Vaultive

User Requirements

Vaultive is an online media-streaming platform offering documentaries, movies and short films. Guests arrive at the welcome page where they can see featured banners and poster thumbnails but cannot interact with them; their only option is to sign up or sign in. Registered users can click any poster to go to a detailed content page where they can watch the trailer, read the description, view reviews, write reviews, and subscribe streaming services in order to watch media content. Users register with a unique email, password, first and last name, a unique nickname, country and status (Student, Normal, Elder). This country selection plays an important role in pricing for subscriptions.

Users may subscribe to a streaming service by selecting from available options. Upon selection they simulate payment and receive a subscription confirmation that records the payment method, price, start date and end date. The system applies discounts based on the user's status and the country in which they are registered. Active subscriptions grant access to streaming content and enable users to write reviews for content they have watched. Once a subscription expires playback is blocked until the user renews.

Whenever a user plays a media content for the first time, Vaultive records a Watch history entry with the watch date and remaining minutes. Watch history gets updated whenever a user resumes the media content. Users may submit a text-only review for any title they have started watching, regardless of how many minutes they viewed. They may update or delete their own reviews at any time they choose.

The back-end employees (administrators) have all the user capabilities and can also manage the platform data. They can add, update or remove any media content record, including its type, description, poster, trailer link and subtitle or audio options. They can create, modify or delete any subscription plan and simulate subscription transactions. They can add, update or remove user accounts and adjust user status or country. They can edit or delete any user review for moderation purposes. The back-end employees does not have restrictions like failed payment, or any some sort of owning validations when managing a property of user e.g review deletion.

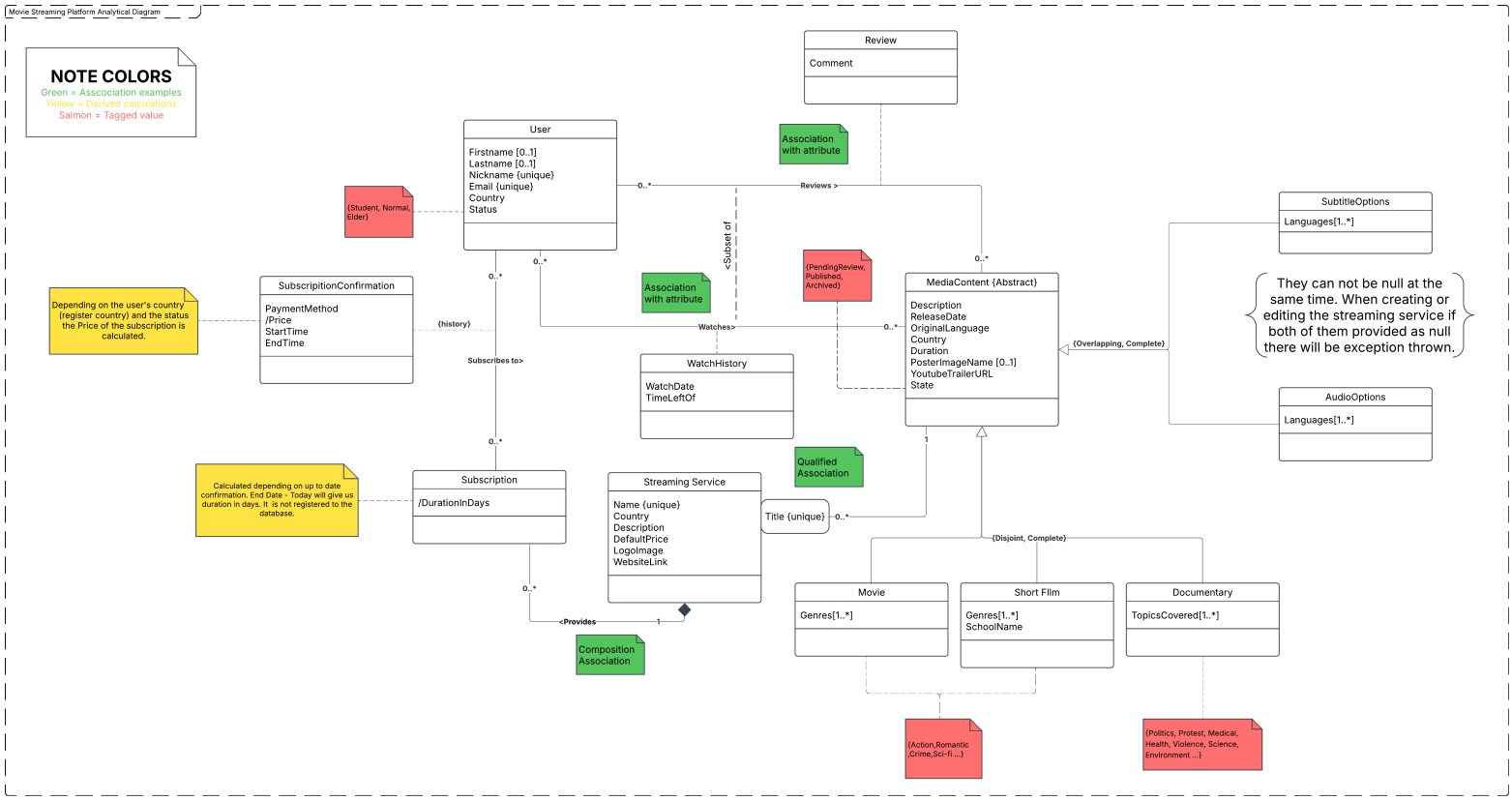

Figure 1: Analytical Class Diagram

Analytical Class Diagram Descriptions

The above analytical class diagram visualizes the Vaultive media-streaming platform's core domain, detailing Users (with statuses Student, Normal, Elder), MediaContent (an abstract parent with Movie, ShortFilm and Documentary subclasses, plus optional SubtitleOptions and AudioOptions), StreamingService, Subscription, SubscriptionConfirmation, WatchHistory and Review, all with their respective multiplicities. Because this is an analytical diagram, it includes many UML constructs that do not exists in modern programming lanuages. For example multi aspect inheritance. It also contains some aspect which not that clear even though they are explained with the notes.

In current diagram we can see User and Subscription classes has a middle class tagged with history showcasing that there is a association with attribute class holding dates for this connection. In the design diagram it becomes more clear, understandable.

In the next section I will transform the analytical class diagram into a design class diagram. UML constructs which modern programming languages does not support will turn into showcase how to actually implement this ideas.

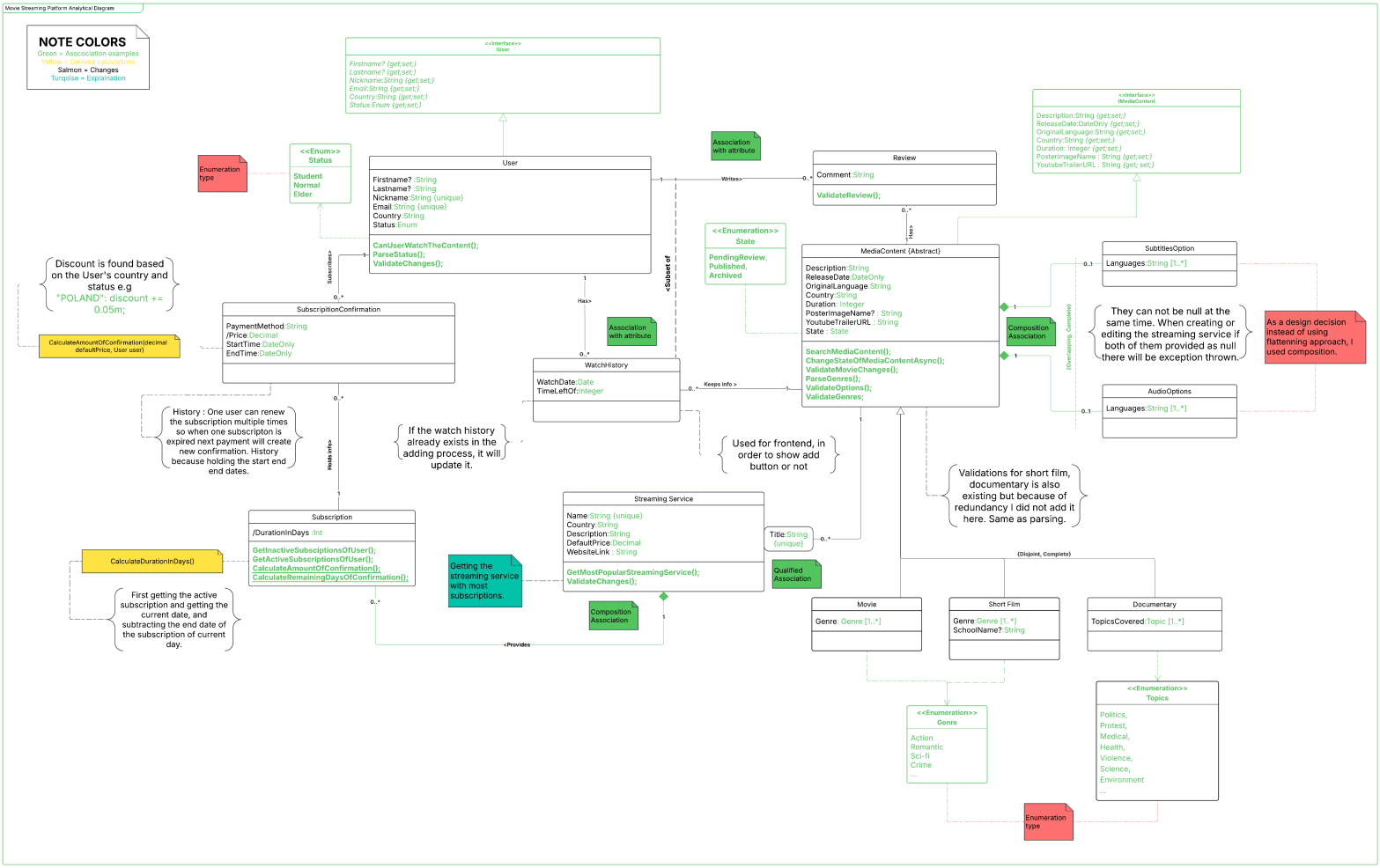

Figure 2: Design Class Diagram

Design Decisions

The above design class diagram illustrates the changes made from the analytical class diagram to design class diagram. The design class diagram has been translated into C# classes with Entity Framework Core handling all persistence. In this diagram we can clearly see that multi aspect inheritance between media content abstract class with the subtypes Movie, Short Film, Documentary with the Subtitle and Audio Options are turned into composition association in options side. Instead of flattening, the composition was chosen here.

We can also see that association with attribute classes now an actual class maintaining the many to many connections. History class now has constraint explaining how the history works; One user may have multiple confirmation for a single subscription.

`SubtitleOptions` and `AudioOptions` are declared as nullable reference properties because they overlap in responsibility and only one or both may be set; they are implemented as composition relationships so that deleting a `MediaContent` entity also deletes its associated options. So options are belonging to media content class. They can not be null at the same time in order to support overlapping inheritance.

Attribute Validations

Required fields are marked with `[.IsRequired()]` in correct configuration files to ensure non-null values, string lengths are bounded via `[.HasMaxLength(50)]` to prevent overlong inputs, and format constraints (for example for email addresses) are validated through custom methods like **IsValidEmail**. These annotations serve two purposes: they provide immediate feedback and error handling at the API model-binding layer, and they inform Entity Framework Core's schema generation so that database columns have the correct nullability and size constraints. In other usage using custom methods, providing customly made messages and printing them to the user where is needed.

Optional Attributes

Optional attributes are attributes that may contain no value. In C#, optional attributes are represented as nullable properties, denoted by a question mark (`?`), indicating they can hold a `null` value. In Entity Framework Core, these optional attributes correspond to database columns defined as nullable. In analytical diagrams, optional attributes are illustrated with a multiplicity range of `[0..1]`, while in design diagrams, this translates into a nullable indicator (`?`) next to the attribute.

We can give example of nullable first name and last name in the user. These are optional attributes which can be null, if user wants to provide it, its welcome.

Multi-value Attributes

While C# natively supports multi-value attributes as classes inheriting from `ICollection`, it is worth noting how this project manages multi-value attributes specifically within Entity Framework Core. In this project, I chose to store multi-value attributes in a single database column. For example, the `Genres` attribute in the `Movie` class, initially defined as an enumeration type, is converted to a comma-separated string and stored within a single column named `Genres` in the `MediaContent` table.

Derived Attributes

Derived attributes utilize that get feature when declaring attributes in c#. In most instances showcased, the derived attributes would have the logics directly created on the getter and would return the final total. Ensuring an automatic creation of the derived attributes based on the other values that create the base of a derived attribute. As an example I can provide the Price attribute in the SubscriptionConfirmation table. This price is calculated depending on the user's registered country and the current status. For instance Elder will provide more discount than the Normal status.

Derived attribute `Subscription.DurationInDays` marked with `[NotMapped]` so that EF Core does not generate database columns for them. Because this will cause to get wrong data if the operation done in data base side. However price attribute in confirmation class is calculated when the payment is done. This attribute could not be [NotMapped] because of the fact that if price was calculating like duration in days, if the the default

price was changed after confirmation the returned price would calculate once again and return not correct data.

Tagged value

Tagged values are now represented as actual enumeration classes, and enumeration handling has been tailored specifically to each attribute. By using enumeration types, each attribute is associated with its own separate enumeration class containing the specific possible values (e.g., `Action`, `SciFi`). Additionally, due to this enumeration usage, the design class diagram explicitly includes these enumeration classes (e.g., `Genres`), transitioning from the previously used note-based tagged-value notation into proper classes directly linked to their respective attributes.

The `Status` enumeration is stored as its string representation in the database, providing readable values instead of numeric codes. The collections for `Genre` and `Topic` are flattened into a single comma-separated string stored within a single column. This approach allows genre values to be persisted and rehydrated efficiently, eliminating the need for additional join tables.

ORM Inheritance Mapping

Inheritance Mapping Strategy: Table-Per-Hierarchy (TPH) I chose the Table-Per-Hierarchy (TPH) inheritance mapping strategy because the subclasses (`Movie`, `ShortFilm`, and `Documentary`) do not have many unique attributes. If the subclasses had numerous distinct attributes, using TPH would lead to many nullable columns, which I aim to avoid.

Since my subclasses are not significantly different, using Table-Per-Concrete-Type (TPC) would not be beneficial, as TPC creates separate database tables for each subclass type, resulting in redundant structures.

Communication Between Front-end and Back-end

The front-end uses web technologies including JavaScript, HTML, and CSS, while the back-end utilizes .NET Entity Framework (EF) and C#. Communication between the two sides occurs via REST APIs. The back-end creates endpoints using controllers derived from `[ControllerBase]`, each having distinct URLs. For example, the endpoint **/api/User/Get/1** retrieves user data for the user with ID `1`. The front-end implements fetch methods corresponding to endpoint types (`GET`, `POST`, `PUT`, `DELETE`) and displays the retrieved data appropriately in the UI.

Base Methods: Add, Update, Remove

The base methods `Add`, `Update`, and `Remove`, shown only in the Swagger API page, have been implemented directly within the backend API. These methods are not included in

the design diagrams to avoid redundancy, but they do exist in the backend and apply to all relevant classes documented in the Swagger interface.

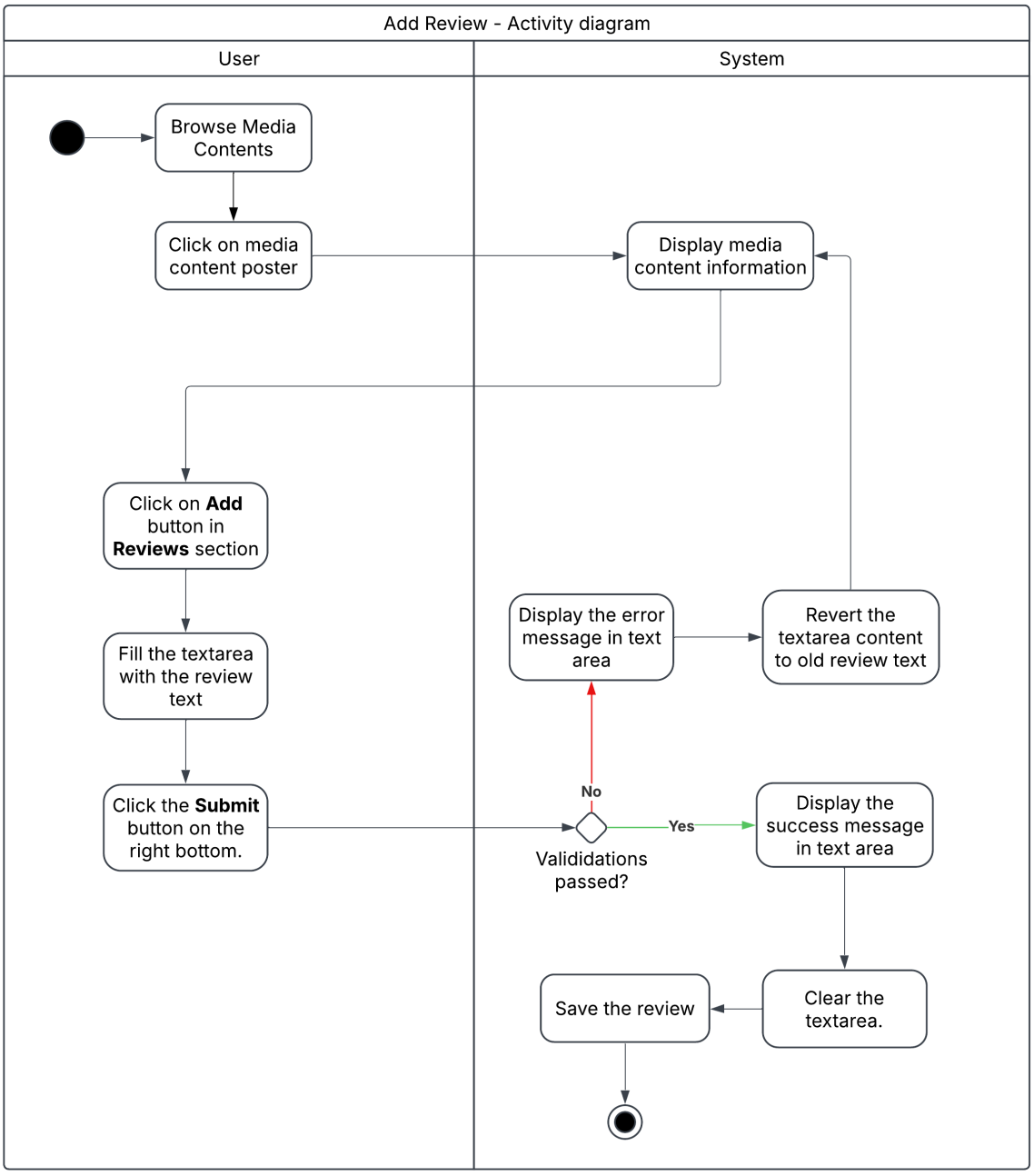

Figure 3: Write Review - Activity Diagram

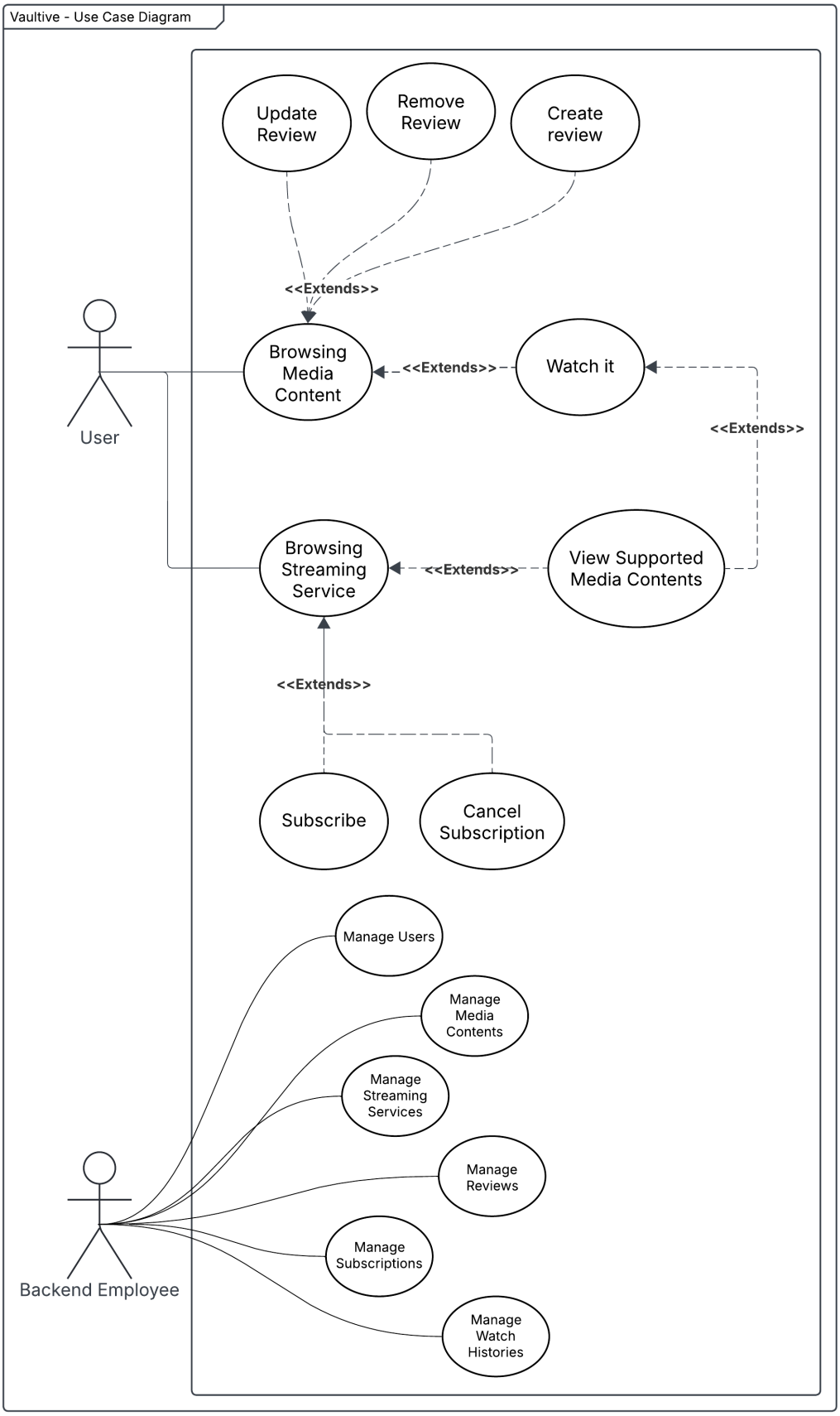

Figure 4: Use Case Diagram

Write Review - UC Scenario

Actor: User

Purpose and Context: To allow a logged-in user write a review to selected media content. In order to write review, user must have watched the media content before.

Assumption: The user's watch history includes the selected media content.

Preconditions:
1. The user is registered already.
2. The user is logged in already.

Basic Flow of Events:
1. System displays media contents.
2. The user clicks the poster of the desired media content.
3. The system shows the title, description, trailer, and available streaming options of the selected media content.
4. The user clicks the **Add Review** button in the Reviews section.
5. The System opens textarea for writing comment.
6. The user types their review text.
7. The user clicks **Submit**.
8. The system checks that the user's watch history confirms they have viewed the content, user exists and media content exists.
9. System displays the success message for 5 seconds in green font: Successfully submitted your review please refresh the page. Thank you!
10. System records the review.
11. Flow ends.

Alternative Flows of Events:

Validation failed in backend:
8a1 - Frontend returned json wrong or something happened in the connection and backend throwed error, no review added to the system.
8a2 - Flow continues from the 5 or if user decides to cancel process by clicking escape button, flow ends.

Postconditions:
Basic: The review is saved and immediately visible in the Reviews section.
Failure: No review is saved.

Figure 5: Write Review - Use Case Scenario

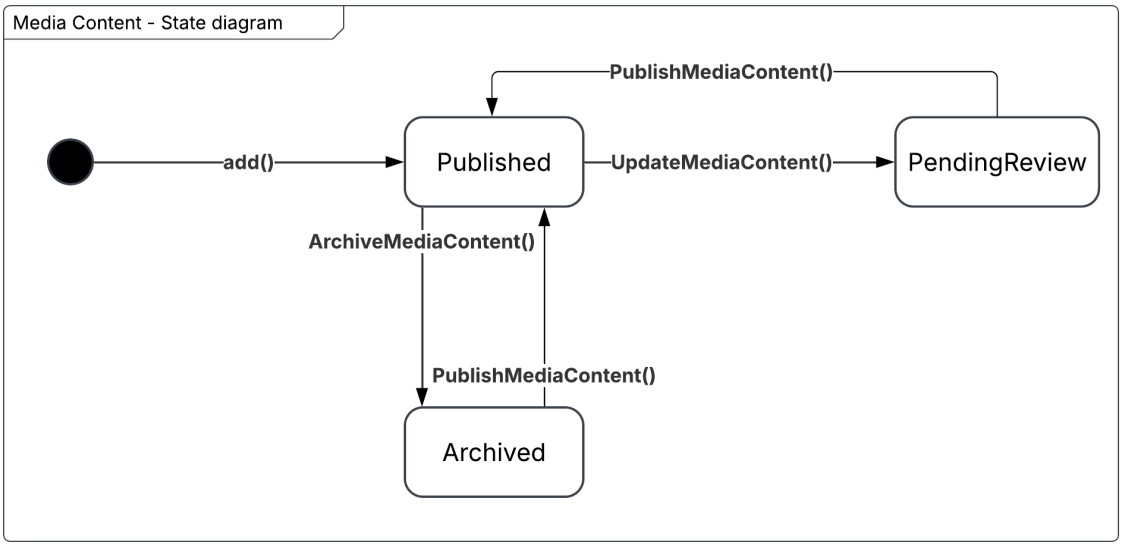

Figure 6: Media Content - State Diagram

State Diagram - Design Description

The state diagram describes the state of Media Content object life time. When media content is created by calling add() method by default it is getting published. Depending on some actions this state of published is changing.

The state of the media content can also be changed by calling the method ChangeStateOfMediaContent() to any state which is supported.

- **Published** - Media Content is visible in frontend. User can watch the content and write review for it. **PublishMediaContent()**
- **PendingReview** - After media content is updated the state changes automatically to pending review. If the back end employee accepts the changes it turns into Published. **UpdateMediaContent()**
- **Archived** - Media content is archived (not deleted). Not visible in front end. User can not watch or review it. **ArchiveMediaContent()**

Dynamic Analysis

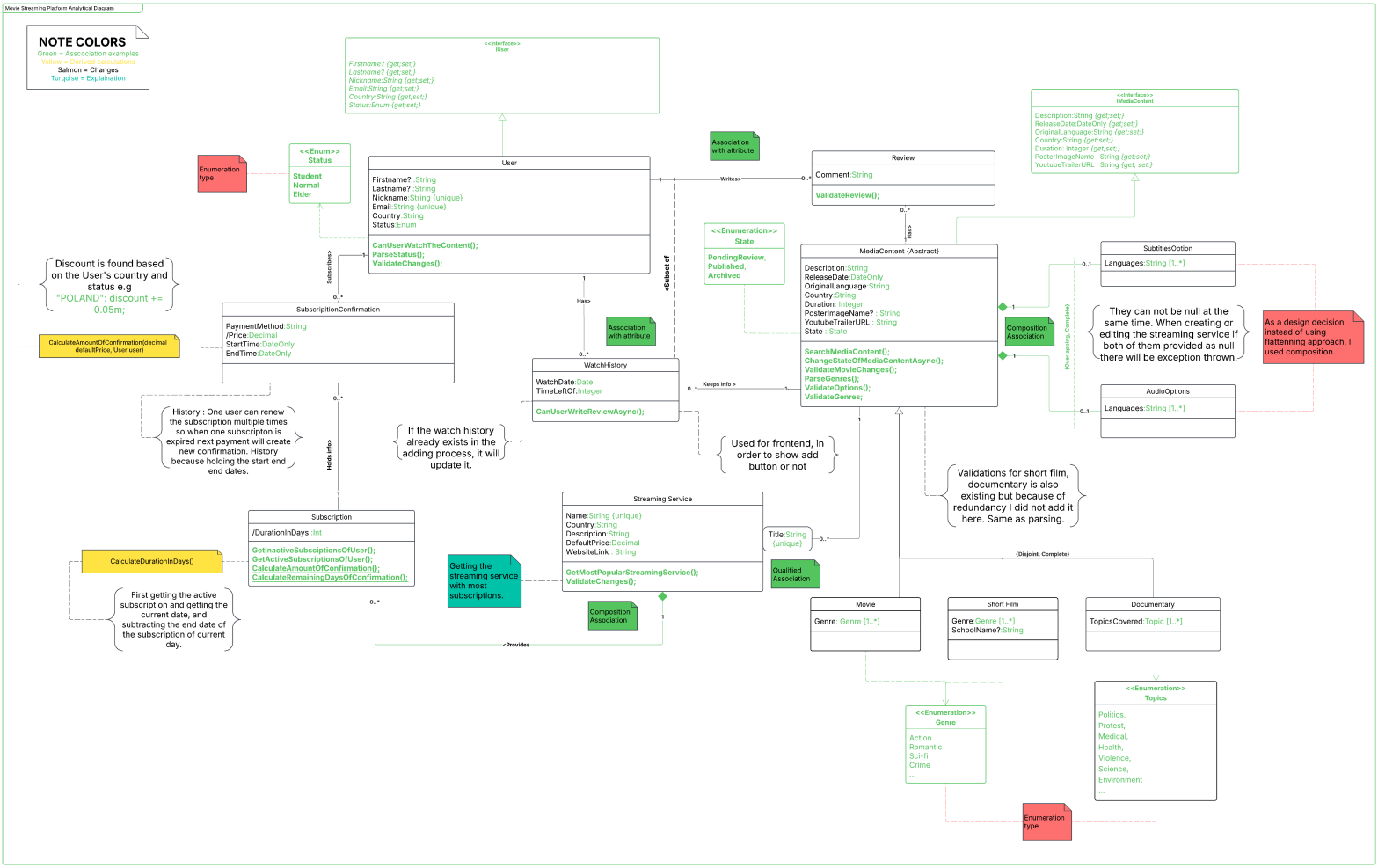

Figure 7: Design Class Diagram

Following the above-presented diagrams, a dynamic analysis was performed. The result of the analysis indicated a need for further expansion of the `WatchHistory` class. After implementing the review-writing feature on the frontend, it became necessary to dynamically display the "Add Review" button. If the user had not watched the movie, there was no reason to show this button.

To achieve this, a method named `CanUserWriteReview()` was implemented. When the user clicks on a movie poster and navigates to the details page, the `CanUserWriteReview()` method is called to validate whether the action is allowed. If the validation succeeds, the button becomes visible to the user.

To support this approach, after implementing the corresponding endpoint, the `reviews` section was removed from the JSON response of the movie-fetching operation. Reviews are now handled independently from the movie retrieval process. Fetching reviews and checking whether the user can write a review are both triggered when the user navigates to the movie details page. This separation improves the flexibility and modularity of the system.

GUI Design Discussion

The following section presents the GUI design. The interface was implemented using JavaScript, HTML, and CSS. The GUI screenshots shown are functional; the movies displayed are hardcoded, and the descriptions are copied and pasted from the internet. Trailers on the details page are embedded from the official YouTube channels of the selected movies.

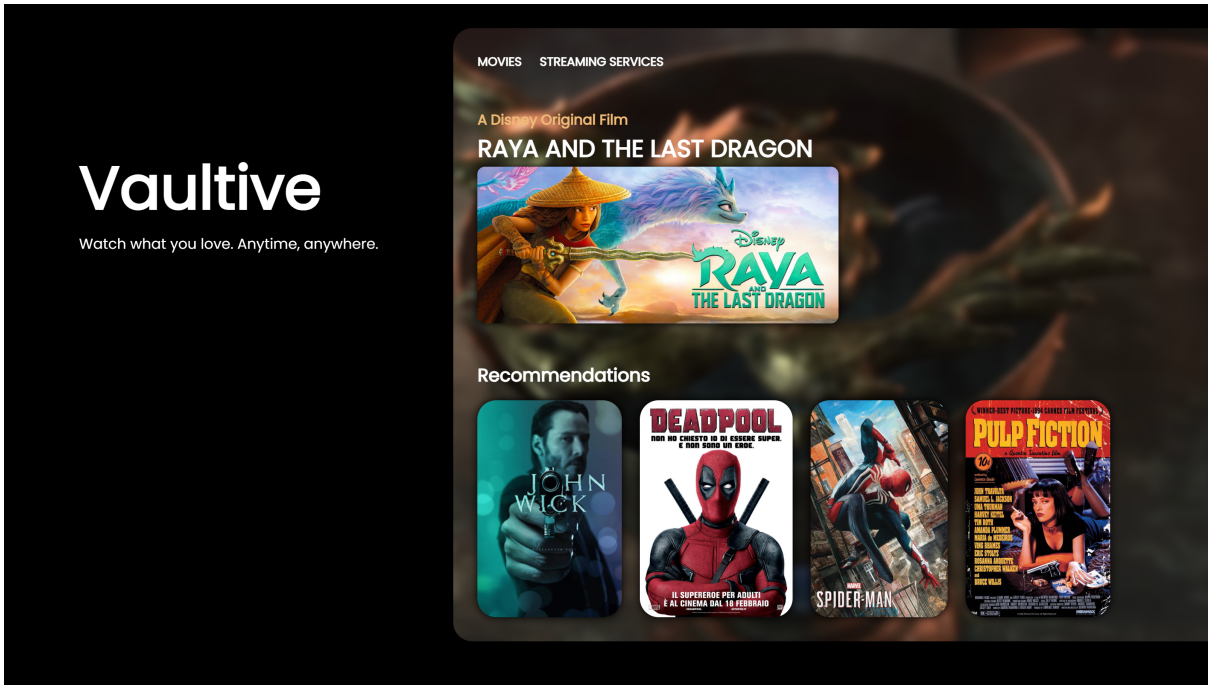

Figure 8: Vaultive - Home Page

Home Page

The screen shown above is the homepage of the website. In this view, there are two buttons: **Movies** and **Streaming Services**. These buttons serve as navigation elements, and their corresponding content will be presented later in the document.

The user can either use these navigation buttons to access the selected page or choose a movie from the recommended section on the homepage. To view details of a recommended movie, the user can simply click on its poster to view details of movie.

See Figures 8 and 9 for the Movies and Streaming Services pages. Figures 10 and 11 show the movie details page for watching trailers and writing reviews.

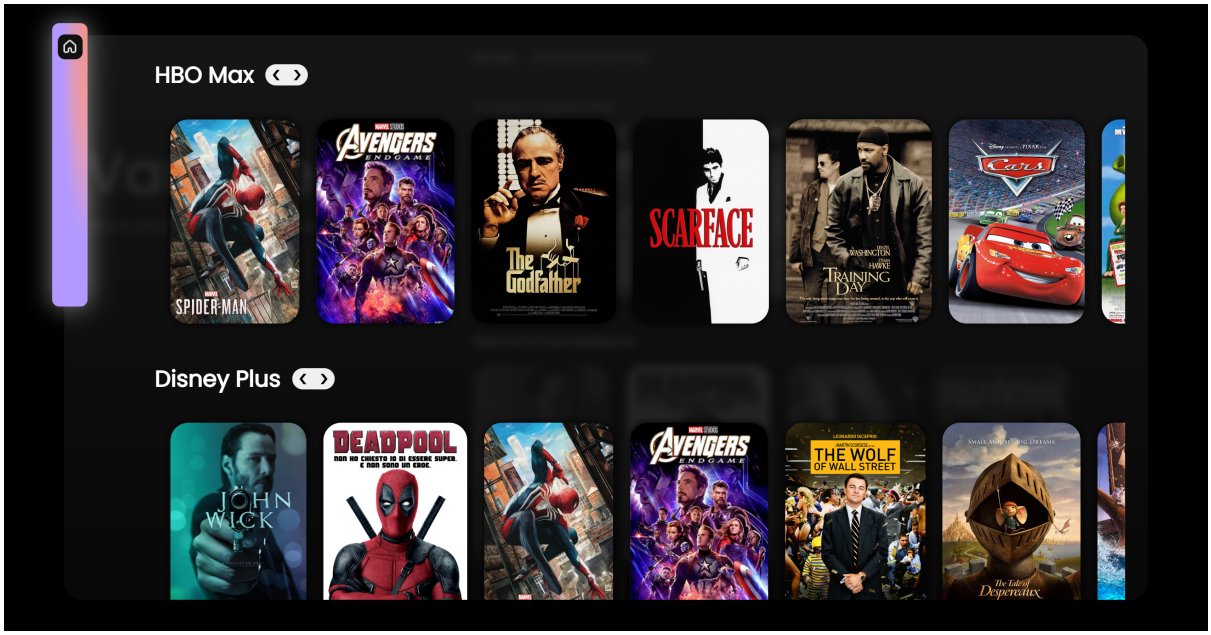

Figure 9: Vaultive - Streaming Services Page

Streaming Services Page

The screen shown above is streaming services page, in this page the user can view the movies supported by each streaming service. Since subscription and payment functionalities are not implemented on the frontend, the user can freely browse all available streaming services and watch trailers, but cannot access full content.

After retrieving the data from the backend in JSON format, the supported streaming services are dynamically mapped and displayed to the user along with their titles. Each time this page is accessed, the order of the streaming services is randomized.

Same as the previous page, user can click on any poster and go details to watch trailers or write a review. (If content is watched before)

See Figures 10 and 11 for the movie details page, which includes trailer viewing and review writing functionalities.

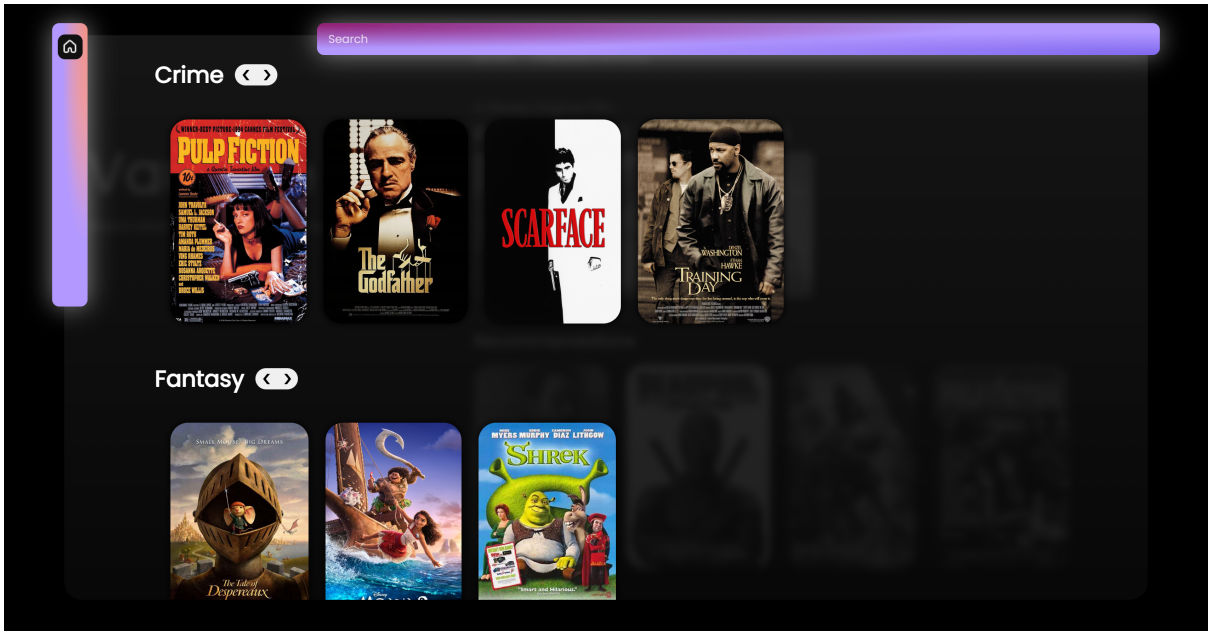

Figure 10: Vaultive - Movies Page

Movies Page

The screen shown above is the Movies page. On this page, the user can view movies categorized by their genres. Since subscription and payment functionalities are not implemented on the frontend, the user can freely browse all available movies and watch trailers, but cannot access the full content.

After retrieving data from the backend in JSON format, movies with the status type *Published* are dynamically mapped based on their genres and displayed to the user along with their titles. Each time this page is accessed, the order of the genres is randomized.

Similar to the previous page, the user can click on any poster to view the movie details, watch its trailer, or write a review (if the content has been watched before).

See Figures 10 and 11 for the movie details page, which includes trailer viewing and review writing functionalities.

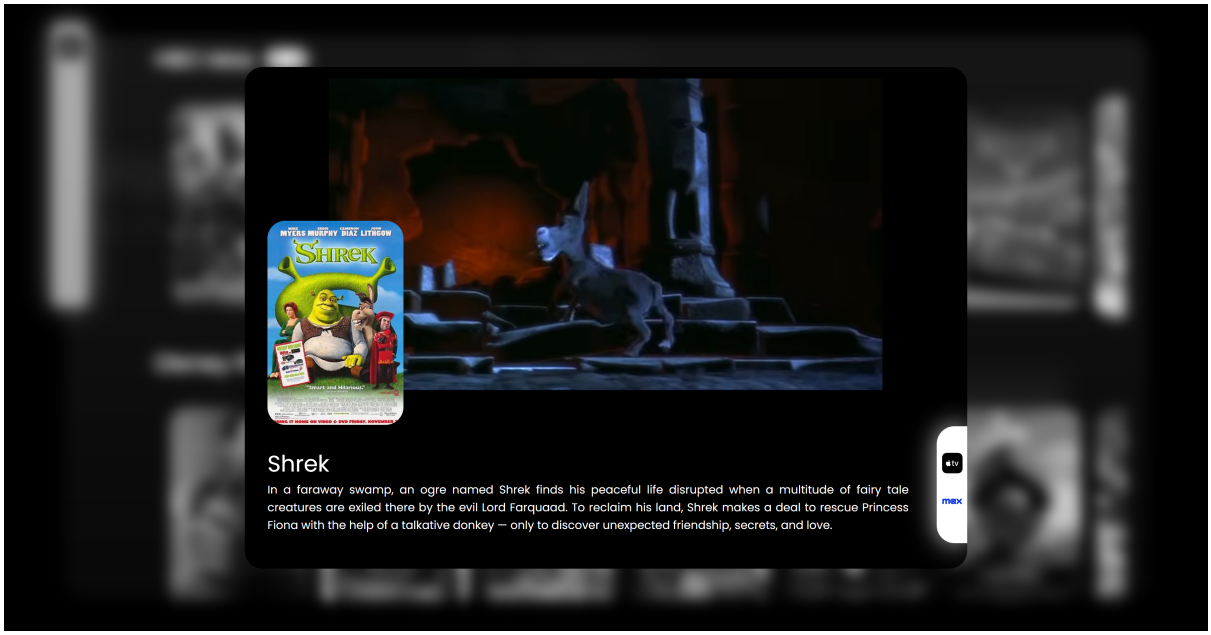

Figure 11: Vaultive - Details of Movie

Details of Movie

The screen shown above is the movie details page. On this page, the user can read the description of the movie, view the supported streaming services, and watch the movie trailer. The supported streaming services are displayed as small icons to the right of the description. By clicking on these icons, the user is redirected to the Streaming Services page to access the content.

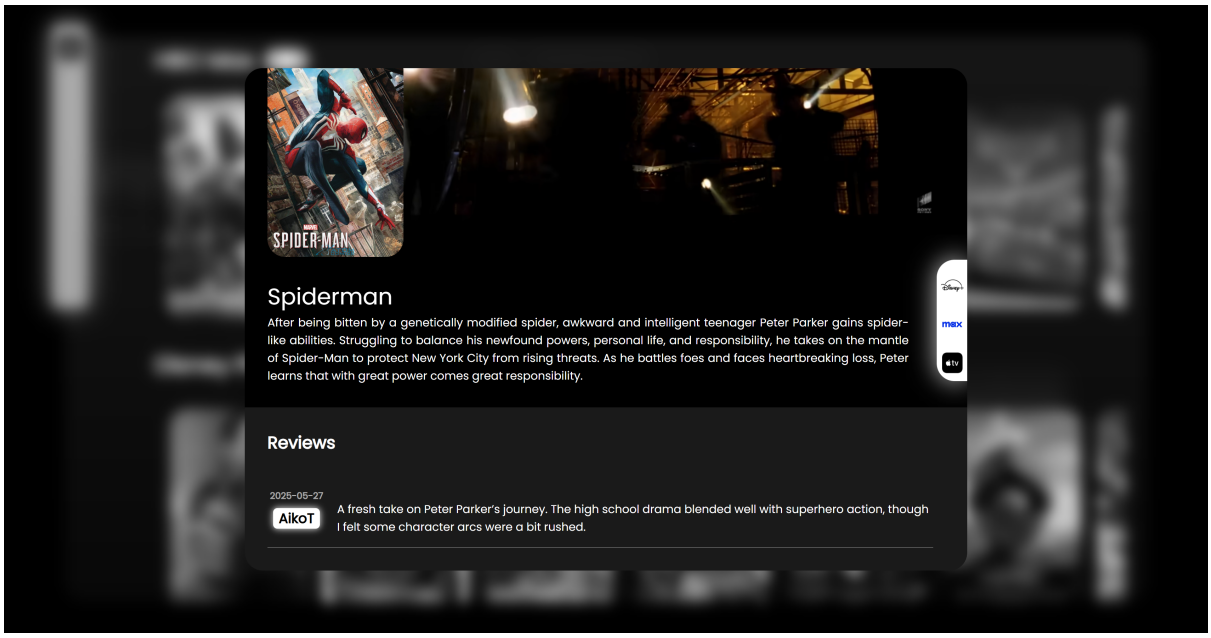

Figure 12: Vaultive - Details of Movie Reviews Section

Details of Movie Reviews

The screen shown above is the continue of the movie details page. On this section of the page user can read/write reviews to the selected media content. To be able to write a review user must have watched the media content before. Users which did not watched the content will not be able to see the **Add** button appears next to Reviews section in order to write a review.