Development of a Media Database Website
ECM3401 - Final Report & Demonstration

Abstract

This report details the design, implementation and evaluation of a media database website. The origin and utility of these contemporary websites has become clear as media services migrate online towards subscription-based content delivery - users are presented with more choice than ever before regarding content consumption. Streaming services and media database websites have risen in popularity at a commensurate rate, retaining high user engagement by succinctly presenting key information about the most relevant media content (films, books, series etc.) to its users and simultaneously providing a wide array of methods to browse the database. Notable design challenges include database structure (associative entities with deletion protections), upholding RESTful URL principles and overcoming the limitations of both content-based and collaborative filtering algorithms for recommender systems.

The previously undertaken literature review, project requirements and evaluation criteria are first summarised. The consideration of human-computer interaction assisted in developing front-end mockups to be followed during implementation. The back-end design describes relationships between Django database models, URL paths, views and templates, giving key challenges and departures from the original specification where appropriate. The implementation covers the processes, challenges and optimisations encountered when developing features such as rating forms, customisable profile sections and the film recommender system. The product will be tested through browser and device compatibility tests, user acceptance testing informing front-end design improvements and finally analysis of recommender system results. An evaluation and critical analysis of the product will involve assessing the project based on the extent to which the initial requirements were fulfilled.

I certify that all material in this report which is not my own work has been identified.

Contents

1 Introduction

In the modern information age where information from a variety of sources is highly abundant, the accessibility and presentation of data is what shapes how users interact with online services (Jain, 2019). The rise in popularity of media content streaming services such as Netflix, Amazon Prime Video and YouTube has led to a broader spectrum of online content being available with a significantly reduced barrier to entry. It is for these reasons that the resource structures, layouts and navigation options of many websites are arguably the most crucial elements in modern, effective web design. This is particularly true when one considers that more corporations are migrating online - ecommerce and subscription-based content delivery are replacing traditional one-time-purchase business models (Wayne, 2018). All of these services are in direct competition with one another (Lee et al., 2018), meaning an acute understanding of how users interact with each page, the basis of human-computer interaction, will help developers maximise the time spent on their website.

In the context of online content, two forms of websites have significantly increased in popularity in recent years: the streaming services themselves and also media database websites. Where the streaming platforms are focused on presenting popular titles for the user to watch, database websites have no discernible vested interest in promoting particular items for users to engage with. Despite the differences in intent, both forms of website aim to maximise user engagement by displaying the most relevant content to the user whenever possible. In terms of page designs, this involves prioritising surface-level and visual information over the smaller details and providing the user with a wide array of options as to how they wish to browse the database (Jain, 2019), ensuring the user never feels trapped or that they have reached a 'dead-end' in the resource hierarchy. In terms of the content itself being presented, this may be enhanced by a recommender system that is aware of what the current user is likely to respond well to, thereby increasing the chance they remain active on the website.

The following report describes the design, implementation and evaluation of such a media database website. A summary of the literature review and project specification will first be given, covering areas such as front-end design principles (human-computer interaction and promoting many ways to browse), structurally sound database design and effective approaches to developing recommendation engines. The specification and design for the system will then be covered, introducing example page designs and justifying the choices made in terms of their layouts and features. The design philosophy of the database's main tables and associative entities will be explored, justifying the use of certain field types, model types and relationship types. The development pipeline for the chosen web development framework, Django, will be given, including a discussion of its advantages and shortcomings in terms of its database abstraction, view types (function vs class-based) and its front-end template language.

The next sections will cover the challenges encountered when implementing some of the website's major features, highlighting the cases where either solutions have been developed or where alterations to the design specification have had to be made. To provide some notable examples, one of these challenges was encountered when optimising the database to simultaneously provide many-many relationships with deletion protections while upholding foreign key constraints. A further challenge was ensuring RESTful resource hierarchy principles were followed with URLs that identify resources via slugs as opposed to iterable IDs. Another problem was found when trying to overcome the limitations of the Django web framework's front-end template language by having to write custom tags to extend its functionality. Additionally, designing collaborative filtering and content-based filtering algorithms in a way that enables their use in isolation and as a hybrid recommender system introduced a number of optimisation issues that had to be overcome. Providing quality-of-life features such as front-end feature interactions without page refreshes through AJAX POST requests provided further implementation challenges.

The final sections will be comprised of the results of compatibility tests, user acceptance tests and recommender system performance tests, followed by an analysis of the extent to which the media database website project has met the requirements based on the evaluation criteria. Here, overall successes and limitations of the website will be analysed to determine if the final product can be seen to have fulfilled the objectives for the project.

2 Summary of Literature Review and Project Specification

2.1 Literature Review

In regards to front-end web design, despite the differences between online content streaming services and database websites, it has been found that both are optimised in similar fashions to maintain their user base and to maximise engagement time (Smith and Linden, 2017). Firstly, they provide a wide variety of means to browse the website - a large proportion of the text on a given page is a link to another resource, page or viewer. If the element or container is not interactive, it often contains very visual information such as a poster or trailer to punctuate and break up the blocks of text; very little text is purely static. Another similarity is that individual article pages present the most broadly-useful terms near the top with more detailed information below (Jain, 2019). For example, a film page may present the title, its length and its director/main cast near the top to catch the user's eye quickly - information such as its budget or producers will lie further down. Furthermore, a study of popular media database websites including the Internet Movie Database (IMDB, 2021), Goodreads (Chandler and Chandler, 2021), Letterboxd (Buchanan and Randow, 2020) and the Internet Games Database (IGDB, 2021), revealed that they all adhere to responsive best-practices. By implementing responsive containers and grids, the elements occupying these structures can be made to fit any browser or device type, a vital consideration in modern web design.

Also of notable importance is these media database websites' implementations of recommender systems to provide suggestions to the user for media to watch, play or read. While the front-end is important to attract the user to various features, it is the powerful, well-designed back-ends of these systems that enable them to present the most relevant information about the most relevant media to its users. The central intent that encompasses a recommender system's design philosophy is that a user is much more likely to engage more positively and for longer with content that they are likely to respond well to. The central problem problem is described below:

$$\forall c \in C, \quad s_c' = arg\ max\ u(c, s) \tag{1}$$

for user $c \in C$, we are aiming to find item $s' \in S$ that maximises the utility function u (Adomavicius and Tuzhilin, 2005). By considering the properties of the user space C or the item space S, respectively, we arrive at contrasting methods to recommending media: collaborative filtering and content-based filtering. The former involves predicting the active user's preferences based on those of others that the system deems 'similar' (Chen et al., 2017). Content-based filtering, on the other hand, use the items' raw characteristics to determine measures of similarity between them (Son and Kim, 2017; Pal et al., 2017), a crucial step to which is weighting the different attributes in a way that more realistic human judgement accounts for (Debnath et al., 2008). In the context of a film recommender, this would allow a developer to set the importance of actors and directors higher than genres and tags, for example.

After a comparison between memory and model-based collaborative filtering techniques, information retrieval methods, cluster analysis algorithms and a variety of similarity distance metrics, the shortcomings of each method were becoming apparent. The ways to create a hybrid of collaborative and content-based filtering were explored to alleviate their individual limitations. These include the cold start problem, where a lack of item interaction cannot provide recommendations (Chen et al., 2017). The new user problem is also present where many ratings will have to be made by new users to create meaningful recommendations (Adomavicius and Tuzhilin, 2005). Further issues include users having niche tastes, presenting a problem where unique rating profiles will not be able to be matched with other users, and overspecialisation, a self-fulfilling problem where a user is only recommended similar items due to their ratings profile being limited to a few select attributes (Leskovec et al., 2016). Feature-augmentation was found to be the most effective technique to hybridise the two types of recommender system to overcome these issues (Burke, 2007). The method proposed by Salter and Antonopoulos (2006), where results of collaborative filtering are fed into content-based filtering, was chosen to form the basis of this project's recommendation engine due to its high coverage of the content space, high rating prediction accuracy and its ability to manually assign the weightings of certain attributes higher than others.

2.2 Functional Requirements

The project's functional requirements for the front-end include being able to browse five types of media (films, television series, video game, books and web series) via people, companies, genres, tags, franchises and, in the case of video games, console hardware. It was a requirement that each of these database tables should have their own detail pages for each record, implemented using Django class-based `DetailView` instances. The structure of the URL hierarchy should uphold RESTful principles, with successive additions to the URL returning gradually more specific resources. Forms should exist for users to be able to contribute to the database by adding records for each media type. Pages for passive recommendations should exist that sort media types by the highest average user rating or box office gross - these results should also be viewable on subsection home pages. Taken together, the specification defines 76 required database tables for standalone models (e.g. Films) and mappings (e.g. Film-Person Mappings) combined.

The navigation bar for the website must provide links to subsection home pages, links to contribution pages, the user's profile and a search bar. Horizontal sliders and image carousels should aim to be implemented as a means to present visual content and simultaneously provide ordered hierarchies. Forms to give ratings and reviews for media items should be present if the user is logged in. A feature to add an item to the user's tracking list is also a requirement, enabling them to track their interests. The user should be able to edit these along with their profile information (name, picture, bio etc.) if they wish. Users should also be able to follow others and see their recent ratings/reviews on an activity feed page. The media database website will also have a recommender system based on the approach from Salter and Antonopoulos (2006). The hybridised system feeds results of collaborative filtering into content-based filtering. Implicit voting (where user data is gathered from browsing history or purchasing trends) is not within the scope of the project - explicit voting will be used instead where users rate the media items on a scale of 1-5 stars.

2.3 Non-Functional Requirements

The agile development methodology has been chosen - each additional aspect of the website is to be completed and tested in isolation, largely due to the clear distinction between the separate features on the front-end and the back-end views and underlying database tables. It was found that upholding responsive web design principles, ensuring compatibility across major browser and device types, was another key non-functional requirement for a project of this nature, meaning select HTML/CSS features should be implemented to resize, reorder or hide certain elements and containers where appropriate. From a security standpoint, the users' chosen passwords will be encrypted using the Django framework's default key derivation function, PBKDF2, meaning the risk of information being compromised is minimal. Additionally, users will not require personal information to create accounts, further reducing this risk. Beyond the Django Framework, hosting on the Heroku infrastructure and using an Amazon Web Services S3 bucket for static files (images and certain styling), there are no other specific software or hardware requirements for the media database website project.

2.4 Evaluation Criteria

The solution's success in general will be evaluated against the extent to which the aforementioned functional and non-functional requirements are met. If the page types are implemented in a visually pleasing and effective manner and if all features needed are present, it can be said to have been a success in this area. Feedback for the usability and style of the front-end will be gathered, informing any iterative changes that must be made to certain pages, the ultimate goal being to make the user experience as pleasant as possible. The extent to which the database, its tables and its records are ACIDic (atomic, consistent, isolated where appropriate and durable), ensuring there are no violations of integrity constraints and foreign key constraints (protecting the large quantity of associative entities), are what make up the criteria for evaluating the database. The results of the recommender system can be evaluated based on: content space coverage (proportion of the media content items that can be recommended in given scenarios) and predicted rating precision. These combined will provide an overall measure of success. Assessing the quality of the recommendations is a largely empirical exercise - comments on its efficacy will be provided by those who gave feedback on the user interface.

3 Design

The system is to be developed in Django, a full-stack web development framework for Python following the model-controller-view software design structure. This choice was motivated primarily due to its feature richness and due to a lack of experience in competing frameworks. Django's ORM (object relational mapper) grants a high level of control over the database through Python classes. This abstraction away from the underlying SQLite towards higher-level programmatic Python presents a much more developer-friendly representation of the database. Django being a full-stack framework means it also has front-end design tools - chief among these is the template language. The various tags, filters and logical blocks that make up the language are placed within the HTML, allowing Python variables and other context to inform a very powerful and dynamic front-end design (Django, 2021).

3.1 Front-End Design - Features, Layouts and Colours

Considering theory behind human-computer interaction is critical when designing a user interface. In the context of front-end web design, the key questions to consider are as follows: 1) "why does a user visit a website?", 2) "what does a user intend to accomplish on a website?" and 3) "what impacts their experience?" (O'Connell and Murphy, 2007). Framing the design of the website around these questions will provide a comprehensive understanding of each page, its purpose and what the user will expect to be able to find there. The user is visiting the website to find information about media items, meaning a wide array of browsing methods must be made available from every page. What they hope to accomplish may vary, but these features (adding to a tracking list, giving a rating etc.) must be visually distinct from their surrounding elements and must be operable by reading minimal text.

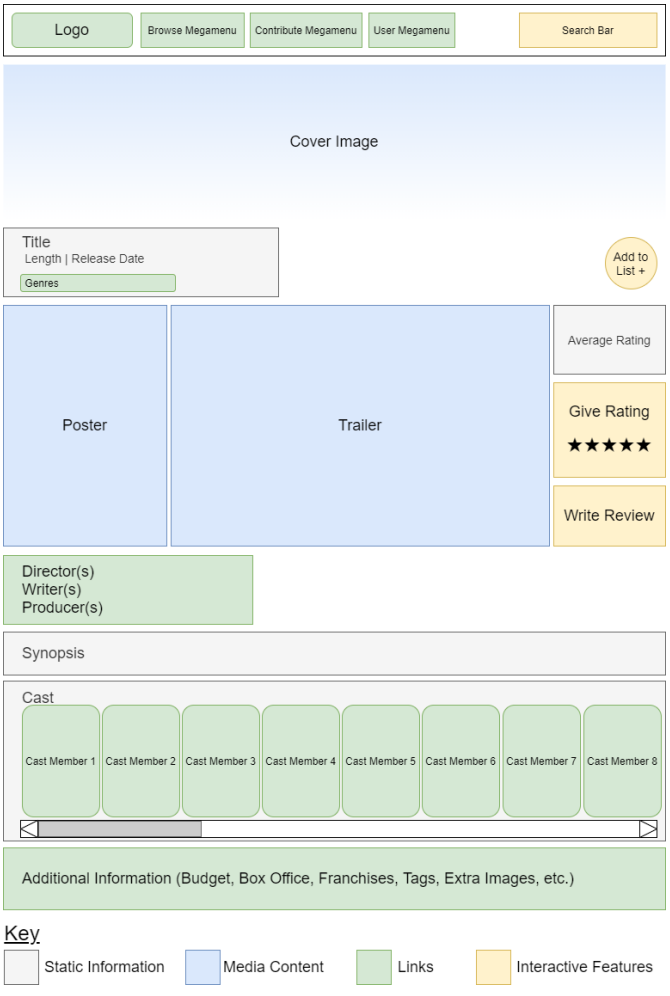


Figure 1 - Initial Layout of Item Detail Page

The interactions a user has on the website when using its features will be impacted primarily by how effectively each feature provides its intended purpose. This means, in addition to being visually distinct from one another, each feature should operate in a logical fashion and should ultimately enhance their experience as opposed to complicating it. With these human-computer interaction considerations in mind, designs for some of the database website's important pages will now be drawn up, described and justified. The goal here is to take advantage of the features and layouts that competitors have found success with, while also addressing the human-computer interaction considerations, creating designs that fit within this database website specifically (in terms of both the theme and the project's scope).

Figure 1 shows the layout of a detail page for a specific item such as an individual film. The visual content consists of the poster, trailer and cover - these serve to help the user to engage with the content and they assist with the page's readability by separating the surface level, key information from the more detailed facts below. A navigation bar will be present at the top with the website logo and browsing options - this is common to all pages and will be implemented on a base HTML template from which all others will inherit. The 'Browse' button will open a mega-menu providing links to the subsections of the website, the 'Contribute' button will reveal

links to add new content to the database. A final dropdown menu will be the user's username, providing links to their profile page, their tracking list and a logout button. If the user is not logged in, this dropdown will be replaced by 'Login' and 'Register' buttons on the navbar.

A large portion of the text will act as links to other types of similar detail pages. Every genre, actor, director, writer, producer, franchise, tag and company (eg: distributor, production company etc.) will be a followable link to see an overview page for that item. In the case of the cast members, a horizontal slider container will not only be an interactive method to view the cast, but will also offer a sense of order. The hierarchy will be based on higher-billed cast members appearing before lower-billed ones. There will be interactive elements beyond just links, the first of these is an 'Add to List' feature. This will be a dynamic +/- button that will correspond to whether the item is currently in the active user's tracking list. Clicking this will add or remove the item accordingly. There will be a 5-star rating input form on the right-hand side, allowing the user to make or change the score for this particular item. Finally, a 'Write Review' button will bring up a modal window overlaid above the rest of the page - this will provide a text area to write a review or update an existing review for this specific piece of media.

The initial design layout for the primary home page of the database website is given in Appendix A (it also applies to subsection home pages for films and for television, for example). Given that it will also inherit from the base HTML template, the navigation bar will be present here once again. A sliding carousel element will sit at the top and loop over a series of banner images. Left and right buttons will allow the user to change the banner manually. Five small, looping videos will sit underneath this, each representing a category of media present in the database (films, television, video games, books and web series). Horizontal sliders will be present once again, this time displaying information such as upcoming and highest rated media of any type, in addition to some recommendations if the user is logged in. In the case of the video game subsection home page, hardware platforms (consoles and operating systems) that games may be available on, in addition to game developers and publishers, will also be present in horizontal sliders - these all serve to provide as many ways to browse as possible.

A user's profile page will be designed based on the layout given in Appendix B. Their profile picture, username and bio should exist near the top and their recent activity is given in a horizontal slider, as are any custom profile sections they may have added, each of which features a selection of media the user wishes to showcase on their page. These custom profile sections are an extension of the initial front-end requirements, the justification for which is as follows: by providing a very dynamic set of tools on a user's profile page, they will be able to showcase any media item in a highly customisable fashion, providing means to personalise their page and potentially entice other users to follow them.

Figure 2 - Analogous colour scheme for the media database website

An analogous colour scheme will be applied to the website's highlight colours, given in Figure 2. This means the main accent colours (the blue and green) are similar to each other, grouped together on the colour wheel, as opposed to complementary or triad schemes that find contrasting colours with opposing hues (Hindman, 2015). The left-most white will be used for general body text and links, the blue will be used to highlight particular usernames on pages such as the activity feed and for confirmation buttons, the central green will be used for the search buttons, the grey will be used for all of the website's container elements and the darker grey on the right will serve as the page body background colour. The website's logo will also be designed to follow this analogous colour scheme - it will be the name of the website "MediaDB" on the far left of the navigation bar, the colour of which will be a transitional gradient between the green and blue in Figure 2. This high-chroma logo being visually distinct from the generic, muted grey containers is an effective, eye-catching design technique (Stone, 2006). The overwhelming majority of the website will use these same elements, containers, structures and colours, unifying all of the pages and features with an overarching design language that simultaneously provides succinct information, visual hierarchy, engaging widgets and a wide variety of sources of visual information. These principles were found to be important following the literature review and the final implementation of the front-end is aiming to incorporate all of these in an effective and visually pleasing manner.

3.2 Back-End Design - Django Development Pipeline and Distinct Apps

The development pipeline when using the Django framework will now be outlined to contextualise the following sections in the project as a whole. Given that the website is highly database-driven, the first key consideration is designing the database models definitions, allowing individual media items such as Films (or links such as Film-Genre mappings) to be represented. Next, particular URL paths must be set to call function-based views or instantiate class-based views, the differences between which will be outlined in 3.2.5. These views are what render the HTML files and provide the context data from the back-end to the front-end. The final step of the process is the creation of these HTML files that inform the layout and markup of the front-end, inheriting from a base HTML file and applying CSS and JavaScript when necessary to enable particular features to be realised.

The design of the back-end must also acknowledge that the Django framework allows developers to create multiple 'applications' within a project - these apps are distinct Python packages containing files for their respective database models, URL paths, views and HTML templates (Django, 2021). It has been decided that two are to be implemented for the media database website project: Media and Users. The Media app will consist of media subsections and home pages, search results, recommendations and the detail pages for each media type, people, genres, tags, franchises, companies and video game platforms - in other words, the bulk of the database. The Users app, on the other hand, will provide login, logout and register functionality, the user profile and user tracking list pages, and finally the user activity feed page. This clear partition between sections of the project will help to separate the implementations of unrelated features in a logical fashion.

3.2.1 Database - Main Tables Rationale

The design of the database tables and relationships is crucial to the success of a media database website. To that end, careful consideration will be taken when designing and planning the structure of the database. Arguably the most important tables are those of the five media types, which are as follows: films, television series, video games, books and web series. These will each require Django `CharFields` to allow their title, synopsis and age rating to be stored. Release dates will be represented through `DateFields`. In the case of films, their budgets and box office earnings must also be implemented as `IntegerFields`. In the case of television or a web series, if it is not currently on-going, their end dates will need to be given in addition to the number of episodes and series. The five media types will also take `ImageFields` as means to store paths to the locations of their hosted posters and cover images. Finally, another `CharField` will store a URL of a trailer video.

Beyond simply the media items themselves are the various ways to browse the website, meaning database models are required for people, companies, genres, tags, franchises and consoles. It has been decided that franchises should be broken down into their constituent subcategories. The justification for this decision is best understood through an example: 'Star Wars' as a brand may be subdivided into the original films, modern films, animated series and video games. By separating which media items are applied to which of the franchise's subcategories, it will provide a far more meaningful and logical way to browse these media items on the front-end, rather than presenting them all in one section.

3.2.2 Database - Associative Entity Relations

In order to deal with many-many relationships between object types, of which there will be a large number in this database, a decision must be made: either create associative entity relations that map tables together using Django `ForiegnKey` fields, or add a `ManyToManyField` to the media's table definition where every single linked instance is listed. The former was chosen for two reasons.

Firstly, if a `ManyToManyField` was used, it would become difficult to find individual Film-Person relationships and whenever that information was needed, the whole Film object with all of the extra fields would be returned. In addition, crucially, it would not allow further fields to be added that describe the many-many relationship. By creating an associative entity relation class to represent Film-Person mappings, fields defining the person's role (e.g. actor, director, writer etc.), their optional character name and their order in the cast billing can all be added as additional attributes.

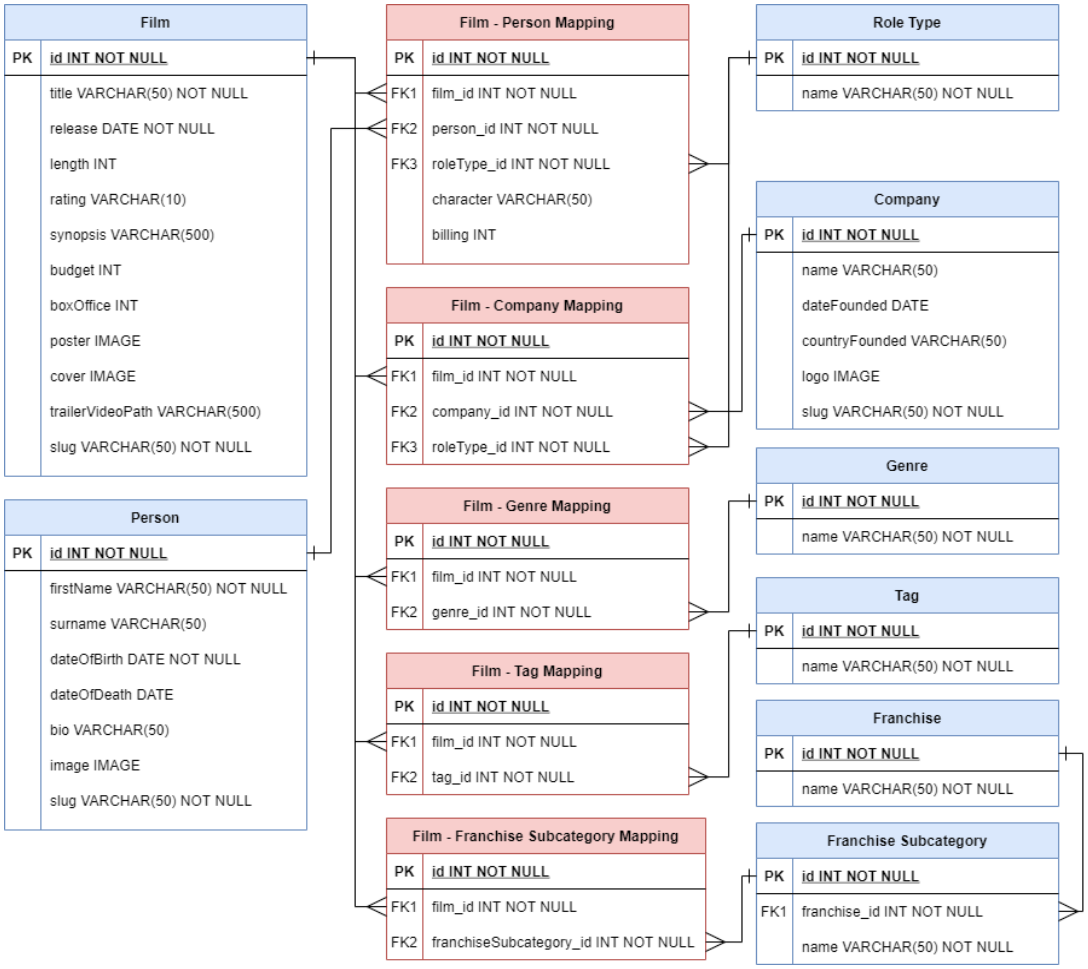


Figure 3 - Entity-relationship diagram for films and their associated mapping tables

To illustrate this, definitions of tables representing associative entity relations such as Film-Person, Film-Company and Film-Genre mappings are given in red in Figure 3. For example, three foreign keys are used in the definition of a Film-Person mapping, making reference to the Film, Person and Role Type tables. This design philosophy has been applied to the other media types and to the database as a whole. There are very many examples of these relationships in the database, Video Game-Console, Book-Franchise and Web Series-Tag mappings to name just three. This has resulted in a very large number of total database tables, but it has also granted a high level of control over every possible many-many relationship in the system.

It is evident from the diagram that, with the exception of titles, release dates and foreign keys, many of the fields do not end with NOT NULL such as a film's length, a person's bio or a company's date founded. This data may not always be available and the system must be able to cope with variations in information availability and quality. If the data is present, the front-end will be able to display it in a pleasing manner and if not, the layout should be designed to not suffer as a result.

3.2.3 Database - Deletion Protections

A further consideration when dealing with Django `ForeignKey` fields is to design them with deletion protection in mind. The `on_delete` parameter of the foreign key allows a number of behaviours. The `CASCADE` choice ensures that dependent instances of Film-Person mappings are removed if the film, the person or the role type objects are deleted from the database. The alternative is to set the foreign key to `PROTECT`, where the records are kept, even if these instances are now dependent on non-existent data. The `CASCADE` option was chosen as the nature of the websites means that the mappings serve no purpose if their referenced parent records are removed (e.g. a Film-Genre mapping has no purpose if the Film record or the Genre record is removed).

3.2.4 Resource Slug Identifiers

```
#---Media Type: Film-------------------------------------------------------------#
class Film(models.Model):
    title = models.CharField(max_length=150, default='NoFilmTitleSpecified')
    release = models.DateField(default=timezone.now)
    length = models.IntegerField(null=True, blank=True)
    rating = models.CharField(max_length=10, blank=True)
    synopsis = models.CharField(max_length=500, blank=True)
    budget = models.IntegerField(null=True, blank=True)
    boxOffice = models.IntegerField(null=True, blank=True)
    poster = models.ImageField(upload_to='posters', blank=True)
    cover = models.ImageField(upload_to='coverImages', blank=True)
    trailerVideoPath = models.CharField(max_length=500, blank=True)
    slug = models.SlugField(max_length=150, editable=True)
```

```
def save(self, *args, **kwargs):
    super().save()
    if not self.slug:
        self.slug = slugify(self.title + "-" + str(self.release))
    super(Film, self).save(*args, **kwargs)
```

Figure 4 - Python class implementation of the Film database model

A sample database model definition is given in Figure 4. Its `save()` method has been overridden to generate the `SlugFied`. The first time this is called (i.e. when a film's information is first entered), the method generates a slug for that film based on its title and release date (e.g. "zodiac-2007-03-02") in lowercase and without spaces (if there is not one already). This is used in the URL for each film's detail page, ensuring it adheres to RESTful best-practices and also remains readable. The design of the project should consider that slug fields need to be generated and applied in this way to many of the database's tables, including other media types (generated from their title and release date) and people (generated from their name and date of birth).

3.2.5 RESTful URLs and Function-Based versus Class-Based Generic Views

The process for making specific addresses render particular pages in Django begins with URL path mappings (different strings in the URL call corresponding views). The design of the website should ensure that practices associated with RESTful resource hierarchies are upheld.

The structure dictates that each addition to the URL should bring the user a gradually more specific resource. An example hierarchy for the media database website is given in Figure 5: '/films/' will direct the user to the website's "Films" subsection home page. Next, adding a particular film's slug (e.g. "/films/zodiac-2007-03-02") will bring up that film's detail page. Finally adding '/crew/' to the end of the URL will render that film's crew detail page where more thorough crew information is displayed.

```
path('films/', views.filmHome, name='film-home'),
path('films/<str:slug>', views.FilmDetailView.as_view(), name='media-film-detail'),
path('films/<str:slug>/crew', views.FilmCrewDetailView.as_view(), name='media-film-crew-detail'),
```

Figure 5 - Sample Django URL resource hierarchy

These each create a call to either a function-based view or a class-based view. It is these views that will query the database, generate context for a particular template and render the appropriate HTML. Function-based views will be used in scenarios such as the website home page, subsection home pages, search results and any other type of page that is not tied to a specific database table record or set of records. By contrast, instances of class-based views such as `DetailView` or `UpdateView` are tied to individual records of database model objects, providing the context and rendering the pages for particular films, people, franchises or user profiles, for example. The contrasting approaches to implementing function and class-based views will be given across section 4.

3.2.6 Django HTML Template Language and Custom Template Tags

The HTML extract in Figure 6 demonstrates the use of the Django Template Language. Using curly braces allows the context passed into the template to be rendered, e.g. `{{film.title}}`. This also allows for logical blocks such as `if` statements and `for` loop structures to inform the front-end design. The example here demonstrates the use of `{% if %} {% else %}` blocks to determine whether the film has an age rating, whether it has a length and a `{% for %}` loop structure to render links to each of the genres the film has been applied to.

```
<div class="container titleSection">
    <h2>{{film.title}}</h2>
    <h5>{% if film.rating %}{{ film.rating }} | {% endif %}{% if film.length %}{{ film.length }} mins | {% endif %} {{ film.release }}</h5>
    {% if genres %}
        {% for genre in genres %}
            <a style="padding-right:10px" href="/genre/{{ genre.genre.title }}"> {{ genre.genre.title }}</a>
        {% endfor %}
    {% endif %}
</div>
```

Figure 6 - Sample use case for the Django template language logical blocks within HTML files

The template language is not feature-complete, however. In order to obtain certain results, the default functionality must be overridden or extended. This is achieved through writing custom template tags. These are Python functions that take information from the HTML context, perform some processing that the template language cannot parse, and return the result. Several custom template tags must be written to fully-realise the project as set out in the requirements, the most important of these will be explained and justified in 4.3 as its implementation and use case is described.

4 Development Process and Feature Implementation

4.1 Media Detail Pages

The view for each type of media item was implemented using an `UpdateView`, a type of class-based generic view. The justification for this is as follows: much like a `DetailView` (that genre, tag and franchise pages use), the `UpdateView` is tied to a particular record of a table such as a Film. However, the `UpdateView` also has a `post()` method (meaning it can handle requests beyond just page GET requests) whose definition can be overridden. There are three types of POST requests the user can make on a page for a media item: creating/updating a rating, creating/updating a review or adding to/removing from their tracking list. The `UpdateView` is necessary to implement these features.

Appendix D shows a fully-implemented film information page containing its: title, release date, poster, cover image, trailer, age rating, synopsis, budget, box office earnings, cast, directors, writers, producers, genres, tags, franchise, production companies, distributors, additional images, average rating and the rating given by the active user. This has been built according to the initial layout design in Figure 1. Appendix E shows an equivalent for a television series and Appendix F shows one for a video game.

4.2 Ratings and Reviews

The templates for each media type (e.g. FilmDetail.html, BookDetail.html etc.) implement the necessary containers and forms allowing users to make ratings and write reviews. The base HTML template implements the JavaScript to handle the forms and POSTs the changes to the database using AJAX, meaning the page does not have to be refreshed for the changes to be reflected on the front-end. A jQuery plugin named RateYo! (RateYo!, 2021) has been adopted to implement this feature. It provides SVG-based star ratings (meaning no images are required) whose colours, sizes, values and behaviours can be customised through simple parameters. The Django template language has been used to set the star rating's default value to the user's rating of that item (if a rating has already been made). A function has been written to run if the `rateyo.set` event fires, performing a POST request to the database, updating only the necessary elements to allow the changes to be viewable without a refresh. Given that media pages are rendered by a class-based generic `UpdateView` (as justified in 4.1), the `post()` method is able to handle these POST requests by calling a handler function named `mediaPagePostRequests()` - this is where database records are created/updated as required. A user creating or updating a review for a particular film is handled in a similar fashion through: front-end containers and forms in the media page HTML, the JavaScript and AJAX in the base HTML, the `post()` method in the `UpdateView` and finally the `mediaPagePostRequests()` handler. The JavaScript and AJAX for making ratings is viewable in Appendix G. Appendix H shows the `post()` method calling the `mediaPagePostRequests()` handler. Appendix I shows the relevant lines of this handler that creates/updates a rating. Appendices J and K show this process for reviews.

4.3 Franchise Detail Pages

Challenges were encountered when implementing the detail page for a particular franchise. Firstly, the various subcategories belonging to that franchise have to be found. The constituent media items that make up each subcategory then have to be combined. The films, television series, video games, books and web series belonging to each subcategory are found based on several conditions. The query cannot only check to see if the media belongs to a certain subcategory name (because multiple franchises may have a subcategory titled "Main Films" or "Spin-offs", for example), the ID of the subcategory's parent franchise must also be equal to the ID of the current franchise being rendered. The media items are then chained together and sorted based on the `orderInFranchise` field, allowing multiple types of media to occupy the same franchise subcategory and in any order. However, after the media items have been successfully combined into a subcategory, there is an issue regarding how to reference this subcategory in the HTML template for it to be rendered out. Typically, supplying context to a template takes the form: `context['keyName'] = value`, but in this case, the key name is dynamic (there could be any number of subcategories which could be titled anything), meaning it must be passed in the following way (full excerpt in Appendix L):

```
#Collect and sort all the media for that subcategory based on the 'orderInFranchise' field
completeSubCategory = sorted(list(chain(SubCat_F, SubCat_TV, SubCat_VG, SubCat_B, SubCat_WS)), key=attrgetter('orderInFranchise'))
#Dynamic context name for the subcategory in the template
context[subcategories[x].title] = completeSubCategory
```

Figure 7 - Dynamic franchise subcategory context names when passing the data to the template

The HTML template must be implemented in such a way that it can reference these dynamically assigned context names - the Django template language cannot facilitate this by default. Achieving this functionality will involve the creation of a custom template tag.

```
#Access contents of dynamically-created variables
@register.simple_tag(takes_context=True)
def dynamicVariableValue(context, DynamicVariableName):
    #Returns value of DynamicVariableName into the context
    return context.get(DynamicVariableName, None)
```

Figure 8 - Custom template tag to enable referencing of dynamic context names

Figure 8 shows the implementation of the custom template tag that alleviates the issue outlined above. Very little processing is involved, but it is using a feature of the back-end Python language to enhance the functionality of the front-end template language. The dynamically-created names for the various subcategories of a franchise cannot be referenced without this new tag. It is used in the template for the `franchiseDetail.html` as follows:

```
{% with var=subcategory.title %}            <!-- Temporary variable var
{% dynamicVariableValue var as var_url %}   <!-- Apply custom tag
{% for entry in var_url %}                  <!-- Iterate through subcategory
    {%  if entry.film != NULL ... %}        <!-- Render if entry is a film
    {%  if entry.television != NULL ... %}  <!-- Render if entry is a tv series
    {%  if entry.videoGame != NULL ... %}   <!-- Render if entry is a video game
    {%  if entry.book != NULL ... %}        <!-- Render if entry is a book
    {%  if entry.webseries != NULL ... %}   <!-- Render if entry is a web series
```

Figure 9 - Usage of the custom template tag in franchiseDetail.html

The HTML extract in Figure 9 (full block viewable in Appendix M) renders each of the items in each of the subcategories of the franchise. A variable `var` is set to take the name of the current subcategory. The newly created custom template tag, `dynamicVariableValue`, is then called on this value. The result can be iterated over, giving the subcategory's entries (its films, books, games etc.), an impossible process using the Django template language alone.

This custom template tag was used again to deal with the custom profile sections a user may wish to add to their profile page. There could be any number of sections named anything, meaning this custom template allows each profile section to be referenced dynamically in memberProfileSection.html.

4.4 Adding Media to Lists and Following Users

A +/- toggle is implemented to allow the active user to add certain media items to their tracking list where they can save particular pieces of content. However, this same feature has been repurposed to also allow the active user to follow other accounts. This functionality was achieved through CSS, JavaScript and an AJAX POST request. Whenever the button is clicked, the CSS dictates that an animation plays and it morphs from a + to - symbol or vice versa. A script in the base HTML template runs a function whenever this happens, creating an AJAX POST request that refreshes just the button and sends either "add" or "remove" to the request handlers described before in 4.2 - the function then adds or removes the item accordingly. This same script is used for following/unfollowing another user. The `post()` method of the `UpdateView` instance for that user either creates or removes a mapping between two users on the UserFollowing database table. It is these mapping records that represent one user following another user in the database. The +/- toggle JavaScript is viewable in Appendix N.

4.5 Recommender System

An outline of the implementation of the film recommendation engine is given in Figure 10. It is based on a modification of the approach proposed by Salter and Antonopoulos (2006). The hybridised recommender system uses collaborative followed by content-based filtering, a hybridisation technique known as feature augmentation (feeding results of one approach into another) (Burke, 2007).

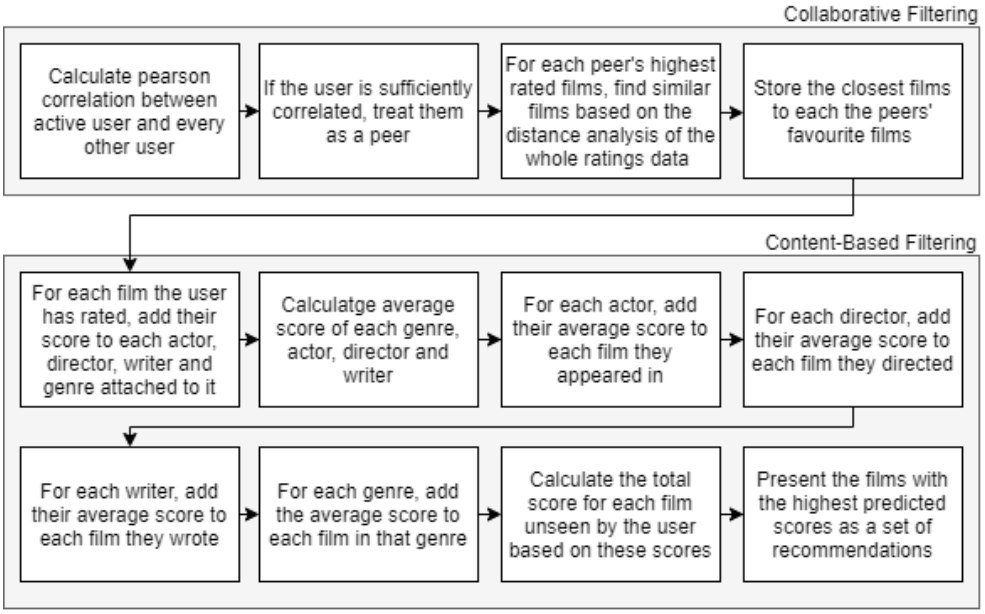


Figure 10 - Process for Generating Film Recommendations

4.5.1 Collaborative Filtering

The first phase of the recommender system, the collaborative filtering, begins with a call to a function that calculates user-user similarities. Initially, the ratings made by the active user, A, are compared to those made by every other user, B. If a film has been rated by both A and B, the scores are added to lists. The Pearson correlation between these lists is then found using `numpy`'s `corrcoef` function.

After finding these peers (the users with the highest correlations), a `collaborativeFiltering()` function is called. Two `pandas DataFrame` structures are made here, the first populated by records of the database's Film objects and the second by FilmRating objects. Where the 'title' field of a Film object matches the 'film' field of a FilmRating object, the records are merged, creating a final dataframe holding all of the required data. Preprocessing then occurs - firstly, records are removed if a Film has no FilmRatings attached to it. Next, a pivot table is created where each film is a row and each column is a user. The pivot table is converted into a CSR (Compressed Sparse Row) Matrix using the `scipy csr_matrix` function - it is a sparse matrix as most elements are zero (most users have not rated

most films). The `scipy` implementation of the CSR matrix was chosen as it is more efficient at slicing rows (films) than columns (users), an important distinction given that the goal is to recommend films.

Following these preprocessing steps, the `sklearn NearestNeighbors` class is instantiated and its `fit()` method is called, taking the CSR matrix as its input dataset. The active user's peers (the most similar users) are then retrieved. For each of these peers' highest rated films, the 20 closest films (the 20 nearest neighbours based on the cosine similarity distance metric) are retrieved. The resulting dataset is then passed into content-based filtering.

4.5.2 Content-Based Filtering

The second phase of the recommender begins in the function `contentBasedFiltering()` where three calls to `personRecommender()` are made (for actors, directors and writers, respectively). This function takes three parameters: the active user, a person type (actor/director/writer) and a set of films (the collaborative filtering results). A function `peopleScores()` is called each time, applying the user's ratings of each item in the film dataset to each of the films' actors, directors or writers. The peoples' average scores are then returned. The `personRecommender()` function continues by iterating through all films the user has not seen, finding a predicted score by summing the average scores for each of the actors, directors or writers, and sorting the result. The function finishes by returning the highest 100 predicted film ratings for that person type.

After `personRecommender()` is run three times to produce recommendations for actors, directors and writers, the function `genreRecommender()` is called to find recommendations based purely on genres. In a similar fashion, this recommender calls a `genreScores()` function to find the active user's average film rating per genre. This result is used when iterating over all films the user has not seen, determining a predicted rating based on the genres that the active user responds well to. The predicted film ratings based on actors, directors, writers and genres are all combined into a list which is presented to the active user on the front-end as a final set of hybrid film recommendations that were generated from both collaborative and content-based filtering.

4.5.3 Challenges, Results and Optimisations

A number of challenges were encountered during the implementation of the recommendation engine which informed changes subsequently being made to the design of the system. The first of these was the presence of inappropriate ratings in the final set of recommendations. A change from cosine similarity to Pearon correlation for finding the active user's peers was implemented. Cosine similarity was consistently exclusively returning similarity scores of 92% or higher and was failing to recognise negative correlations - this was resulting in many users that would otherwise not be considered a peer to the active user being treated as such. This was providing final recommendations that were in fact the opposite of what should have been recommended. After implementing Pearson correlation, values of the similarity coefficients ranged from -1 to 1 with a much more expected distribution and resulting recommendations were as expected, with fewer to no outlier inappropriate film suggestions.

An additional change was made when determining the active user's peers. It was found, in some cases, that active users were not able to find peers that met the minimum correlation requirement of 70%, resulting in no input films from which to perform collaborative filtering. This was alleviated by removing the minimum correlation threshold entirely. Instead, the 20 users with the highest rating correlations are selected to become the active user's peers. In most cases, this will assign the same peers as before, but the system will now accommodate users with fewer, moderately-correlated peers.

Given that film recommendations are displayed on the website home page if the user authenticated, an unforeseen issue arose where the page's loading times increased drastically. It was considered vital to the success of the recommender system that the recommendations are able to be displayed on the home page, meaning these load times had to be reduced. The approach to overcome this challenge involved restructuring how and when recommendations are generated - by storing a particular user's recommendations in the database, these could be accessed at any time, significantly reducing the number of times these computationally intensive operations would have to run. The first implementation attempt involved creating a new database table, UserFilmRecommendations, where

records would act as associative entity relations, with foreign keys linking users to recommended films - this worked as intended. However, after testing recalculations of film recommendations, it was found that removing and creating new records for every recommendation was highly inefficient, load times were still too long. An optimisation was soon found and implemented in its place: by serialising the unique IDs of the recommended films into the JSON format, a single record could be used to store all of the IDs of the films in order. This was later adopted to also store the IDs of the active user's peers to allow rating recalculations to process faster. These greatly reduced the loading times of the home page - it is able to quickly present all of the film recommendations as set out in the requirements.

A further unexpected result was encountered when implementing collaborative filtering. With every output set, several outlier films were still consistently present. After observing that these films always had either 1, 2 or 3 ratings applied by a small subset of users, it became clear that the system was frequently recommending these based on them always being close to one another in the CSR matrix. This was alleviated by adding an additional preprocessing step. Several `pandas` functions are called: `groupby()` (sorting all the ratings by film title), `count()` (counting the number of ratings) and finally `reset_index()` and `rename()` (to reapply indexes of each FilmRating record and to create a column storing the number of ratings per film). Films with fewer than 5 ratings are then filtered out before the `Dataframe` is converted into the pivot table. This ensures that these small self-contained subsets of the film ratings dataset are not accounted for when providing recommendations. By placing greater significance on higher-level trends in the data, the collaborative filtering will be able to provide more accurate similarity measures between more relevant films for a higher proportion of the users.

4.6 Customisable Profile Page Sections

The ability for a user to create a customisable profile section was described in 3.1 as an extension to the original set of functional requirements for the website's front-end featureset. These sections can be created to present media items in any fashion the active user wishes and in any order, introducing a significant element of customisability. The first step involved in their implementation was to create six new database tables. The first of these tables is `ProfileSection` - it stores a name and a media type as two `CharField` attributes. The section belonging to a particular user is represented through a `ForeignKey` field, mapping it to a user profile object. It also contains the fields 'order' (to represent the drawing order of the various sections on the user's profile page) and a `SlugField` to uniquely identify each profile section (created from the user's username and the profile section name). The other five database tables are associative entity relations that map each media type (Films, Television, Video Games, Books and Web Series) to a particular profile section. Again, these relations contain an 'order' attribute to represent the media item's position in the profile section.

Having enabled the database to represent a profile section and store mappings to each media type, the front-end must now be made to visualise them. The challenges encountered when implementing the franchise detail page (described in 4.3) apply closely to this scenario. Given that there could be any number of profile sections (that could be named anything and could contain any number of media items), the custom template tag must be used once again. Here, the dynamic context variable names that allow the profile section to be referenced in the HTML template, are assigned as follows:

```
if profileSections != None:
    for x in range(profileSections.count()):
        #Retrieve the Films, Television, Games, Books or web Series that make up the profile section
        films = ProfileSectionFilmMapping.objects.filter(profileSection__profile__user=self.object.id)\
            .filter(profileSection__sectionName=profileSections[x].sectionName)
        television = ProfileSectionTelevisionMapping.objects.filter(profileSection__profile__user=self.object.id)\
            .filter(profileSection__sectionName=profileSections[x].sectionName)
        videoGames = ProfileSectionVideoGameMapping.objects.filter(profileSection__profile__user=self.object.id)\
            .filter(profileSection__sectionName=profileSections[x].sectionName)
        books = ProfileSectionBookMapping.objects.filter(profileSection__profile__user=self.object.id)\
            .filter(profileSection__sectionName=profileSections[x].sectionName)
        webSeries = ProfileSectionWebSeriesMapping.objects.filter(profileSection__profile__user=self.object.id)\
            .filter(profileSection__sectionName=profileSections[x].sectionName)

        #Compile these media items into the full profile section and pass to the context
        completeProfileSection = sorted(list(chain(films, television, videoGames,books, webSeries)), key=attrgetter('orderInSection'))
        context[profileSections[x].sectionName] = completeProfileSection
```

Figure 11 - Dynamic profile section context names when passing the data to the template

By passing the name of each profile section to the context dynamically, as shown in Figure 11, the same template tag from Figure 8 can be used in the same way as demonstrated in Figure 9 to reference and render each section. After these sections are displayed, the next challenge is to implement a way to customise the order of these sections on the page. The jQuery plugin Sortable (Sortable, 2021) was found. It provides a function that takes the elements of an HTML unordered list (`<ul>`), serialising them based on a certain property (in this case, their HTML element IDs), and allows them to be dragged and dropped into a new order. The axis of the direction in which this occurs is set: x (horizontal) or y (vertical). Figure 12 below shows the JavaScript function handling this feature:

```
$(document).ready(function() {
    $("#sections").sortable({
        axis: 'y',
        update: function(event, ui) {
            var data = $(this).sortable('serialize', {attribute: "id"});
            $.ajax({
                url: "/user/{{ memberProfile.username }}",
                type: "POST",
                data: {'changing': "confirm", 'content': data, 'csrfmiddlewaretoken': '{{ csrf_token }}'}
            });
        }, }).disableSelection(); });
```

Figure 12 - JavaScript function using jQuery Sortable plugin to customise the order of custom profile sections

After a profile section has been dragged and dropped up or down (y-axis) into a new position, all of the items in the list (the profile sections) are assigned new HTML element IDs. An AJAX POST request passes this new order back to the `post()` method of the Django view for this user's profile page, which in turn handles the POST request accordingly, given in Figure 13:

```
#Gather all of the profile sections from the POST request
entries = QueryDict(request.POST.get('content'))
#Enumerate the sections and reorder them by setting their 'order' field to their index in the loop
for index, entry_id in enumerate(entries.getlist('section[]')):
    entry = ProfileSection.objects.get(id=entry_id)
    entry.order = index
    entry.save()
#Redirect to the user profile page with the changes applied in real time without a refresh
return http.HttpResponseRedirect('/user/' + object.username)
```

Figure 13 - POST Request handler to reorder the custom profile sections

The `section[]` list from the POST request is enumerated over. The 'order' field of each profile section is assigned to the index in this enumeration loop. This updates their database entries to match the values after the user dragged and dropped the sections into a new order. The AJAX request means this happens without the need for a page refresh at any point in this process.

A page was then required to allow media items to be added, reordered and removed. This was achieved through an `UpdateView` instance - as described before, these class-based views correspond to specific records in the database (in this case, each profile section). They have a `post()` method, meaning POST requests can be handled. The technique to reassign each item's 'orderInSection' value was the same as the approach to reorder the sections themselves, given in Figure 13. The only notable difference is that the reordering within a section occurs horizontally on the x-axis (viewable in Figure 14) as opposed to vertically on the y-axis, as before. Figure 14 below demonstrates how an item in the horizontal scroller section can be dragged into a new position. This fact is reflected in the database seamlessly with no page refresh required.

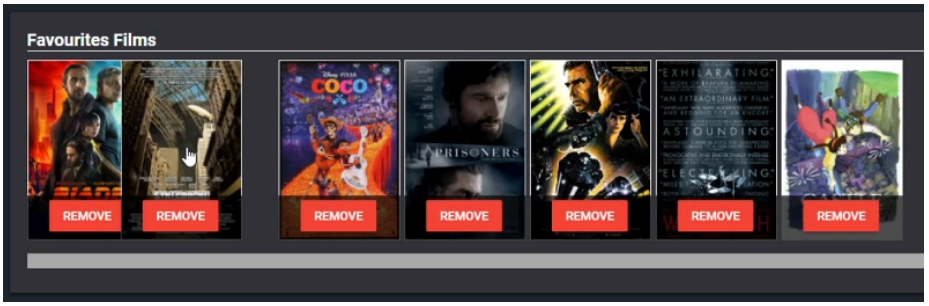


Figure 14 - Dragging and dropping list elements to reorder a profile section

Finally, the ability to add and remove these media items was developed. Deleting entries of a profile section is simply achieved through a 'Remove' button attached to each item - the `post()` method handles this request by removing the mapping record between the profile section and that media item. Adding a new item was achieved through a search bar and a series of 'Add' buttons. Clicking the button corresponding to a particular film adds it to the end of the profile section. Figure 15 gives an extract of the `post()` method handling a new film being added to a section (this applies to all media types).

```
#Retrieve the film object to be added and add it to the end of the section, saving the new record
if object.type == "Film":
    getFilm = Film.objects.filter(slug=mediaSlug)[0]
    sectionLength = 0
    #Retrieve the number of items (length) of the profile section
    for psfm in ProfileSectionFilmMapping.objects.filter(profileSection=object):
        sectionLength += 1
    #Save this new mapping record, specifying its 'orderInSection' to be the length of the section + 1 (the new end)
    newMap = ProfileSectionFilmMapping(film=getFilm, profileSection=object, orderInSection=sectionLength+1)
    newMap.save()
```

Figure 15 - Extract from the `post()` method allowing a film object to be added to a profile section

Where the unique `SlugField` of a `Film` object matches the slug of the new media item in the POST request, a mapping record is created between this `Film` object and the profile section (in other words, it is added to the end of the section). To make the profile sections as customisable as possible, a form was also added to this page allowing the user to change its name. Once again, this is handled through an AJAX POST request, meaning a page refresh is not required for this to be reflected on the front and back-end simultaneously. Finally, a 'Delete' button also exists on the page for a particular profile section, removing it from the database if clicked. Checks have been successfully implemented to ensure only the user to whom the profile section belongs can rearrange the media items, add or remove items, change the name or remove the section entirely.

5 Testing and User Feedback

The following section will consist of three forms of testing. The first will focus on compatibility with browsers and device types, the second will highlight front-end design changes made following user acceptance testing and the final tests describe the performance of the recommender system.

5.1 Browser and Device Compatibility Testing

The following outlines high-level compatibility tests, ensuring pages and features are usable on several browsers and devices. The testing took the following form: loading the home page, registering a new user, logging in, rating and reviewing one of each media type, adding one of each media type to the tracking list, contributing a new media item, creating, reordering and removing profile sections and finally adding, reordering and removing items in those profile section. The browsers' support of the website's interactive features were included (such as the horizontal scrollers and image carousel viewers). After testing Google Chrome, Microsoft Edge and Mozilla Firefox, all pages rendered correctly, all containers followed intended layouts and all interactive features operated as expected. The detailed breakdown is given in Appendix O. Using the in-built Google Chrome 'toggle device' tool, the viewport size was changed to emulate the user experience on a standard small smartphone (iPhone 5/SE), larger smartphone (Google Pixel 2 XL) and tablet (iPad). The same test was run for each of these. On the small and large smartphones, all features worked as intended, all pages were accessible and all layouts fit within the new viewport size by reordering and adjusting containers accordingly. On the tablet, however, this was not entirely successful: while the layouts changed correctly and the features remained functional, the navigation bar experienced issues. Being marginally narrower than desktops, links were being cut off on the right on the tablet. This was fixed by applying a new Bootstrap class to the navigation bar. The Bootstrap toolkit (Bootstrap, 2021) provides spacing and layout classes that can apply to all element types. Applying the class `navbar-expand-lg`, the navbar now responsively adjusts itself to hide the links in the collapsable burger bar menu at a wider viewport width, ensuring no links are cut off before this transition.

5.2 User Acceptance Testing

User acceptance testing was undertaken, the feedback from which informed iterative improvements to the design of the website, the goal being to optimise the usability and approachability of the front-end. Five students agreed to provide feedback - while three were computer science students who were more likely to consider implementation details, two had no background in web development or programming in general. The feedback from these students in particular was deemed crucial in making sure the system's approachability and usability was sufficient for any type of end user.

The questions involved the ways to browse the website and the general front-end style. 3/5 students strongly agreed with the statement: "I can easily access any other major page with 3 or fewer clicks" and one answered "I agree". The fifth student gave the answer "I neither agree nor disagree", justifying the response with: when attempting to search for a particular item, the search results page presenting a long text list of people, media and franchises was not visual enough. This was particularly true when the search query was very short as a large number of results were hard to distinguish between in the list. This informed a notable design change for the search results page where responsive grid-based image viewers now display all of the results in a far more visually engaging layout.

Following this change, 3/5 agreed the usability of the search page was improved. The two remaining students described that far too many results were being presented per object type, again making it hard to immediately find the desired item. This prompted a further design change: a limit of 72 results for people and media items, and a limit of 25 for franchises and consoles. These were chosen to simultaneously provide a relatively large number of results, but also to fill each object type's container on the page, visually illustrating that the user will benefit from narrowing-down their query. If people or companies do not have images, they are presented in a list at the bottom in a similar fashion to the original design. The original and new versions are visible in Appendices Q and R. Following these improvements, items can be found faster thanks to a more engaging and approachable search page.

Another notable design change resulted from feedback to the statement: "Every page feels optimised to present its information in a visually pleasing manner". 4/5 strongly agreed with the statement, but one user noted, in the case of the video games home page, a number of the horizontal scroller elements could be placed side-by-side instead of in a list. The company and console sliders in particular did not need to occupy the full page width. Half-width scrollers were implemented and also applied to the genre and franchise viewers. They were also adopted on the website's main home page, making the recent activity feed and famous birthdays scrollers each take up half of the page width. In order to uphold responsive design principles, however, these layouts were made to adapt to smaller devices by occupying the full width of the page when the viewport is 675 pixels or narrower. These mixtures of full-width and half-width scrollers allow pages to feel much more space-efficient and visually engaging, making them a worthwhile addition. The improved video game home page is given in Appendix S.

5.3 Recommender System Testing and Performance Analysis

The first metric through which the recommender system was tested was content space coverage. A script was written to generate recommendations for 100 random users. The process of calculating user-user similarities, performing collaborative filtering and then passing results into content-based filtering was identical to how a user would use the system on the front-end. Each new film encountered was recorded in a list - after all recommendations were made, this list was compared to all film records in the database, providing a percentage of how many items can be suggested by the recommender.

Run	Films Encountered / Total Films	Content Space Coverage (%)
1	6303 / 9267	68.01
2	6342 / 9267	68.43
3	6499 / 9267	70.13

Table 1 - Results of the film recommender's content space coverage tests

After three runs, the average of 68.9% was a result slightly lower than the 71% achieved by Salter and Antonopoulos (2006). A value between 65-75% is expected as the dataset used, the 2018 MovieLens

Small dataset (100,000 ratings of 9,000 films by 600 users) (GroupLens, 2018), has a slight bias towards mainstream films that are considered popular, and the implementation of the recommender favours highly-rated films. It is expected that roughly a quarter to a third of the films present do not have high enough average ratings to be present in the collaborative filtering results for most users (GroupLens, 2018). Furthermore, a known limitation of the recommender's hybridisation technique is that the suggestions from the content-based filtering are limited to those films that have already been returned by the collaborative filtering first, reducing the overall number of films that can be recommended (Burke, 2007). The difference between the 68.9% and the 71% achieved by Salter and Antonopoulos (2006) could also be due to differences in trends in the datasets (based on the source of the data, the potential variation in the types of users involved in providing ratings and also the 12-year gap between the paper's dataset and the 2018 MovieLens Small dataset).

However, this value of content space coverage does not provide the full picture of the quality of the recommender system. The films themselves that are being recommended by the system are an empirical, but ultimately crucial, measure of success. 5 users were first asked to create new accounts, rate 15 films and judge the quality of the recommendations. This was repeated after having rated 25 and then 50 films. A statement, "The system has primarily generated suitable recommendations", was asked. If the user was recommended a film they had already seen, they were asked to compare the predicting rating to their own real rating of that item, if the rating was within +/- 1 of their own score, the prediction was said to be accurate. 3/5 agreed the recommendations were mostly suitable after rating 15 films and 5/5 agreed after rating 25 and 50, demonstrating the system having a more comprehensive and more representative user profile returns more suitable suggestions. Rating prediction accuracy scores for the users averaged 76% after 10 ratings, 87% after 25 and 88% after 50 - these are very promising scores that suggest the system is able to account for nuanced opinions in determining films to ultimately recommend. However, in order to gain a complete picture of rating prediction accuracy, a much larger set of users would have to analyse for a wider array of scenarios.

6 Evaluation and Critical Analysis

The following critical analysis of the media database website project will be comprised of three sections. Initially, the project's successes and limitations will be assessed against the project requirements set out before implementation began and against the explicit evaluation criteria. The implementation's limitations will then be given, alongside potential strategies to alleviate these shortcomings if more time for the project was provided. Improvements that could be made to the media database website will then be outlined, tying in to a reflection outlining any potential changes that must be considered if the project were to be restarted, continued or attempted by another party.

6.1 Assessment Against Project Requirements and Evaluation Criteria

The project is now to be evaluated against the project's functional requirements. Initially, the front-end requirements outlined that a wide variety of methods to browse the database's five media types (films, television, video games, books and web series) must be implemented to enable the user to find different information and specific resources with ease. These media types and the various mappings to browse them, in addition to all records having their own detail pages (implemented using Django `DetailView` and `UpdateView` instances), have been successfully realised. This can be seen as a major success of the project - it was found in the literature review (in the study of human-computer interaction and during the study of competitor websites) that implementing database tables for people, tags, franchises and other similar mappings are vital to allow the user to feel that they have control over their interaction with the website and that they are always very few clicks/pages away from any other resource. A further functional requirement, that these resources' URLs' identifiers should be slug fields (comprised of titles, names, release dates and dates of birth) as opposed to iterable, numerical IDs, has also been achieved. This is an important consideration when designing a website with a large number of articles as it means scripts cannot be made to crawl over these database records. Furthermore, the requirement that these URLs adhere to RESTful conventions has also been met - this is also important as it means every step of the hierarchy points to a resource or group of resources (and also that the exclusive use of '/', '-' and lower case alphanumeric characters are compatible with all browser and device types). The other miscellaneous front-end pages have also

been successfully implemented, meeting all of their respective functional requirements. These include search results, subsection home pages for each media type, in addition to pages displaying highest-grossing and top-rated media items being made available to users, providing passive recommendations and further ways to browse the media items.

From a visual, user-interaction standpoint, it can be concluded that the implementation has succeeded in all respects when analysed against the project requirements. By creating a base HTML that defines the layout and structure of the navigation bar and the page body's colour and width, in addition to linking the necessary CSS, JavaScript and jQuery plugins, the website's overall style and feel has been unified across every page. In terms of layout elements, the dropdown menus and search text box on the navigation bar, in addition to the horizontal sliders and image carousel on many of the website's pages, have been implemented successfully, fulfilling each of these requirements respectively. Regarding specific features, the forms to provide ratings worked fully as intended - scores can be applied or changed without page refreshes and reviews can be written in a pop-up modal window. The feature to add media items to the user list has been implemented with a +/- toggle button, the logic for which was repurposed to allow users to follow others, demonstrating that the website implementation has stuck closely to the initial requirements.

As set out in the requirements, the hybrid recommender system was developed based on the approach given by Salter and Antonopoulos (2006). The method to calculate users' peers falls closely in line with this paper, the favourite films of whom are used to perform collaborative filtering. These results are then passed into content-based filtering to produce the final recommendations based on their actors, directors, writers and genres. The result was optimised and numerous minor tweaks were made, as described in 4.5.3. While the implementation of this recommender system can be improved in a number of minor ways, as will be discussed in 6.2, it still performs as intended and generates hybrid recommendations to the active user following the algorithm laid out by Salter and Antonopoulos (2006) - this is another primary success of the media database website, fully meeting its requirements.

There are also non-functional requirements to evaluate the media database website against. The first of these was that the agile approach should be adopted during development. This was followed and it can be confidently said that this was the correct approach. Given that the final solution is a collection of numerous, distinct features, it proved to be very valuable to perform the implementation and testing concurrently for each of these. By improving the implementation quality based on the context of the website's other features and based on its own results, it can be said that this helped to optimise the development workflow - this was particularly true for the collaborative and content-based filtering algorithms for the recommender system as its results informed iterative improvements to the code.

Remaining non-functional requirements included the need for principles of responsive web design to be adopted. These were successfully integrated into the HTML and CSS by applying class names whose layouts dynamically respond to the size of the web browser, thereby supporting a variety of device types and form factors. Several font sizes and image sizes were accounted for, meaning these do not appear too large when loading pages on mobile phone browsers. Container elements that exist to the left or right of a page on the desktop website were made to fall underneath one another in a traditional list, meaning information is not lost to the gutters on either side of the viewport. The final primary non-functional requirement revolved around user data protection and security - as was described before, by using the default key derivation function of PBKDF2, the Django framework is able to securely store users' passwords without the need for any modifications, meaning this non-functional requirement is met.

6.2 Limitations and Proposed Solutions

Improvements that could be implemented if there was additional time to work on the project or if the media database website was restarted, continued or attempted by another party, will now be given. Firstly, sources that provide extensive datasets for all media types should be located and utilised to fully populate the database. While the implementation as it exists does support all media types and all mappings, as set out in the requirements, having larger datasets here would fully-realise the idea that any information can be retrieved on the website. Film (omdb-api, 2021), Book (Kaggle, 2021) and

Video Game (IGDB API, 2021) information are present, but not television or web series. Another potential improvement would be possible once these media items had been retrieved from a dataset or queried from an API - recommender systems could be implemented for each media type, as opposed to just films. The website's database and front-end is versatile enough to support the implementation of these new recommender systems with the existing collaborative and content-based filtering functions.

A primary limitation of the front-end is related to loading times for media content. While page loading times have been optimised by minimising background processing, the sources of the posters, cover images, sliding carousel banners and people's head shot images are only stored as one file size. It has been found that each database model's `save()` method can be overwritten to resize an uploaded image before saving each record to make each faster to load. However, this does not change the fact that multiple different sizes for all types of images would have to be stored and hosted in the AWS S3 bucket. Furthermore, a very large number of small modifications would have to be made to the view functions and HTML to correctly render each image at its appropriate size on a given page. Nonetheless, this remains a notable limitation - multiple sizes for all images should be aimed to be stored in the future, thereby further optimising the load times for pages with a large number of images.

A further limitation comes in the form of the implementation of the recommender system. While it has been optimised by storing the results of peer-peer similarity metrics and the final recommendations in the database, the content-based filtering in particular could be further optimised. The current implementation involves many `for` loops iterating over a very large number of mappings such as Film-Person and Film-Genre records. A similar approach that was adopted for the collaborative filtering using `pandas Dataframe` stores should aim to be implemented as a future improvement. After preprocessing, the converting of the datasets into pivot tables and then into compressed sparse row (CSR) matrices would make the content-based filtering notably faster, as these are highly efficient at slicing rows (representing each film in the database).

The content-based filtering algorithm is also subject to another limitation. The approach currently uses genres, actors, directors and writers as the raw characteristics to find similar films. However, by also finding scores for each film based on their tags and franchises, this would provide an even more comprehensive content-based filtering implementation that accounts for a very wide variety of characteristics and attributes. These extra attributes would broaden the knowledge base the recommender system uses to suggest content to its users.

6.3 Potential Future Improvements and Additional Features

In terms of additional features that could be implemented, it could be argued that the recommender system not being customisable is a further limitation. The ability for a user to choose exclusively collaborative filtering or content-based filtering would provide greater flexibility to fine-tune the system to fit the user's tastes. Allowing the user to manually assign weightings to different attributes would further enhance the system - a user would be able to customise the system to recommend media based on actors and not directors or genres, for example.

While the database website does provide a large number of ways to browse media items (as was deemed crucial in the initial requirements) an additional browsing method could be introduced: 'Most Popular' pages. These would sort media items by metrics such as their weekly page visits or monthly ratings. This would help to assist items that have been recently released that have few ratings or it could be used to promote certain items that have gained temporary spikes in popularity on a notable anniversary of their release date, for example.

Another potential change could be to improve the usability of the contribution forms by adding search menus as opposed to scrolling through dropdown menus. Another method to make contribution more approachable would be to add forms to detail pages for particular items such as films. In the example of a film, buttons, dropdowns and search boxes could be added to allow a user to provide Film-Person, Film-Genre, Film-Genre, Film-Tag etc. mappings for that particular film. A final additional feature to help improve contribution would be to introduce a system of moderation: a user contributing a piece of media or updating an existing item could be sent to an buffer where requested edits wait to be approved by moderators. Tiers to moderation privileges could be implemented, where the top tier can

access the back-end via the Django Administrator website, and a regular moderation user group would allow users to approve certain contributions and edits. Users could be enticed to assist in maintaining the database through the introduction of a rewards scheme that keeps a tally of the number of contributions, introducing certain new privileges or customisation options for their profile page.

Beyond the potential additional features outlined above, there are several longer-term potential improvements to be considered. Implementing a direct-messaging feature would allow users who follow one another to message and discuss a particular media item or review, greatly increasing the website's ability to accommodate social interactions. This would also serve to maximise user retention by providing more features to entice users to regularly check the website and interact with other members on a regular basis. A final potential improvement is to create a dedicated mobile application for the website. This would provide an experience specifically tailored to mobile operating systems that would use the device's native text areas and input fields to create an efficient, seamless client that could also potentially cache data to work offline. While the website has successfully implemented HTML and CSS that make use of responsive web design principles to work on mobile devices within the browser, this would not match the experience on a custom-made, downloadable mobile app, making this an important potential consideration if the project was to undergo further development.

7 Conclusion

Taken together, it can be confidently said the development of the media database website has been a success. Users are able to view films, television series, video games, books and web series through various people (actors, directors, writers, producers, authors), companies (producers, developers, publishers), genres, tags, franchises and consoles. Users can follow one another to curate an activity feed, contribute new items, add items to their tracking list, rate items, review items and add items to customisable profile sections. A film recommender system has also been successfully designed and implemented. Using visually engaging layouts, key information is displayed succinctly, a large number of ways to browse are accommodated, customisation options are present and interactive features grant the user freedom with how they wish to engage with the website. It was found that focusing on these principles and features are what determine the success of a database website.

By considering human-computer interaction to appreciate what users hope to achieve on particular pages, combined with an analogous colour scheme for accenting, the layouts and designs have been optimised to present key facts in logical ways, while also supporting very visual elements. All of the ways to browse different media items outlined in the project requirements have been implemented, in addition to a variety of extra page types. The database is structurally secure, supporting deletion protections to ensure records cannot become dependent on non-existent data. It also consists of many associative entity relations that allow additional fields to describe many-many relationships. In total, 95 database tables have been created to realise the database (increased from the original 76 set out in the requirements due to additions such as custom profile sections and storing recommendation results in the database). Structured in a carefully designed hierarchy, resources' URLs fully adhere to RESTful principles, uniquely identified by slug fields. 22 function-based views and 58 class-based views display pages that are either stand-alone or linked to specific database records. The Django template language programmatically informs front-end blocks and containers, allowing the website to render context passed from the back-end. The functionality of the template language is extended through custom template tags, allowing dynamic context variables to be referenced in the HTML. 55 unique HTML templates describe layouts of different page types. 23 JavaScript/jQuery functions handle features such as making ratings, writing reviews, adding items to tracking lists, creating profile sections and following other users. Many of these handle POST requests using AJAX, preventing page refreshes, enhancing the front-end user experience. The Bootstrap toolkit has been used to provide fonts and container spacings and the 'RateYo' and 'Sortable' jQuery plugins provide star-rating input forms and drag-and-drop features, respectively. The recommender system follows the approach given in Salter and Antonopoulos (2006), passing collaborative filtering into content-based filtering, recommending films using opinions of like-minded individuals and also the items' raw characteristics. Despite several improvements that should aim to be implemented, the successful creation and optimisation of the recommender system can be said to be a primary success of the media database website project, given its importance in modern web design as a tool to maximise user retention and engagement.

References

Adomavicius, G. and Tuzhilin, A., 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. IEEE transactions on knowledge and data engineering, 17(6), pp.734-749,
DOI: https://doi.org/10.1109/TKDE.2005.99

Bootstrap, 2021. Bootstrap 4.1 Toolkit Documentation, [online]. Available at: <https://getbootstrap.com/docs/4.1/getting-started/introduction/>
[Accessed 02/02/21]

Buchanan, M. and Randow, K.V., 2021. Letterboxd: Social Film Discovery (2021)
[online] <https://letterboxd.com> [Accessed 21/03/21].

Burke, R., 2007. Hybrid web recommender systems. The Adaptive Web, Springer, pp. 377-408,
DOI: https://doi.org/10.1007/978-3-540-72079-9_12

Chandler, E.K. and Chandler, O., 2021. Goodreads: Meet your next favorite book.
[online] Available at: <https://www.goodreads.com/> [Accessed 21/03/21].

Chen, H.W., Wu, Y.L., Hor, M.K. and Tang, C.Y., 2017. Fully content-based movie recommender system with feature extraction using neural network. International conference on machine learning and cybernetics (ICMLC), 2, pp. 504-509. DOI: https://doi.org/10.1109/ICMLC.2017.8108968

Debnath, S., Ganguly, N. and Mitra, P., 2008. Feature weighting in content based recommendation systems using social network analysis. Proceedings of the 17th international conference on World Wide Web, pp. 1041-1042.
DOI: https://doi.org/10.1145/1367497.1367646

Django, 2021. Django Software Foundation - Django Framework v3.2 Documentation
[online] Available at: <https://docs.djangoproject.com/en/3.2/> [Accessed 07/04/21]

GroupLens, 2018. MovieLens Small Dataset, 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users. [online] Available at: <https://grouplens.org/datasets/movielens/> [Accessed 20/01/21]

Hindman, T., 2015. Color theory & web design: Choosing a color scheme for your website. TBH Creative.
[online] Available at: <https://blog.tbhcreative.com/2015/12/choose-color-scheme-website.html>
[Accessed 15/04/21]

IGDB, 2021. The Internet Games Database
[online] Available at: <https://www.igdb.com> [Accessed 21/03/20].

IGDB API, 2021. The Internet Games Database API - Documentation
[online] Available at: <https://api-docs.igdb.com/#about> [Accessed 28/03/20]

IMDb, 2021. The Internet Movie Database, IMDb: Ratings, Reviews, and Where to Watch the Best Movies & TV.
[online] Available at: <https://www.imdb.com/> [Accessed 21/03/21].

Jain, Y. 2019. Redesigning Wikipedia to be more productive and useful. Redesigning UI and UX. Medium - Blogs
[online] Available at: <https://medium.com/@uiuxyash/redesigning-wikipedia-to-be-more-productive-and-useful-redesign-ui-ux-case-study-89f8a7173b0> [Accessed 04/03/21]

Kaggle, 2021. Goodreads Books Dataset - 31 Features (Collected by user: Austin Reese) (original source: Goodreads)
[online] Available at: <https://www.kaggle.com/austinreese/goodreads-books> [Accessed 05/04/21]

Lee, C.C., Nagpal, P., Ruane, S.G. and Lim, H.S., 2018. Factors affecting online streaming subscriptions. Communications of the IIMA, 16(1), p.2, [online] Available at:
<https://scholarworks.lib.csusb.edu/ciima/vol16/iss1/2> [Accessed 04/03/21]

Leskovec, J., Rajaraman, A. and Ullman, J.D., 2016. Lecture 43 - Collaborative Filtering. Video (13 April 2016). Stanford University - Mining of Massive Datasets - Recommender Systems, YouTube Channel: Artificial

Intelligence - All in One. [online] Available at: <https://www.youtube.com/watch?v=h9gpufJFF-0> [Accessed 01/03/21]

O'Connell, T.A., Murphy, E.D., 2007. The Usability Engineering Behind User-Centered Processes for Web Site Development Lifecycles, Human Computer Interface Research in Web Design and Evaluation, pp. 1-21. DOI: https://doi.org/10.4018/978-1-59904-246-6.ch001

omdb-api, 2021. The Open Movie Database RESTful Web Service (original source: IMDb) [online] Available at: <https://www.omdbapi.com/> [Accessed 20/01/21]

Pal, A., Parhi, P. and Aggarwal, M., 2017. An improved content based collaborative filtering algorithm for movie recommendations. In 2017 tenth international conference on contemporary computing (IC3), pp. 1-3, DOI: https://doi.org/10.1109/IC3.2017.8284357

RateYo!, 2021. A jQuery Star Rating Plugin [online] Available at: <https://rateyo.fundoocode.ninja/> [Accessed 28/01/21]

Salter, J. and Antonopoulos, N., 2006. CinemaScreen recommender agent: combining collaborative and content-based filtering. IEEE Intelligent Systems, 21(1), pp. 35-41, DOI: https://doi.org/10.1109/mis.2006.4

Smith, B. and Linden, G., 2017. Two decades of recommender systems at Amazon.com. IEEE Internet Computing, 21, pp. 12-18. DOI: https://doi.org/10.1109/MIC.2017.72

Son, J. and Kim, S.B., 2017. Content-based filtering for recommendation systems using multiattribute networks. Expert Systems with Applications, 89(33), pp. 404-412, DOI: https://doi.org/10.1016/j.eswa.2017.08.008

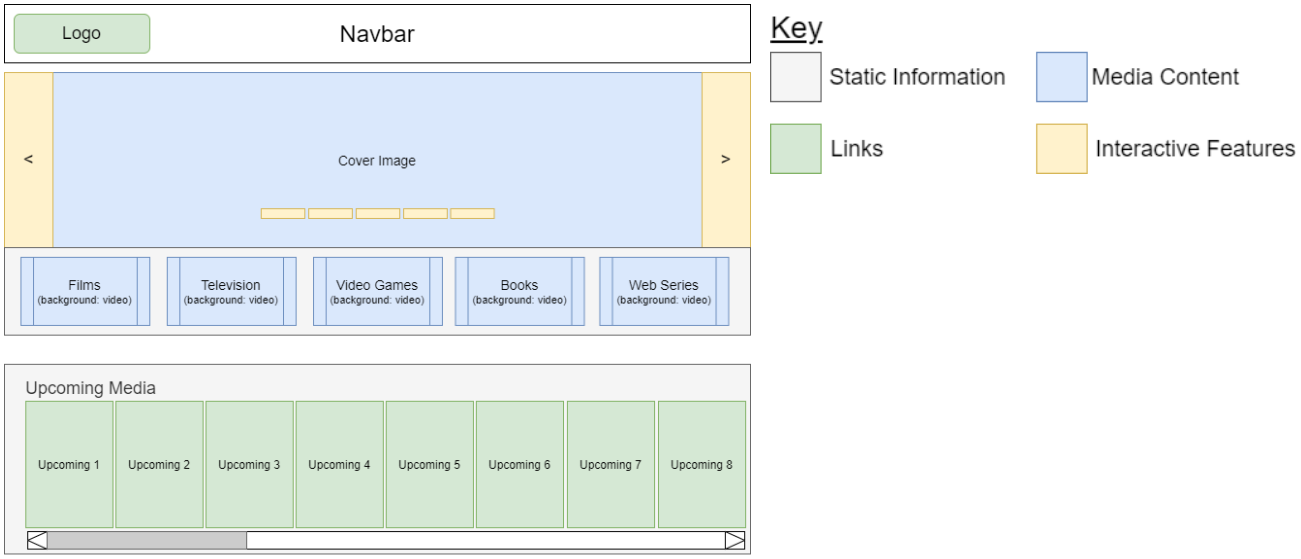
Sortable, 2021. jQuery UI Plugin [online] Available at: <https://jqueryui.com/sortable/> [Accessed 23/02/21]

Stone, M., 2006. Choosing colors for data visualization. Perceptual Edge. Business Intelligence Network, 2. [online] Available at: <https://www.perceptualedge.com/articles/b-eye/choosing_colors.pdf> [Accessed 15/04/21]

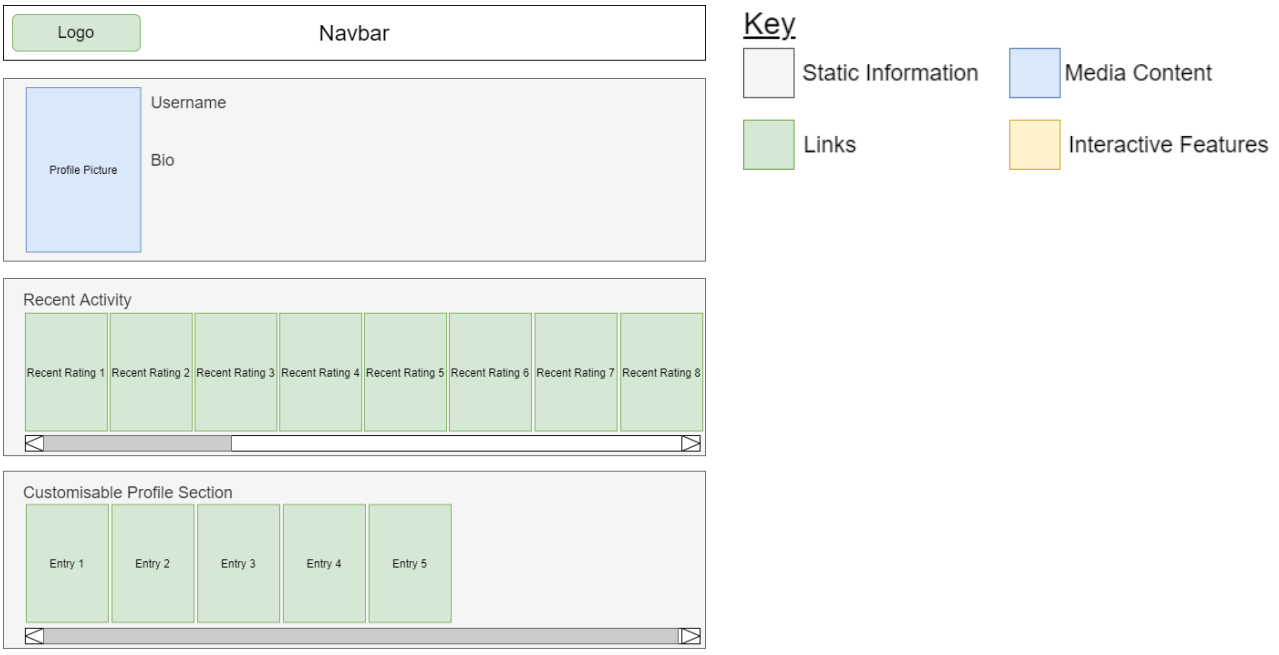
Wayne, M.L., 2018. Netflix, Amazon, and branded television content in subscription video on-demand portals. Media, Culture & Society, 40(5), pp.725-741, DOI: https://doi.org/10.1177/0163443717736118

Appendices

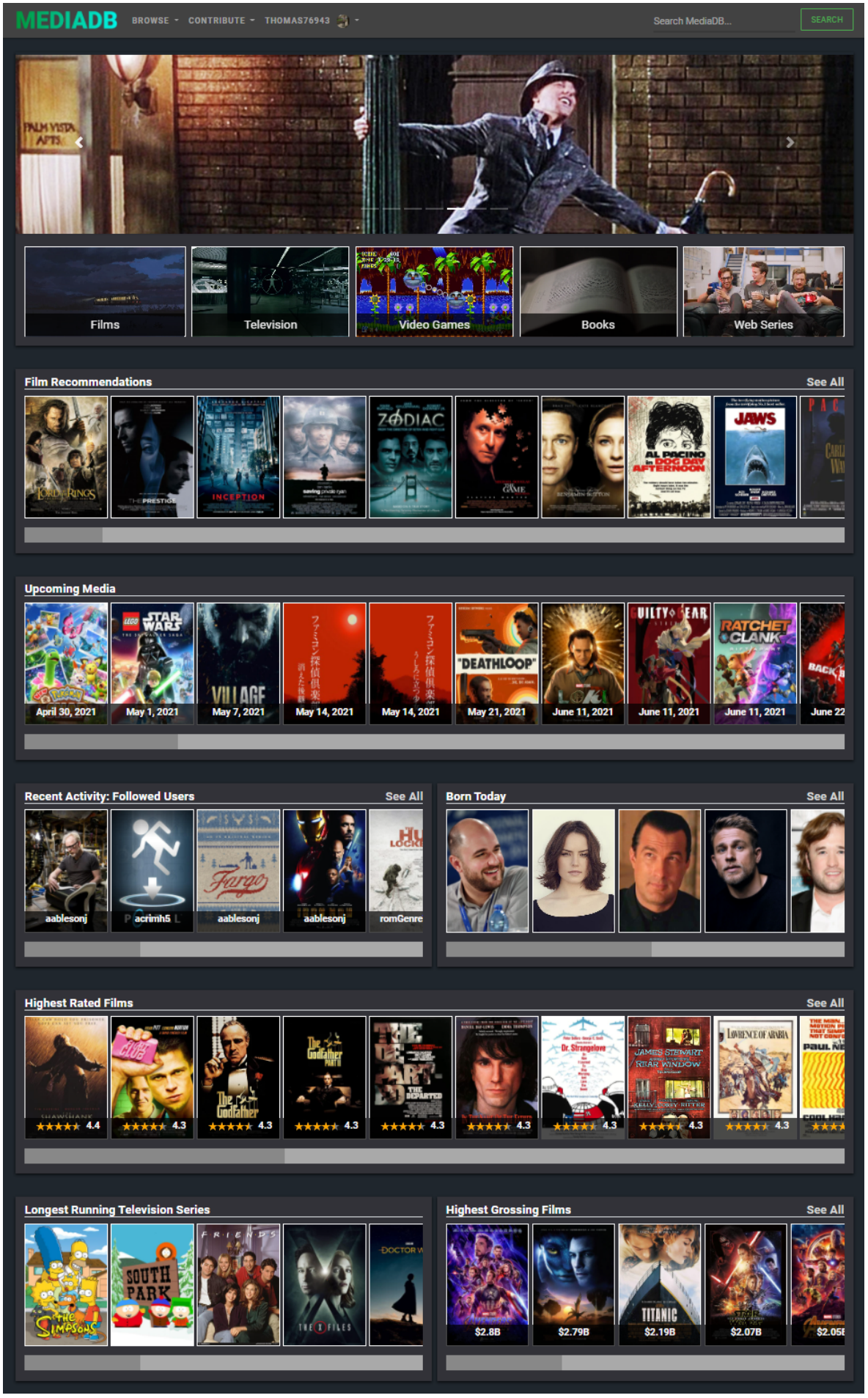
A - Initial Design: Layout of Home Page



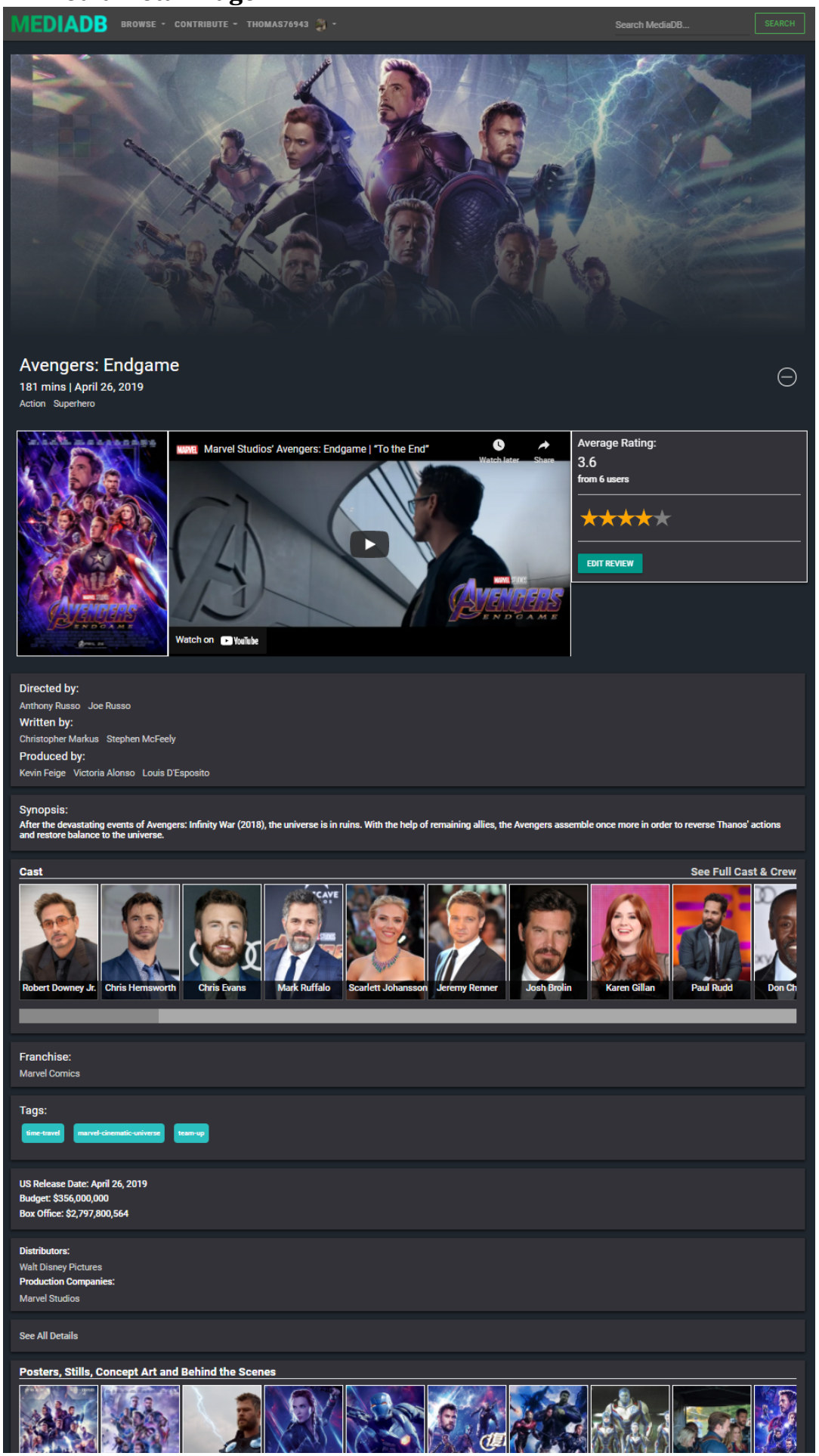
B - Initial Design: Layout of Profile Page



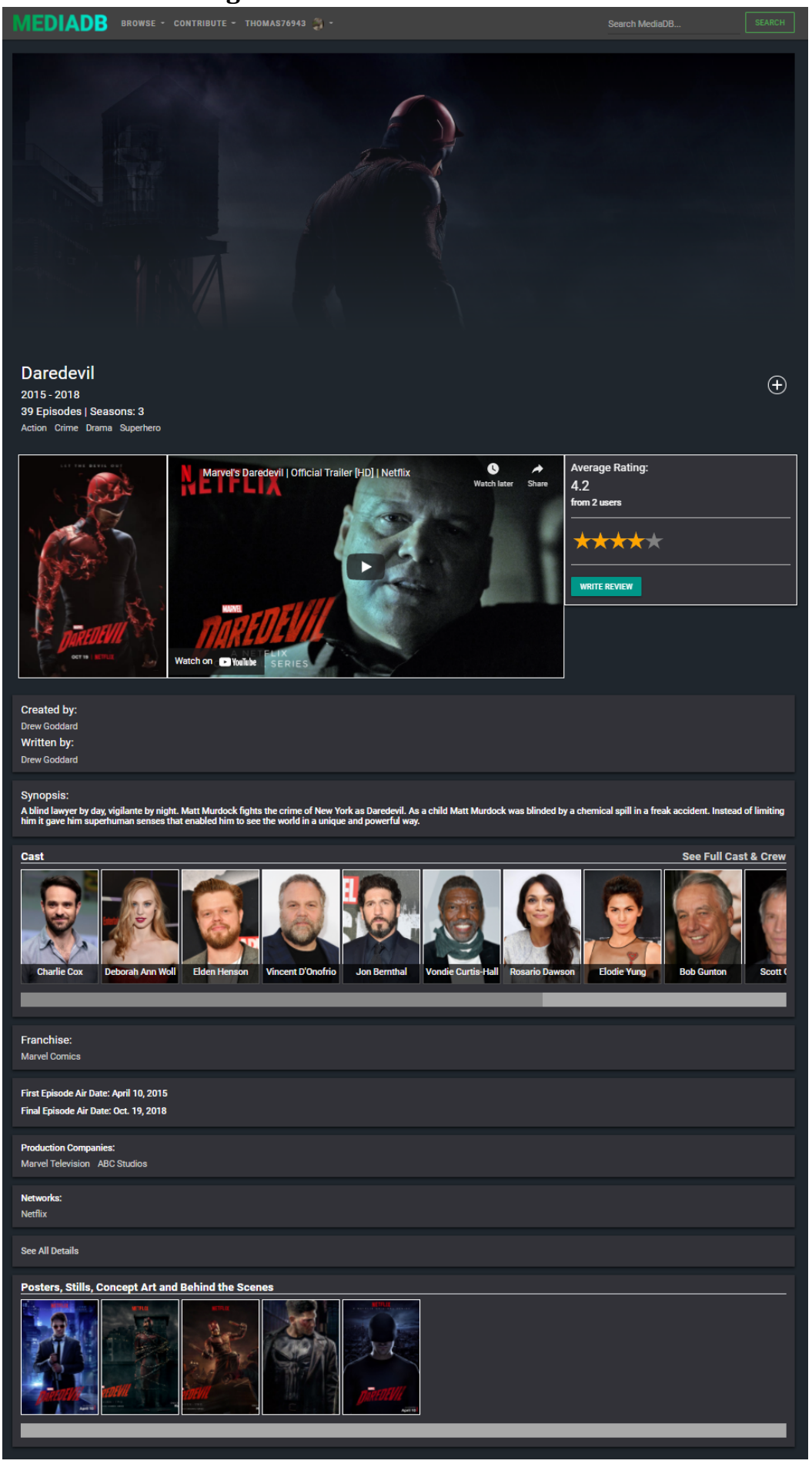
C - Home Page


MEDIADB
BROWSE
CONTRIBUTE
THOMAS76943
Search MediaDB...
SEARCH
Films
Television
Video Games
Books
Web Series
Film Recommendations
See All
Upcoming Media
April 30, 2021
May 1, 2021
May 7, 2021
May 14, 2021
May 14, 2021
May 21, 2021
June 11, 2021
June 11, 2021
June 11, 2021
Recent Activity: Followed Users
See All
aablesonj
acrimh5
aablesonj
aablesonj
romGenre
Born Today
See All
Highest Rated Films
See All
4.4
4.3
4.3
4.3
4.3
4.3
4.3
4.3
4.3
Longest Running Television Series
Highest Grossing Films
See All
$2.8B
$2.79B
$2.19B
$2.07B

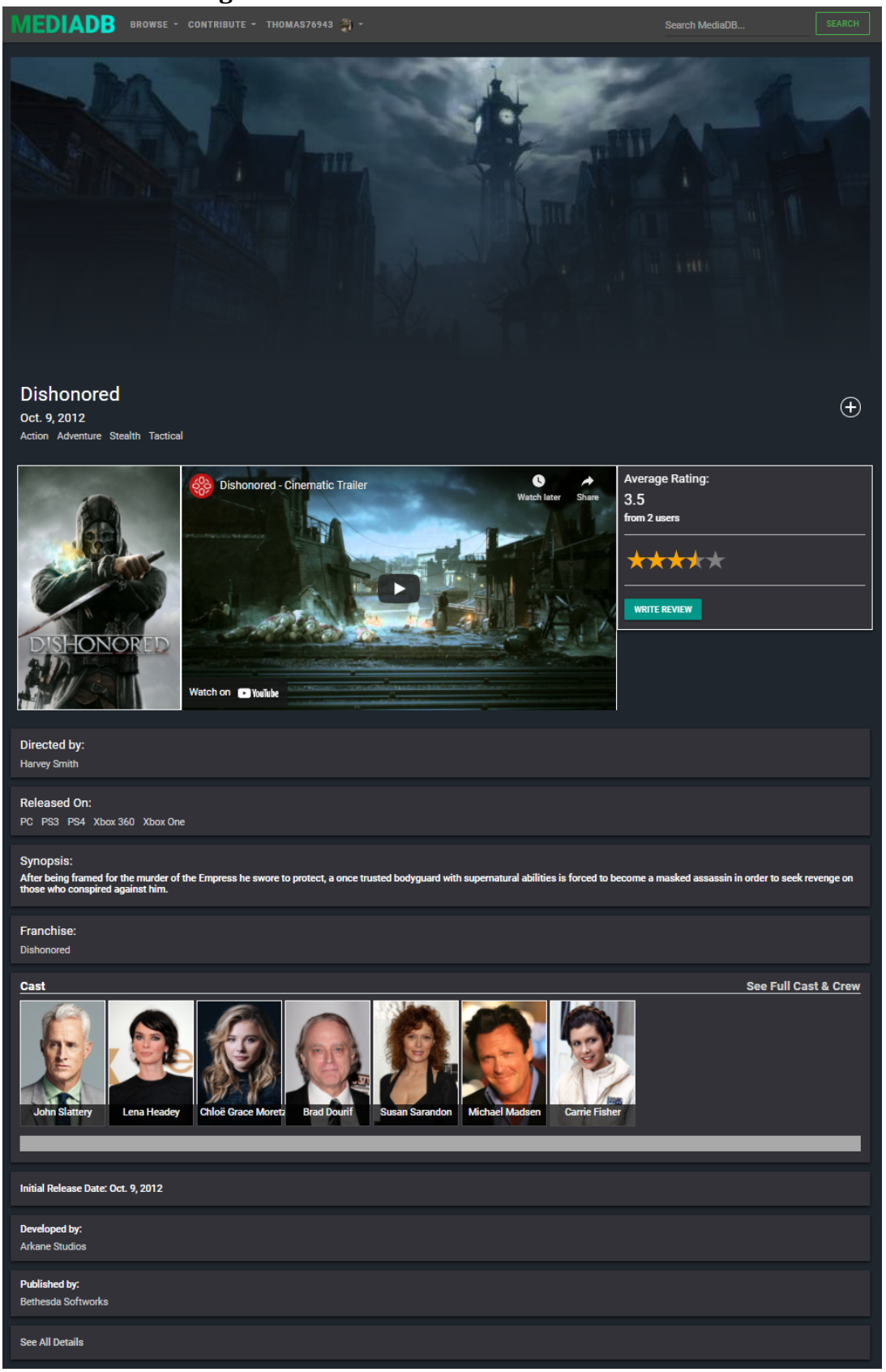
D - Media Detail Page: Film



E - Media Detail Page: Television Series



F - Media Detail Page: Video Game



G - JavaScript to perform AJAX POST requests to send a rating value to the database

```
$('#rate').rateYo({
    {% if userRating %} rating:'{{userRating}}', {% else %} rating:'0', {% endif %}
    minValue:0, maxValue:10, halfStar:true,
    normalFill:'grey', ratedFill:'orange'
/*If the rating is changed, make an AJAX POST request*/
}).on("rateyo.set", function (e, data) {
    var newRating = data.rating
    $.ajax({
        url:document.URL,
        type: "POST",
        data: { 'type': "rating", 'nr': newRating, 'csrfmiddlewaretoken': csrftoken },
        success: setTimeout(function(){
            /*Refresh the 5-star rating input, average score text and the review button*/
            $('#reviewButtonContainer').load(' #reviewButtonContainer', function (){$(this).children().unwrap()})
            $('#refreshAverageNoneText').load(' #refreshAverageNoneText', function (){$(this).children().unwrap()})
            $('#refreshAverageText').load(' #refreshAverageText', function (){$(this).children().unwrap()})
            $('#refreshCountText').load(' #refreshCountText', function (){$(this).children().unwrap()})
        }, 1000),
    });
```

H - The class-based view for each Film instance (using an `UpdateView` to be able to override its `post()` method to call `mediaPagePostRequests()`)

```
class FilmDetailView(generic.UpdateView):
    model = Film
    template_name = 'media/filmDetail.html'
    slug_field, slug_url_kwarg = "slug", "slug"
    fields = []

    def post(self, request, *args, **kwargs):
        mediaPagePostRequests(self, request, *args, **kwargs)
        return super(FilmDetailView, self).post(request, *args, **kwargs)

    def get_context_data(self, **kwargs):...
```

I - Excerpt from the `mediaPagePostRequests()` handler function that interprets the POST request of a new rating or an updated rating for any type of media

```
#POST Request: Adding/Changing a Rating
if request.POST['type'] == "rating":
    newRating = float(request.POST['nr'])
    if isinstance(object, Film):
        r = FilmRating.objects.filter(user=self.request.user, film=object.id).first()
        if r is not None:                           #Updating an existing rating
            r.rating = newRating
            r.dateTime = datetime.datetime.now()
            r.save()
        else:                                       #Creating a new rating
            FilmRating(user=self.request.user, film=object, rating=newRating).save()

    elif isinstance(object, Television):...
    elif isinstance(object, VideoGame):...
    elif isinstance(object, Book):...
    elif isinstance(object, WebSeries):...
```

J - JavaScript to perform AJAX POST requests to send a review data to the database

```
/*Retrieve the review form*/
var frm = $('#writeReview')
frm.submit(function(e) {
    e.preventDefault();     /*Prevent the page from reloading*/
    $.ajax({                /*Make the AJAX POST request*/
        url:document.URL,
        type : "POST",
        data : { 'type': "review", 'newReview': document.querySelector('#reviewTextArea').value, 'csrfmiddlewaretoken' : '{{ csrf_token }}', },
        success: setTimeout(function(){
            /*Close the modal window containing the review text box*/
            var modal = document.getElementById("writeReviewModal");
            modal.style.display = "none";
            /* Refresh only the two buttons and the text area*/
            $('#writeReviewButton').load(' #writeReviewButton', function (){$(this).children().unwrap()})
            $('#reviewTextArea').load(' #reviewTextArea', function (){$(this).children().unwrap()})
            $('#submitReview').load(' #submitReview', function (){$(this).children().unwrap()})
        }, 1000),
    })
    return false;
```

K - Excerpt from the `mediaPagePostRequests()` handler function that interprets the POST request of a new review or an updated review for any type of media

```
newReview = request.POST['newReview']
print(newReview)
if isinstance(object, Film):
    r = FilmRating.objects.filter(user=self.request.user, film=object.id).first()
elif isinstance(object, Television):
    r = TelevisionRating.objects.filter(user=self.request.user, television=object.id).first()
elif isinstance(object, VideoGame):
    r = VideoGameRating.objects.filter(user=self.request.user, videoGame=object.id).first()
elif isinstance(object, Book):
    r = BookRating.objects.filter(user=self.request.user, book=object.id).first()
elif isinstance(object, WebSeries):
    r = WebSeriesRating.objects.filter(user=self.request.user, webSeries=object.id).first()

r.review = newReview
r.dateTime = datetime.datetime.now()
r.save()
```

L - Excerpt from the `get_context_data()` method of the `DetailView` that renders a particular Franchise page

```
if subcategories != None:
    for x in range(subcategories.count()):
        SubCat_F = FilmFranchiseSubcategoryMapping.objects.filter(franchiseSubcategory__parentFranchise=self.object.id).filter(franchiseSubcategory__title=subcategories[x].title)
        SubCat_TV = TelevisionFranchiseSubcategoryMapping.objects.filter(franchiseSubcategory__parentFranchise=self.object.id).filter(franchiseSubcategory__title=subcategories[x].title)
        SubCat_VG = VideoGameFranchiseSubcategoryMapping.objects.filter(franchiseSubcategory__parentFranchise=self.object.id).filter(franchiseSubcategory__title=subcategories[x].title)
        SubCat_B = BookFranchiseSubcategoryMapping.objects.filter(franchiseSubcategory__parentFranchise=self.object.id).filter(franchiseSubcategory__title=subcategories[x].title)
        SubCat_WS = WebSeriesFranchiseSubcategoryMapping.objects.filter(franchiseSubcategory__parentFranchise=self.object.id).filter(franchiseSubcategory__title=subcategories[x].title)

        #Collect and sort all the media for that subcategory based on the 'orderInFranchise' field
        completeSubCategory = sorted(list(chain(SubCat_F, SubCat_TV, SubCat_VG, SubCat_B, SubCat_WS)), key=attrgetter('orderInFranchise'))
        #Dynamic context name for the subcategory in the template
        context[subcategories[x].title] = completeSubCategory
```

M - Excerpt from franchiseDetail.html that uses a custom template tag to render out each subcategory and each of their constituent media items

```
{% load media_tags %}
{% if subcategories %}
    {% for subcategory in subcategories %}
        <div class="container infoSection">
            <h5 class="font-weight-bold border-bottom mb-0">{{ subcategory.title }}</h5>
            <div class="scroller">
                <ul class="hs full">
                    {% with var=subcategory.title %}            <!-- Temporary variable var
                    {% dynamicVariableValue var as var_url %}   <!-- Apply custom tag
                    {% for entry in var_url %}                  <!-- Iterate through subcategory
                        {%  if entry.film != NULL ... %}        <!-- Render if entry is a film
                        {%  if entry.television != NULL ... %}  <!-- Render if entry is a tv series
                        {%  if entry.videoGame != NULL ... %}   <!-- Render if entry is a video game
                        {%  if entry.book != NULL ... %}        <!-- Render if entry is a book
                        {%  if entry.webseries != NULL ... %}   <!-- Render if entry is a web series
                    {% endfor %}
                    {% endwith %}
                </ul>
            </div>
        </div>
    {% endfor %}
{% endif %}
```

N - JavaScript and AJAX POST request to handle a piece of media being added to a list or a user following another user

```
$('#plus-toggle').click(function() {
    var toggle;
    {% if inList == True %} toggle = "remove"
    {% else %} toggle = "add" {% endif %}
    $.ajax({
        url:document.URL,
        type: "POST",
        data: { 'type': "listToggle", 'toggle': toggle, 'csrfmiddlewaretoken': csrftoken },
        success: setTimeout(function(){
            $('#addListToggle').load(' #addListToggle', function (){$(this).children().unwrap()})
        }, 500),
    });
});
```

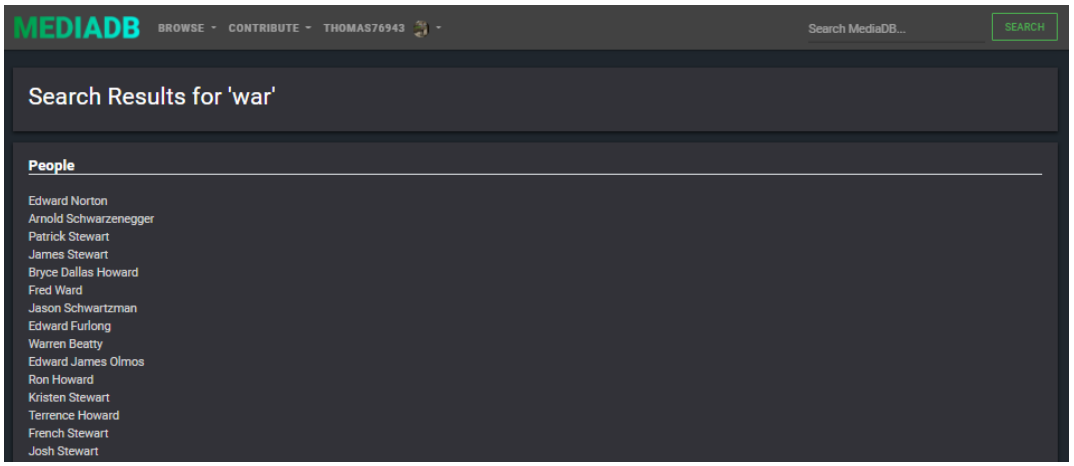
O - Browser Compatibility Testing

Test:	Google Chrome v 89.0.4389.128	Microsoft Edge v 89.0.774.76	Mozilla Firefox v 87.0
Loading home page	✓	✓	✓
Registering a new user	✓	✓	✓
Logging in	✓	✓	✓
Rating + reviewing a film	✓	✓	✓
Rating + reviewing a tv series	✓	✓	✓
Rating + reviewing a video game	✓	✓	✓
Rating + reviewing a book	✓	✓	✓
Rating + reviewing a web series	✓	✓	✓
Adding/removing a film from user list	✓	✓	✓
Adding/removing a tv series from user list	✓	✓	✓
Adding/removing a video game from user list	✓	✓	✓
Adding/removing a book from user list	✓	✓	✓
Adding/removing a web series from user list	✓	✓	✓
Contributing a new film	✓	✓	✓
Contributing genre + tag mappings	✓	✓	✓
Contributing people + company mappings	✓	✓	✓
Creating custom profile sections	✓	✓	✓
Adding/removing profile section items	✓	✓	✓
Reordering profile section items	✓	✓	✓
Reordering profile sections	✓	✓	✓
Removing profile sections	✓	✓	✓
Interactive / Dynamic / Animated Features			
Horizontal sliders and carousel viewers	✓	✓	✓
Navigation bar + mega menus	✓	✓	✓
Media browsing grid viewers	✓	✓	✓
Star rating forms and review modal windows	✓	✓	✓
+/- toggle buttons	✓	✓	✓

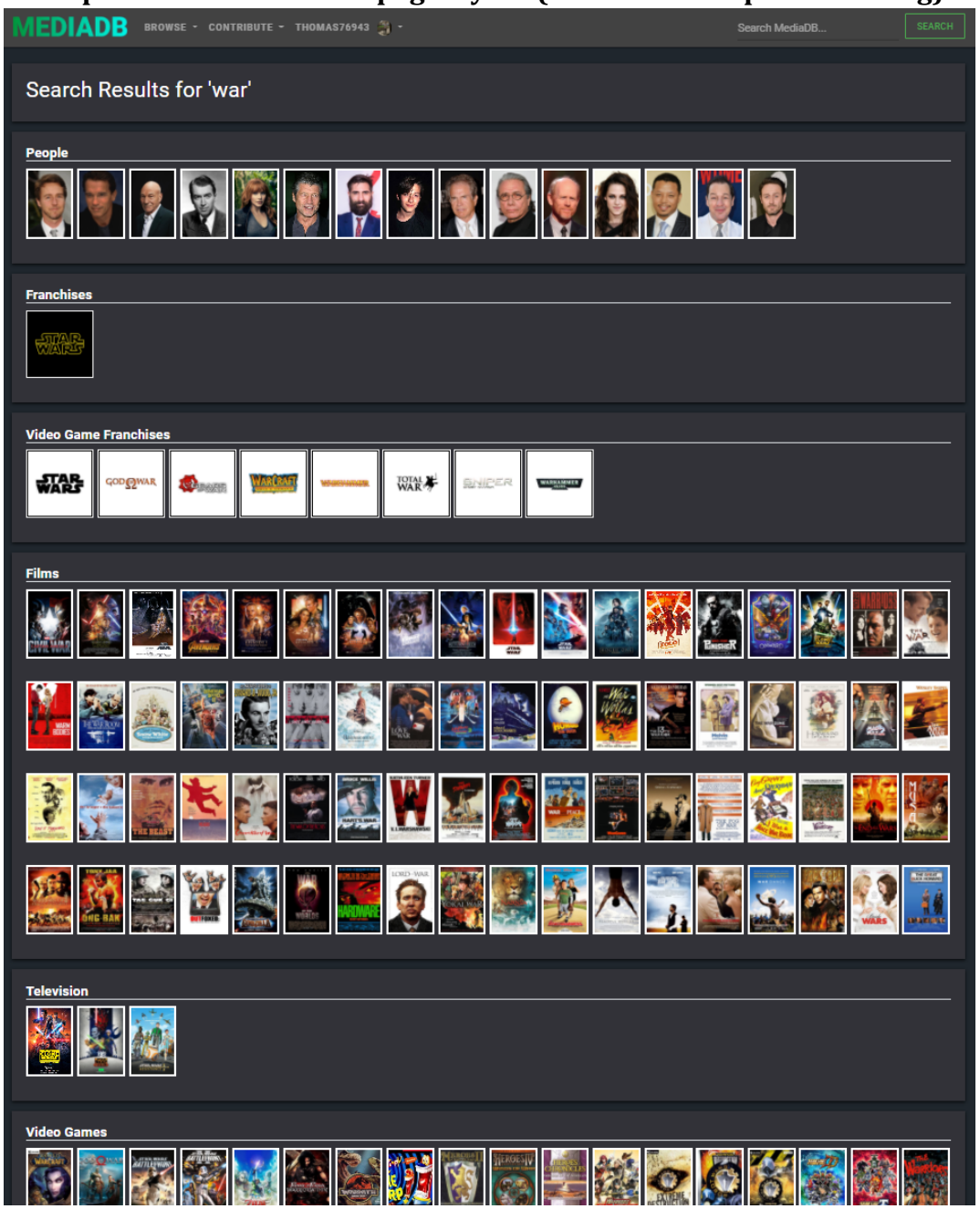
P - Device Compatibility Testing

Test:	iPhone 5/SE (small smartphone)	Google Pixel 2 XL (large smartphone)	iPad (tablet)
Loading home page	✓	✓	✓
Registering a new user	✓	✓	✓
Logging in	✓	✓	✓
Rating + reviewing a film	✓	✓	✓
Rating + reviewing a tv series	✓	✓	✓
Rating + reviewing a video game	✓	✓	✓
Rating + reviewing a book	✓	✓	✓
Rating + reviewing a web series	✓	✓	✓
Adding/removing a film from user list	✓	✓	✓
Adding/removing a tv series from user list	✓	✓	✓
Adding/removing a video game from user list	✓	✓	✓
Adding/removing a book from user list	✓	✓	✓
Adding/removing a web series from user list	✓	✓	✓
Contributing a new film	✓	✓	✓
Contributing genre + tag mappings	✓	✓	✓
Contributing people + company mappings	✓	✓	✓
Creating custom profile sections	✓	✓	✓
Adding/removing profile section items	✓	✓	✓
Reordering profile section items	✓	✓	✓
Reordering profile sections	✓	✓	✓
Removing profile sections	✓	✓	✓
Interactive / Dynamic / Animated Features			
Horizontal sliders and carousel viewers	✓	✓	✓
Navigation bar + mega menus	✓	✓	X
Media browsing grid viewers	✓	✓	✓
Star rating forms and review modal windows	✓	✓	✓
+/- toggle buttons	✓	✓	✓

Q - Original search results page layout (before user acceptance testing)



R - Improved search results page layout (after user acceptance testing)



S - Improved video games subsection home page with a mixture of full-width and half-width scrollers following user feedback