FRAMES

PWA PORTFOLIO

Unlimited reviews of Movies,
TV, shows and more

Table of Contents

Project Requirements

Functional Requirements

- The Project's PWA format:
    - The project must be either a movie or a game review app, in the form of a PWA (Progressive Web App).
    - It must include Service Workers to allow the project to work offline.
- Accessibility
    - Visitors (those who have just accessed the site without interacting with it in any way) can register new user accounts, login, and post their own reviews.
    - Visitors that are not logged in, cannot post reviews. Reviews should include a review date, name of user reviewing, rating, and review text and more.
    - There must be procedures and algorithms in place for user creation, authentication and authorisation.
- Structure and internal design
    - The project is to be built using HTML, CSS, JavaScript, SQLite, and Python Flask (or equivalent).
    - The data should be stored in an SQL database, and must be interacted with by the PWA using SQL.
- Maintenance
    - There should be security measures in place to protect sensitive data such as passwords, etc. to prevent Cross-Site Scripting

Non-Functional Requirements

- The project is to be completed by 10:00 pm on the 16/12/2025
- The project is to showcase an understanding of the concepts of building and maintaining a Progressive Web App (PWA), and of various coding languages for front-end and back-end development, including HTML, CSS, Python, SQL and JavaScript.

Constraints

- The project had to be completed within the allocated school timeframe, due date being 16/12/2025 - Time 2200
- Application had to be hosted locally
- Team members worked remotely at times, requiring collaboration through GitHub

- The database system was limited to SQLite which was new to the team members at the time, limiting advanced user management features.
- External AI APIs and paid services such as Gemini were avoided to complete AI overviews of films, to ensure that the project remained self-contained and suitable for the assessment.

Acceptance Criteria

In order for this project to be of acceptable quality, the project needs to:

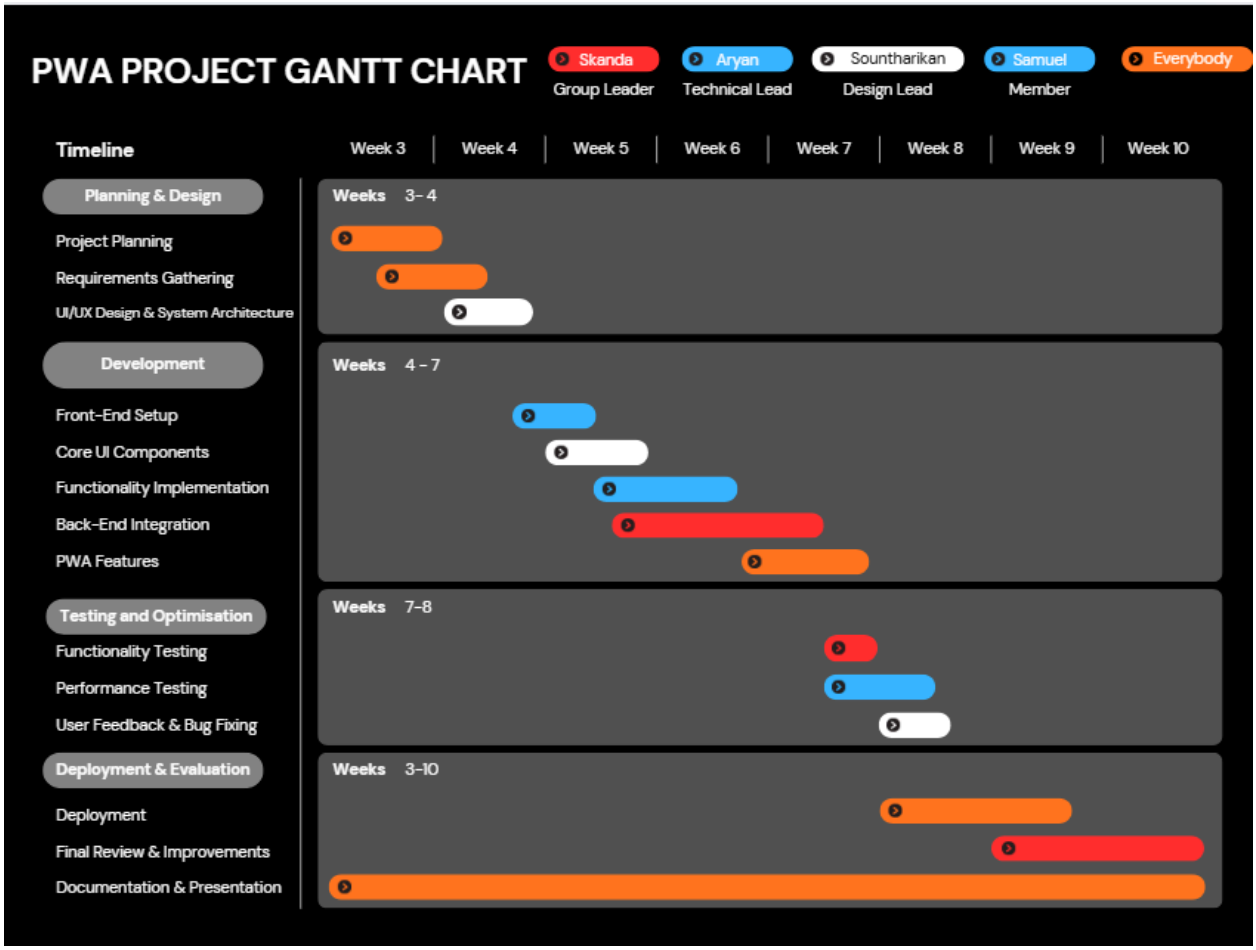
- Have a fully functioning and self intuitive, user friendly interface.
- Allow for people to register an account, and log in to that account, alongside logging out of and deleting your own account.
- Allow people to browse the movies, and apply filters for ease of browsing, and has the ability to allow users to search for movies, even when not logged in.
- Allow for people to add reviews (which should consist of a rating out of 10, a comment, and time of commenting), but only if you are logged in to an account.
- Allow people to see other's reviews and their profiles.
- Work across multiple platforms and device aspect ratios.

Planning

Gantt Chart

[Canva Link to Gantt Chart]

(GANTT CHART IS UPLOADED BELOW - IF RESOLUTION IS POOR, OR IF LINK IS BROKEN, A COPY IS ATTACHED IN FOLDER)


PWA PROJECT GANTT CHART
Skanda
Group Leader
Aryan
Technical Lead
Sountharikan
Design Lead
Samuel
Member
Everybody
Timeline
Week 3
Week 4
Week 5
Week 6
Week 7
Week 8
Week 9
Week 10
Planning & Design
Project Planning
Requirements Gathering
UI/UX Design & System Architecture
Development
Front-End Setup
Core UI Components
Functionality Implementation
Back-End Integration
PWA Features
Testing and Optimisation
Functionality Testing
Performance Testing
User Feedback & Bug Fixing
Deployment & Evaluation
Deployment
Final Review & Improvements
Documentation & Presentation
Weeks 3- 4
Weeks 4 - 7
Weeks 7-8
Weeks 3-10

IPO Chart

Input	Processing	Output
• Username/Email & Password	• Checks if there is such a user with the username. • Checks if the password matches the username	• If all match, it gives the user all the privileges of being a logged in user. • If they do not match, the user is denied login and the ability to post reviews.
• Search Query	• Checks through database for titles that match the Query	• If there are matches, it will display all the relevant titles that match. • If there are no matches, it will display none.
• Title choice & Consent to view comments	• Checks for details on the title. • Checks if there are any comments posted on the title	• Displays all the details of the listed title • Displays any and all comments and reviews posted on the title, when the user wishes to see the comments.
• Sign-up details • Email, Username, Password	• Check username length (<=16) • Check username for profanity (via better_profanity) • Check password length (>=8) • Validates email format using regex • Searches DB to see if username / email already exists • Hashes password	• If valid: account is created and saved in database • Redirects the user to /login • If invalid: shows an error message on signup.html (e.g. username too long, invalid email, user exists, etc.)

	with bcrypt + salt • Generates join date/time • Inserts new account into *AccountDetail* with defaults (PFP/title/bio)	
Account data submission (Create account request)	• Opens DB connection • Executes *INSERT INTO AccountsDetail (...)* • Commits changes as closes DB	• New row stored in *AccountsDetail* • Users can then log in with username or email
Login identifier + password (identifier = username OR email)	• Looks up the user by *Username=? OR UserEmail=?* • Retrieves stored hashed password + authority level • Verifies password using *bcrypt.checkpw()*	• If correct: session cookie is created and the user is directed to /browse • If incorrect: shows “Incorrect password” on login.html • If not found: shows "User not found” on login.html
Successful login event	• Stores *session[“user”] = username* • Stores *session[“level”] = authority level* • Session is kept client-side as a signed cookie (Flask session)	• User becomes “logged in” across pages • Pages can display the username / use authority level for permissions
Attempt to access protected pages (/browse, /top10, /movie, /profile)	• Checks *if “user” not in session:* • If missing, blocks access	• If not logged in: user is redirected to /login • If logged in: requested page loads normally

Browse page request	• Confirms session exists • Passes session username into template *(user=session["user"])*	• Renders browse.html • Shows user-specific UI text (e.g., "Welcome, {username}")
Profile page request	• Confirm session exists • Queries DB for the logged-in user's profile fields: Username, Email, PFP (Profile Picture), JoinDate, ReviewCount, Title, Description • If user record missing, forces logout to avoid broken session	• Renders profile.html populated with user info • If account is missing: redirects to /logout (clears session)
Logout request	• Clears all session values using *session.clear()*	• User is logged out immediately • Redirects to /login • User can no longer access protected pages until logging in again
App startup / database setup	• Runs *init_db()* • Creates *AccountsDetail* table if it doesn't exist	• Database is ready for signups/logins • Prevents crashes when app runs the first time
Username moderation (profanity filter)	• Loads censor word list *(profanity.load_censor_words())* • Checks input username against profanity list	• If it is flagged: signup is rejected with "Inappropriate username" • Helps enforce community standards

Backend		

Browsing movies (filtering and ordering)	• Validates the inputs against the allowed sets (genres / classification / year orders)
	• Builds SQL *WHERE* clause (e.g. classification)
	• Builds *ORDER BY* clause:
	  ○ Prioritises the chosen genre using a weighted CASE statement (P/S/T/Q)
	  ○ Sorts rating by DESC by default
	  ○ Optionally sorts by release year ascending/descending
		• Displays a list of movies ordered by "best match":
		  ○ ID, title, rating, genres, year, classification
		• If none found: shows "No movies found."

Search movies
• Search query (movie title text)
• Optional genre filters
• Optional classification filters
• Rating sort order (asc/desc)
• Year sort order (new→old / old→new)
	• Runs SQL *LIKE %query%* search on *FilmName*
	• If no direct matches:
	  ○ Loads all film names
	  ○ Uses *difflib.get_close_matches()* to find similar titles
	  ○ Searches DB using *WHERE FilmName IN (...)*
		• Displays matching movies as a list: *[FilmID] Title (Year | Classification)*
		• If no query entered: "Search cannot be empty."
		• If nothing matches: "No matching or similar movies found." / "No movies match your filters."

	• Applies filters:
	  ○ Checks if any of the movie's 4 genres match selected genres
	  ○ Checks classification match
	• Sorts filtered results by rating and/or year order	

View Movie List • User chooses "List all movies"	• SQL query: *SELECT FilmID, FilmName, FROM MovieList*	• Prints all movies with ID + name • If DB is empty: "No movies found in the database."

Adding a comment + rating
• Movie selected (FilmID)
• Rating (0-10, one decimal allowed)
• Comment text
	• Displays available movies from *MovieList*
	• Validates FilmID exists
	• Validates rating is numeric and between 0 and 10
	• Rounds rating to 1 decimal
	• Checks comment is not empty
	• Checks comment text for banned words
	• Inserts comment into *CommentList* with timestamp
	• Updates *MovieList:*
	  ○ Increments *FilmReviewCount*
	  ○ Recalculates
		• Reads "Comment added successfully!"
		• Movie average rating and review count updated
		• User review count updated + title may change.

	average rating using old average + count • Updates *AccountsDetail:* ○ Increments *UserReviewCount* ○ Calls *update_user_title()* to apply milestone titles	
View user statistics • Selecting a user	• Checks the UserID, and attempts to find a match in AccountsDetail. • Extracts these from AccountsDetail for the UserID match: ○ Username ○ Profile picture link ○ Join date ○ User title • Extracts these from CommentList for all instances of a UserID match: ○ MovieID ○ CommentRating ○ CommentContents ○ CommentTime • Extracts these from MovieList for all instances of a MovieID match: ○ FilmImageLin	Outputs all of the information with user profile picture, name, title, and join date at the top, with all their comments in chronological order from newest to oldest, with each comment showing the movie image, title, and comment rating, date, and contents.

	k ○ FilmName	
Delete Account button pressed, and user enters: ● Username/Email ● Password	● Searches AccountsDetail for a match of username/email and password. ● Retrieves the hashed UserPassword from AccountsDetail. ● Uses bcrypt to ensure entered password matches the password in the database ● Deletes user's record from AccountsDetail, displays a success message.	"Account deleted successfully" if successful. "No such user." if the username/email doesn't exist in the database. "Incorrect password", if password doesn't match.
<u>User registration</u>		
Database initialisation	● Opens SQLite connection to *PWAFramesDatabase.db* ● Runs *CREATE TABLE IF NOT EXISTS AccountsDetail (...)* ● Commits changes so that the table exists before any queries / inserts happen	● Database is guaranteed to have the *AccountsDetail* table ● The program avoids crashing from missing-table errors during registration / login.
Username profanity moderation during registration	● Loads profanity word list once *(profanity.load_censor_words())* ● Checks username using	● If inappropriate: message shown ("Username contains inappropriate language.") and requires the user to

	profanity.contains_profanity(username) • Rejects username if it contains inappropriate language	re-enter. • If appropriate: username passes onto the next validation step
Navigation + session-based UI • Current session state *(sesson.get("user"))* • User clicks: Home / Browse / Top10 / Profile / Logout / Sign in / Sign up	• Jinja checks if a user exists in session • If logged in: renders username + profile dropdown with kinks • If logged out: renders Sign in / Sign up buttons • Clicking links routes the user to the correct Flask route	• Correct navbar options displayed depending on login status • User can reach Profile / Logout if logged in, or Login/Signup if not
Browse page - movie grid browsing • User opens */browse* • User clicks "View Details" on a movie card	• Page displays a grid of movie cards (poster, title, year, star rating) • Clicking "View Details" loads */movie?id=<movie_id>*	• User sees a list of recommended movies in a grid • User is taken to the movie details page for the chosen movie
Movie page - URL query parameter selection • URL query string: *?id=inception* (or avatar, interstellar, darkknight) • Browser loads */movie*	• JS reads URL query: *new URLSearchParams(window.location.research)* • Extracts *id* • Look up that id in the *movies* object dataset	• If id exists: correct movie details are loaded • If id doesn't exist: displays "Movie not found."
Movie page dynamic movie details rendering (DOM injection) • Selected movie object (title, director,	• *renderMovie(movie)* builds a full HTML layout using template strings • Injects content into	• Movie poster + title + director/year + star rating + genres + synopsis appear on screen

year, poster, genres, rating, description)	* *	
Movie page - review form visibility toggle • User clicks "Write a Review"	• JS finds *#review-form* • Toggles the *hidden* class *(classList.toggle("hidden"))*	• Review form appears when requested, disappears when toggled again

Movie page - client side review submission (temporary, in-page)
• Author name
• Rating (1-5)
• Review text
• User clicks "Submit"
	• JS validates: checks for empty fields
	• If invalid: *alert("Please fill in all fields!")*
	• If valid:
	  ○ Pushes review into *movie.reviews []*
	  ○ Adds date via *new Date().toISOString().split("T")[0]*
	  ○ Re-renders the whole movie page *(renderMovie(movie))*
	  ○ Hides the review form again.
		• If invalid: user gets an alert and nothing posts
		• If valid: new review appears immediately in the reviews list (but only for the current session / page load)

Movie page - reviews list rendering (empty v populated)
• *movie.reviews.length*
• *Reviews array objects (author, rating, text, date)*
	• If *reviews.length === 0*: prints "No reviews yet."
	• Else: maps over reviews and generates formatted review blocks
		• Either:
		  ○ Displays "No reviews yet." message
		  ○ OR it shows a list of displayed reviews with the name, star

		rating, date, and text.
Horizontal scrolling in trending now gallery • User scrolls row manually OR clicks left/right arrows	• Poster displayed in an overflow-x horizontal container (*overflow-x-auto*) • Right arrow: *scrollBy({left: 300, behaviour: "smooth"})* • Left arrow: *scrollBy({left: -300, behaviour: "smooth"})*	• Users can smoothly scroll through the top 10 poster row.
Top 10 page -- arrow visibility • Current scroll position (*trendingRow.scrollLeft*) • Scroll events + page load event	• *updateArrows()* checks if *scrollLeft > 0*. • If yes: show left arrow • If no: hide left arrow	• Left arrow is hidden at the start, appears only after the user scrolls right
Profile page - server-rendered user stats display • Backend-rendered variables: *username, title, joined, bio, reviews, pfp* • User clicks "Log out"	• Jinja inserts variables into the profile layout • "Log out" routes to */logout*	• User sees their profile information and review count • Users can navigate to log out.

Storyboard

Below are the planned designs for the pages for the front-end:

Canva link to storyboard

(IF LINKS DO NOT WORK - DOWNLOADED COPIES ARE ATTACHED IN FOLDER)

Data dictionary

Variable	Data Type	Description
app	Flask	Main Flask application object.
app.secret_key	str	Secret key used to sign session cookies.
DB_Path	str	SQLite database file path (*"PWAFramesDatabse.db"*)
profanity	module/object	*Better_profanity* profanity filter instance (used to validate usernames).
Database connection + initialisation		
get_db()	An sqlite function	Opens and returns a new SQLite connection using *DB_PATH*.
init_db()	function	Creates tables if they do not exist: *AccountsDetail*, *MovieList*, *CommentList*.
conn	sqlite3.Connection	A connection object used inside DB functions/routes.
cursor	sqlite3.Cursor	Executes SQL statements for a given connection.
Database tables and fields -		
AccountsDetail	TABLE	Stores user accounts and profile data.
UserID	INTEGER (PK, AUTOINCREMENT)	Unique user identifier.
Username	TEXT (UNIQUE, NOT NULL)	Public username (limited to 16 chars in signup validation)
UserPassword	TEXT (NOT NULL)	bcrypt hash string of user password.
UserEmail	TEXT (UNIQUE, NOT NULL)	Email address (validated by regex at signup).
UserPFP	TEXT	Profile picture filename (default *"default.png"*)
UserJoinDate	TEXT	Datetime string when user registered (*"%Y-%m-%d %H:%M:%S"*)

UserReviewCount	INTEGER (DEFAULT 0)	Count of reviews/comments the user has posted.
UserTitle	TEXT	Display title/rank (default *"New User"*)
UserDescription	TEXT	User bio text (default *"No bio yet."*)
UserAuthorityLevel	INTEGER (DEFAULT 1)	Permission level stored in session on login (*session["level"]*)
MovieList (TABLE)		
MovieList	TABLE	Stores movie catalogue and derived stats (average rating, review count).
FilmID	INTEGER (PK, AUTOINCREMENT)	Unique movie identifier
FilmName	TEXT (NOT NULL)	Movie title.
FilmGenreP	TEXT	Primary genre
FilmGenreS	TEXT	Secondary genre
FilmGenreT	TEXT	Third genre
FilmGenreQ	TEXT	Fourth genre
FilmReleaseDate	TEXT	Release year/date string (e.g., *"2014"*)
FilmClassification	TEXT	Age classification (e.g., *G, PG, M, MA15+*)
FilmDescription	TEXT	Synopsis / description
FilmDirectors	TEXT	Director(s) string
FilmActors	TEXT	Actor(s) string
FlimImageLink	TEXT	Poster URL string
FilmRating	REAL (DEFAULT 0)	Average rating (recomputed using AVG(CommentRating))
FilmReviewCount	INTEGER (DEFAULT 0)	Review count field exists but is not currently updated in your POST logic.
FilmTrailerURL	TEXT	Its used in /movie SELECT as column 10, and hosts the links for all youtube trailers.

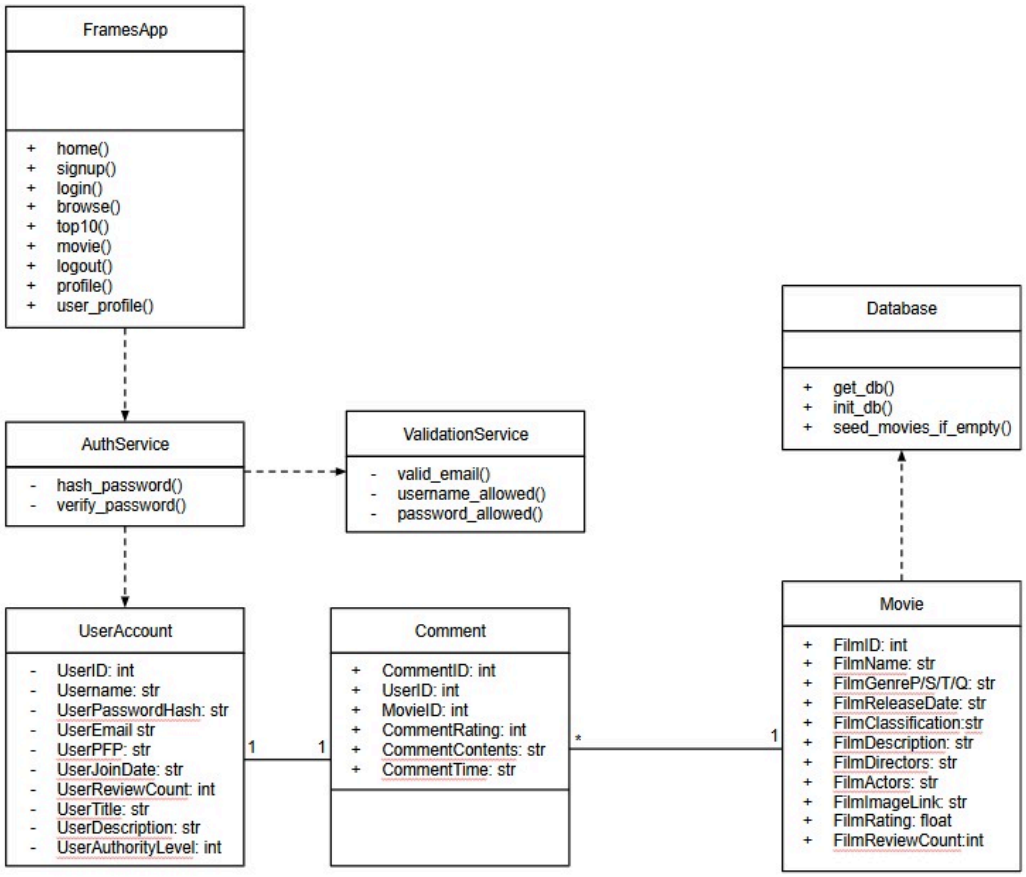
CommentList (TABLE)		
CommentList	TABLE	Stores user ratings + written comments per movie.
CommentID	INTEGER (PK, AUTOINCREMENT)	Unique comment identifier.
UserID	INTEGER (FK)	Foreign key referencing AccountsDetail(UserID)
MovieID	INTEGER (FK)	Foreign key referencing MovieList(FilmID)
CommentRating	REAL	Numeric rating a user gave the movie (you cast POST input to *float*)
CommentContents	TEXT	Written review content.
CommentTime	TEXT	Datetime string for when comment was posted.
Movie seeding variables (seed_movies_if_empty)		
seed_movies_if_empty()	function	Inserts a predefined list of movies into MovieList if the table is empty.
count	int	Number of rows currently in MovieList (*SELECT COUNT(*))*
movies	list(tuple)	List of movie tuples, each matching the INSErT column order (11 values).
cursor.executemany(...)	Method call	Bulk inserts seed movies into MovieList.
Flask request/session variables		
request	Flask request	Used to read form data (*request.form*) and URL query params (*request.args*)
session	dict-like (Flask session)	Cookie-signed session store for login state.
session{“user”]	str	Logged-in username (set on successful login)
session{“level”}	int	UserAuthorityLevel stored after login.
“User” in session	bool	Used as login-gate in */top10*, */movie* POST,

		/profile
Route level input variables and outputs /signup		
email	str	The email that the user inputs.
username	str	The username that the user inputs.
password	str	The password that the user inputs.
hashed	str	The password after its been hashed through bcrypt.
now	str	A string carrying the time of when the variable was defined. Format: YYYY-MM-DD HH:MM:SS
error	str	Carries a message that describes what type of error was encountered, such as “Password must be 8+ chars” and “Invalid email”.
Route level input variables and outputs /login		
identifier	str	The email or username the user inputs to login.
password	str	The password the user inputs to login.
user	tuple	As a result of cursor.fetchone(), it is a tuple with all the columns of the user’s row in the database.
username	str	The username of the user which has a match with who is logging in.
hashed	str	The stored hashed password in the database of the user.
error	str	Carries a message that describes what type of error was encountered, such as “Incorrect password” and “User not found”.
Route level input variables and outputs /browse		
query	str	The search string from the URL

genre	str	Genre filter from the url
sql	str	SQL string query
params	list(str)	Parameter list bound to SQL placeholders (LIKE and genre comparisons)
movies	list(tuple)	Query results: *(FilmID, FilmName, FilmReleaseDate, FilmRating, FilmImageLink)*
user	Str or None	Passed into template as *user=session.get("user")*
Route level input variables and outputs /top10		
movies	list(tuple)	Random 10 movies: (*FilmID*, *FilmName*, *FilmImageLink*) rows.
Route level input variables and outputs /movie (GET + POST)		
movie_id	Int or None	FilmID from URL param *id* (cast using *type=int*)
movie	Tuple or None	Full movie row returned by FilmID SELECT
comments	list(tuple)	Joined comment rows to render reviews
rating	float	Parsed POST rating: *float(request.form["rating"])*
comment	str	Parsed POST comment text: *request.form["comment"]*
user_id	Int or None	UserID
already_reviewed	Tuple or None	The result of "has this user reviewed this movie already?" check.
error	str	Template error message when user already reviewed the movie.
Route level input variables and outputs /logout		
session.clear()	Method call	Clears all session keys (hence, logs the user out)
Route level input variables and outputs /profile		

user	Tuple or None	DB row for logged-in user profile data
username	str	Taken from user[0] and passed onto PWA
email	str	Taken from user[1] and passed onto PWA
pfp	str	Taken from user[2] and passed onto PWA
joined	str	Taken from user[3] and passed onto PWA
reviews	int	Taken from user[4] and passed onto PWA
title	str	Taken from user[5] and passed onto PWA
bio	str	Taken from user[6] and passed onto PWA
/user/<username>		
username (route param)	str	Username taken from the URL path.
user	Tuple or None	DB row for that user's profile data.
reviews	list(tuple)	List of that user's reviews joined with movies: *(FilmID, FilmName, FilmImageLink, CommentRating, CommentContents, CommentTime)*.

UML Diagram

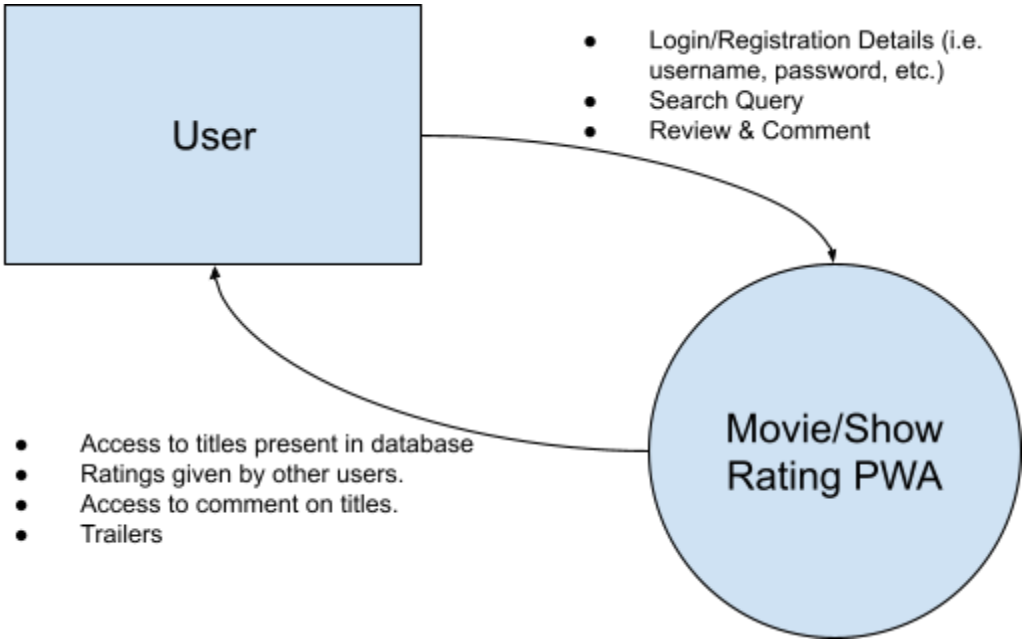


Flowchart

[Lucidchart link to Flowchart]

(IF FLOW CHART LINK IS BROKEN PLEASE, CHECK FOLDER WHERE THIS DOCUMENT IS LOCATED)

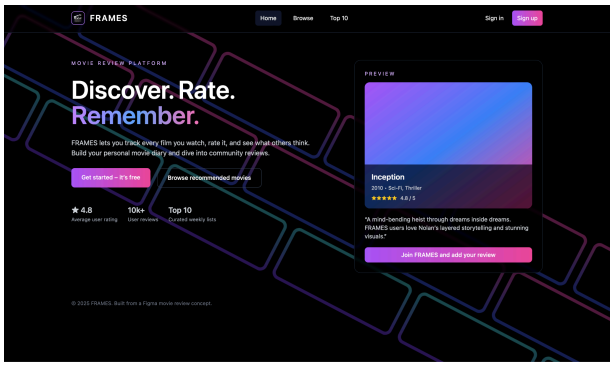
Context Diagram

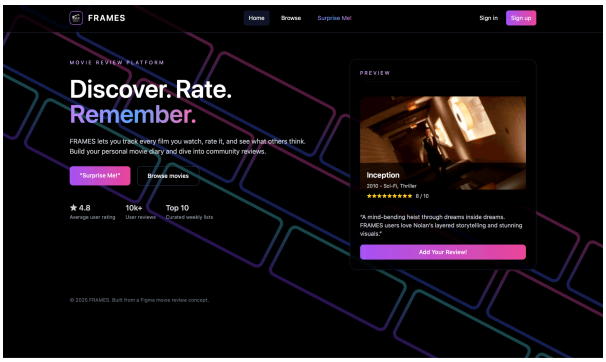


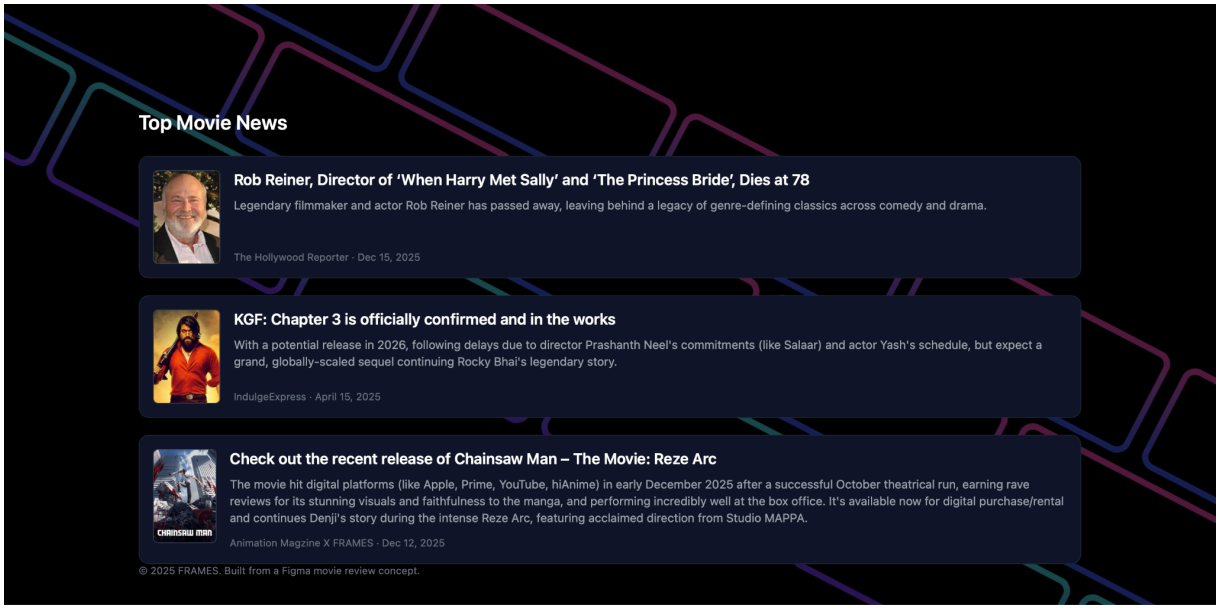
Implementation

Website Aspect

Landing Page

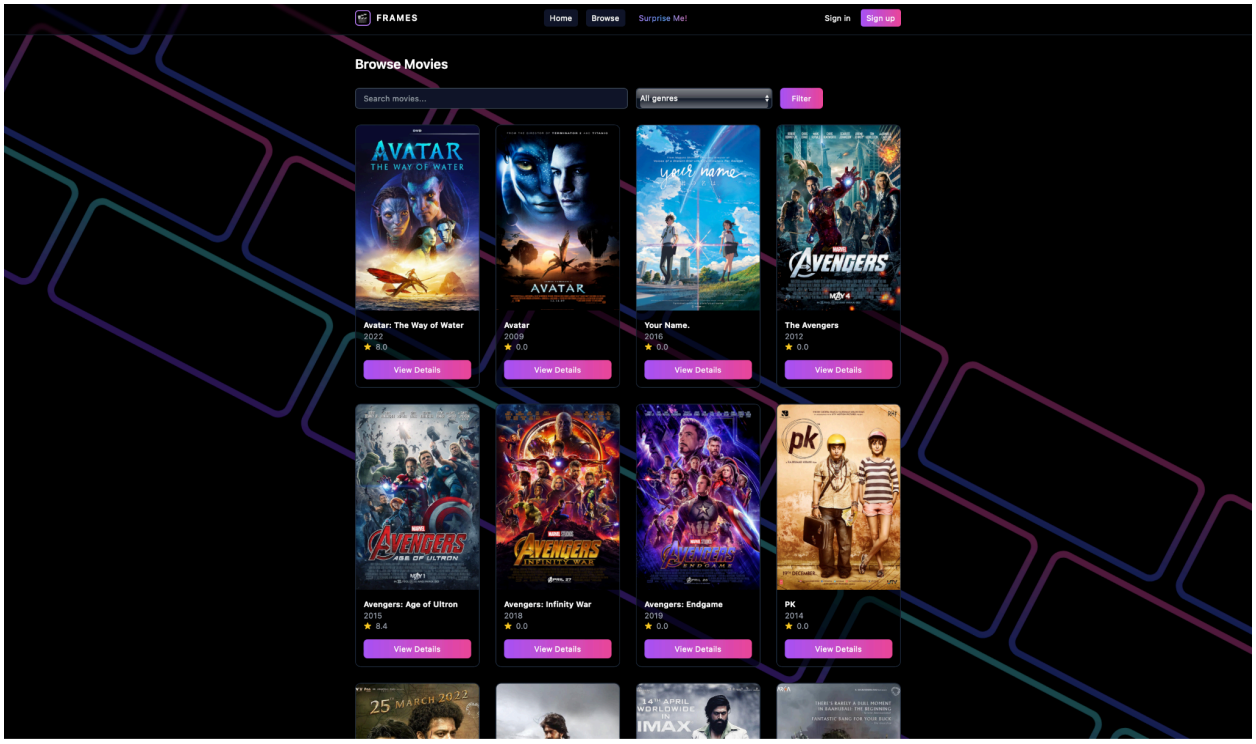






- This landing page design is somewhat similar to the original planned design, but has been designed to appear more spacious and have more navigation options, with the images above showcasing how the page has changed from the first functioning, runnable version to the more recent versions.
- The top right shows an animated GIF of a scene from the movie *Inception*, and
- The bottom image above shows the addition of relevant news articles on movies and related media to further educate users on current affairs within the industry.

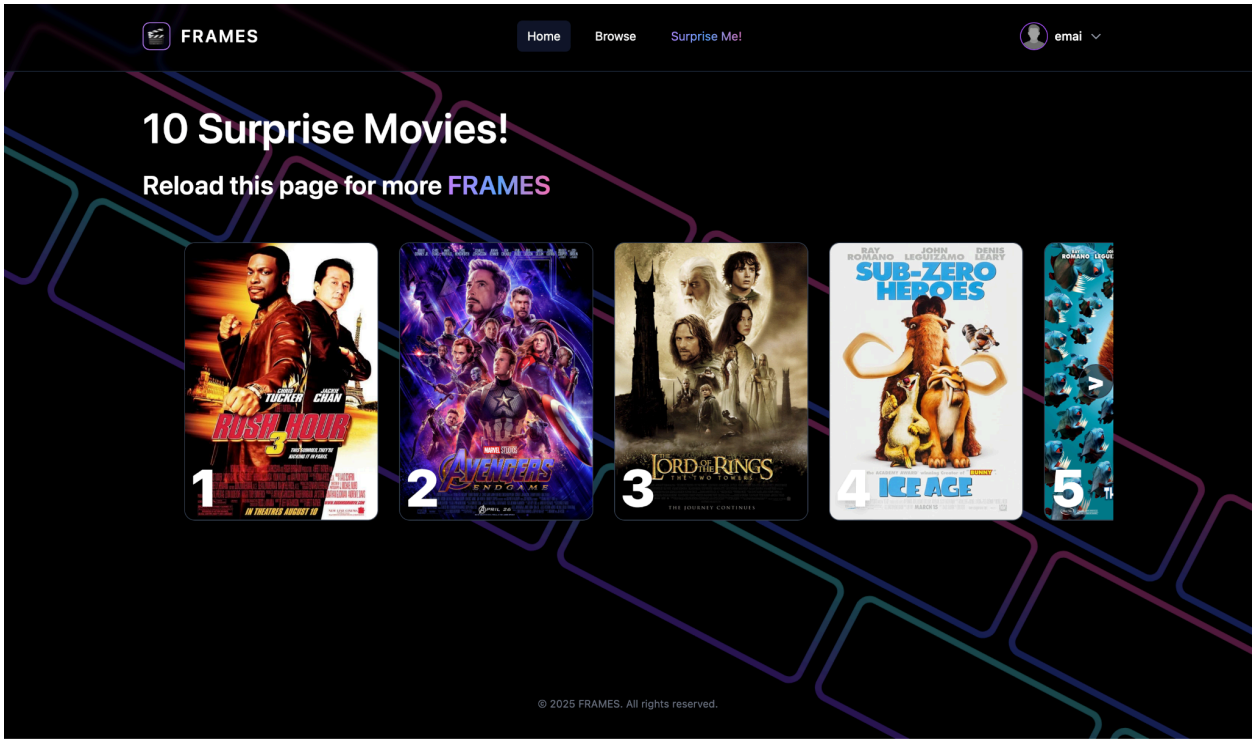
Browsing Page



(Note the above image is zoomed out)

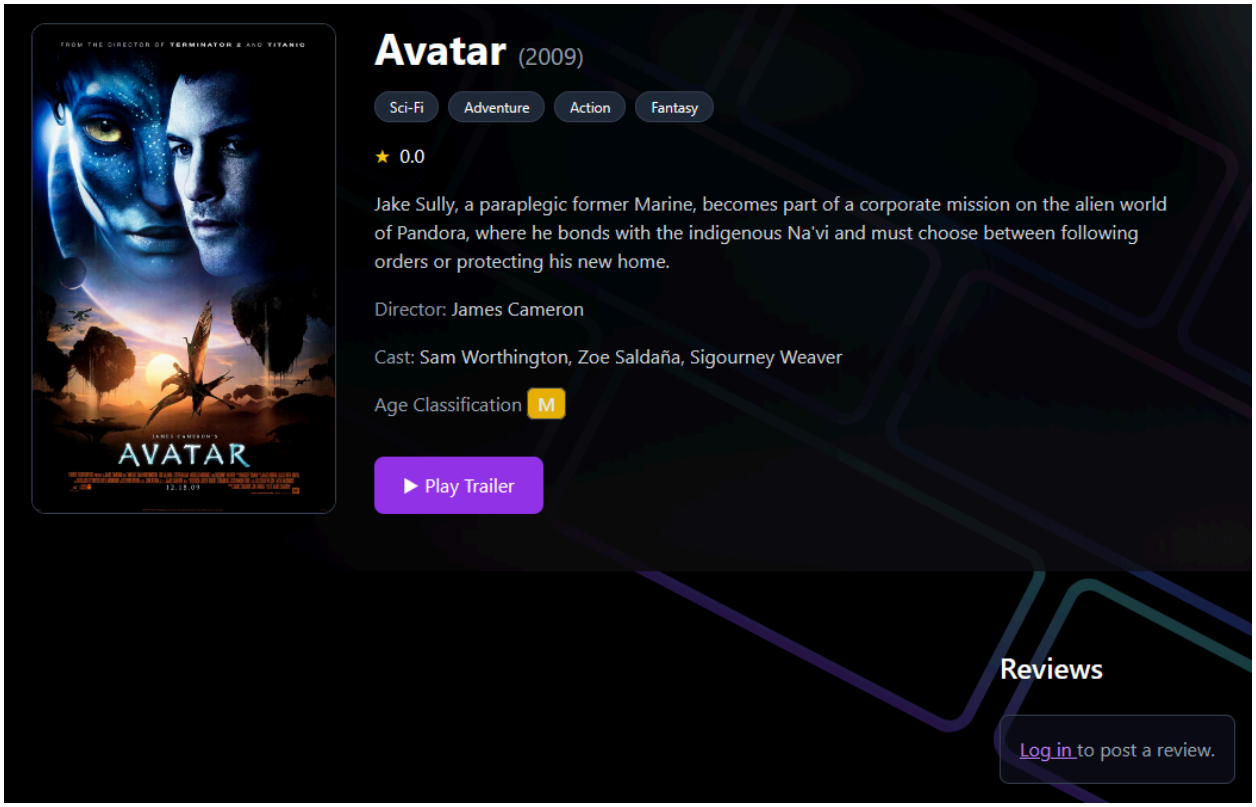
- Each of the individual movie review pages can be accessed by clicking the “View Details” button below the film.
- The page is in a gallery format, in contrast to the scrollbar used for the Top 10 page, which was made to be this way to make the pages more appealing and unique, and for ease of access to each title, making the integration with the search feature much more smooth.
- The browse menu also includes search filters such as the genre, to allow users to narrow down titles to rate, and the name, if a user is searching for one specific title.

Surprise Page



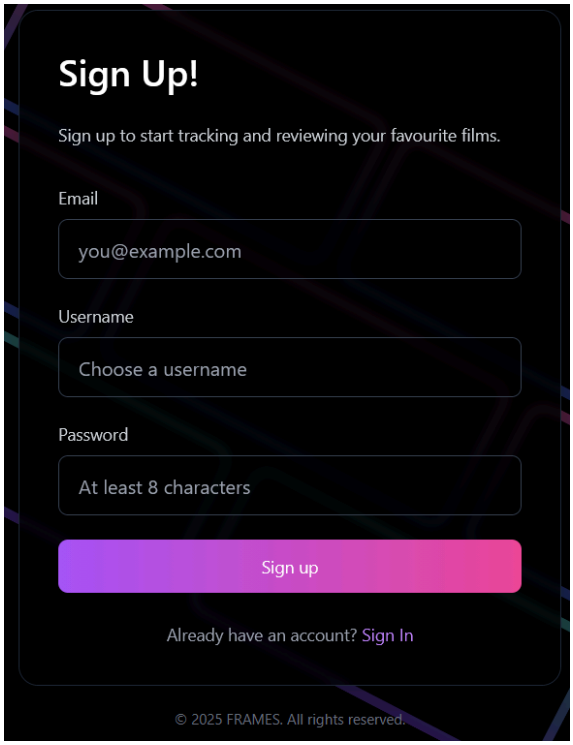
- This page was appropriated from the top 10 movie list that was made on the storyboard. When refreshed, the page displays a different assortment of movies
- This page has horizontal scrolling, as intended according to the storyboard plan.

Movie Page



- The movie page showcases the movie poster on the left-hand side; the title of the movie; the year it was created; the genres it falls under; the star rating; the description; director; cast; age classification; button that sends you to the trailer link; and option to post reviews.
- When logged in, users are able to post their reviews and give their own star ratings.
- The app also takes an average of all the ratings posted and makes the average visible when browsing and when on its page.

Login/Signup Page



- On this page, users are prompted to enter their email, username, and password for account creation.
- If the user enters an inappropriate word for their username, profanity filters are in place and prevent the account from being created.
- The username only allows between 3 to 16 characters in total and the password accepts passwords of greater than 8 characters.

Code Snippets

Password Hashing Script (Using Bcrypt)

```
        hashed = bcrypt.hashpw(password.encode(),
bcrypt.gensalt()).decode()
        now = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        cursor.execute("""
        INSERT INTO AccountsDetail
        (Username, UserPassword, UserEmail, UserPFP, UserJoinDate,
UserTitle, UserDescription)
        VALUES (?, ?, ?, ?, ?, ?, ?)
        """, (username, hashed, email, "default.png", now, "New User", "No
bio yet"))
```

Hashing passwords allows for the full security of password handling and storage, by using BCRYPT which is an industry-standard hashing algorithm, preventing database leaks from exposing passwords.

SQL Database Design with Foreign Keys

```
    CREATE TABLE IF NOT EXISTS AccountsDetail (
        UserID INTEGER PRIMARY KEY AUTOINCREMENT,
        Username TEXT UNIQUE NOT NULL,
        UserPassword TEXT NOT NULL,
        UserEmail TEXT UNIQUE NOT NULL,
        UserPFP TEXT,
        UserJoinDate TEXT,
        UserReviewCount INTEGER DEFAULT 0,
        UserTitle TEXT,
        UserDescription TEXT,
        UserAuthorityLevel INTEGER DEFAULT 1
    )
```

Using foreign keys ensures that the reviews will ALWAYS link to valid users and movies, additionally it prevents broken data. This is a proper database, where data isn't simply just "dumped" and not managed.

Preventing Decimal Abuse (Input Validation)

```
        try:
            rating = int(request.form["rating"])
        except ValueError:
            return render_template(
                "movie.html",
                movie=movie,
                comments=comments,
                error="Rating must be a whole number between 0 and 10."
            )
```

This forces only whole numbers to be entered. Previously a test was done where exploits were asked to be discovered and someone had typed in 1.100000003 as a rating, and it disrupted the pleasantness of the website, this led to us preventing decimal ratings, and keeping everything to integers.

```
        <!-- RATING -->
        <div class="flex items-center gap-2 text-yellow-400">
          ★ <span class="text-white">{{ "%.1f"|format(movie[3] or 0)
}}</span>
        </div>
```

This also cleans up the existing data, before the python was integrated into our program to prevent new bad reviews, utilizing defensive UI design, so that even if bad data were to occur it wouldn't affect the UI.

Preventing Duplicate Reviews

```
        # CHECK IF USER HAS ALREADY REVIEWED THIS MOVIE
        cursor.execute("""
            SELECT 1
            FROM CommentList
            WHERE UserID = ? AND MovieID = ?
        """, (user_id, movie_id))

        already_reviewed = cursor.fetchone()
```

This feature prevents a user from mass uploading reviews, spoiling the purpose of a review website, hence improving data reliability which is important for software engineering applications, as it is a real constraint used by IMDb, and other review sites.

Ad integration into Review Site (Monetisation Concept)

```
      <!-- LEFT AD -->
      <aside class="hidden lg:block">
        <div class="sticky top-28">
          <a href="https://raidshadowlegends.com/"
            class="block h-[600px] rounded-xl overflow-hidden border
border-gray-800
                   hover:scale-[1.01] hover:shadow-xl transition">

            <img
              src="../static/ads/sample-ad1-300x600.png"
              alt="Sponsored Ad"
              class="w-full h-full object-cover"
            >

            <span class="absolute top-2 right-2 text-xs text-gray-400
bg-black/70 px-2 py-0.5 rounded">
              Sponsored
            </span>

          </a>
        </div>
      </aside>
```

If the website were to be monetized later, ads are a good place to start. This was industrial-standard 300 x 600 banner ads. They are hidden on mobile for a neat minimalist style, and to minimize frustration, but it's visible on desktop. They exist without breaking the scroll UX, making the PWA usage as pleasant as possible.

Profanity checker (Input validation)

```
        if profanity.contains_profanity(username):
            return render_template("signup.html", error="Inappropriate
username")
```

If a username is deemed inappropriate by the profanity checker API that was implemented, the user would not be able to create that username and use it on our website. It improves autonomous community moderation and promotes safe user-generated content.

Data Security Measures

Data is kept very securely on the PWA, using tactics such as encryption and limiting access to certain segments of information to prevent the leakage of sensitive information.

- Encryption of Data
    - The Progressive Web-Application employs the use of hashing to encrypt sensitive data, such as the passwords to access accounts, using the bcrypt library on python, which is a well-known platform for hashing and making data unreadable when intercepted.
    - The encrypted data is what is stored in the databases to prevent leaking and if such information were to be compromised, the data would be encrypted.
- Authorisation firewalls
    - Contents of the database are strictly controlled by users that have administrative (admin) permissions, not by those with only moderator access. This data is also stored within the database in hashed form, so when data is taken from it, it is already encrypted.
    - Accounts follow the principle of least privilege, with new accounts having basic access and only being able to gain trust and become a possible candidate for a moderator or admin through writing large quantities of well-received reviews.
    - For an account to be able to interact directly with databases, a user must be an admin, and they may only receive such a position by being entrusted with the role by the original admin(s).
    - There is currently only one admin account to keep admin activity easier to be tracked, which is the only account which is able to interact with the database, and even the admin is not able to see the passwords of accounts, because of their encrypted nature, and only their emails are visible.
- Removing loose ends
    - Data that is no longer needed, such as that of a deleted account, is immediately wiped from the database, including emails, passwords, etc.

This system ensures the PWA is secure and data stays out of the hands of hackers and, if leaked, the data is unreadable, thus making it safer to use without risking exposure of personal information for users.

Logbook

Date	Sountharikan	Skanda	Samuel	Aryan
16/10/25 - 28/10/25	> Brainstormed ideas for the site. > Contributed to Gantt Chart to outline what is to be done at different points in time. > Started working on some pages for the design of the pages by creating a shared Canva.	> Brainstormed ideas for the site. > Contributed to Gantt Chart to outline what is to be done at different points in time. > Created the GitHub Repository for everyone's work to be uploaded. > Started working on some CSS to be implemented for each of the pages.	> Brainstormed ideas for the site. > Contributed to Gantt Chart to outline what is to be done at different points in time.	> Brainstormed ideas for the site. > Contributed to Gantt Chart to outline what is to be done at different points in time. > Started working on a Microsoft Access Database for the project.
28/10/25-10/11/2025	> Continued refining page layouts in Canva. > Explored converting Canva designs to Figma and then to HTML/CSS	> Created the Github repository, and added everyone.	> Started adding features to Sountharikan's design Canva. > Explored converting Canva designs to Figma and then to HTML/CSS	> Continued on the microsoft access database, now connecting it with a python file in order to test some queries.
11/11/2025 - 15/11/2025	>Committed multiple versions of the front-end, ensuring	>Made changes to the base CSS, customizing core features to make it more	>Worked on top 10 HTML, fixing up issues from the unpolished figma design to HTML.	>Committed multiple versions of the backend

	navigation was flawless.	aesthetically pleasing.		
16/11/2025 - 22/11/2025	>Integrated front-end pages with Flask Routing. >Improved responsiveness across all devices.	>Began writing sections of the documentation and portfolio. >Created Diagrams to Support System Explanation.	>Assisted with testing navigation and page loading. >Provided usability feedback.	>Implemented core backend features for Sountharikan to adapt and modify.
6/12/2025 - 12/12/2025	>Finalized user review with Aryans database. > Ran review programming that was working with Aryan's backend.	>Reviewed progress of the entire system to ensure that the project met all guidelines. >Contributed to portfolio sections	> Tested what was made so far. > Made significant contributions to the portfolio.	>Completed more of the backend functionality. > Researched and added all 68 movies into the database with all columns.
13/12/2025 - 16/12/2025	>Integrated Google Forms Contact Section into website. >Stopped Decimal Abuse >Finalized all other means of data validation. >Utilized Aryans delete user function (Users can delete themselves)	>Reviewed final documentation and ensured alignment with the project. >Assisted with final testing.	>Ran compliance tests, assisting with the final testing. >Aided Skanda with documentation and portfolio work.	> Finished a few more functions of the backend, namely, updateUserTitle, editProfile(), and ViewStatistics(). > Made a commit on the 13th, so Sountharikan could integrate these last few functions.

Self & Group Evaluation

Sountharikan

Throughout this project, I mainly focused on the aesthetics of the front-end trying to mimic streaming websites, and this one project that I've worked on in year 9, the fairy tale website, with its grid-like background and neat design. I've also had experience with flask and tunnelling applications used to host websites/stream videos as I've used it for my bicycle safety project (Project R.A.S). Initially the project seemed straightforward, but while developing the complexity has increased significantly, particularly when modifying and adapting the database Aryan has developed, such as adding a trailer URL column. One of the biggest challenges with this project was balancing it with other assessments and my outside of school FTC Robotics, which caused some delays towards the start of this project. However once FTC robotics had ended, and my other school assessments had finished, I was able to work more consistently and commit more often, with effective changes over the last few weeks. Overall, this project significantly improved my understanding of Flask's Jinja templating, web hosting workflows, collaborative coding using GitHub, and modular integration between front end and back end modules. . I am satisfied with both the final product and the teamwork that me, Skanda, Samuel and Aryan have exercised.

Skanda

The project started out seeming quite simple, although not much was known on certain aspects like how to use GitHub and how to integrate code from different coding languages. One of the main challenges, especially towards the mid-late end of the term, was finding the time to work on the project amidst the myriad of assessments and HSC exams. Despite all of that, once the exams started to finish up, we were able to be a lot more productive and get more of the project done. Maybe it would have been less stressful towards the end if I had planned more effectively and been better at enforcing the mini-deadlines for individual aspects of the project to be completed, but it went surprisingly smooth towards the end. With everyone having completed all other assessment tasks for the term, we were able to get more work done on the project with less distractions.

Samuel

In the end, we were able to complete the project to a more than satisfactory level of effort. The final product showed a consistent design scheme with a set and easy on the eyes colour scheme. The backend integration of the movies, comments, and accounts work and the website is able to retain information across different parts of the website. I started a little

late into the project but I was able to find my place in creating some of the initial designs for the project along with coding the top10 feature. One of the major challenges we faced was knowing how to integrate multiple types of languages with one another and time-management. We found ourselves lagging slightly behind schedule according to the Gantt Chart which meant that some features had to be rushed over or not utilised at all. If I were to do one thing differently, it would be to ensure that we delegate our time more efficiently at the beginning of the task and that we have a clearer plan on the features to be integrated as well.

Aryan

This assignment seemed somewhat trivial at first, but upon commencing, we soon encountered numerous issues. For example, I had originally made the database on Microsoft Access, but that doesn't work well with github or flask, hence I had to shift it over to SQLite. But SQLite had very difficult to work with UI, so I had to make the database through python itself, by forming a connection with the database in order to make changes to it. At the start, progress was slow, for we had foolishly assumed that time was in our hands. However towards the middle of the term, it seemed as though progress was cut off entirely, for we had a sudden spike in priorities, which mainly consisted of exams. While we did manage to finish the project with relatively little stress, and had a complete product that worked, I feel as though if we had been more wary of time's innate nature to slip free from our grasp, we would have been able to have a much better project than what we ended up with. I had, for the most part, dealt with all the backend, and the database, except for the flask section, and feel as though my comprehension in SQL and python has greatly increased over the duration of this task, and I am glad to have gotten the chance to work on this project. I believe that our teamwork was good, and we had good synergy in that everyone knew what had to be done. Overall, project and teamwork was good, time management was iffy at the beginning and middle, but towards the end we became very time efficient.

Group

As a group, we believe that we did well in this assignment, as we worked well together, with good communication and balancing work allocation. Although the main challenge was the imbalance of available time throughout the majority of the time given within the term, due to the presence of other exams and assessment tasks from other subjects. Despite this however, we did manage to overcome this challenge, by effectively managing our time towards the later segments of the project, after the stress from other exams had died down. Overall we worked well together, but our time management could have been better if we had started earlier and consistently managed our time and stayed up-to-date with all

mini-deadlines for individual tasks of the assignment, and we will keep this in mind for future tasks.

Statement of Authenticity

Statement of Authenticity and Academic Integrity

Names: Sountharikan.Thirukkumaran, Aryan.Patel, Samuel.Naicker, Skanda.Nagendra
Class: Software Engineering - HSC 2026
Teacher: Mrs Saki
Subject: HSC Software Engineering
Assessment: HSC Task1 - Progressive Web App Development

I certify that: the planning, development, content and presentation of this assessment task is my own work in every respect this assessment task has not been copied from another person's work or from books or the internet (including AI) or any other source I have used appropriate research methods and have not used the words, ideas, designs, music, images, skills or workmanship of others without appropriate acknowledgement in the assessment task or its development I have read, understand and have followed the assessment policies outlined in the assessment policy book.

Student Signatures 

Date: 16/12/2025