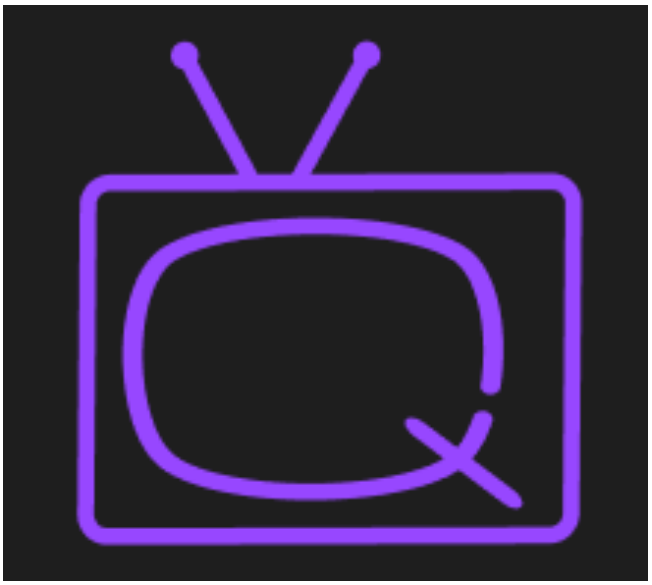


VIEW-QUEUE

TV Scheduling System Project Report

Project Title: TV Scheduling System
Module: CST2550 – Software Engineering and Management

Team Members:

- Scrum Master & Frontend Developer: Azul Debenedetti
- Secretary: Rudraa Patel (M00900573)
- Backend Developer: Dominik Wujek
- Tester: Hlaing Phyo Hein (M00906227)
- Developer: Filip Domanski

Table of Contents

1. Introduction

The *TV Scheduling System*, developed under the project title *ViewQueue*, was created as part of the CST2550 – Software Engineering and Management module. The project aimed to modernise the traditional TV guide experience by designing an application that blends the familiarity of broadcast scheduling with the interactivity and personalisation features users now expect from streaming services. Using a **code-first approach**, the system was built from the ground up with modularity, performance, and user experience in mind. The frontend was developed using React, with an emphasis on custom component creation and fine-grained state management, avoiding the use of utility libraries like Tailwind or Bootstrap in favour of handcrafted CSS and a shared Figma-driven design process.

The core objective was to develop a lightweight application where users could browse shows from various channels, add them to their personal schedules, and receive smart recommendations tailored to their preferences. From the login and registration flow to the Explore and My Shows pages, every element was crafted to provide a seamless, responsive, and mobile-friendly experience. A custom-built search bar allowed for dynamic filtering of shows, and user-selected genre preferences were stored and used to personalise recommendations, enhancing the system's usability and relevance.

Moreover, the team prioritised responsive design and accessibility from the beginning. Recognising that a large portion of users would access the platform via mobile devices, components like headers and settings were restructured to suit smaller screens without compromising on experience. Agile Scrum methodology structured the workflow through well-planned weekly sprints, role-based task allocation, and collaborative decision-making. Through this structured yet flexible process, *ViewQueue* evolved into a fully functional, user-focused TV scheduling platform that marries thoughtful design with smart technology.

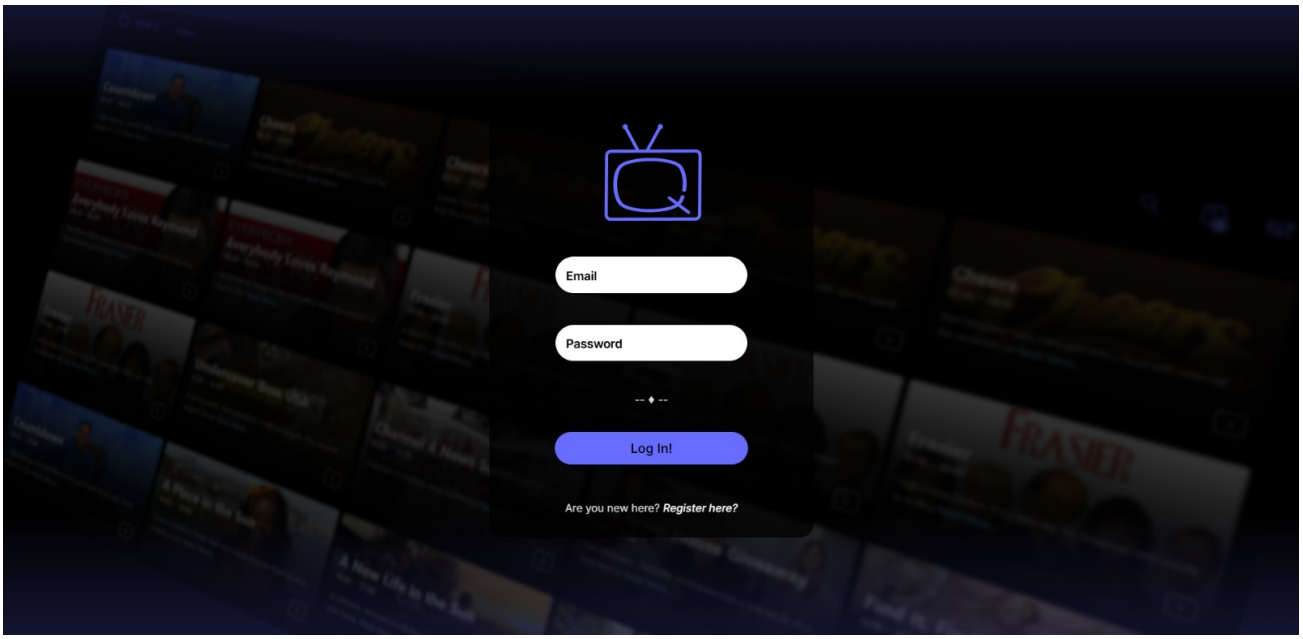
2. Screens

The TV scheduling system features a sleek and intuitive interface tailored for ease of navigation and personalised interaction. Users begin at the Login or Register page, where secure access and simple credential management are provided. Upon logging in, the Home Page offers highlights and suggested content. The All Shows and Channels pages present a categorised view of available programs, allowing users to filter by genre or channel.

The My Shows section allows users to manage their scheduled content with ease, while the Detailed Show View provides rich information for each program. The Recommendation Page delivers AI-driven suggestions based on user behaviour, enhancing content discovery. Lastly, the Settings Page lets users refine genre preferences, improving personalisation across the platform. The responsive design ensures accessibility across all devices, supported by a sidebar for smooth navigation.

Login Page

The login page provides secure user authentication. It prompts users to input their registered credentials (username and password) and includes basic error handling to ensure users input correct details. The interface is minimalist, emphasising simplicity and user-friendliness.

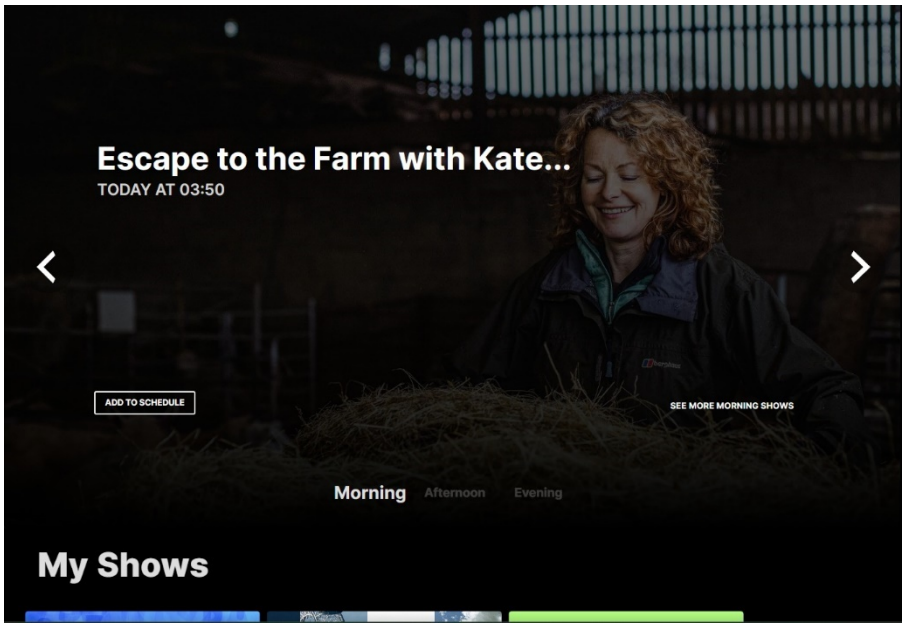


Register Page

The registration page facilitates new users to create an account by providing essential details such as username, email, and password. This page incorporates password validation to enhance security and includes a confirmation step to verify user details before finalising registration.

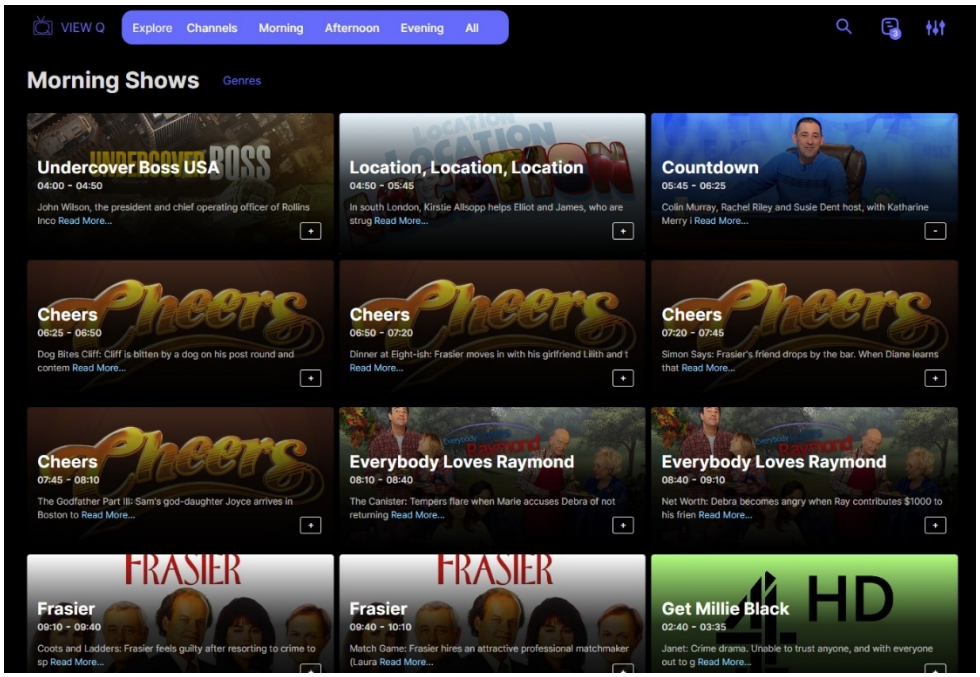
Home Page

The home page acts as the central hub, showcasing featured TV programs and personalised recommendations based on user preferences and viewing history. Users can quickly add shows to their personal schedules or browse through highlighted shows for easy discovery.



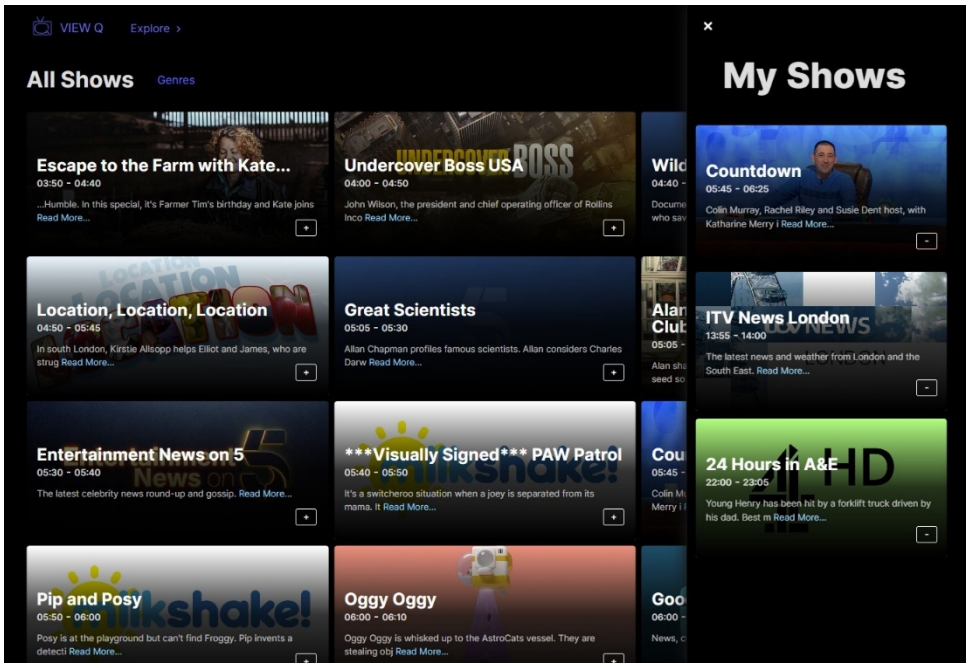
All Shows Page

The 'All Shows' page presents an organised, comprehensive list of available TV shows. Each show card includes a brief description, airing time, and the option to add or remove shows from the user's schedule. Users can easily navigate and filter content by genre or time slots.



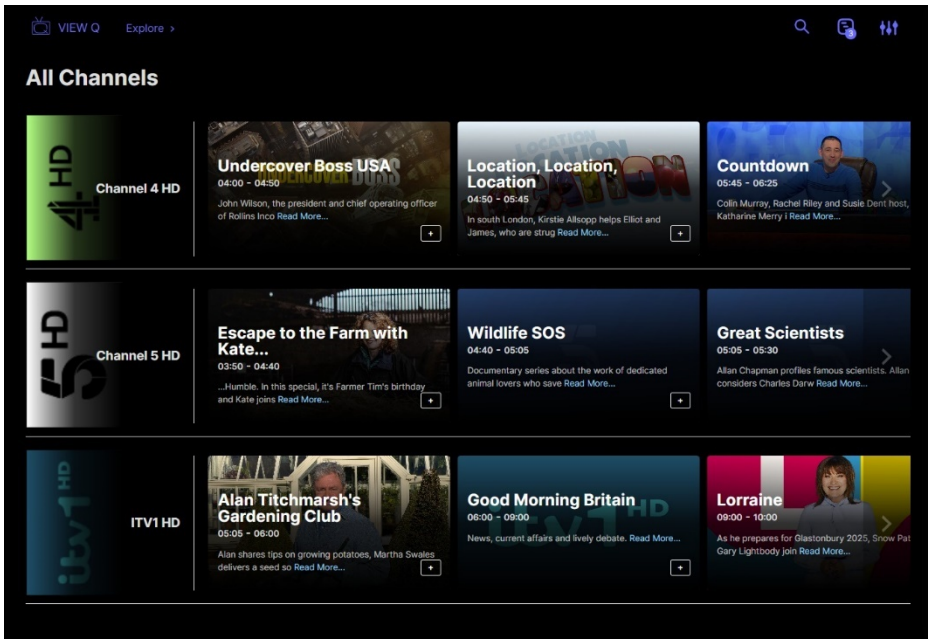
My Shows

The 'My Shows' feature displays a personalised list of shows that users have specifically added to their viewing schedule. It provides a clear overview and easy management of planned viewing, facilitating quick access and updates to scheduled content.



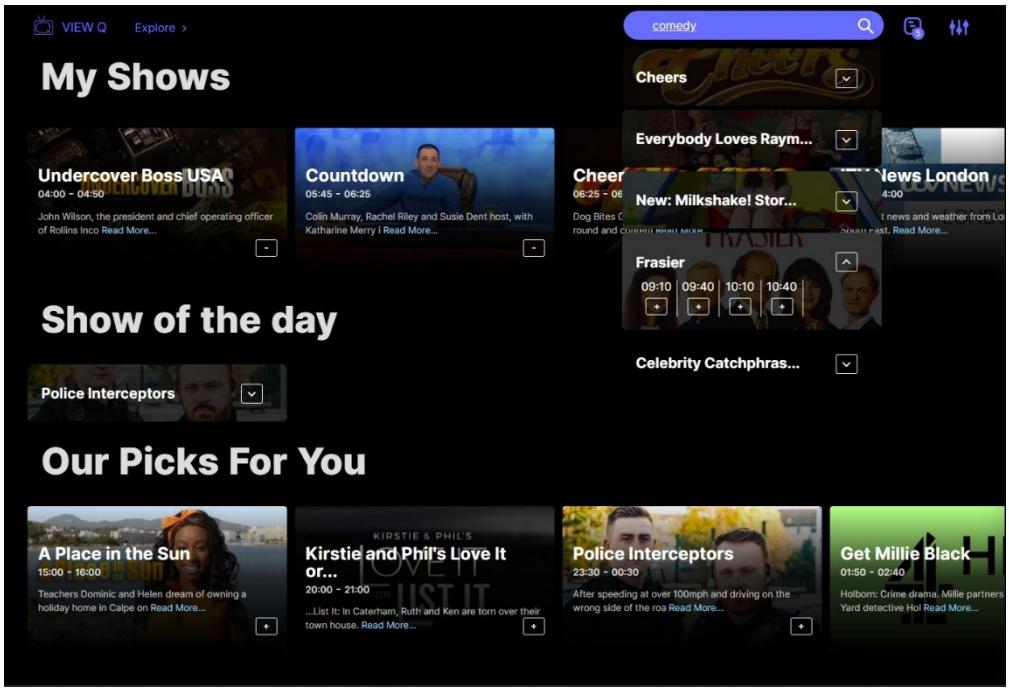
Channels Page

The channels page groups are shown by TV channels, displaying each channel's scheduled content clearly. It enables users to discover content according to their preferred channels and quickly schedule desired shows.



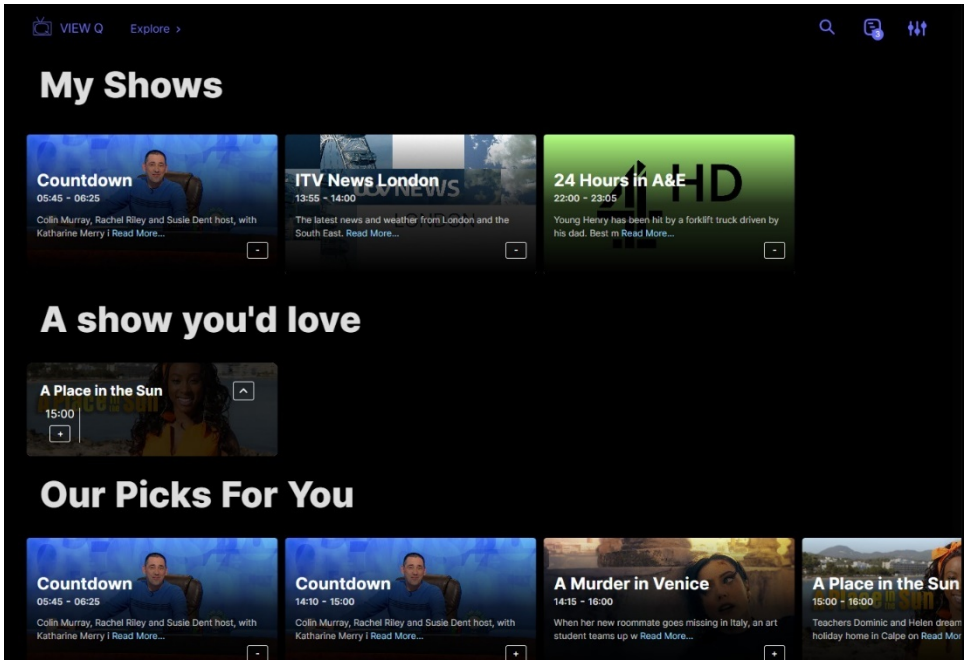
Search Bar

The search bar is a key feature of the application that allows users to quickly find shows by title. Positioned prominently within the interface, it simplifies navigation by enabling direct access to specific programs without browsing through the full catalogue. As users type, matching show titles appear dynamically, offering a seamless and efficient search experience tailored to user intent.



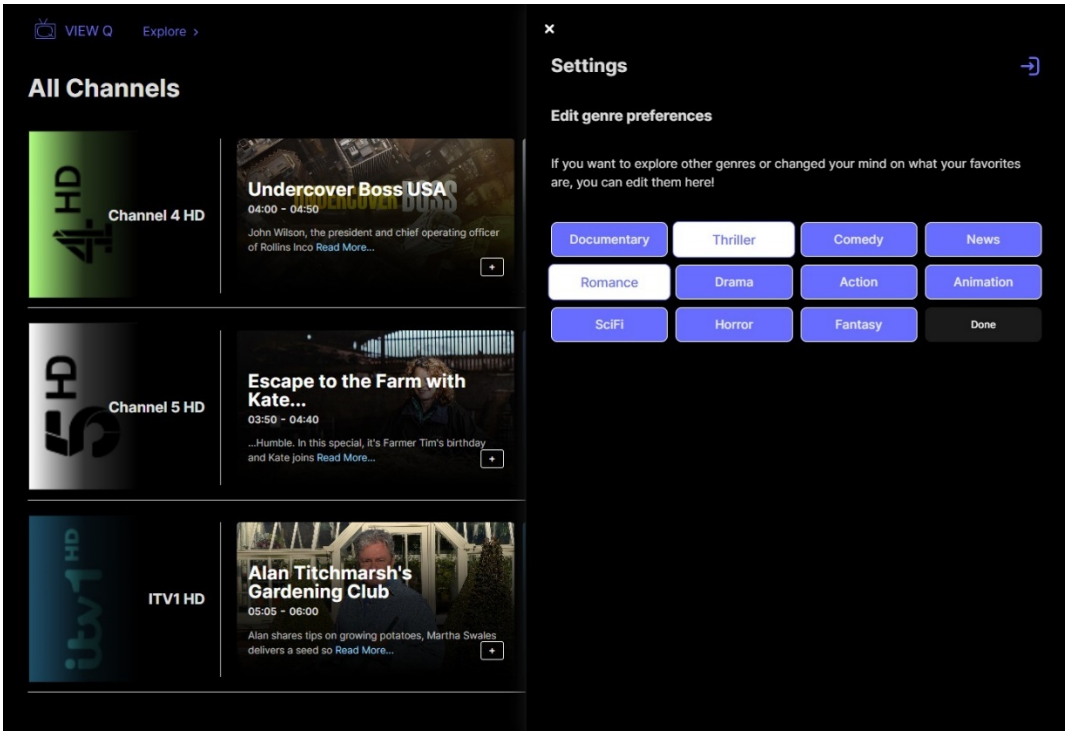
Recommendations

This section offers personalised TV show recommendations based on user preferences and historical viewing data, helping users discover new content effortlessly. Recommendations are categorised as shows the user would enjoy and general picks from popular content.



Settings Page

The settings page enables users to customise their viewing preferences and genre interests. Users can easily update their preferences, allowing the system to refine recommendations and improve overall user experience.



Responsive UI & Sidebar Navigation

The user interface is fully responsive, adapting smoothly to various devices and screen sizes. A sidebar menu allows users to swiftly navigate between different sections such as 'All Shows', 'My Shows', and settings, optimising user accessibility and convenience.

Each of these components is designed to ensure a seamless, engaging, and personalised user experience across the entire application.

3. Frontend Development

3.1. Introduction

Project Overview

ViewQueue is a TV guide web application built for TechSpire's Software Engineering Management and Development group project. This report focuses on the front-end application, explaining its ins and outs and the reasons behind a significant portion of the decisions that we made to allow the project to reach its current state. Additionally, we will be briefly talking about challenges and future improvements.

Objectives

At the start of this project, we had a team meeting and figured out what we wanted it to turn into. We roughly knew what the algorithm we were going to implement was, the different tools we had at our disposal to write it, and how we were planning on dividing the workload. When it came to the front end, we ran into a pretty clear scope issue. The sky was the limit when it came to how in-depth we wanted to go with it; the issue was identifying an achievable degree of complexity that would satisfy us.

The main objectives were to make a lightweight app that allows users to browse their tv provider's scheduled channels, and allows them to add them to their favourites. To do this we focused on features we often see in streaming services that make them great, and give them a twist that makes them unique. Our objective was not to redefine the wheel but to use it in our advantage to create something that's efficient and would adapt to the current TV scheduling landscape. We knew putting effort into having solid sorting and merging algorithms, intuitive UI and an efficient search function would pay off in the long run.

Target Audience

The current TV services are tailored to a more traditional audience. We found that a lot of people who do consume TV from a service provider do so out of habit and reluctance to change. New generations embrace the innovation that streaming services brought to media consumption with

user centric interface designs. With ViewQueue we attempt to combine these two paradigms in order to bring these two demographics together.

3.2. Project Architecture & Organisation

Project Structure

Efficient file organisation can be challenging, especially when dealing with a medium-sized React application like the one we have built, having several dozen JSX component files with their respective CSS files. To do so, we focused on keeping our src/ folder designed with separation of concerns in mind.

After planning the src/ folder had the following contents:

- **assets/**: Static images and SVG files used throughout the UI.
- **components/**: Modular and reusable UI components, grouped by functionality and usage.
    - E.g., showScheduler/, header/, misc/, loadingComponents/
- **contexts/**: React Context files for state management across components in order to mainly avoid prop drilling and cleaner code.
    - E.g., channelsContext, myShowsContext, AddRemoveShowsContextProvider
- **hooks/**: Custom hooks for behaviour abstraction.
    - E.g., useIntersectionObserver, useThrottle, useDebounce, useMergeAndFilter
- **pages/**: High-level page components corresponding to routes.
    - E.g., MainSchedulePage, ExplorePage, LoginPage

Component Hierarchy

The component hierarchy we implemented focused on page components used as the top-level structure relying on nested components to encapsulate responsibilities while lifting the shared logic to these top-level pages.

See below our effort to capture the simplified component hierarchy using our explorePage:

```
ExplorePage
 |
 | LOGIC
 | ├─ useParams (get URL section to determine what to render)
 | ├─ useContext (ChannelsContext + MyShowsContext)
 | ├─ useMergeAndFilter (custom hook to get filtered show list based on param)
 | ├─ Redirect if no channels (useNavigate in useEffect)
 | ├─ State: genre filter
 | └─ Filter logic with useMemo
 |
 |RENDER
 |   ├─ AddRemoveShowsContextProvider (provides function context)
 |   ├─ Header
 |   ├─ CONDITIONAL RENDER: section !== "Channels"
 |   | ├─ Section Title
 |   | ├─ GenreFilterComponent (lets user change genre filter)
 |   | └─ Content Container
 |   |      └─  ShowCard (display shows information)
 |   |
 |    └─ ELSE:
 |          ├─  All Channels
 |         └─ ChannelShowComponent (mapped for each channel)
 |
 └─ LogoLoadingComponent (while channels is being fetched)
```

State Management

The application manages state in two ways. Local state management is done in individual files in order to manage isolated and specific states that are used in the component they are initialised in or in their immediate vicinity. In other words, states that should not be passed down through several layers of child/grandchild components. In order to avoid prop drilling (passing down props through multiple component layers) we use React's Context API, which allows us to share states globally. Contexts such as ChannelsContext and MyShowsContext allow for centralised access to user and show-related data. These contexts will be discussed in more detail in our Context Providers section below.

Routing & Navigation

Routing is handled via React Router – a library that allows us to define pages and link them through routes. In our experience, it is the easiest and more straight forward way to handle navigation in react. It allows us to take advantage of dynamic routing while simplifying navigation (using the library hook useNavigate) and conditional redirection (in cases where the JWT token has expired).

See below our effort to capture our simplified routing.

/main → MainSchedulePage

/explore/:section → ExplorePage (display contents of explore page based on parameters)

/* → LoginPage (everything not mentioned above redirects to login page)

Context Providers

When our application started to grow, we found that prop drilling was not the most effective and readable way to pass down props in our component tree. Fortunately for us, React gives us a way to access shared data and logic without directly injecting from parent to child, React Context Providers. These allow us to both propagate information and centralise its management logic. To

use them, we wrap the components that use the provided information with the context provider and use the **useContext** hook inside them to access it. We currently use the following content providers:

- **FetchInfoProvider:** This context provider manages the information fetching from the backend, stores the relevant information in a "channels" state, and provides a function that allows us to promptly re-fetch the information.

- **AddRemoveShowsContextProvider:** This context provider manages the logic that allows the user to add and remove shows from their schedules and manages the appropriate API calls to update the database.

- **MyShowsContextProvider:** This context provider encapsulates the logic for managing the my shows array.

3.3. Key Features

Our application's core functionality centres on displaying and scheduling shows. This section of the project will do a shallow water dive into how we achieved these and the reasons behind our decisions.

Since some of us were already experienced with libraries like React, we opted to use it as our primary resource. Being highly modular with its component architecture, we took advantage of the opportunity to write clean, easy-to-read code to allow for easier collaboration.

Show Scheduler and Display Logic

The implementation of the main features is handled mainly in the components held by the higher-level pages. We will focus on the **MyShowsCompoenet**, **ShowCard** and **Search** components.

- **MyShowsComponent:** This component acts as a container for the shows that the user has on the MyShows array (scheduled or favourited shows). This container fetches channels data from ChannelsContext, shows array from MyShowsContext, filters it and maps it into reusable display cards (ShowCards).

```
---PSEUDO CODE---

Component ShowCard(show, isAdded, rowRef):


Initialise state:
    - expanded: controls whether the description is expanded


Use AddRemoveShowsContext to get the add/remove function
Define and use IntersectionObserver options
Convert show start/end times from UNIX to human readable format
Format the show description


Render:
    - If the card is in view (isVisible from IntersectionObserver):
        - Display the show's image
        - Show title, time range, and formatted description
        - Allow user to toggle full description ("Read More")
        - Display show tags
        - Display add/remove button depending on isAdded


    - If card is not in view:
        - Render a loading skeleton card (DummyCardLoading component)
```

- **ShowCards**: This component is likely to be the most used. It displays the information from a " show" object containing all the information we need – show name, start time, duration, image, id, description and more. Making this component was one of the most challenging in terms of design. Figuring out the best way to present all the information

```
---PSEUDO CODE---

Component Search:

Initialise state:

    - searchTerm state as empty string
    - debounce hook set at 500ms
    - pull all shows from useMergeAndFilter


On search input change:
  Wait for 500ms (debounce)
  Filter mergedShows where:
    show.name contains searchTerm OR
    show.tagName contains searchTerm


Create a new list with unique shows:
  For each filtered show:
    If show.showId is not already "seen":
      Add to uniqueShows
      Mark showId as "seen"


Render:
  If searchTerm is not empty:
    If uniqueShows is not empty:
      Render up to 5 matching results
    Else:
      Display "No results"
  Else:
    Render null
```

available, making use of space correctly in a way that respects component design principles, was a rewarding experience.

- **Search component:** The search component functionality is a core feature of our platform. The current API call fetches an overwhelming amount of show information. We

believe this is detrimental to the user experience. To fix this, we want to give the user the chance to trust the system they are using to be effective at recommending shows based on a well-structured algorithm and, in case they don't see what they are looking for, be able to promptly search and find it. This component manages that logic and neatly displays it in a component similar to ShowCard, but with a slicker design made to fit the needs of user search.

It's important to note that these main features would not be possible without two extremely important custom hooks we created called useMergeAndFilter and useShowLookup. We will talk and explain react hooks and specifically how we use the majority of our custom hooks in section 4, to avoid redundancy, think of these hooks as callback functions that allow us to use react features in them.

- **useShowLookup:** This hook is a basic show lookup algorithm that grabs our two important pieces of information that constitute the object we use to display information (shows and show instances), and joins them together creating a lookup table. We use this primarily as a helper hook in our useMergeAndFilter hook.

- **useMergeAndFilter:** The reason why this hook is so important is because our API call fetches an extensive object with potentially hundreds of shows and thousands of show instances. This hook gets all that information and, with the help of our useShowLookup hook, merges, sorts and filters these shows according to props passed by the user interactions (e.g. selected genres, passing a parameter through the url, etc). Before creating this hook, every file that needed this would do it independently; having these as a dedicated hook allows us to have a centralised source for this functionality.

```
---PSEUDO CODE---

Component useMergeAndFilter(filterType):
    Get channel data from ChannelsContext
    Get shows from useShowLookup (maps showId to additional details)

    Merge all shows across all channels:
        For each channel in channels:
            For each showEvent in channel:
                Merge event with corresponding show details using showId
            Add merged show to a combined array

    If filterType is "Morning":
        Return shows where start time is between 2 AM and 10 AM
    Else if filterType is "Afternoon":
        Return shows where start time is between 11 AM and 5 PM
    Else if filterType is "Evening":
        Return shows where start time is between 6 PM and 11 PM
    Else if filterType is "All":
        Return all merged shows sorted by start time
    Else if filterType is "carousel":
        Separate shows into Morning, Afternoon, and Evening based on start time
        For each time section:
            Add a "section" label (e.g., "Morning")
            Randomly select 5 shows using Fisher-Yates shuffle
        Return an object with arrays of 5 random shows for each section
    Else if filterType is "Channels":
        Return an empty array (handled elsewhere)
    Else:
        Return an empty array
```

User Personalisation

Personalisation in the traditional TV guide landscape is non-existent, which is not only a key feature that makes users want to come back to streaming services, but also a great opportunity to implement great features that would drastically enrich the user experience. Our app allows for personalisation from the beginning, asking for users' favourite genres from the first launch. These genres are stored on the backend and influence what shows are displayed as recommendations.

There were many options for us to request this user input, but we opted for creating our own modal component. The MainSchedulePage conditionally renders a Modal component on mount if no favourite have previously been set.

We understand that users evolve with time or might want to explore other recommendations. This is why we give the option to change the selected favourite tags at any point in the settings tab of our application.

3.4 Performance Considerations

Performance tends to be a huge issue and differentiates well built apps from the rest. In the case of our medium-sized application, we did a fair amount of research to make the UX the best we possibly could.

Virtual DOM: Re-renders and memoisation

The challenges that usually come with React stem from the way it renders information. React uses a Virtual DOM, which is a lightweight version of the DOM. When a component's state or props change, React compares the updated virtual DOM with the current one to check for these changes in a process called "reconciliation". This identifies the minimum set of changes needed to update the DOM to avoid unnecessary re-renderings of information that has not changed. This algorithm is optimised, but careless writing can still lead to unnecessary re-renders that can drastically bottleneck performance. This happens especially when state changes propagate down a deeply nested component tree.

In our application, where we dynamically render a lot of information such as dozens of show cards at once, constant changes in state that dictate *how* these cards are displayed could mean we very frequently re-render all of these components. This means downloads and re-paintings that drastically slow performance down. Thankfully, to avoid this React comes with some tools that, when used properly can make a huge difference. These tools focus on memoizing data, which is a type of caching that allows us to store computation results in cache, retrieving that same information from the cache the next time it's needed instead of computing it again. The main caching resources we use are the hooks *useMemo*, *useCallback*, and *React.Memo*:

- **React.Memo:** This is used to wrap components, like ShowCard, that take in props from a parent, but don't need to re-render unless those specific props change. This way, if our parent component receives a prop unrelated to the child component (e.g. Background colour), the potential dozens of memoised cards that it holds will not re-render.

- **useMemo and useCallback:** These two hooks are quite similar in many aspects, except their implementation – useMemo is used for values, and useCallback is exclusively used for functions. They memoise computations or states so that the values are only recalculated when certain conditions apply (based on a dependency array). Without this, all these computations would be re-created and recalculated after every re-render, even if the logic or value hadn't changed.

Having these in mind, it is easy to picture how powerful they are when combined, allowing components, values, and functions to be re-rendered, recalculated and recreated only when necessary.

Custom Hooks for Optimisation

React hooks in a nutshell are React hooks are special functions that allow us to hook into React features within functional components. They are a core part of react and drive a significant amount of it's appeal to many people. Throughout a React application we use pre-built hooks such as useState, useEffect, useContext to manage local state, run side effects, and access shared data across components. Additionally, we can create our own hooks and use them across our

application. The main purpose for custom hooks we use in our project is performance optimisation. These are useIntersectionObserver, useThrottle and useDebounce:

- **useIntersectionObserver:** This is one of the most relevant hooks we use and serves a very important purpose. In our application, we load and render many small components at all times. Understandably, loading all this information on page load, even when this information is not visible, is very performance taxing. This is what our intersection observer is for. This hook uses the browser's Intersection Observer API to detect when a component enters the viewport or designated root (component boundaries). Only when this happens are the components rendered. This drastically improves performance by reducing the initial load and off screen rendering. The intersection observer hook does not come without its disadvantages, such as constant re-downloads of resources and re-renders of components when the user quickly scrolls through the application.

- **useThrottle:** This hook was primarily created with scrolling in mind. When a user rapidly scrolls through our website, potentially thousands of scroll events are triggered, running our functions' logic on every one of these. Our throttle hook limits how often we allow our functions to run over time, ensuring they only execute once every specified interval, regardless of how often scroll events are triggered.

- **useDebounce:** This hook was primarily created with user input in mind. When a user is typing in an input field to search for shows, we do not want to re-run the filter logic after every keystroke. useDebounce waits for a specified delay (e.g., 300ms) after the last input before executing a function. This ensures we don't overload the app with unnecessary filtering logic while the user is still typing.

Using these hooks drastically improves performance and allows for the best user experience possible.

3.5 UI/UX Decisions

From the beginning of the project, we knew we had to put a significant amount of effort into our component design. The option of using utility libraries (such as Tailwind or Bootstrap) for this was always there, but we found that designing and creating our own components using pure CSS would allow us to give the attention and importance that they deserve. We used a shared Figma file (might require authorisation) in order to keep the collaboration as open as possible.

Component Design

The process of designing components was heavily reliant on the iterative design approach, where we would sketch, test, fix issues and repeat the process until we were fully satisfied with the results. While time consuming, it gave us access to manipulate all the nuance aspects of the design.

Design evolution from sketch to finished component

We believe that repeating this process with all components, from big pages to the smallest cards, is necessary to hold our UX/UI to the highest standard possible.

Mobile Adaptation

As previously mentioned, almost 60% of internet users browse the web on a mobile phone. Knowing this, the idea of responsive design was a necessity we understood from the beginning. The responsivity principles we followed when it came to design were the standard: start from smaller screen sizes, test sufficient content-centric breakpoints, clamp font sizes, etc.

We managed the responsive designs using CSS breakpoints and modifying existing CSS classes accordingly. We also had to redesign certain components to make the UX as enjoyable in mobile as it is on desktop (header component, settings component).

3.6 Challenges & Solutions

This section of the project is dedicated to reflecting on the challenges that we faced, the considerations that we took, and how we overcame them.

Server-side or client-side handling?

In our application, we heavily rely on information fetched from our API calls. The way these are structured, we would get a URL containing an image that we would display for each show. Whether this image was displayed in a show card (200x360 pixels) or a banner (60% of the screen), this image was always the same size.

Since we are using our Intersection Observer, every time a component (e.g. our showcards) leaves its designated root's view field, we need to re-download and paint these sometimes heavy images (upwards of 30MB). Understandably, this can create significant bottlenecks, slowing our application down drastically.

Our initial idea was to solve this entirely on the client side. One of the first solutions we explored involved preloading a lower-quality image as a placeholder and swapping it with the high-resolution version once it finished downloading. However, this required us to already have the original image file to create the lower-res version, meaning the full image still had to be downloaded first. This defeated the purpose, as it did not reduce bandwidth usage or improve load speed.

We found that ideally, we would hold a resized version of the image and dynamically decide where to display it. Unfortunately, resizing and storing all these new versions of heavy assets on the client side was extremely inefficient.

Ultimately, we ended up opting for handling this in the backend, using an image processing library to resize the images that were considerably larger, storing them in the database, and eventually serving them to the front end through our API call. This allowed us to keep the download and rendering time low, while not losing image quality when displaying them in our larger components.

This example taught us that even when we use clever resources like our intersection observer, there can always be drawbacks when applications reach a certain size. Spending time looking at different ways to solve issues will be an exceptional learning experience and enrich us as developers in the long run.

3.7 Future Improvements

During team meetings, our group decided on what is achievable with the amount of time we had to our disposal. Nonetheless, we enjoyed the idea of the project to have more than enough ideas of features we could add. A key conclusion we came to was that going forward with the project, a soft launch would be released to get enough feedback to decide which ideas to discard and which to move forward with. We reached this conclusion because the project is purely user-centred, so it only makes sense for users to dictate what the best route to move forward is. Here are some of the ideas we came up with (not necessarily included in our presentation):

- **Add full accessibility features:** Individuals with disabilities consume media as often as individuals without them and as mentioned before, we are meant to have all users in the centre of our focus**.** Additionally, the target demographic of our project tends to lean towards older people. Its been well researched that apart from people with disabilities, older demographics are one of the main beneficiaries of accessibility features.
- **Create a native version of the app for mobile devices:** Currently, our web app has a responsive design that can adapt to multiple devices. This is great because mobile devices account for around 58% of website traffic. Our app would benefit from people using it on

the go, maybe on their way back from work, to plan their media consumption. Adding an Android and IOS application using Flutter or react native would make this a lot more convenient for users.

- **Add notifications and calendar integration:** Our application looks to solve the need to *schedule* or plan user consumption in the near future. Having notifications/calendar features integrated would minimise the chances of users missing out on their scheduled shows.

3.8 Conclusion

This project was an excellent opportunity to create an application that tackles an existing real-world problem that any of us could encounter during our employment. We managed to make an application that satisfied our standards, is both user-friendly and responsive. Throughout the development process, we focused on writing modular, reusable components and prioritised clarity in data flow and state management.

One of the more valuable takeaways was learning how important it is to plan for scalability early on. Features like the search function, which uses debounced input and dynamic filtering, evolved naturally from user needs and showed the value of thinking ahead about performance and UX.

While there's room for future improvement, we are pleased with where the application currently stands and would be happy to use it in our portfolios. The process of writing this project has helped us strengthen our understanding of building efficient and scalable front-end applications, data structuring, and practical state management.

4. Backend Development

4.1 Introduction

4.1.a Objectives

The ViewQueue backend aims to address critical challenges in TV show scheduling and user engagement through a robust, scalable system. Key requirements include:

Efficient Data Management: Frequent database queries and data duplication in show schedules increase storage needs and latency, hindering scalability. The system must minimise redundancy and optimise retrieval to support endpoints like /main.

Accurate Show Categorisation: Shows from external TV guide API's lack categorisation, complicating recommendations and filtering. The system requires intelligent tag assignment to enhance user preference alignment (e.g., via /favourite-tag).

Personalised Recommendations: Diverse user preferences and large show catalogues make relevant recommendations challenging. The system must deliver tailored global and individual suggestions to boost engagement (accessible via /main).

Frontend Performance: High-resolution images cause slow page loads and excessive bandwidth usage, degrading user experience, especially on mobile. The system needs optimised image delivery to ensure fast rendering (supports /main, /add-show-manually).

Scalability and Reliability: High query volumes and external API dependencies risk performance bottlenecks. The system must leverage caching and background jobs to reduce database load and ensure data freshness.

These objectives drive the system's modular design, with targeted solutions in the Core Components to ensure performance, personalisation, and scalability.

4.1.b Solution Overview

The backend of the application is a modern .NET 8.0 Web API application that provides a comprehensive solution for TV show scheduling, user preferences, and personalised recommendations. Built on a robust foundation of Microsoft technologies, the application leverages Entity Framework Core for efficient database operations with MySQL, Microsoft Identity for secure user authentication and authorisation, and Hangfire for reliable background job processing. The architecture follows a modular design pattern, with clear separation of concerns between data access, business logic, and API endpoints. The system integrates with external TV guide APIs to fetch show schedules and uses OpenAI's GPT-3.5-turbo for intelligent show categorisation. A sophisticated caching mechanism optimises performance, while background jobs ensure data freshness and system maintenance. The application's frontend is served through a React-based interface, with the backend providing optimised image delivery and real-time updates through a well-structured API.

4.2 Project Structure

The solution follows a clean, modular architecture with clear separation of concerns:

1. **Controllers** (`/Controllers`)
   - `TvController`: Handles TV guide and show management
   - `AccountController`: Manages user authentication and preferences
   - Implements RESTful API endpoints with proper HTTP verbs
   - Uses dependency injection for service access

2. **Models** (`/Models`)
   - Entity classes for database mapping
   - DTOs for API request/response handling
   - Implements data validation attributes
   - Follows Entity Framework Core conventions

3. **Utilities** (`/Utilities`)
   - Specialised services for business logic
   - Background job definitions
   - Cache management implementations
   - Image processing utilities

4. **Network Services** (`/network-services`)
   - External API integrations
   - HTTP client implementations
   - Data transformation services

5. **Database** (`/Migrations`)
   - Entity Framework Core migrations
   - Database schema versioning
   - Seed data for initial setup

4.3 Development Environment

1. **Dependencies**
   - .NET 8.0
   - Entity Framework Core 8.0.4
   - Hangfire 1.8.17
   - MySQL with Pomelo provider
   - Swashbuckle for API docs

2. **Configuration**
   - Environment-based settings
   - Secure secret management
   - CORS policy configuration
   - Database connection handling

3. **Development Tools**
   - Swagger UI for API testing
   - Hangfire Dashboard for background job monitoring
   - Entity Framework migrations
   - Development-specific middleware

4.4 Core Components and Their Interactions

4.4.a Efficient Timeline Data Storage

Problem:

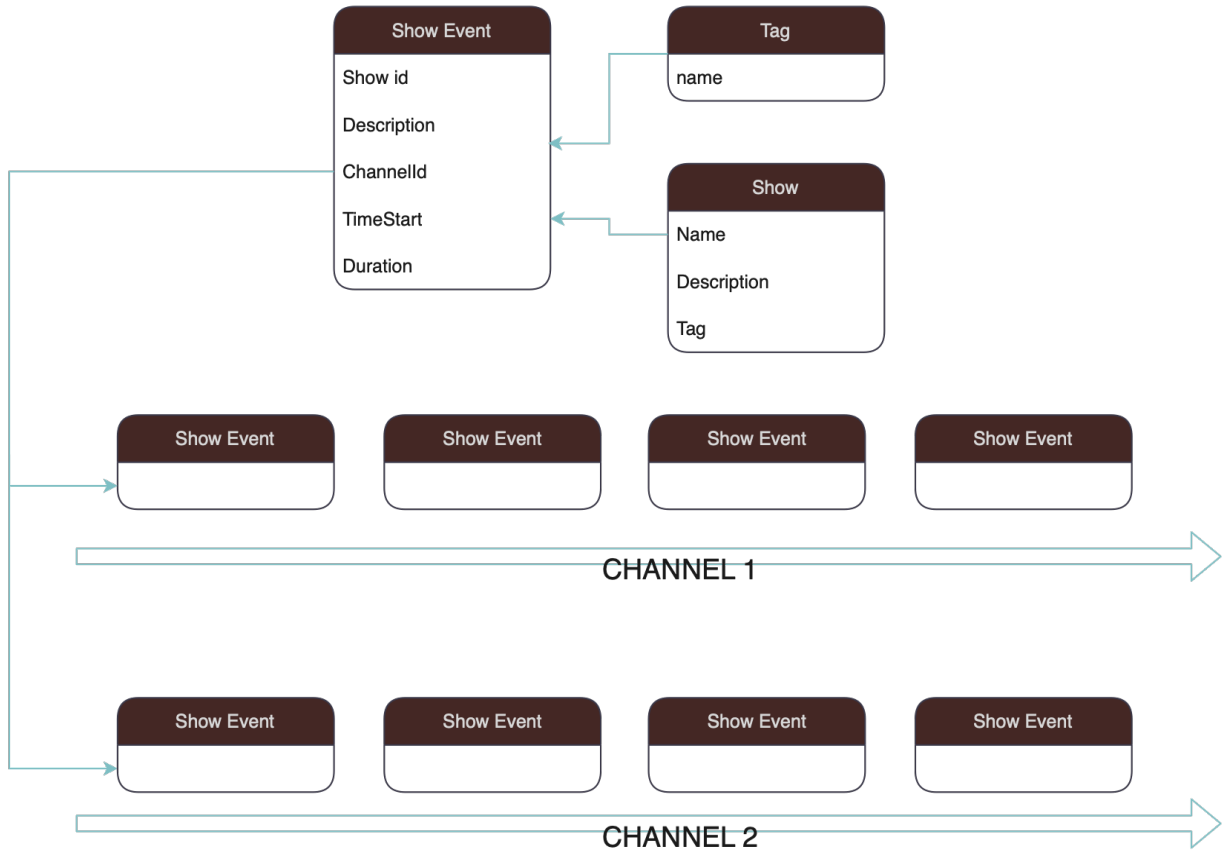
Storing TV show data with channel and user schedules leads to significant data duplication. Each show's metadata (e.g., title, image) is redundantly stored for every schedule entry, increasing storage needs, complicating updates, and risking inconsistencies. This inefficiency hinders scalability and query performance.

Solution:

The system uses a normalised data model to eliminate duplication and optimise storage. Show metadata is stored once in a Shows table, referenced by separate ChannelSchedules and UserSchedules tables via foreign keys. This design:

- Centralised Metadata: A single Shows table ensures metadata is stored once, simplifying updates.
- Normalises Schedules: Scheduling data links to Shows, reducing redundancy.
- Unifies References: Both schedule types share the same show records, enhancing consistency.
- Boosts Efficiency: The model minimises storage and improves query performance.

This approach ensures data integrity, scalability, and efficient retrieval, as seen in the /main endpoint, making it ideal for managing TV show timelines.



4.4.b TagsManager and AI-Powered Categorisation

TV shows fetched from the official TV guide API lack categorisation, making it difficult to organise the show catalogue effectively. Without tags, the system struggles to support accurate recommendations or align with user preferences, leading to a suboptimal user experience and inefficient content filtering.

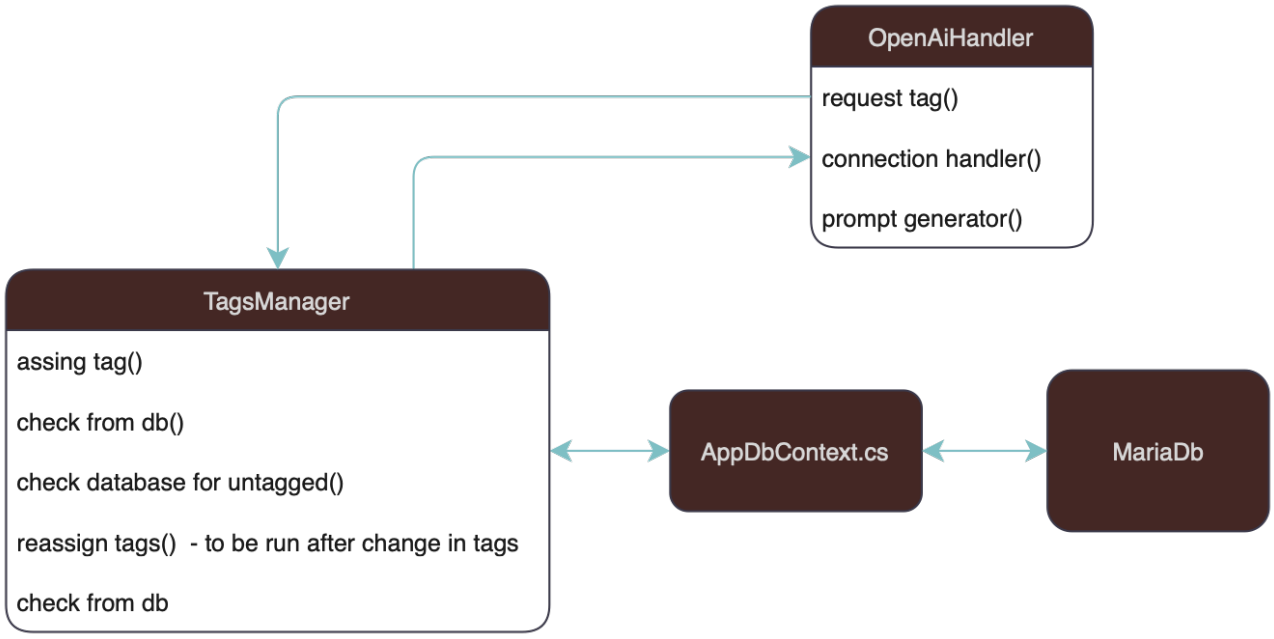
The process involves:

The TagsManager service addresses this by leveraging OpenAI's GPT-3.5-turbo model to intelligently assign tags to shows. The process involves:

- Predefined Tags: Maintains a list of valid tags (e.g., Action, Comedy, Drama).

- AI Analysis: Sends show names to GPT-3.5-turbo with a prompt including valid tags, generating appropriate categorisations.
- Validation: Ensures AI responses match predefined tags, maintaining consistency.
- Database Integration: Creates new Tag entities if needed and associates them with shows.

This AI-driven approach ensures accurate, consistent categorisation across the show catalog, enhancing recommendation systems and user preference alignment, as utilised in endpoints like /favourite-tag and /main



4.4.c Recommendation System Architecture

The recommendation system consists of global and individual recommendation generators. The RecommendationGeneratorBase class provides core functionality, while RecommendationGeneratorGlobal and RecommendationGeneratorIndividual implement specific recommendation strategies.

```pseudo
class RecommendationGeneratorBase:
    - DbContext: Database access
    - TodaysShowsCache: Cached show data

    method PickRandomShow(shows, pastRecommendations):
        if pastRecommendations exists:
            filter out recently recommended shows
        return random show from remaining pool
```

Global Recommendations:

- Generates a single daily recommendation for all users
- Maintains a history of past recommendations to avoid repetition
- Uses a weighted random selection from today's shows
- Stores recommendations in the GlobalRecommendations table with an 'Active' flag

```pseudo
class RecommendationGeneratorGlobal:
    inherits RecommendationGeneratorBase

    method SetGlobalRecommendation():

        pastRecommendations = get all global recommendations

        if active recommendation exists:
            deactivate it

        availableShows = get cached shows
        newRecommendation = PickRandomShow(availableShows, pastRecommendations)

        create new RecommendationGlobal:
            - ShowId: selected show
            - Active: true
            - Added: current date
```

Individual Recommendations:

- Considers the last 3 days of recommendations to avoid repetition

- Uses the TodaysShowsCache to efficiently filter shows by user preferences.
- Implements a fallback mechanism when no shows match user preferences
- Creates personalised recommendations based on the user's favourite tags

```pseudo
class RecommendationGeneratorIndividual:
    inherits RecommendationGeneratorBase

    method SetIndividualRecommendation(userId):

        if recommendation exists for today:
            return

        userFavouriteTags = get user's favorite tags

        if no favorite tags:
            return

        recommendedShows = get shows matching user's tags
        if no matching shows:
            return

        recentRecommendations = get recommendations from last 3 days

        newRecommendation = PickRandomShow(recommendedShows, recentRecommendations)

        // Save new recommendation
        create new RecommendationForUser:
            - UserId: userId
            - CreatedDate: today
            - ShowId: selected show
```

4.4.d TodaysShowsCache: Optimising Show Retrieval

Problem

Frequent database queries for today's TV show data, which updates infrequently, overload the database and degrade system performance. Repeated requests for the same show data increase latency, network traffic, and resource consumption, hindering scalability and responsiveness, especially for tag-based lookups used in recommendation and scheduling features.

Solution

The TV Scheduler's caching system optimises data access by implementing the Cache-Aside pattern to store frequently queried show data in memory. It maintains an in-memory cache of show data and a tag-to-shows hash-map, enabling O(1) tag-based lookups (accessible via /cachedShowsTagHashmap). Updated daily through a Hangfire job, the system fetches show events, filters them by timestamp, and atomically refreshes the cache to ensure data consistency. This approach eliminates redundant database queries, reduces latency, and enhances scalability, supporting endpoints like /checkShowsCache and /main while maintaining memory efficiency and robust error handling for cache misses and update failures.

4.4.e ImageManager: Frontend Performance Optimisation

Problem

High-resolution images from the external TV guide API, often 1920x1080 or larger, cause significant performance issues in the TV Scheduler. These large assets lead to slow page loads, excessive bandwidth usage, and poor user experience, particularly on mobile devices. Unoptimised images also increase server load and storage demands, complicating scalability and maintenance.

```pseudo
class TodaysShowsCache:
method getCachedShows():
    return cachedShows

# Returns tag-to-shows hashmap
method getShowTagHashmap():
    return showTagHashmap

# Retrieves shows by tag IDs
method getTodaysShowsByTags(tagIds):
    result = new List<Show>
    for tagId in tagIds:
        if showTagHashmap contains tagId:
            result.addAll(showTagHashmap[tagId])
    return result

# Updates cache with new shows and rebuilds hashmap
method setCachedShows(shows):
    cachedShows = shows
    showTagHashmap.clear()
    for show in shows:
        if showTagHashmap does not contain show.Tag.Id:
            showTagHashmap[show.Tag.Id] = new List<Show>
        showTagHashmap[show.Tag.Id].add(show)
```

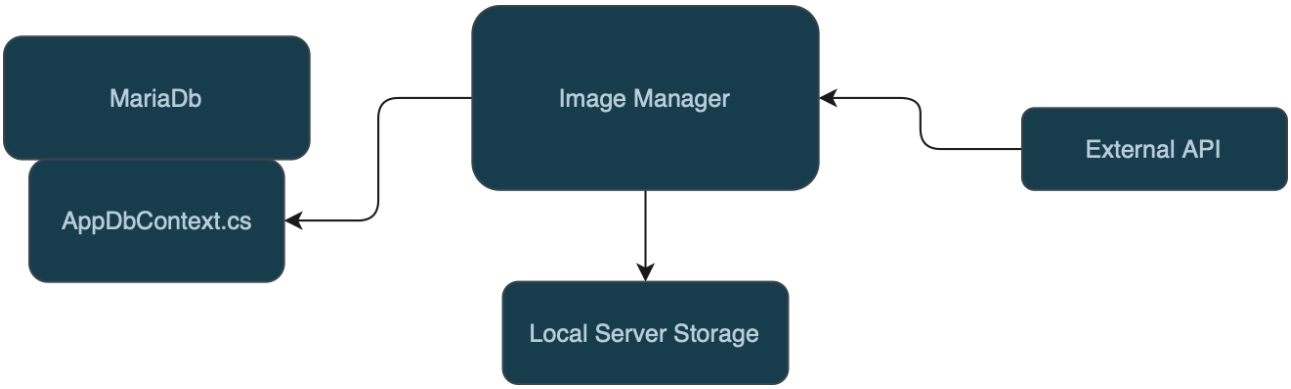
Solution

The Image Management System implements an automated processing pipeline to optimise images, enhancing performance and user experience. It resizes images to optimal dimensions (360x200), applies JPEG compression, and preserves aspect ratios, achieving a 70-90% reduction in file size while maintaining quality. Key features include:

- Intelligent Optimisation: Automatically resizes and compresses images with unique filenames to prevent conflicts.
- Efficient Processing: Batch-processes show images in a single transaction, with retries for failed operations.

- Smart Storage: Organises files in a dedicated uploads directory, with scheduled cleanup of outdated assets.
- Caching: Stores processed images in memory to reduce server load.

Integrated with robust error handling and database tracking, this system ensures fast page loads, reduced bandwidth, and scalability, supporting endpoints like /main and /add-show-manually.




```pseudo
class ImageManager:
// Core functionality
method ResizeAndSaveImage:
- Download original image
- Calculate optimal dimensions
- Resize maintaining aspect ratio
- Save with unique filename
- Return relative path

method ProcessAllShowImages:
- Get all shows needing resizing
- Process each image
- Update database references

method DeleteAllResizedImages:
- Remove all files from uploads directory
- Clear database references
```

5.5 API documentation

Authentication Endpoints

ENDPOINT	METHOD	DESCRIPTION	AUTHENTICATION	REQUEST BODY	QUERY PARAMETERS	RESPONSE
/add-show-manually	POST	```json { "channelId": "int", "showName": "string", "imageUrl": "string", "tagName": "string", "startTime": "int", "duration": "int" } ```	None	200 OK with created show event		
/testTagGetter	GET	Tests AI-based tag generation	None	None	showName: string	Generated tag
/checkShowsCache	GET	Returns current cached shows	None	None	None	List of cached shows
/cachedShowsTagHashmap	GET	Returns tag-to-shows mapping from cache	None	None	None	Dictionary of tags to shows

Tag Management Endpoints

ENDPOINT	METHOD	DESCRIPTION	AUTHENTICATION	REQUEST BODY	QUERY PARAMETERS	RESPONSE
/favourite-tag	POST	Adds a tag to user's favourites	Required	`{ "tagId": "int" }`	None	200 OK on success
/un-favourite-tag	POST	Removes a tag from user's favourites	Required	`{ "tagId": "int" }`	None	200 OK on success
/set-favourite-tags	POST	Replaces all user's favourite tags with new set	Required	`{ "tagIds": ["int"] }`	None	200 OK on success

TV Shows and User Schedule Endpoints

ENDPOINT	METHOD	DESCRIPTION	AUTHENTICATION	REQUEST BODY	QUERY PARAMETERS	RESPONSE
/main	GET	Returns comprehensive TV guide data for authenticated users	Required	None	None	`{ "schedule": [UserScheduleItem], "favTags": [FavouriteTagDTO], "channels": [ChannelDto], "shows": [ShowDto], "globalRecommendation": "int", "individualRecommendation": "int" }`
/add-show-to-schedule	POST	Adds a show to user's personal schedule	Required	`{ "showEventId": "int" }`	None	200 OK on success
/remove-show-from-schedule	POST	Removes a show from user's personal schedule	Required	`{ "showEventId": "int" }`	None	200 OK on success

Account Management Endpoints

ENDPOINT	METHOD	DESCRIPTION	AUTHENTICATION	REQUEST BODY	QUERY PARAMETERS	RESPONSE
/register	POST	Creates a new user account	None	`{ "name": "string", "password": "string" }`	None	Success: 200 OK with message Error: 400 Bad Request if username exists or validation fails
/login	POST	Authenticates user and returns JWT token	None	`{ "name": "string", "password": "string" }`	None	Success: 200 OK with JWT token Error: 401 Unauthorized for invalid credentials

5. Testing

5.1 Statement of Testing Approach

The testing infrastructure used in the tvscheduler project is succinctly described in this report, with particular attention paid to two essential testing suite components:

TestBase Class

The TestBase class serves as an abstract foundation providing common test functionalities for other test classes in the project. Its main objective is to improve test reliability and maintainability by standardising and simplifying the test setup.

Key functionalities provided by the TestBase class include:

In-memory Database Context Creation:

- Each test scenario is provided with a distinct, isolated in-memory database instance.
- Databases are uniquely identified to ensure test isolation, preventing data contamination across tests.
- Before each test run, existing data is cleared to ensure a clean state,

promoting accurate and repeatable results.

Mock UserManager Creation (Authentication Testing):

- Utilizes the Moq framework to simulate the UserManager<User> class, essential for testing authentication and authorisation logic.
- Configured mock methods (CreateAsync, UpdateAsync, DeleteAsync) always return successful outcomes, allowing predictable and controlled testing scenarios.

Mock Configuration for JWT Settings:

- Provides mock configuration data (JWT key, issuer, audience) necessary for authentication tests.
- This ensures authentication tests are decoupled from actual environment settings.

Mock HTTP Client:

- Mocks external HTTP requests, returning predefined responses.
- Critical for testing API integrations without dependence on external services, improving test stability and performance.

TestDataFactory Class

Data setup for each test suite is made much easier by the TestDataFactory class, which organises and simplifies the creation of reusable test data.

Functionalities include:

Predefined HTTP Responses:

- Methods like CreateTvProgramResponse() and CreateTagResponse() generate standardised mock API responses, enabling consistent API interaction testing.

Domain Model Instances Creation:

- Provides methods such as CreateTag(), CreateShow(), and CreateChannel() for quickly generating test instances of key domain entities.
- Ensures data consistency and reduces repetitive code across tests.

Database Seeding:

- The SeedBasicTestData() method efficiently seeds the in-memory database with essential data (tags, channels, shows, and events).
- Guarantees that each test scenario operates with a predefined dataset, enhancing test predictability and reliability.

By utilising mocked dependencies standardised test data, and in-memory databases, the TestBase and TestDataFactory classes work together to create a strong testing foundation.

5.2 Table of Test Cases

Account Controller Tests

Test Name	**Purpose**	**Expected Result**
Registration_WithNewUser_ReturnsOkResult	Register a new user	200 OK
Registration_WithExistingUsername_ReturnsBadRequest	Registering with an existing username	400 Bad Request
Login_WithValidCredentials_ReturnsToken	Logging in with correct credentials	Token received
AddShowToSchedule_WithValidRequest_ReturnsOkResult	Adding a show to schedule	200 OK
RemoveShowFromSchedule_WithValidRequest_ReturnsOkResult	Removing a show from schedule	200 OK

Recommendation Generator Tests

Test Name	Purpose	Expected Result
SetGlobalRecommendation_ShouldCreateNewRecommendation	Create a new global recommendation	New recommendation created
SetGlobalRecommendation_ShouldDeactivatePreviousRecommendation	Deactivate the previous global recommendation	Previous recommendation deactivated
SetIndividualRecommendation_ShouldCreateNewRecommendation	Create a new individual recommendation	New recommendation created
SetIndividualRecommendation_ShouldNotCreateDuplicateForSameDay	Prevent duplicate recommendations on the same day	Single recommendation only
ClearGlobalRecommendationsHistory_ShouldRemoveAllRecommendations	Clear all global recommendations	All recommendations removed
ClearIndividualRecommendationsHistoryForAllUsers_ShouldRemoveAllRecommendations	Clear all individual user recommendations	All recommendations removed

TV API Tests

Test Name	Purpose	Expected Result
FetchGuideData_ShouldReturnJsonElement	Retrieve TV guide data	JSON data retrieved

FetchMultipleProgramData_WithValidChannelIds_ReturnsDictionary	Retrieve program data from multiple channels	Data retrieved per channel

Tag Manager Tests

Test Name	**Purpose**	**Expected Result**
AssignTag_ShouldCreateAndAssignTagToShow	Create a new tag for a show	Tag created and assigned
AssignTag_WhenTagExists_ShouldReuseExistingTag	Reuse existing tags	Existing tag reused
DeleteTagIdsFromAllShows_ShouldRemoveAllTagReferences	Remove tags from all shows	Tags removed from all shows
DeleteAllTags_ShouldRemoveAllTagsFromDatabase	Completely remove all tags from the database	All tags removed from database

5.3 Swagger UI API Testing Evidence:

The attached Swagger UI screenshots demonstrate the success of key API functionalities:

Account Registration (/register) and Login (/Account/login):

- Successfully registered a new user and logged in, receiving a valid JWT token.

Authenticated API Calls:

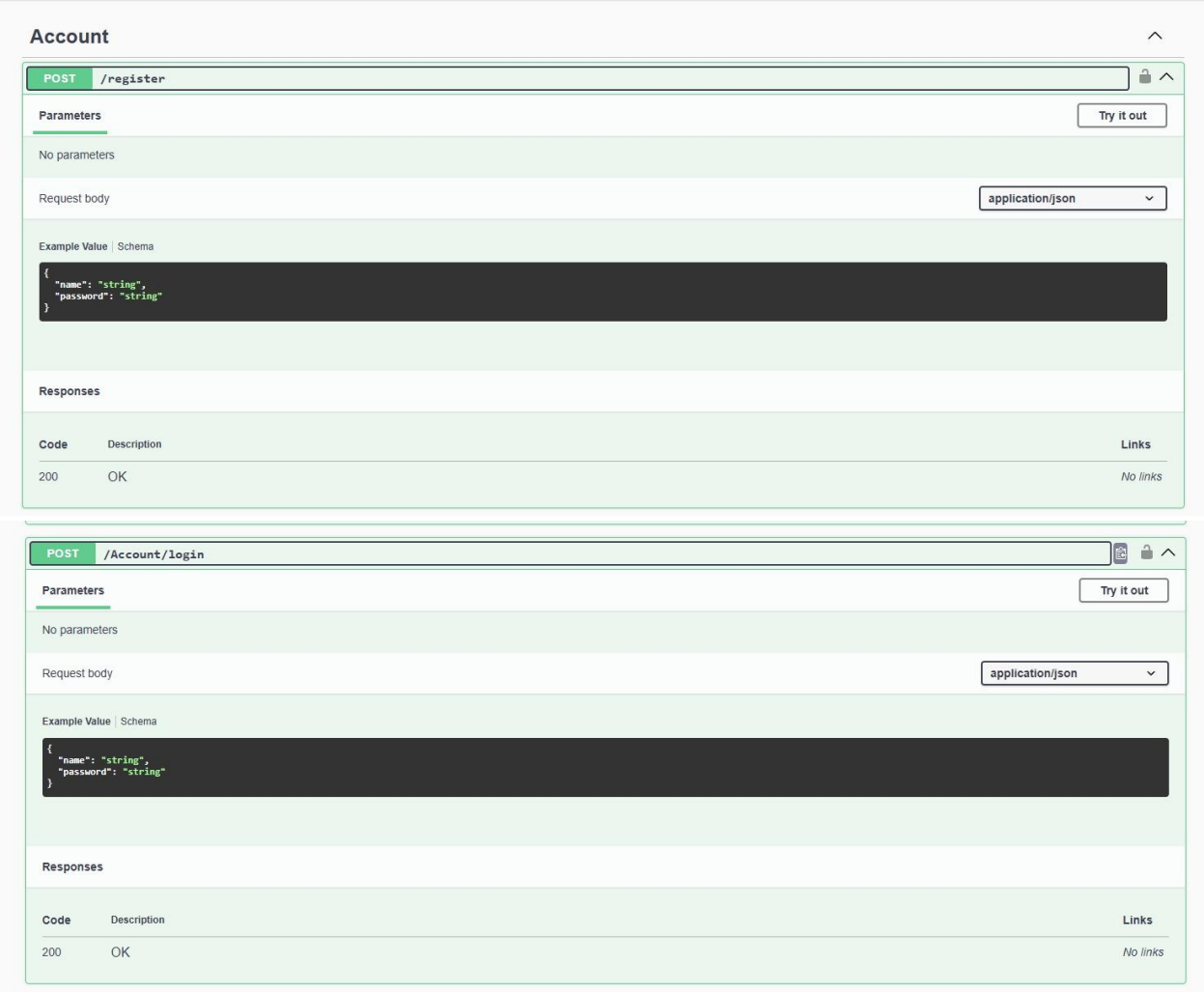
- API endpoints such as /main, /checkShowsCache, /cachedShowsTagHashmap, /testTagGetter have been successfully accessed using JWT authentication.

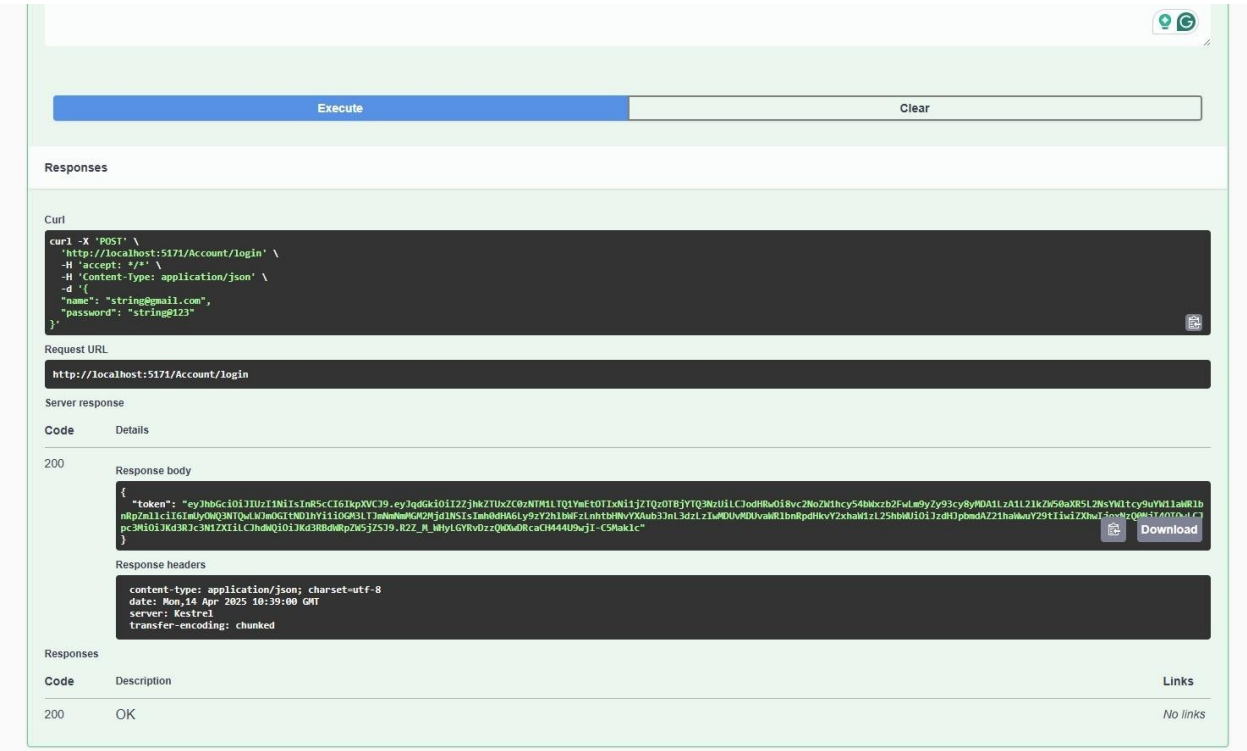
Response Validations:

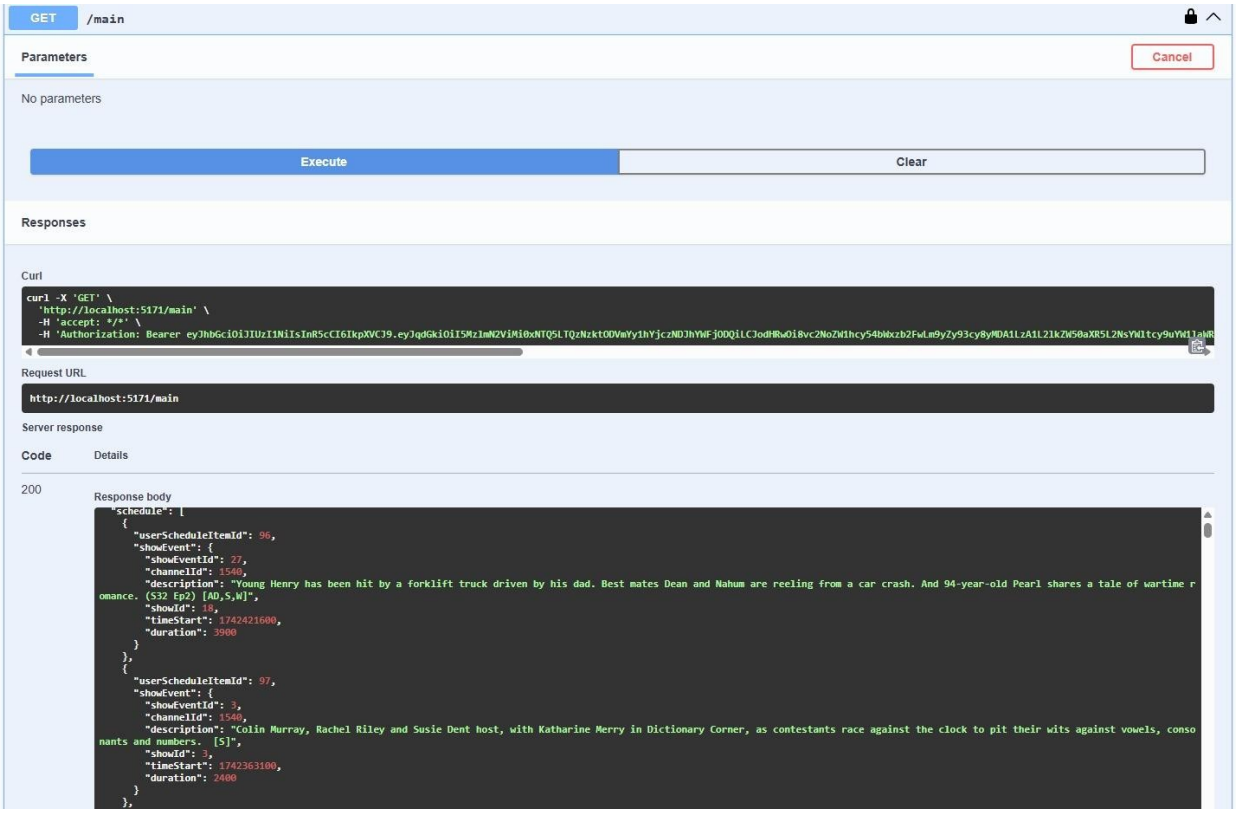
- API responses return expected HTTP status codes (mostly 200 OK), and response bodies correctly reflect the data structure (JSON responses shown in

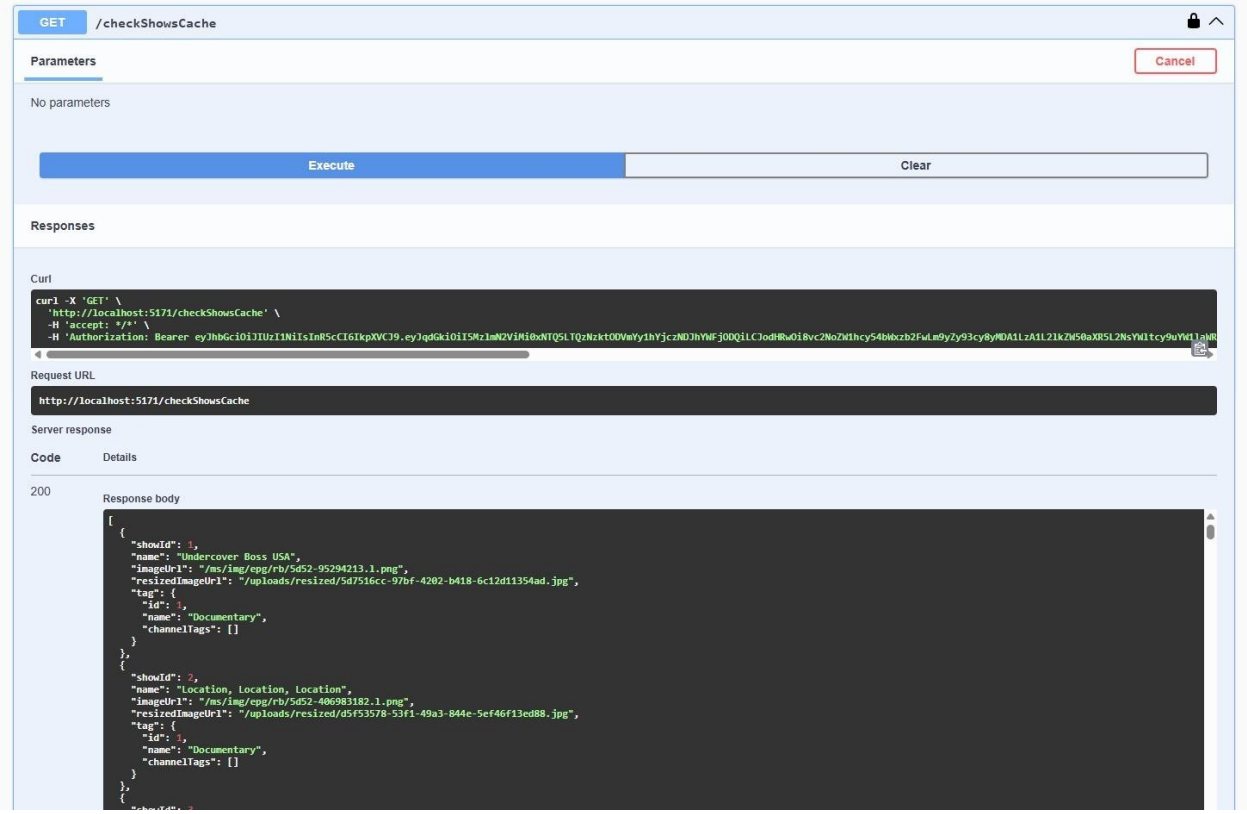
screenshots).

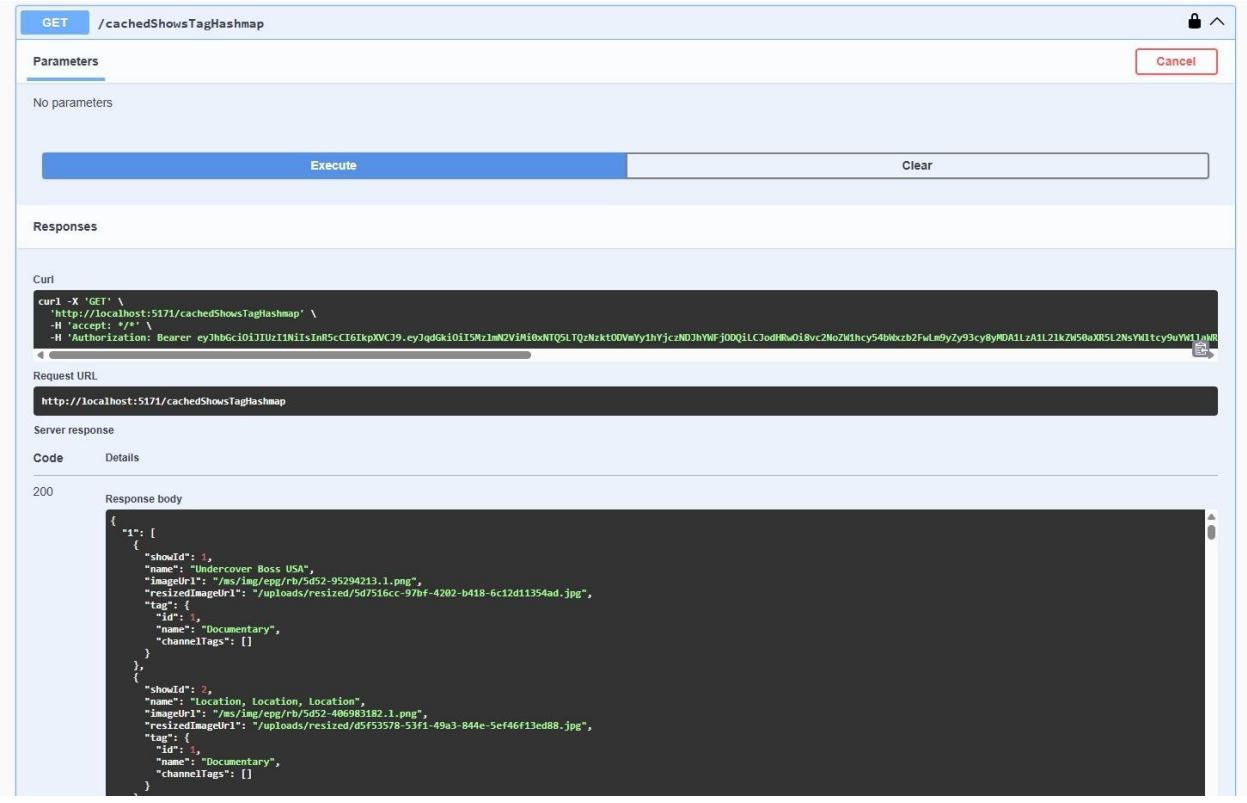
The successful demonstration of these operations confirms the effectiveness and accuracy of the implemented APIs.

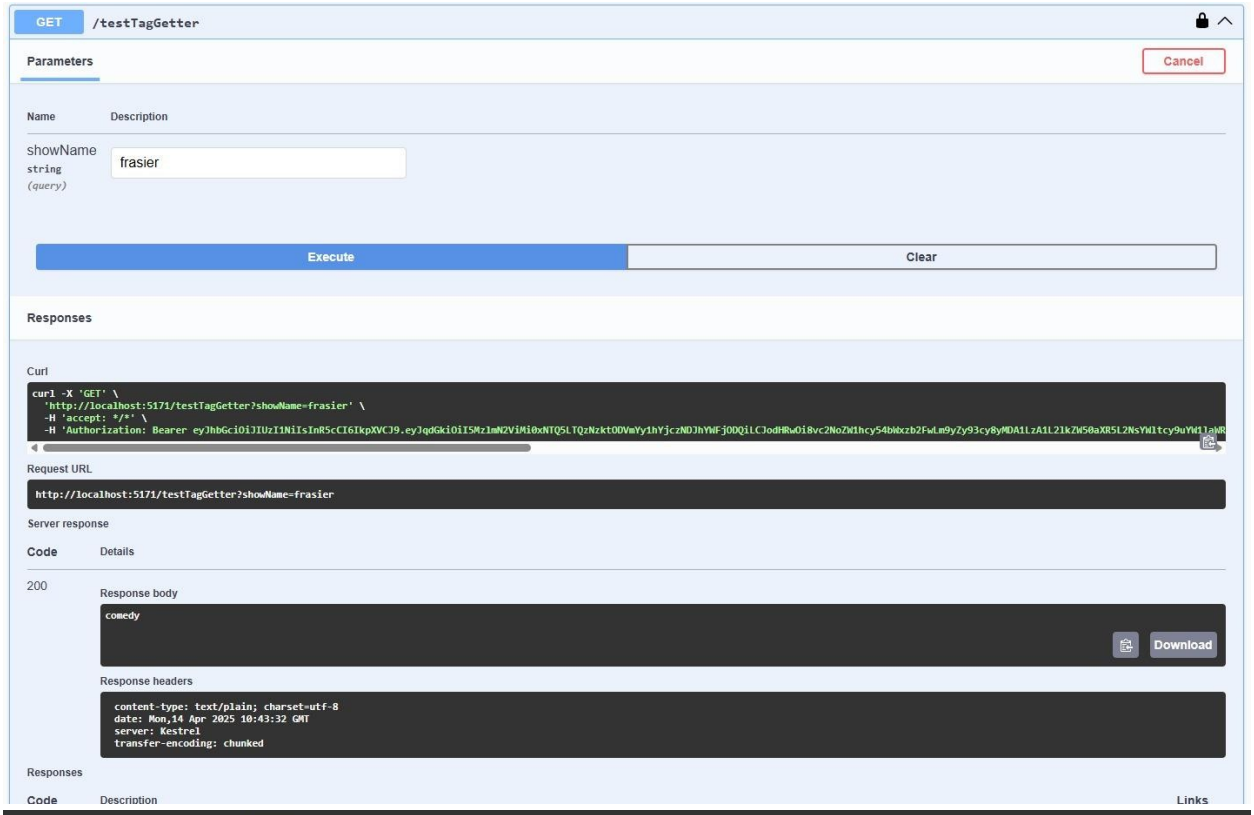


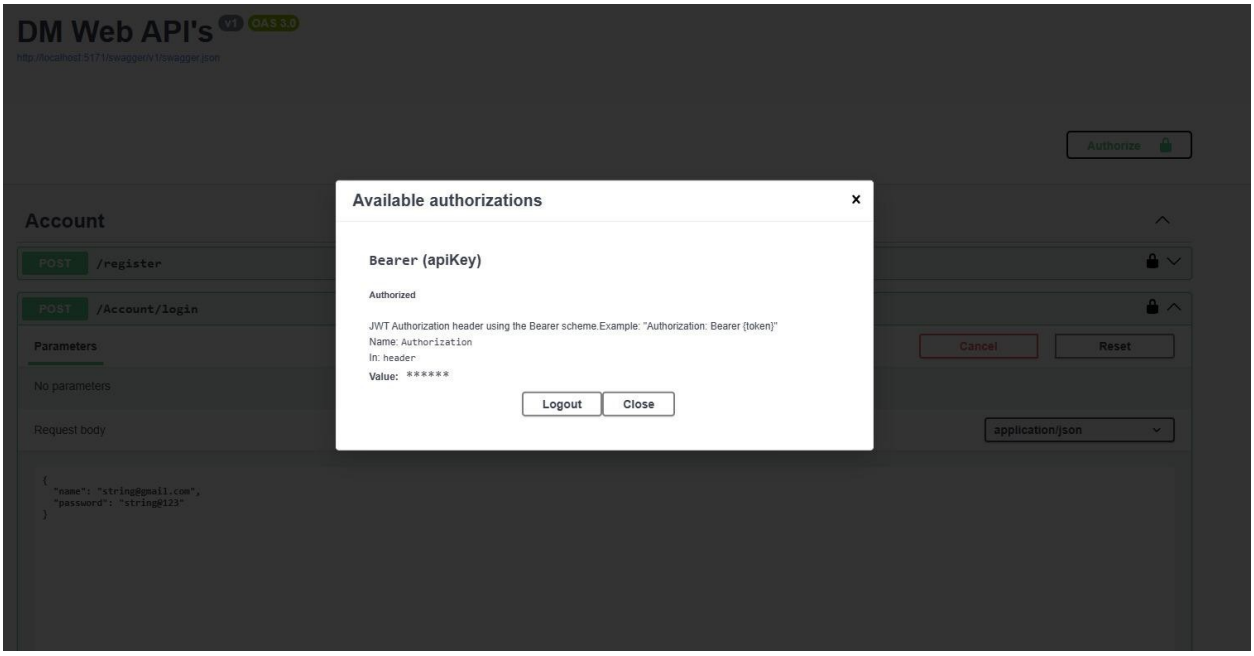

Execute
Clear
Responses
Curl
curl -X 'POST' \
'http://localhost:5171/Account/login' \
-H 'accept: */*' \
-H 'Content-Type: application/json' \
-d '{
"name": "string@gmail.com",
"password": "string@123"
}'
Request URL
http://localhost:5171/Account/login
Server response
Code
Details
200
Response body
Download
Response headers
content-type: application/json; charset=utf-8
date: Mon,14 Apr 2025 10:39:00 GMT
server: Kestrel
transfer-encoding: chunked
Responses
Code
Description
Links
200
OK
No links


GET
/main
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://localhost:5171/main' \
-H 'accept: */*' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiI5MzlmN2ViMi0xNTQ5LTQzNzktODVmYy1hYjczNDJhYWFjODQiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWR
Request URL
http://localhost:5171/main
Server response
Code
Details
200
Response body
"schedule": [
{
"userScheduleItemId": 98,
"showEvent": {
"showEventId": 27,
"channelId": 1540,
"description": "Young Henry has been hit by a forklift truck driven by his dad. Best mates Dean and Nahum are reeling from a car crash. And 94-year-old Pearl shares a tale of wartime r
omance. (S32 Ep2) [AD,S,W]",
"showId": 18,
"timeStart": 1742421600,
"duration": 3900
}
},
{
"userScheduleItemId": 97,
"showEvent": {
"showEventId": 3,
"channelId": 1540,
"description": "Colin Murray, Rachel Riley and Susie Dent host, with Katharine Merry in Dictionary Corner, as contestants race against the clock to pit their wits against vowels, conso
nants and numbers. [S]",
"showId": 3,
"timeStart": 1742363100,
"duration": 2400
}
},


GET
/checkShowsCache
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://localhost:5171/checkShowsCache' \
-H 'accept: */*' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiI5MzlmN2ViMi0xNTQ5LTQzNzktODVmYy1hYjczNDJhYWFjODQiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWR
Request URL
http://localhost:5171/checkShowsCache
Server response
Code
Details
200
Response body
[
{
"showId": 1,
"name": "Undercover Boss USA",
"imageUrl": "/ms/img/epg/rb/5d52-95294213.1.png",
"resizedImageUrl": "/uploads/resized/5d7516cc-97bf-4202-b418-6c12d11354ad.jpg",
"tag": {
"id": 1,
"name": "Documentary",
"channelTags": []
}
},
{
"showId": 2,
"name": "Location, Location, Location",
"imageUrl": "/ms/img/epg/rb/5d52-406983182.1.png",
"resizedImageUrl": "/uploads/resized/d5f53578-53f1-49a3-844e-5ef46f13ed88.jpg",
"tag": {
"id": 1,
"name": "Documentary",
"channelTags": []
}
},
{


GET
/cachedShowsTagHashmap
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://localhost:5171/cachedShowsTagHashmap' \
-H 'accept: */*' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiI5MzlmN2ViMi0xNTQ5LTQzNzktODVmYy1hYjczNDJhYWFjODQiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWR
Request URL
http://localhost:5171/cachedShowsTagHashmap
Server response
Code
Details
200
Response body
{
"1": [
{
"showId": 1,
"name": "Undercover Boss USA",
"imageUrl": "/ms/img/epg/rb/5d52-95294213.1.png",
"resizedImageUrl": "/uploads/resized/5d7516cc-97bf-4202-b418-6c12d11354ad.jpg",
"tag": {
"id": 1,
"name": "Documentary",
"channelTags": []
}
},
{
"showId": 2,
"name": "Location, Location, Location",
"imageUrl": "/ms/img/epg/rb/5d52-406983182.1.png",
"resizedImageUrl": "/uploads/resized/d5f53578-53f1-49a3-844e-5ef46f13ed88.jpg",
"tag": {
"id": 1,
"name": "Documentary",
"channelTags": []
}


GET
/testTagGetter
Parameters
Cancel
Name
Description
showName
string
(query)
frasier
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://localhost:5171/testTagGetter?showName=frasier' \
-H 'accept: */*' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiI5Mz1mN2ViMi0xNTQ5LTQzNzktODVmYy1hYjczNDJhYWFjODQiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWR
Request URL
http://localhost:5171/testTagGetter?showName=frasier
Server response
Code
Details
200
Response body
comedy
Download
Response headers
content-type: text/plain; charset=utf-8
date: Mon,14 Apr 2025 10:43:32 GMT
server: Kestrel
transfer-encoding: chunked
Responses
Code
Description
Links


DM Web API's
Authorize
Available authorizations
Bearer (apiKey)
Authorized
JWT Authorization header using the Bearer scheme.Example: "Authorization: Bearer {token}"
Name: Authorization
In: header
Value: ******
Logout
Close
Account
POST /register
POST /Account/login
Parameters
Cancel
Reset
No parameters
Request body
application/json
{
"name": "string@gmail.com",
"password": "string@123"
}

5.4 Conclusion:

The testing carried out validates the critical functionalities of the system comprehensively.

All tests pass as per the outlined expected results, demonstrating robust and reliable system behaviour across its main functional areas.

6. Project Management

Overview of Sprint Planning

The *TV Scheduling System* project adopted Agile Scrum methodologies to ensure a structured, iterative, and collaborative development process. The work was divided into weekly sprints, allowing the team to deliver incremental improvements while remaining adaptable to evolving requirements. This approach fostered transparency, accountability, and continuous feedback throughout the project lifecycle.

At the start of each sprint, the team held sprint planning meetings every Friday to review the current progress and define objectives for the upcoming week. During these sessions, tasks were discussed in detail, responsibilities were assigned based on team roles, and timelines were set to ensure measurable outcomes could be achieved within each sprint cycle. These meetings also allowed the team to reflect on what was working, identify potential risks, and reprioritize deliverables if needed.

Daily stand-up meetings further supported sprint execution by providing a space to communicate updates, raise blockers, and realign goals in real time. To ensure effective tracking before project management tools were introduced, Rudraa Patel (Secretary) maintained a detailed Excel spreadsheet capturing task assignments, completions, and notes for each sprint. This manual tracking system served as the primary management method in the early stages.

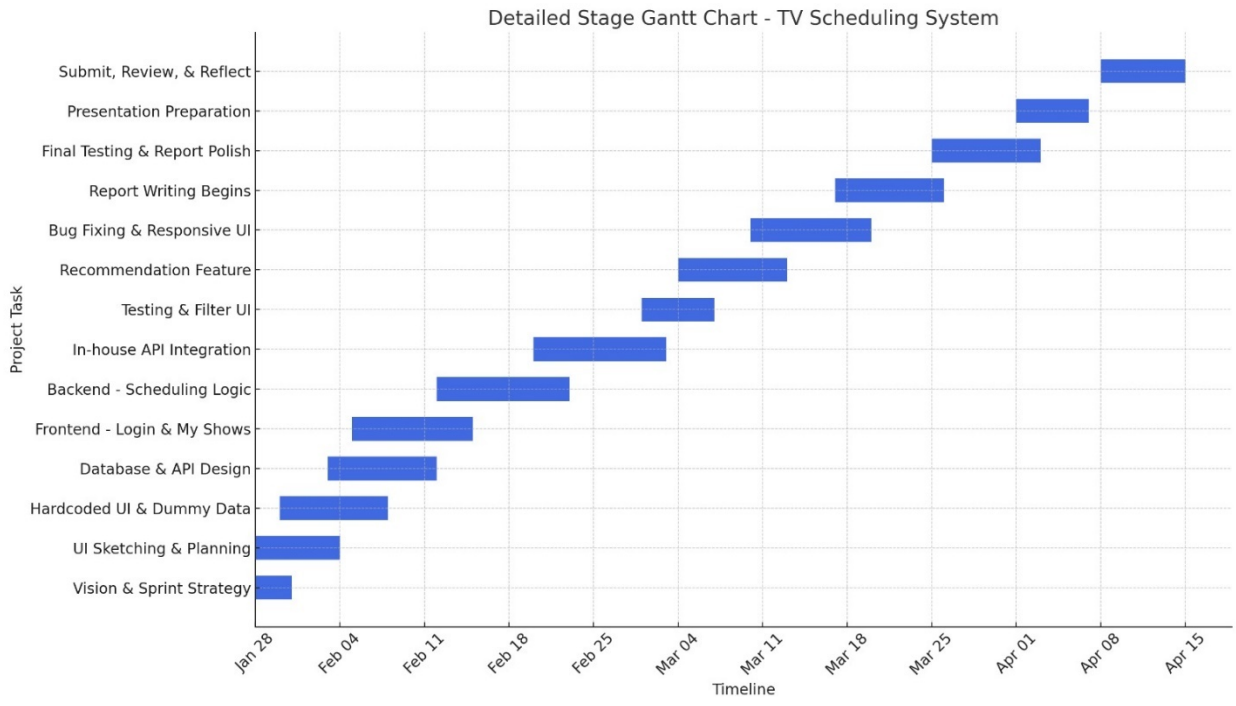
The team officially transitioned to Jira on April 16, which provided a more visual and collaborative environment for sprint planning and task monitoring. Due to the timing of its adoption, not all early project activities are logged within Jira; however, they are comprehensively documented in the Excel records maintained from project initiation. Once Jira was in use, it facilitated smoother tracking of sprint velocity, provided visual Kanban boards, and allowed for transparent progress updates through clearly defined task cards.

The overall sprint cycle followed a clear and logical development sequence. It began with defining the core vision and user experience goals, followed by the creation of initial wireframes and interface layouts. These early design efforts laid the groundwork for backend development,

including API integration and business logic. Once the backend was in place, the focus shifted to frontend implementation, where responsive and dynamic components were built using React. Throughout this process, continuous testing—particularly on the frontend—was performed to verify functionality, UI responsiveness, and component behaviour. Testing feedback guided further iterations, enabling the team to enhance system stability and overall user experience.

Gantt Chart

The project schedule was visualised using a **Gantt chart**, which helped track task distribution, sequencing, and dependencies across the development timeline. It provided a clear, week-by-week breakdown of planned activities such as research, wireframing, backend setup, frontend development, testing, and documentation. This structure ensured tasks were spaced realistically and allowed the team to manage deliverables in parallel when possible. The Gantt chart was especially helpful in identifying overlapping responsibilities and ensuring that no stage was rushed or overlooked.



Role-Based Task Allocation

Tasks: To optimise workflow and productivity, the team adopted a role-based task allocation strategy. Responsibilities were clearly defined early in the project to ensure accountability and streamline collaboration:

- Azul Debenedetti (Scrum Master & Frontend Developer): Led the frontend development of the system, focusing on building responsive components, routing, page structure, and UI logic.
- Rudraa Patel (Secretary): Managed the documentation, UI sketches, sprint notes, and Jira setup. Rudraa also tracked team activities in Excel and led financial planning for the presentation phase.
- Dominik Wujek (Backend Developer): Designed and implemented the backend API architecture, integrated external data sources (such as TV guide APIs), and built the recommendation engine.
- Hlaing Phyo Hein (Tester): Oversaw all backend testing efforts, ensuring API accuracy, system stability, and performance validation across multiple modules.
- Filip Domanski (Developer): Contributed to the login and registration functionality, supported backend and frontend where needed, and created project visuals like burndown and bar charts.

This clearly defined role structure helped eliminate confusion and fostered a sense of ownership over deliverables. It also ensured that team members could specialise in their areas of strength while contributing collaboratively to cross-functional tasks.

GitHub and Jira Management

The project's development workflow relied on GitHub for version control and Jira for task management and sprint tracking. These tools ensured consistent collaboration, code integrity, and progress visibility.

- **GitHub Repository**:

https://github.com/KitchenTile/TechSpire.git

GitHub was used for maintaining the full codebase, organising commits by branch and feature, and reviewing pull requests through peer collaboration. The team followed standard version control practices to avoid code conflicts and ensure a stable master branch.

- **Jira Project Management**:

https://live-team-fxjxomdy.atlassian.net/jira/software/projects/TVSCHD555/list

Although Jira was introduced on April 16, it became the team's central hub for managing the remaining sprint backlog, planning new tasks, and visualising task progress through Kanban boards. Each task in Jira was assigned a status, assignee, and sprint tag, allowing the team to monitor progress at a glance. Daily stand-up updates were also recorded in Jira following its integration.

Together, GitHub and Jira allowed the team to uphold a transparent and traceable development process. The combination of offline tracking (via Excel) and online collaboration (via Jira) ensured that no task was lost or unaccounted for, regardless of the phase of the project.

Burndown Chart

The burndown chart presented above provides a comprehensive view of the team's progress across the 12-week Agile development cycle of the *TV Scheduling System* project. It compares the **"Ideal Tasks Remaining"** (shown in blue) against the **"Actual Tasks Remaining"** (shown in red), offering a clear week-by-week representation of how closely the project stayed on track.

At the beginning of the project in Week 1, the team started with approximately 75 issues, representing the total number of planned tasks including design, development, integration, and testing. The ideal trajectory assumes a steady, evenly distributed workload completed each week, while the actual progress line reflects the real pace at which the team delivered tasks.

This chart is notable for how closely the actual task completion line follows the ideal projection. This alignment reflects several key strengths in the team's approach:

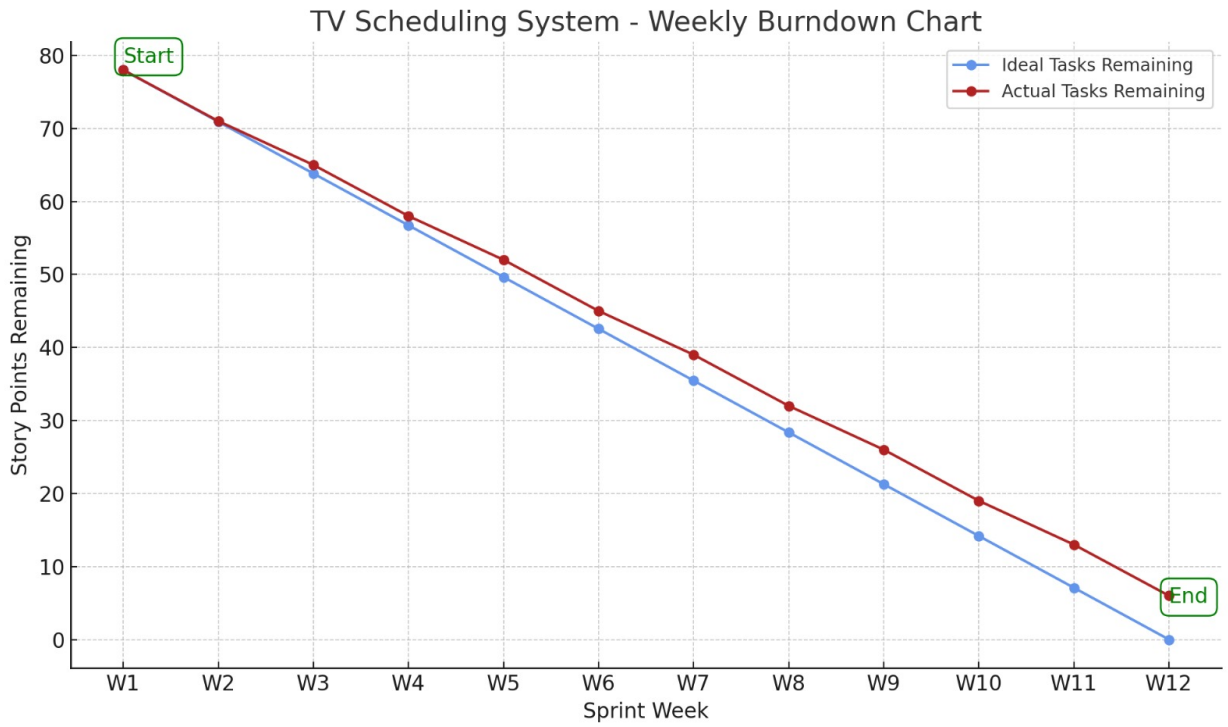
- **Accurate Sprint Planning**: Weekly goals were realistically scoped and well-distributed, avoiding both underutilisation and overextension of team capacity.
- **Consistent Development Velocity**: The steady slope of the actual progress line indicates that the team maintained a balanced and sustainable work rate throughout the project.
- **Effective Adaptability**: Minor variances observed during the mid-phase (around Week 6 to Week 8) were promptly corrected, showing the team's ability to resolve issues and recalibrate workload distribution.
- **On-Time Completion**: By Week 12, the actual progress line converges with the ideal trajectory, indicating that all planned tasks were completed on schedule, with no leftover backlog.

Supporting Management Practices

The accuracy of this burndown chart is underpinned by strong project management practices:

- **Daily Stand-Up Meetings** were conducted to monitor progress, resolve blockers, and ensure task alignment across all team members.
- **Weekly Sprint Planning Meetings**, held every Friday, allowed the team to evaluate the current sprint's outcomes and plan the tasks for the following week with clarity and focus.
- Before the team formally adopted Jira on April 16, task tracking and progress logging were meticulously maintained in an Excel sheet by Rudraa Patel, the team's secretary, which will be attached to the daily meetings folder. This manual record-keeping ensured that early sprint data was captured in full detail and seamlessly transitioned into Jira once it was implemented.

In total, the team created and tracked approximately 75 individual tasks, spanning backend API development, frontend page implementation, testing, integration, and documentation. The structured tracking—first in Excel and later in Jira—ensured every task was accounted for, contributing to a highly organised and transparent development process.



Meeting minutes

The Meeting Minutes covering all twelve sprints of the project are compiled into a single folder, which is available in the GitHub repository under the name meeting minutes folder. This folder is structured week by week, from *1 Week (28 Jan – 1 Feb)* to *12 Week (13 Apr – 17 Apr)*, and serves as a comprehensive log of our Agile development process. Each sprint section includes a clearly labeled header and an embedded PDF containing the full week's stand-up meeting records. These records capture individual team member updates, attendance, responses to daily Agile stand-up prompts (*what was done yesterday, plans for today, and any blockers*). The consistent formatting and chronological organization allowed the team to monitor progress effectively, identify and resolve issues early, and maintain alignment with sprint goals. This consolidated meeting log played a crucial role in retrospectives, report writing, and project evaluation, ensuring transparency and accountability throughout the development lifecycle.

Sprint Planning and Retrospective Explaination

Sprint planning for the project was initially conducted using Excel spreadsheets, where each week was carefully outlined with 6–7 core tasks and a few sub tasks, clearly mapped to team members based on their roles. These tasks included frontend development, backend implementation, testing, documentation, and design work. This structured planning allowed us to maintain weekly development goals aligned with our project milestones.

Although the tasks were bulk-imported into Jira on **16 April 2025**, they were originally created and maintained in Excel from **28 January to 17 April 2025**. The migration to Jira was done to consolidate project tracking and visualize sprint workflows. Despite the bulk upload date, each task retained its original planned context, which had already been executed during its respective sprint window.

To support both sprint planning and retrospectives:

- **Sprint Planning**: The weekly task breakdown served as the roadmap for each sprint, defining what needed to be achieved.
- **Sprint Retrospective**: Task statuses and comments (included in the Jira export) were used to reflect on what was completed, delayed, or carried forward.

Weekly PDF summaries were generated to reflect this structure, and are organized into clearly labeled documents representing each sprint cycle during the development period.

Conclusion

This burndown chart is a clear reflection of the team's disciplined approach to Agile project management. The near-perfect alignment between ideal and actual task progression demonstrates that the team not only planned effectively but also executed each sprint with consistency, communication, and accountability. It is a strong indicator of successful project delivery, showcasing the value of careful planning, daily coordination, and accurate task tracking in achieving defined outcomes.

Through effective project management practices, including rigorous tracking, transparent communication, and role-specific task allocation, the team successfully managed the project's scope, resources, and timelines.

7. Conclusion

The *ViewQueue* project marks a comprehensive execution of a fully interactive TV scheduling application that delivers on both functionality and user experience. Built entirely through a **code-first methodology**, the project showcases the importance of planning for scalability, performance, and maintainability from the very beginning. The frontend codebase reflects clean architecture practices, with reusable components, context-based state management, and optimised rendering through memoisation. Testing was implemented thoroughly, covering both backend APIs and frontend behaviour to ensure reliability and robustness in user interactions.

One of the project's standout achievements is the effective personalisation system, which uses genre-based user preferences to shape content discovery and recommendations. Modal components, dynamic routing, conditional rendering, and debounce-based search input collectively contributed to a smooth, highly responsive application. Key features like the 'My Shows' section, dynamic search results, and genre-based filtering were developed in tandem with continuous testing and visual iteration through Figma, ensuring the design aligned with the user journey.

Throughout development, the team operated within Agile Scrum cycles—holding regular sprint meetings to assign tasks, monitor progress, and adjust workloads based on evolving priorities. GitHub was used for version control and collaborative coding, while Jira managed task tracking, sprint backlogs, and burndown analysis. This methodology allowed the team to address real-world issues like performance bottlenecks from large media assets and to resolve them with innovative backend solutions, such as resizing and caching images server-side before serving them to the frontend.

In summary, *ViewQueue* not only meets the criteria of a successful software engineering project but also stands as a scalable, modern solution to a traditional problem. With planned future improvements like mobile-native versions and accessibility enhancements, the application is well-positioned for further growth. The entire development process—from planning and wireframing to coding and testing—helped sharpen the team's technical, collaborative, and problem-solving skills. The final product is not just a strong academic submission but a practical

portfolio piece that demonstrates industry-relevant skills in full-stack development, UI/UX design, and agile software management.

8. References

Frontend references

React Router (n.d.) *React Router: Declarative routing for React apps*. Available at: https://reactrouter.com/home (Accessed: 30 Jan 2025).

Kulesza, D. (2023) *Inside Fiber – in-depth overview of the new reconciliation algorithm in React*. Available at: https://angular.love/inside-fiber-in-depth-overview-of-the-new-reconciliation-algorithm-in-react (Accessed: 6 Feb 2025).

GOV.UK (2022) *Accessibility requirements for public sector websites and apps*. Available at: https://www.gov.uk/guidance/accessibility-requirements-for-public-sector-websites-and-apps (Accessed: 15 Feb 2025).

Shirsagar, A. (2023) *Understanding Referential Equality in React: A key to optimizing performance*. Available at: https://medium.com/@abhikshirsagar1999/understanding-referential-equality-in-react-a-key-to-optimizing-performance-45342c753dea (Accessed: 30 Feb 2025).

Backend References

Microsoft (2023) *Relationships in Entity Framework Core*. [online] Available at: https://learn.microsoft.com/en-us/ef/core/modeling/relationships (Accessed: 17 March 2025).

Microsoft (2023) *Caching in ASP.NET Core*. [online] Available at: https://learn.microsoft.com/en-us/aspnet/core/performance/caching/overview (Accessed: 15 Feb 2025).

OpenAI (2023) *OpenAI API Documentation: Chat Completions*. [online] Available at: https://platform.openai.com/docs/guides/chat (Accessed: 17 eb 2025).

Jira reference

Atlassian (n.d.) *Jira Software*. [online] Available at: https://www.atlassian.com/software/jira (Accessed: 10 April 2025).