring the “generateID” method, the SQL operations interface is responsible for all SQL related operations as alluded to in the name of the interface.

By refactoring these methods into their own interface, the single-responsibility principle is adhered to as the DAO is now only responsible for returning the results of each method such as ‘create film’ or ‘get all films’, and defining the statement to be used. Once this statement is passed to the appropriate method in the interface the SQL statement is executed, with the results returned to the DAO, which is then returned to the servlet for formatting.

SQL statements can be split into two separate types, each with its own Java method for executing them. “executeUpdate()” is responsible for DML statements such as “DELETE, INSERT and UPDATE”, which return an integer-based upon the number of rows affected by the statement. “executeQuery()” is responsible for “SELECT” queries, which return a set of results to be parsed.

Select query:

src > interfaces > ISQLOperations.java

```
26    static ArrayList<Film> sqlSelect(String SQL, ArrayList<Object> paramVals) {
27      // creating connection closes connection regardless of whether
28      // it succeeds or fails, freeing up connection to be reused
29      try (Connection conn = ConnectionPoolSingleton.getConnectionPool().getPool().getConnection()) {
30        // convert list of MySQL query parameter's into query
31        PreparedStatement statement = prepareStatement(conn.prepareStatement(SQL), paramVals);
32
33        // execute SQL query and convert to array list of films
34        ArrayList<Film> results = resultsToList(statement.executeQuery());
35        statement.close();
36        return results;
37      } catch (Exception e) {
38        e.printStackTrace();
39      }
40
41      return null;
42    }       You, a month ago • Attempt to load data
```

DML query:

src > interfaces > ISQLOperations.java

```
52    static int sqlManipulate(String SQL, ArrayList<Object> paramVals) {
53      int numResults = -1;
54
55      // creating connection closes connection regardless of whether
56      // it succeeds or fails, freeing up connection to be reused
57      try (Connection conn = ConnectionPoolSingleton.getConnectionPool().getPool().getConnection()) {
58        // convert list of MySQL query parameter's into query
59        PreparedStatement statement = prepareStatement(conn.prepareStatement(SQL), paramVals);
60
61        // execute query that manipulates data
62        numResults = statement.executeUpdate();
63        statement.close();
64      } catch (Exception e) {
65        e.printStackTrace();
66      }
67
68      return numResults;
69    }
```

As each SQL select query will return a Film object, it is logical to convert the functionality of turning an SQL result set into a usable ArrayList of Film POJOs. This functionality also makes use of the film’s Builder pattern:

src > interfaces > ISQLOperations.java

```
134   static ArrayList<Film> resultsToList(ResultSet results) {
135     ArrayList<Film> films = new ArrayList<>();
136
137     try {
138       // convert all results to usable Film POJOs
139       while (results.next()) {
140         films.add(resultToFilm(results));
141       }
142
143       // return array list of films
144       return films;
145     } catch (SQLException e) {
146       e.printStackTrace();
147     }
148
149     return null;
150   }
151
152 >  /** ...
158   static Film resultToFilm(ResultSet result) {
159     // create new film POJO from result parameter
160     try {
161       // return film
162       return new Film.Builder(null).id((int) result.getObject(1)).title((String) result.getObject(2))
163           .year((int) result.getObject(3)).director((String) result.getObject(4)).stars((String) result.getObject(5))
164           .review((String) result.getObject(6)).build();
165     } catch (Exception e) {
166       e.printStackTrace();
167     }
168
169     return null;
170   }
```

Prepared Statements

For this project, I have decided to use prepared statements (parameterised queries), a process that replaces unspecified data with provided variables. The benefit of using prepared statements over non-prepared statements is that they reduce the risk of SQL injection attacks but are not 100% invulnerable. Another reason that prepared statements are beneficial in this project is that the attributes in the film object “stars” and “review” may contain commas which would break an SQL statement as commas are used as separators. When inserting a film with two or more actors, a non-prepared statement would insert the second actor as the value of the film in the below example. One method of subverting this behaviour would be to escape the values but this would lead to wordier code and is not considered a good practice compared to the usage of prepared statements.

My implementation receives the SQL statement to be used, and an ArrayList using the “Object” type of the parameters to be used in the SQL statement. The reason for using the Object type is that all types in Java are subclasses of the Object type. Therefore, an ArrayList of the Object type can contain variables of any type inside of it, this is beneficial as parameters in the ArrayList can be either strings or ints.

Example flow of logic related to the creation of a prepared statement is as follows:

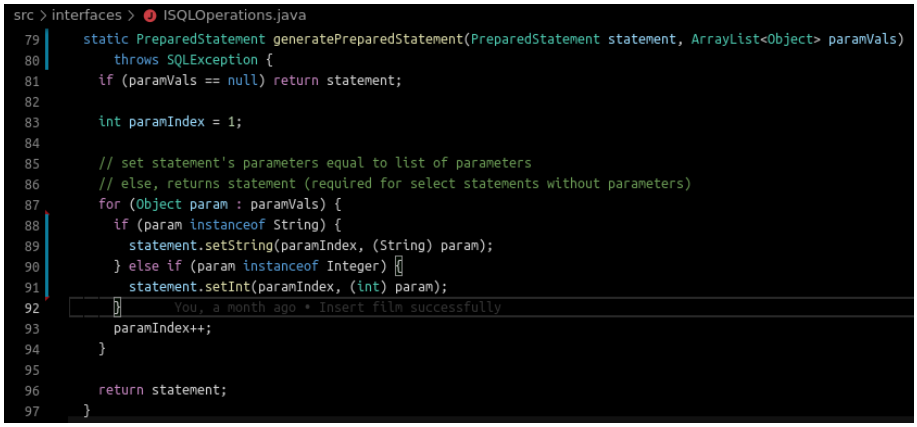
1. Originating in the Film’s DAO Singleton, the SQL statement is defined, i.e. inside the createFilm method
2. The Film object that was passed as a parameter, has the values of its attributes converted into an ArrayList
3. The SQL statement and ArrayList is passed to the method responsible for DML statements

src > dao > FilmDAOSingleton.java

```
110     public int createFilm(Film film) {
111       String SQL = "INSERT INTO films VALUES (?, ?, ?, ?, ?, ?)";
112
113       try {
114         // if film id = -1, i.e. the value the DAO returns
115         // if an error has occurred, throw an exception
116         if (film.getId() == -1) throw new Exception("Invalid film ID");
117
118         // convert film attributes to list of parameters for insert statement
119         ArrayList<Object> paramVals = film.attributesToParamList();
120         // execute SQL
121         return ISQLOperations.sqlManipulate(SQL, paramVals);
122       } catch (Exception e) {
123         e.printStackTrace();
124       }
125
126       return -1;
127     }
```

4. The sqlManipulate method calls the generatePreparedStatement method, the SQL statement is converted from a string into a PreparedStatement object and passed as a parameter, along with the ArrayList of parameters

5. Starting with an index of 1 (as required by the PreparedStatement object), each parameter in the ArrayList's type is checked as either a string or int, sets the parameter at the index's values position to the current parameter in the ArrayList
6. The index is incremented
7. Steps 5 & 6 are repeated until each parameter has been looped through



Connection Pool

Creating connections to a database can be a time-expensive process that can become expensive in a monetary sense when using Cloud platforms. As such, I implemented the creational Object Pool pattern. Typically, an application makes a request to the database, creates a connection, performs an SQL query, then closes the connection. When database pooling is used, the application checks the pool for unused connections, uses an existing connection if there is one, if not creates a new one, uses the connection, then once a specified amount of time has passed, the connection is closed to save resources. By reducing the likelihood of needing to create a new connection, the time and money spent executing SQL queries can be reduced on average. My implementation uses the previously mentioned Singleton pattern as only one instance of the connection pool should ever exist.
Implementation of Connection Pooling:

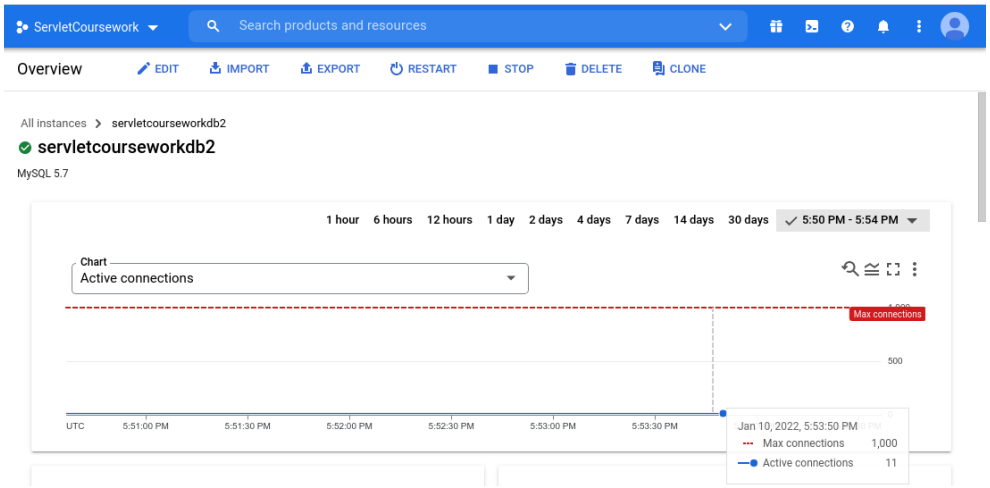
src > connectionPool > ConnectionPoolSingleton.java

```
11  public class ConnectionPoolSingleton {
12    private static ConnectionPoolSingleton connectionPool;
13    private static DataSource pool;
14
15  /** ...
18    private ConnectionPoolSingleton() {
19    }
20
21  /** ...
26    public static synchronized ConnectionPoolSingleton getConnectionPool() {
27      if (connectionPool == null) {
28        connectionPool = new ConnectionPoolSingleton();
29        pool = createConnectionPool();
30      }
31
32      return connectionPool;
33    }
34
35  /** ...
40    private static DataSource createConnectionPool() {
41      HikariConfig config = new HikariConfig();
42
43      String connectionName = "servletcoursework-336513:europe-west2:servletcourseworkdb2";
44      String databaseName = "servletcoursework";
45      String password = "wu31wMas9nclNh05";
46
47      // connection settings:
48      config.setJdbcUrl(String.format("jdbc:mysql:///%s", databaseName));
49      config.setUsername("root");
50      config.setPassword(password);
51      config.addDataSourceProperty("ipTypes", "PUBLIC");
```

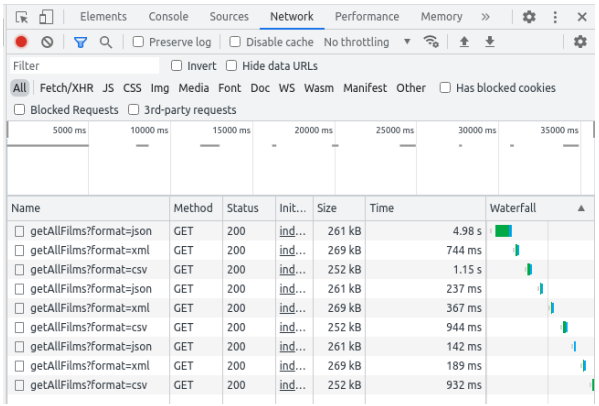
src > connectionPool > ConnectionPoolSingleton.java

```
54        config.setMaximumPoolSize(99);
55        config.setMinimumIdle(5);
56
57        // establish connection timeout: 10 seconds
58        config.setConnectionTimeout(10 * 1000);
59        // timeout: 5 minutes
60        config.setIdleTimeout(5 * 60 * 1000);
61        // max connection length: 30 minutes
62        config.setMaxLifetime(30 * 60 * 1000);
63
64        return new HikariDataSource(config);
65    }
66
67  /** ...
72    public DataSource getPool() {
73      return pool;
74    }
75
76  /** ...
81    private static void setPool(DataSource pool) {
82      ConnectionPoolSingleton.pool = pool;
83    }
84
85  }
```

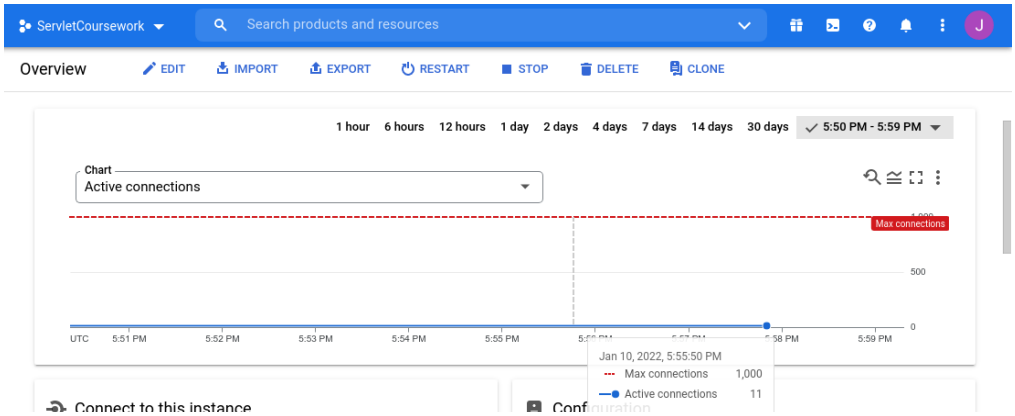
Without connection pooling, the expected number of connections should increase by one per request, but with pooling, the number of connections should remain constant. An example of the database pooling in action is that there are currently 11 active connections in GCP SQL:



Multiple requests dependent on SQL queries being executed:



Number of active connections after executing queries:



After an initial request, creating a new connection, subsequent requests on average are sub-one second:

Name	Method	Status	Init...	Size	Time
getAllFilms?format=json	GET	200	ind...	261 kB	4.98 s
getAllFilms?format=xml	GET	200	ind...	269 kB	744 ms
getAllFilms?format=csv	GET	200	ind...	252 kB	1.15 s
getAllFilms?format=json	GET	200	ind...	261 kB	237 ms
getAllFilms?format=xml	GET	200	ind...	269 kB	367 ms
getAllFilms?format=csv	GET	200	ind...	252 kB	944 ms
getAllFilms?format=json	GET	200	ind...	261 kB	142 ms
getAllFilms?format=xml	GET	200	ind...	269 kB	189 ms
getAllFilms?format=csv	GET	200	ind...	252 kB	932 ms

getAllFilms?format=json	GET	200	ind...	261 kB	4.33 s
getAllFilms?format=xml	GET	200	ind...	269 kB	609 ms
getAllFilms?format=csv	GET	200	ind...	252 kB	891 ms
getAllFilms?format=json	GET	200	ind...	261 kB	259 ms

Converting to data exchange formats

Strategy pattern

My method of converting the data to different data exchange formats such as JSON, XML and CSV is implemented via the use of the Strategy behavioural pattern. The Strategy pattern is used to select which pattern is used at runtime which is beneficial when determining what and how data from the client/database should be formatted in order to be parsed or sent. The strategy pattern is implemented by writing an interface that strategies will implement, this interface should contain the default method to be overridden by the strategies:

src > interfaces > IPojoFormatStrategy.java

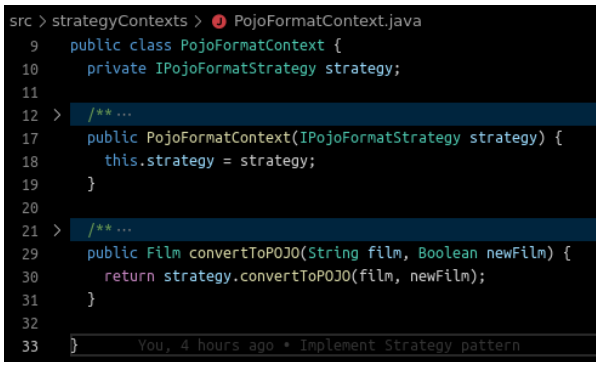
```
5   public interface IPojoFormatStrategy {
6   >   /** ...
13      public Film convertToPOJO(String film, Boolean newFilm);
14  }       You, 4 hours ago • Implement Strategy pattern
```

Strategies are then implemented in the form of classes that implement the interface and override the default method:

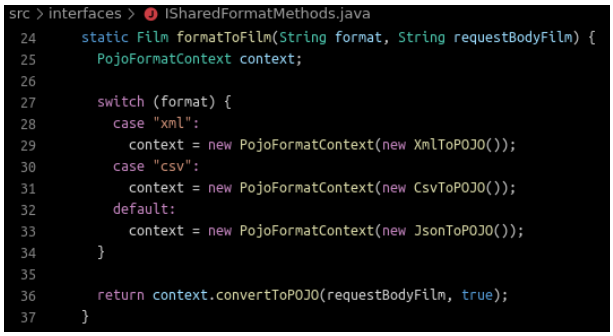
src > strategies > JsonToPOJO.java

```
10  public class JsonToPOJO implements IPojoFormatStrategy {
11  >   /** ...
19      @Override
20      public Film convertToPOJO(String jsonString, Boolean newFilm) {
21          // if result is a new film then create a new film POJO
22          // from JSON but generate new ID, otherwise create a
23          // new film POJO from JSON using existing ID
24
25          Builder film = new Film.Builder(new Gson().fromJson(jsonString, Film.class));
26
27          if (newFilm) film.id(FilmDAOSingleton.getFilmDAO().generateNewID());
28
29          return film.build();
30      }
31
32  }
```

A Context class is also used such that the strategy containing the overwritten method is passed to the Context's constructor so that the overwritten method can be used as a method from the Context.



Strategy pattern in use:



Formatting libraries

Each strategy used to override the conversion methods inside the strategy's interface contains the appropriate method for converting data. In order to convert data to/from a format, I have used standard libraries where possible. The methods used for converting between formats depending on what actions are being performed:

- Getting all films, a film by title from the database does not receive a film object from the client but must return films in a specified format so it must convert multiple film POJOs to a format
    - Getting by ID acts the same but only converts one film POJO to format
- Getting a film by title from the database can return multiple film objects so functions the same as getting all films
- Creating and updating a film receives a film object from the client in a specific format so it must convert from the format to a POJO

An example of converting JSON to singular POJOs is that I have used Google's Gson serialisation library:

The Gson library can be used to map the JSON which is stored as a string in the

variable “jsonString”, to Film class. As the Builder which was described previously can receive a Film object as a parameter to its constructor, the Film object created from the Gson library is then passed to the Film Builder. Alternatively, the Jackson library can be used for converting to and from JSON but Gson appears to be quicker based on benchmarks found online.

Converting JSON to POJO:

Gson:

```
Film film = new Gson().fromJson(jsonInput, Film.class);
```

Jackson:

```
Film film = new ObjectMapper().readValue(jsonInput, Film.class);
```

Converting an ArrayList of POJOs to JSON is also similarly simple using Gson:

src > strategies > FilmsToJSONArray.java

```
18      public String convertArrayToFormat(ArrayList<Film> data) {
19        return new Gson().toJson(data);
20      }
```

To convert to and from XML I have used the XStream library. An alternative to XStream would be to use Java’s own JAXB marshal and unmarshalling framework to convert to and from XML and POJOs. The main issue I found with JAXB was that it required a lot more boilerplate code compared to XStream, by removing excess code, it allows for cleaner and more readable code.

Converting XML to POJO:

XStream:

```
XStream xstream = new XStream();
return xstream.fromXML(film);
```

JAXB equivalent:

```
JAXBContext context = JAXBContext.newInstance(Film.class);
Unmarshaller m = context.createUnmarshaller();
StreamSource source = new StreamSource(new StringReader(film));
return m.unmarshal(source, Film.class).getValue();
```

My implementation of converting XML to POJO:

src > strategies > XmlToPOJO.java

```
20      public Film convertToPOJO(String xmlString, Boolean newFilm) {
21        XStream xstream = new XStream();
22        xstream.allowTypes(new Class[] { Film.class });
23        xstream.processAnnotations(Film.class);
24
25        Builder film = new Film.Builder((Film) xstream.fromXML(xmlString));
26
27        if (newFilm) film.id(FilmDAOSingleton.getFilmDAO().generateNewID());
28
29        return film.build();
30      }
```

ArrayList of POJOs to XML:

src > strategies > FilmsToXMLArray.java

```
19    public String convertArrayToFormat(ArrayList<Film> data) {
20      XStream xstream = new XStream();
21      // change xml tag name from class name to value of first function parameter
22      xstream.alias("root", List.class);
23      xstream.alias("film", Film.class);
24
25      // convert films to XML
26      String xmlString = xstream.toXML(data);
27
28      // return XML and XML declaration
29      return "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>" + "\n" + xmlString;
30    }
```

My implementation for CSVs does not use a library as converting to text is very trivial. I have overridden the “toString()” method inside the Film’s model:

src > models > Film.java

```
193    @Override
194    public String toString() {
195      // returns Film's attributes separated by double comma
196      // to avoid issues with attributes containing commas
197      return this.getId() + ",," + this.getTitle() + ",," + this.getYear() + ",," + this.getDirector() + ",,"
198          + this.getStars() + ",," + this.getReview();
199    }
```

By using a very simple method to convert a film object’s attributes to a string, makes converting a film POJO to a CSV very simple even without a library. As each of the film object’s attributes will be divided by two commas, as string can be created by using two commas as a delimiter:

src > strategies > CsvToPOJO.java

```
16    @Override
17    public Film convertToPOJO(String csvFilm, Boolean newFilm) {
18      String[] filmAttributes = csvFilm.split(",,");
19
20      // remove quotations from all attributes
21      for (String attribute : filmAttributes) {
22        attribute = attribute.replace("\"", "");
23      }
24
25      // create film object from csv values
26      Builder film = new Film.Builder(null);
27      if (newFilm) {
28        film.id(FilmDAOSingleton.getFilmDAO().generateNewID()).title(filmAttributes[0])
29            .year(Integer.valueOf(filmAttributes[1])).director(filmAttributes[2]).stars(filmAttributes[3])
30            .review(filmAttributes[4]);
31      } else {
32        film.id(Integer.valueOf(filmAttributes[0])).title(filmAttributes[1]).year(Integer.valueOf(filmAttributes[2]))
33            .director(filmAttributes[3]).stars(filmAttributes[4]).review(filmAttributes[5]);
34      }
35      return film.build();
36    }
```

Converting a POJO to an ArrayList of CSVs is made very simple by the “toString()” method shown above, after each toString method, a line terminator is added, as each line will act as a single film:

src > strategies > FilmsToCSVArray.java

```
16    public String convertArrayToFormat(ArrayList<Film> data) {
17      String films = "";
18
19      // use film model's toString function to convert
20      // film's attributes to CSV string
21      for (Film film : data) {
22        films += film.toString() + "\n";
23      }
24
25      // remove additional line terminator, and return CSV film array
26      return films.substring(0, films.length() - 2);
27    }
```

Example servlets

I have attempted to refactor each servlet as much as possible, ensuring only the highest-level methods required to complete the servlet's behaviour are included. What I found whilst producing the servlets was that they all share a lot of functionality such as: setting headers based upon the HTTP method, retrieving the format to be used from the servlet's request's URL parameters, converting either a request body or database response to or from a format, interacting with the DAO, and sending the response.

As a result of lots of reusable functionality, each servlet (without comments or whitespace) ranges between five and six lines of code.

Example servlets:

Get Films by Title servlet:

src > coreservlets > GetFilmByTitle.java

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

    // set relevant headers
    response = IRequestHelpers.setHeaders(response, "GET");

    // get format from url
    String format = IRequestHelpers.getFormat(request);

    // get film title from url
    String title = request.getParameter("title");

    // get films with title containing title from url
    ArrayList<Film> films = FilmDAOSingleton.getFilmDAO().getFilmsByTitle(title);

    Object formattedFilms = ISharedFormatMethods.filmPOJOsToFormat(films, format);

    // send response containing formatted list of films
    IRequestHelpers.sendResponse(response, formattedFilms);
}
```

Create Film servlet:

src > coreservlets > CreateFilm.java

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    // set relevant headers
    response = IRequestHelpers.setHeaders(response, "POST");

    // get format from url
    String format = IRequestHelpers.getFormat(request);

    // get film from HTTP body
    String requestBodyFilm = IRequestHelpers.getRequestBody(request);

    // set film equal to film object converted based on relevant format
    Film film = ISharedFormatMethods.formatToFilmPOJO(format, requestBodyFilm);

    // send response containing number of rows affected by creating new film
    IRequestHelpers.sendResponse(response, FilmDAOSingleton.getFilmDAO().createFilm(film));
}
```

RESTful servlet

The servlets I have produced during this project already conform to the RESTful requirements and as such implementing a single servlet to act as the REST implementation was possible by reusing code from my servlets. Ignoring GET requests, each CRUD-HTTP method is implemented by a single servlet. Java Servlets have a limitation of only being able to have one implementation of each HTTP method, i.e. doGet, doPut, doPost, etc. As there are two required operations for the project (get all films, get by title) with my implementation having an additional "Get Film by ID" operation, I had to create a switch to handle each get functionality inside the servlet's doGet method:

src > coreservlets > REST.java

```java
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
    // set relevant headers
    response = IRequestHelpers.setHeaders(response, "GET");

    // get format from url
    String format = IRequestHelpers.getFormat(request);

    FilmDAOSingleton filmDAO = FilmDAOSingleton.getFilmDAO();

    // get get type from url
    String getType = request.getParameter("getType");

    Object films;       You, seconds ago • Uncommitted changes

    // determine which type of get function to use based on get type from url
    switch (getType) {
      case "title":
        String title = request.getParameter("title");
        films = ISharedFormatMethods.filmPOJOsToFormat(filmDAO.getFilmsByTitle(title), format);
        break;
      case "id":
        int id = Integer.parseInt(request.getParameter("id"));
        films = getFilmByID(filmDAO, format, id, response);
      default:
        films = ISharedFormatMethods.filmPOJOsToFormat(filmDAO.getAllFilms(), format);
    }

    // send response containing film(s)
    IRequestHelpers.sendResponse(response, films);
}
```

All other methods have been copied from their respective servlets. In order to access the RESTful servlet, I created a toggle in my frontend which is used to determine which URL the frontend client should make a request to. When the toggle is active and the client makes a request, rather than use the endpoint for the appropriate method, instead the “/REST” endpoint is called and the HTTP method used causes the appropriate servlet method to be called, e.g. HTTP GET causes doGet to be called.

Frontend

Design decisions

Frontend library

For the frontend I decided to use React, the main reason being that I already have experience using React but I do not have experience with other frontend frameworks/libraries except Angular. Alternatives to React include jQuery, Angular and Vue, but React is by far the most popular JavaScript framework and it has out-of-the-box support for TypeScript. Additionally, React is the most popular means that it has a larger community and a number of community-developed packages which can aid with development. I have also used the functional approach instead of the class-based approach to React as functional components in React are encouraged and are a replacement for classes in React.

I decided to use TypeScript in order to aid the collaborative TypeScript is a programming language that allows for static typing and is transpiled directly to JavaScript. The main benefits of static typing afforded by TypeScript are:

- Type declarations act as a form of documentation as the type of data required is clearly defined, this also makes reading statically typed code easier to read and maintain by other developers
- Static typing helps reduce runtime errors as TypeScript will cause type errors due to incorrect/inconsistent typing. This, in turn, leads to fewer runtime errors due to functions being less likely to receive incorrectly typed parameters

Some disadvantages I have experienced with TypeScript is that the non-dynamic nature of typing means that there is an increased usage of single-use variables which increases memory usage, however, eliminating intermediary variables can cause it to be more difficult to debug code and can decrease the readability of code.

TypeScript:

```
const monoXMLFilmToJSON = (xmlToParse: string): IFilm => {
  const parsedXML = new DOMParser().parseFromString(xmlToParse, 'application/xml');
  const xmlDocument = parsedXML.getElementsByTagName('film')[0].children
  …
}
```

JavaScript:

```
const monoXMLFilmToJSON = (xmlString) => {
  let xml = new DOMParser().parseFromString(xmlString, 'application/xml');
  xml = xml.getElementsByTagName('film')[0].children;
  …
}
```

As seen above, as JavaScript can use dynamic typing the same variable can be used twice whilst retaining readability times saving memory, whereas TypeScript requires a new variable for each time as there are three separate variable types used. It is possible to override this behaviour in TypeScript by using a union type, signified by the pipe “|” symbol:

```
const monoXMLFilmToJSON = (xml: string | Document | HTMLCollection): IFilm => {
  xml = new DOMParser().parseFromString(xml as string, 'application/xml');
  xml = xml.getElementsByTagName('film')[0].children;
  …
}
```

The main disadvantage of using union types is that code can become more difficult to comprehend quickly and overuse type assertions (variable “as” type).

Formatting and Linting

To keep code as clean as possible, in both a stylistic and quality sense, I have used Prettier code formatter in addition to ESLint which is a code-analysis and bug fixing tool. ESLint and Prettier will format code automatically upon saving. ESLint has the option to configure rules, and I have done so such that all function return types and variables must have their type explicitly defined. Prettier is an opinionated formatter so there are almost no rules to configure. By using defined rules if other developers were to work on the project there would be a homogenous code style across the project.

Styling components

In standard HTML + CSS applications, it is typical to use an external style sheet or in-line styling when appropriate. React is able to use an external style sheet, inline styling in addition to CSS modules, JSS, Styled-Components, Stylable and more. My approach is to use CSS modules in conjunction with SCSS. To create smaller and tidier components, I have used CSS modules, as CSS modules store the CSS in separate files which it is then imported into the React component via the syntax:

```
import classes from ‘ComponentName.module.css’;
…
<ComponentName className={classes.CssClass} />
```

ComponentName.module.css:

```
.CssClass {
  float: left;
  padding: 0;
  width: 20vw;
}
```

CSS modules can also be used programmatically to swap modules dynamically or to set multiple classes at one time. For example, the className of the Input component uses both its own CSS module as well as a CSS class from its props:

client > src > components > Input > TS Input.tsx > ...

```
13  const Input: React.FC<IProps> = ({ className, label, name, onChange, placeholder, value }) => {
14    return (
15      <MDBInput
16        className={`${classes.Input} ${className}`}
17        label={label}
18        name={name}
19        onChange={onChange}
20        placeholder={placeholder}
21        type="text"
22        value={value}
23      />
24    );
25  };
26
27  export default Input;
28
```

Scss offers benefits compared to CSS such as logically nesting CSS classes, importing other scss files such as variables or other CSS classes.

React App

React creates a Single-Page Application that consists of one HTML file with a JavaScript file imported as a script that dynamically generates pages based upon internal routing rather than via server routing like a PHP application would use. A typical React application consists of multiple pages and would have an index file with a render method as the application's entry point, which would render a separate "App" component. Typically the App component would contain the routing logic like so:

```
import { BrowserRouter, Redirect, Route, Switch } from 'react-router-dom';
…
return (
  <BrowserRouter>
    <Switch>
      <Route exact path="/">
        <Home />
      </Route>
      <Route exact path="/page1">
        <Page1 />
      </Route>
    </Switch>
  </BrowserRouter>
);
```

However, as this project has only one page, I have not used an App component or included routing. Instead, my index file renders the Home page component directly. The home page component is wrapped in the "StrictMode" component from React which is used to detect issues with the application such as deprecated functionality and unexpected side effects during runtime:

Home page

Inside the Home component, React's "useState" hooks are used as an equivalent to local variables. When returning JSX as shown below, values known as props can be set equal to the value inside the braces, e.g. inside SidePanel, "format={format}", sets the SidePanel prop to the value of the format state. In order to share state between non-parent-child components, I have taken advantage of the concept of "lifting state up" and storing state in a parent component: the SidePanel component has multiple props that are defined as functions with a parameter, and when executed the value in the parameter is set to an appropriate state. E.g. the "setFIlms" prop in SidePanel is set to a function that receives a variable called "films" as a parameter. In the SidePanel, the "setFilms" prop can be called, passing the value for the "films" parameter from the prop as a parameter. Finally, the "setFilms" hook is called, setting the films state:

```
client > src > pages > Home > SidePanel > TS SidePanel.tsx > [e] SidePanel > useEffect() callback
104         setFilms(await getAllFilms(endpoint, format, useREST));
```

```
client > src > pages > Home > TS Home.tsx > ...
 8  const Home: React.FC = () => {
 9    const [films, setFilms] = useState(null as unknown as IFilm[] | string | null);
10    const [format, setFormat] = useState('json');
11    const [formatChanged, setFormatChanged] = useState(false);
12    const [selectedFilmID, setSelectedFilmID] = useState(null as number | null);
13
14    return (
15      <MDBContainer className={classes.PageWrapper}>
16        <MDBRow className={classes.PageContent}>
17          <SidePanel
18            format={format}
19            selectedFilmID={selectedFilmID!}
20            setFilms={(films: IFilm[] | string | null): void => setFilms(films)}
21            setFormat={(format: string): void => setFormat(format)}
22            setFormatChanged={(formatChanged: boolean): void => setFormatChanged(formatChanged)}
23            setSelectedFilmID={(selectedFilmID: number | null): void => setSelectedFilmID(selectedFilmID)}
24          />
25          <Output
26            films={films as IFilm[] | string | null}
27            format={format}
28            formatChanged={formatChanged}
29            setSelectedFilmID={(selectedFilmID: number | null): void => setSelectedFilmID(selectedFilmID)}
30          />
31        </MDBRow>
32      </MDBContainer>
33    );
34  };
```

Now that the "films" state is set, it can be passed to the Output component as a prop to be rendered.

Example - Sending data

The side panel is comprised of four main parts, a radio group responsible for selecting which data exchange format to send data in, another radio group responsible for which operation to use, a switch responsible for what JSX should be rendered depending on the selected operation from the operation radio group, and the logic for sending requests to the backend. The FormatRadioGroup component receives the “onClick” prop that uses the “lifting state up” concept in order to set the format state inside the SidePanel equal to the event.target.id from the event inside the FormatRadioGroup component:

client > src > pages > Home > SidePanel > TS SidePanel.tsx > SidePanel

```
248                <FormatRadioGroup
249                  onClick={(): void => {
250                    setFormatChanged(true);
251                    // @ts-expect-error id does exist on event.target
252                    setFormat(event!.target!.id);
253                  }}
254                />
```

The event.target.id value is equal to the ID of the Radio component that was clicked:

client > src > components > RadioGroups > FormatRadioGroup > TS FormatRadioGroup.tsx > ...

```
9   const FormatRadioGroup: React.FC<IProps> = ({ onClick }) => {
10    return (
11      <MDBBtnGroup className={classes.FormatRadioGroup}>
12        <Radio defaultChecked label="JSON" id={'json'} name="formatGroup" onClick={onClick} />
13        <Radio label="XML" id={'xml'} name="formatGroup" onClick={onClick} />
14        <Radio label="Text" id={'csv'} name="formatGroup" onClick={onClick} />
15      </MDBBtnGroup>
16    );
17  };
```

The operation to use works through the same concept, with each operation’s Radio component setting which endpoint to use:

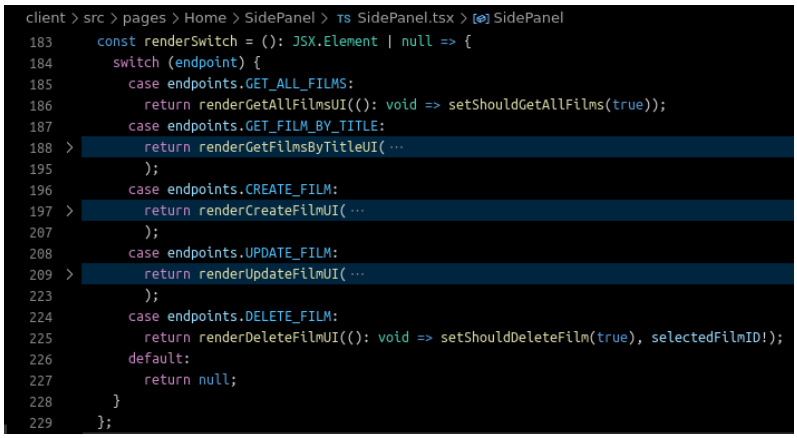
client > src > components > RadioGroups > OperationRadioGroup > TS

```
11      <MDBBtnGroup className={classes.OperationRadioGroup}>
12        <Radio
13          className={classes.TopOperationRadio}
14          id="GET_ALL_FILMS"
15          label="Get all films"
16          name="operationGroup"
17          onClick={onClick}
18        />
19        <Radio
20          className={classes.TopOperationRadio}
21          id="GET_FILM_BY_TITLE"
22          label="Get film by title"
23          name="operationGroup"
24          onClick={onClick}
25        />
26        <Radio
27          className={classes.TopOperationRadio}
28          id="CREATE_FILM"
```

client > src > pages > Home > SidePanel > TS SidePanel.tsx > SidePanel

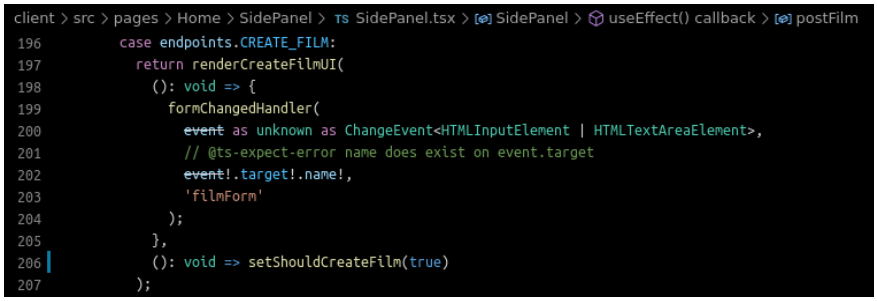
```
252                <OperationRadioGroup
253                  onClick={(): void => {
254                    // @ts-expect-error event does exist on event.target
255                    setEndpoint(endpoints[event!.target!.id]);
256                  }}
257                />
```

Once the endpoint is set, the value of the endpoint is used in the switch:



To refactor the code where possible, I have placed both the operation's logic and the method in the switch to render inside a single file:

Each operation's render method contains a Button component that has an onClick function parameter passed to it. When the Button is clicked the onClick function is executed in the SidePanel component, which sets a boolean state for the operation to true:



When a variable in the array that is the second parameter of a useEffect hook's value changes, this will cause the code inside the first parameter to execute, so by putting the boolean "shouldCreateFilm" into the array of the second parameter, this will cause the logic for creating a film to execute:

client > src > pages > Home > SidePanel > TS SidePanel.tsx > SidePanel > useEffect() callback > putFilm

```
139     // create new film
140     useEffect(() => {
141       const postFilm = async (): Promise<void> => {
142         setShowSpinner(true);
143
144         await createFilm(endpoint, format, formData!, useREST);
145
146         setShouldCreateFilm(false);
147         setShowSpinner(false);
148       };
149
150       if (shouldCreateFilm) postFilm();
151     }, [shouldCreateFilm]);
```

As JavaScript is an asynchronous language, it is important to use the "await" keyword to make JavaScript stop executing code until the awaited function has finished executing. useEffect hooks cannot run asynchronously but can execute asynchronous functions, so wrapping the logic in an asynchronous function allows the await keyword to be used.

Inside the CRUD operation files, XML and CSV format requests use the same request function as XML as the backend functionality responsible for handling the format receives strings as inputs. Once the function responsible for creating the request is called, e.g. "createFilm", a function is called in order to create a URL:

client > src > crudFunctionality > TS createFilm.tsx > createFilm

```
9   const createFilm = async (endpoint: string, format: string, formData: IFilm, useREST: boolean): Promise<void> => {
10    const url = generateURL(endpoint, format, useREST);
11
12    try {
13      switch (format) {
14        case 'xml':
15          const xmlFilm = `<Film>${jsonToXML(formData)}</Film>`;
16          await textRequest(url, 'POST', xmlFilm);
17          break;
18        case 'csv':
19          const csvFilm = new jsonToCSV({ header: false, delimiter: ',,' }).parse(formData!);
20          await textRequest(url, 'POST', csvFilm);
21          break;
22        default:
23          await jsonRequest(url, 'POST', formData);       You, a week ago • Refactor
24      }
25    } catch (e) {
26      console.error(e);
27    }
28  };
```

client > src > utils > TS generateURL.ts > generateURL

```
3   const generateURL = (endpoint: string, format: string, useREST: boolean): string => {
4     if (useREST) {
5       return `${ROOT}/${REST}?format=${format}`;
6     } else {
7       return `${ROOT}/${endpoint}?format=${format}`;
8     }
9   };
```

Depending on the value of the format state, the updateFilm function will use a switch to determine what type of request needs to be made.
Request functions:

client > src > utils > TS requests.ts > ...

```
3   export const jsonRequest = async (url: string, method: string, body?: IFilm | IFilm[]): Promise<IFilm[] | IFilm> => {
4     const options = body ? { method, body: JSON.stringify(body) } : { method };
5
6     const response = await fetch(url, options);
7     return await response.json();
8   };
9
10  export const textRequest = async (url: string, method: string, body?: string): Promise<string> => {
11    const response = await fetch(url, {
12      method,
13      body
14    });
15
16    return await response.text();
17  };
```

In order to create requests, the request method receives the created URL, the HTTP method that should be used, and a parameter containing the body for the request which is optional. Alternatives to JavaScript's Fetch API would be to use a third-party HTTP client such as Axios or for XML to use JavaScript's "XMLHttpRequest". I decided against using Axios as it seemed unnecessary to add extra npm packages when the existing Fetch API was more than sufficient for my application. XMLHttpRequest was also not used due to being "wordier" and less simple to use compared to the Fetch API.

Fetch API:

```
const request = await fetch(url, {
  method: "GET"
});
const response = await request.text();
```

XMLHttpRequest:

```
const xhttp = new XMLHttpRequest();
let response;
xhttp.onreadystatechange = () => {
    if (this.readyState == 4 && this.status == 200) {
       response = xhttp.responseText;
    }
};
xhttp.open("GET", url, false);
xhttp.send();
```

Example - Retrieving data

When retrieving data from the backend/database, the code works exactly the same, but inside the useEffect hook, it will contain a method for setting the films state in the Home page component:

Once the films state value is set, it is passed as a prop to the Output component:

```
client > src > pages > Home > TS Home.tsx > Home
25          <Output
26            films={films as IFilm[] | string | null}
27            format={format}
28            formatChanged={formatChanged}
29            setSelectedFilmID={(selectedFilmID: number | null): void => setSelectedFilmID(selectedFilmID)}
30          />
```

The JSX for the Output component is quite small as it is very logic dependent, if the films prop is not null it will render the return value of the “handleFormat” function:

```
client > src > pages > Home > Output > TS Output.tsx > ...
75      return (
76        <MDBCol className={classes.RightContent}>
77          <h1 className={classes.Header}>Films:</h1>
78          <ul className={classes.List}>{films ? handleFormat() : null}</ul>
79        </MDBCol>
80      );
81    };
```

The handleFormat function is responsible for converting the films in the films prop to JSON, as JSON is required by JavaScript in order to parse objects.

```
client > src > pages > Home > Output > TS Output.tsx > Output
15      const handleFormat = (): JSX.Element | null => {
16        if (formatChanged) return null;
17
18        switch (format) {
19          case 'xml':
20            const xmlFilms = new xmlToJSON().parse(films as string).root.film;
21            return printFilms(xmlFilms);
22          case 'csv':
23            return printFilms(csvToJSON(films as string));
24          default:
25            return printFilms(films as IFilm[]);
26        }
27      };
```

Once the film prop is converted to JSON, the printFilms function is called with the JSON films as a parameter. The map function is used on the array of films called “preparedFilms” to render return each film object in a table:

client > src > pages > Home > Output > TS Output.tsx > Output

```tsx
const printFilms = (preparedFilms: IFilm[]): JSX.Element => {
  return (
    <MDBTable className={classes.FilmTable} striped>
      <MDBTableHead> ···
      </MDBTableHead>
      <MDBTableBody>
        {preparedFilms
          ? preparedFilms.map((film: IFilm) => {
              return (
                <tr key={film.id} onClick={(): void => setSelectedFilmID(film.id!)}>
                  <td className={classes.IDCell}>{film.id}</td>
                  <td className={classes.TitleCell}>{film.title}</td>
                  <td className={classes.YearCell}>{film.year}</td>
                  <td className={classes.DirectorCell}>{film.director}</td>
                  <td className={classes.StarsCell}>{film.stars}</td>
                  <td className={classes.ReviewTextCell}>
                    <div className={classes.ReviewText}>{film.review}</div>
                  </td>
                </tr>
              );
            })
          : null}
      </MDBTableBody>
    </MDBTable>
  );
};
```