

# Project-Description

## Introduction and Project Goal

The goal of this project is to develop a social networking platform for movie and TV show enthusiasts, blending social media features, recommendation systems, and external data integration (such as TMDb). Existing movie review and recommendation platforms like Netflix and IMDb are generally closed ecosystems with limited real-time user interaction, and their recommendation systems are often not transparent or customizable for user needs. This application will allow users to rate, review, and share opinions, track friend activities, receive personalized recommendations, and actively participate in a film-loving community.

## Problem Analysis

Modern users of streaming platforms want to share their viewing experiences with friends or a wider online audience. Key problems being addressed include: - The closed nature of streaming and review platforms (Netflix, IMDb) prevents meaningful user interaction. - There is no straightforward way to follow what friends or influential users are watching. - Recommendation algorithms are often not transparent or sufficiently personalized. - There is a need for a place where recommendations originate from a mix of personal taste, group trends, and genuine social engagement.

This project will solve these through: - User profiles with options for branding and choosing preferred genres. - Real-time tracking/commenting on activity from other users. - Both collaborative and content-based recommendations plus manual watchlist additions. - Automated data import from TMDb for up-to-date film and show information.

## Potential Benefits

This platform delivers multiple benefits: - Enables users to express and share their film tastes and personal recommendations. - Encourages the development of communities built around shared interests. - Enhances recommendation quality through advanced algorithms and social engagement. - Opens possibilities for advanced analytics (viewing statistics, trending topics). - Acts as a space for experimentation and research in recommendation systems and media-focused social networking.

## Existing Similar Solutions

The most comparable existing solution is **Letterboxd**, a globally popular social platform for film fans. Launched in 2011, Letterboxd has over 13 million users and is focused on cataloging, rating, reviewing, and tracking movies viewed. Each user can maintain a film diary, create lists, share reviews, and interact with other community members.

Key differences vs. this project

Feature	Letterboxd	This Project
Recommendations	No advanced algorithms, mostly manual	Advanced collaborative and content-based AI
Social Interaction	Comments, following, diaries	Messaging, live feeds, deeper interaction
Data Sources	TMDB, automated sync	TMDB, automated sync, expanded APIs
Customization	Limited	Highly customizable profiles, genres, notifications
Product Growth	Films focus, TV planned	Films, TV, flexible expansion

Letterboxd is a film-focused social network, but it does not provide algorithmic recommendations; instead, it emphasizes lists, film logs, and manual sharing. This project, by contrast, adds personalized and automated recommendations, deeper messaging and live feed options, more sophisticated customization, and is designed for expansion into other media domains.

Similar Solution Illustrations

- See official site: <https://letterboxd.com/>



Letterboxd app screenshot

Target User Groups

Users who could be interested: - Movie enthusiasts of all ages wanting to share opinions and recommendations - Streaming platform users seeking better recommendations and more interaction - Industry professionals or critics who want a space for direct dialogue with their audiences - Educators and media studies students - Developers and researchers working with recommendation engines and social media for multimedia content.

## Solution Adaptability

- Modular architecture makes it easy to add new features (e.g., TV shows, podcasts, notifications).
- Responsive design for various platforms (mobile/web).
- Recommendation algorithms can be replaced, combined, or improved according to community needs.
- Advanced filtering by genre, year, and metadata is planned.

## Project Scope

This project will include: - Frontend developed with Next.js and React - Backend service and PostgreSQL database - Dedicated Python microservice for recommendations - OAuth 2.0 authentication - Implementation of user profiles, feeds, messaging, rating, and reviewing - Automated import of movie and show data from TMDb - Comprehensive testing and documentation

## Possible Upgrades

- Implementation of advanced machine learning methods for recommendations
- Expansion into podcasts
- Real-time notifications and group features
- Integration of analytics tools and personalized reports
- Gamification: awards, badges, leaderboards
- Internationalization for global reach

## References

- Letterboxd official website, Wikipedia
  - Comparative analysis of social movie networks
- 

## Meeting-records

### Meeting Records

Date	Attendees	Conclusion
8.10.2025.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Luka Volarević, Helena Floreani, Nora Milolović, David Grabovac	First meeting and first mandatory laboratory exercise; discussion about our own topic
21.10.2025	Nora Milolović, Helena Floreani	Discussed frontend architecture and UI design, defined overall visual style, and agreed on task

Date	Attendees	Conclusion
		distribution and responsibilities.
23.10.2025.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Nora Milolović, David Grabovac	Second mandatory laboratory exercise; progress review and defining details
30.10.2025.	Matija Akrap, Jurica Šlibar, David Grabovac	Creating pipeline for DobbySense and nextJS app
5.11.2025.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Helena Floreani, Nora Milolović, David Grabovac	Agreement on future project work and defining details
12.11.2025.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Luka Volarević, Helena Floreani, Nora Milolović, David Grabovac	Third mandatory laboratory exercise; discussion about progress to date
10.12.2025.	Matija Akrap, Helena Floreani, Nora Milolović	Commenting on distributed points after first submission
16.12.2025.	David Grabovac, Luka Volarević, Helena Floreani	Fixing frontend/backend bugs
7.1.2026.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Luka Volarević, Helena Floreani, Nora Milolović, David Grabovac	Setting a plan for the 2nd cycle of the semester. Making bullet points for next implementations.
10.1.2026.	Fran Kramberger, Helena Floreani, Nora Milolović	Review of frontend development features and further plans
12.1.2026.	Matija Akrap, Jurica Šlibar, David Grabovac	Implementing and discussing DobbySense pipeline and features
14.1.2026.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Luka Volarević, Helena Floreani, Nora Milolović, David Grabovac	Presentation of alpha version of the app
15.1.2026.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Luka Volarević, Helena Floreani, Nora Milolović, David Grabovac	Mandatory meeting for discussing final progress and tasks
16.1.2026.	Fran Kramberger, Helena Floreani, Nora Milolović	Reviewed frontend progress and discussed the remaining tasks.
21.1.2026.	Matija Akrap, Helena Floreani	Reviewed final progress, resolved remaining issues, and finalized

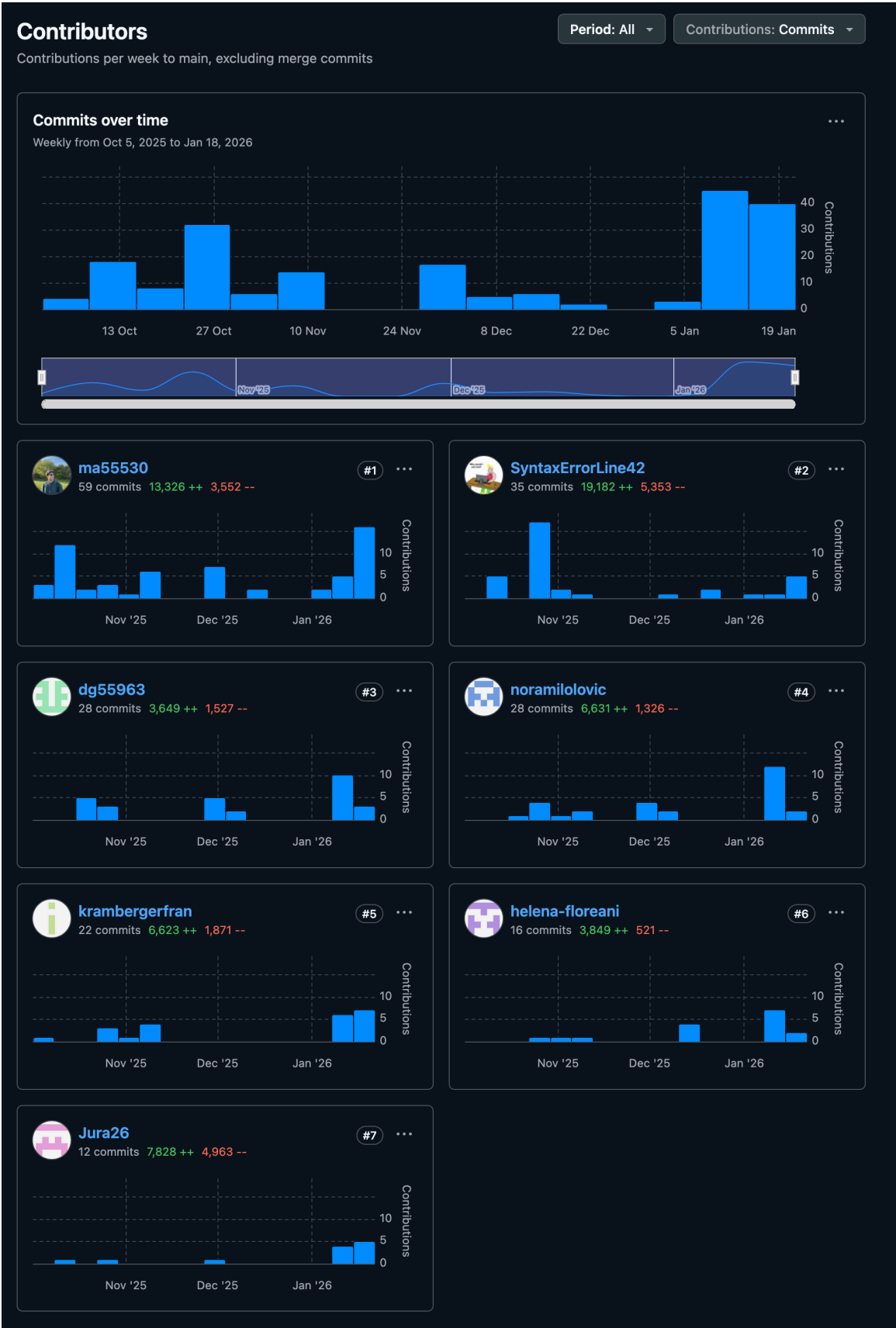
Date	Attendees	Conclusion
23.1.2026.	Matija Akrap, Jurica Šlibar, Fran Kramberger, Luka Volarević, Helena Floreani, Nora Milolović, David Grabovac	documentation and submission tasks.  Final meeting.

## Activity-Table

Activity	Matija Akrap	Helena Floreani	David Grabovac	Fran Kramberger	Nora Milolović	Jurica Šlibar
Project management	28h					
Project task description	2h					
Functional requirements	1h			3h		
Description of use cases				3h		
Use case diagrams				4h		
Sequence diagrams				3h		
Description of other requirements	3h			2h		
System architecture and design	7h		7h			8h
Database			5h			6h
Class diagram			3h			
State diagram				1h		
Activity diagram	1h				4h	
Component diagram			2h			
Adoption of technologies and tools			8h			2h
Solution testing						4h
Deployment diagram		1h				
Deployment instructions						2h

Activity	Matija Akrap	Helena Floreani	David Grabovac	Fran Kramberger	Nora Milolović	Jurica Šlibar
Meeting log	1h	1h			1h	
Conclusion and future work		2h				
Key challenges and solutions	4h					2h
Database creation			2h			
Database connection			2h			
Presentation creation	3h					
Application design	3h	36h		30h	35h	
Front end	4h	48h	2h	35h	50h	3h
Back end	20h		50h	3h		14h
Web app deployment	2h					
README update	2h					1h
Wiki editing	3h		5h	5h		2h
Documentation change log	1h					
DobbySense	14h					30h

## Contributions diagrams



# Documentation-Change-Log

# Documentation Change Log

Rev.	Description of change/addition	Authors	Date
0.1	Initial phase wiki page & Home page setup	Luka Volarević	11.10.2025
0.2	Initial setup of Shadcn, Supabase and Auth	Luka Volarević	12.10.2025
0.3	Routes overview, Home update, Auth initialization, Sidebar & Footer creation	Matija Akrap, Luka Volarević	13.10.2025
0.4	Database wiki initialization and documentation	Luka Volarević	17.10.2025
0.5	Created Requirements analysis documentation	Fran Kramberger	22.10.2025
0.6	Updated Sidebar and Requirements analysis	Matija Akrap	23.10.2025
0.7	Created System Requirements Specification (SRS)	Fran Kramberger	28.10.2025
0.8	Added DobbySense wiki, Updated Routes & Functionality, SRS updates	Matija Akrap, Jurica Šlibar, Fran Kramberger	29.10.2025
0.9	Added Documentation change log, Meeting records and Installation & Deployment	Matija Akrap	15.1.2026
1.0	Added the initial version of the activity table, reorganized the footer and changed the home page	Matija Akrap	15.1.2026
1.1	Added the Dynamic Behavior of the Application and UML Activity Diagram	Nora Milolović	20.1.2026
1.2	Added the UML Component Diagram	David Grabovac	20.1.2026
1.3	Added the UML Deployment Diagram	Helena Floreani	20.1.2026
1.4	Added final conclusion and project review documentation	Helena Floreani	20.1.2026
1.5.	Updated DobbySense and ForYou wiki, updated DB wiki	Matija Akrap, Luka Volarević	22.1.2026
1.6	Added contributions diagrams, updated activity table and added a meeting record	Matija Akrap	23.1.2026

## Requirements-analysis

## Dobby - Requirements Analysis

# 1. Functional Requirements

Requirement ID	Description	Priority	Source	Acceptance Criteria
F-001	The system allows users to sign in or register using their Google account (OAuth 2.0).	High	Stakeholder Requirement	The user can authenticate via Google, and the system successfully creates or retrieves their Supabase profile.
F-002	The system allows users to recover access to their account via Google OAuth recovery options.	Medium	Stakeholder Requirement	If a user cannot log in, they are informed that account recovery is handled through Google. After successful Google recovery, the user can log in again via "Sign in with Google."
F-003	The system allows users to edit their profile (username, avatar, preferences, privacy settings).	Medium	Stakeholder Requirement	User can modify and save profile information; changes persist in Supabase.
F-004	The system allows browsing and searching for movies and TV shows.	High	Stakeholder Requirement	The user can search media content and view metadata (title, poster, genre, rating).
F-005	The system displays detailed media content information via the TMDB API.	High	External API	Media details are correctly displayed with all relevant information.
F-006	The system supports advanced search filters.	Low	Stakeholder Requirement	Search results dynamically update based on applied filters.
F-007	The system allows users to add media to favorites and watchlist.	High	Stakeholder Requirement	The user can add or remove content from favorites/watchlist, and the change is synchronized with the backend.
F-008	The system generates personalized recommendations based on user history and similarity with other users.	High	Stakeholder / Backend	Recommendations are generated and displayed in the user interface in real time.

Requirement ID	Description	Priority	Source	Acceptance Criteria
F-009	The system allows users to rate and review movies/TV shows.	High	Stakeholder Requirement	The user can submit ratings and reviews, which are stored in Supabase.
F-010	The system allows users to create and share custom lists.	Medium	Stakeholder Requirement	User can create, edit, and share thematic lists; lists are visible to other users.
F-011	The system allows users to follow or unfollow other users.	Medium	Stakeholder Requirement	User can follow another user and see their activity in a social feed.
F-012	The system allows sending recommendations and messages to other users.	Medium	Stakeholder Requirement	Messages and recommendations are successfully sent and visible to recipients in their feed.
F-013	The system provides a personalized activity feed (recent reviews, ratings, and recommendations from followed users).	Medium	Stakeholder Requirement	User sees an updated feed of relevant social activity.
F-014	The system provides a modern and responsive UI suitable for both mobile and desktop devices.	High	Design Requirement	The interface is responsive, functional, and visually aligned with Tailwind CSS and shadcn/ui specifications.
F-015	The system supports notifications (e.g., when someone follows you, reacts to your review, or sends a message).	Medium	Stakeholder Requirement	User receives in-app or email notifications for relevant events.
F-016	The system allows administrators to manage user accounts and reported content.	High	Admin Requirement	Admin can deactivate users, remove reviews, and moderate reports.

## 2. Non-Functional Requirements

Requirement ID	Description	Priority
NF-001	The system must ensure secure authentication and data encryption through Supabase.	High
NF-002	Recommendation response time must not exceed 2 seconds per user request.	High
NF-003	The system must support at least 10,000 concurrent users.	High
NF-004	The system must be designed to allow easy maintenance and scalable modular architecture.	High
NF-005	The interface must be responsive and accessible on all modern web browsers and mobile devices.	High
NF-006	The system must have documented backend and frontend code and provide appropriate manuals for users and administrators.	High

## 3. Stakeholders

Stakeholder	Role / Interest
User	End user of the system; initiates registration, browsing, rating, and recommendations.
Administrator	Manages users and content, supervises the system, and maintains the backend.
Development Team	Responsible for development and maintenance of frontend, backend, and recommendation microservice.
Client / Stakeholder	Provides project requirements and guidelines, monitors development and system functionality.

## 4. Actors and Their Functional Requirements

### Primary Actors

#### A-1 User (Initiator)

- Can register, log in, and manage their profile (F-001, F-002)
- Can browse, search, and view media details (F-003, F-004)
- Can add content to favorites/watchlist and rate media (F-005, F-006)
- Can send recommendations and messages to other users (F-007)
- Receives personalized recommendations from the system (F-008)
- Uses a responsive and modern interface (F-009)

# Supporting / External Systems (Secondary Actors)

## A-2 Supabase (System)

- Serves as the backend database for user data, media preferences, and interactions (F-001, F-002, F-005, F-006)
- Stores user profiles, favorites/watchlists, ratings, and messages (F-003, F-004, F-007)
- Ensures secure authentication and data retrieval for the Dobby Web App (F-001, F-002)

## A-3 TMDB API (External System)

- Provides media information including movies, series, and related details (F-003, F-004)
- Supplies metadata required for browsing, searching, and viewing media (F-003, F-004)
- Acts as the authoritative source for media content displayed to users (F-003, F-004)

## A-4 DobbySense (Recommendation Engine)

- Generates personalized media recommendations for users (F-008)
  - Analyzes user behavior, ratings, and preferences to suggest relevant content (F-008)
  - Communicates with the Dobby Web App to display recommendations (F-008)
- 

# System-Requirements-Specification

## 1. Use Case Overview

This section presents the main use cases derived from the functional requirements. The diagrams and descriptions outline how users and external systems interact with the Dobby system.

---

## 2. Use Case Diagrams

### 1. High-Level Use Case Diagram of the Entire System

#### Purpose:

Illustrates the main functionalities from the perspective of end users and supporting systems.

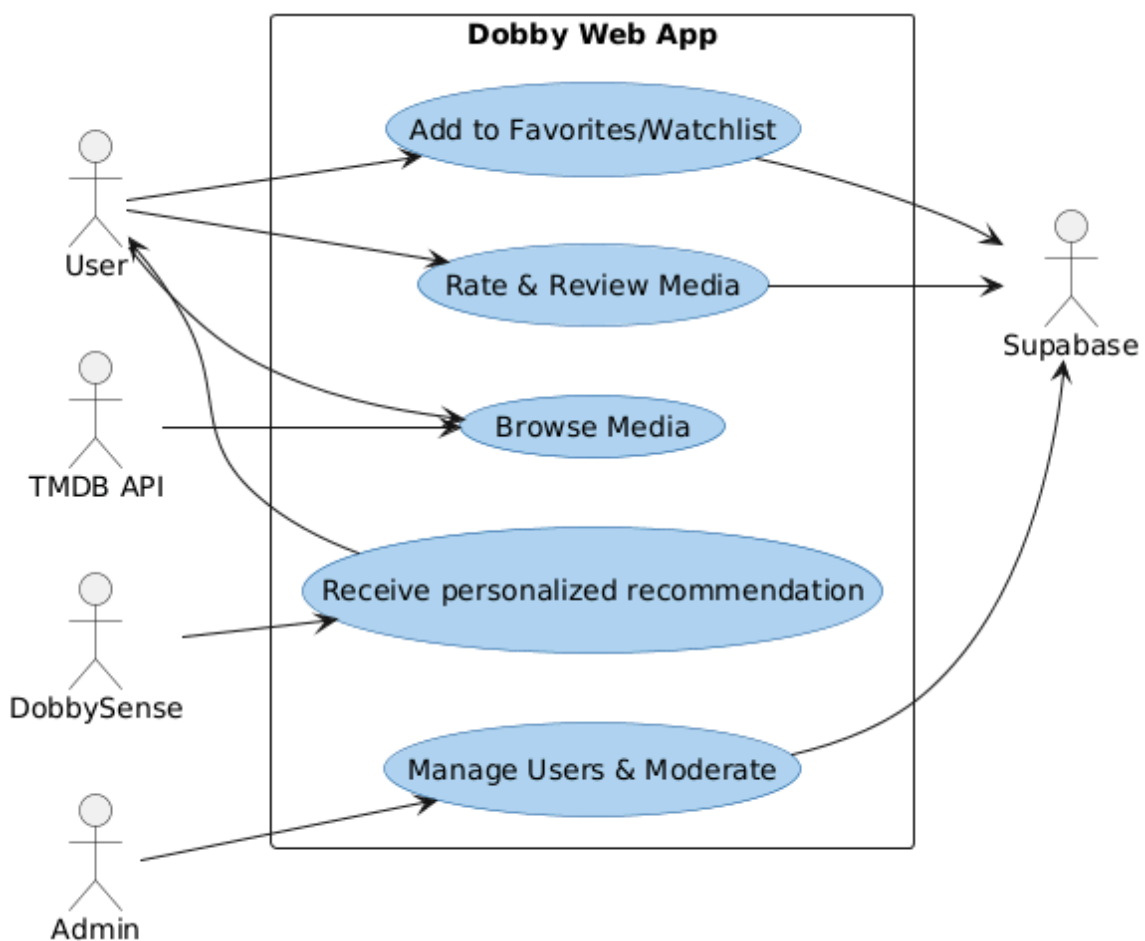
#### Main Actors:

- **User:** interacts with the web application.
- **Administrator:** manages system data and users.
- **Supabase:** backend service for authentication, storage, and data persistence.

- **TMDB API:** external provider of media information.
- **DobbySense Engine:** recommendation subsystem.

### Main Use Cases:

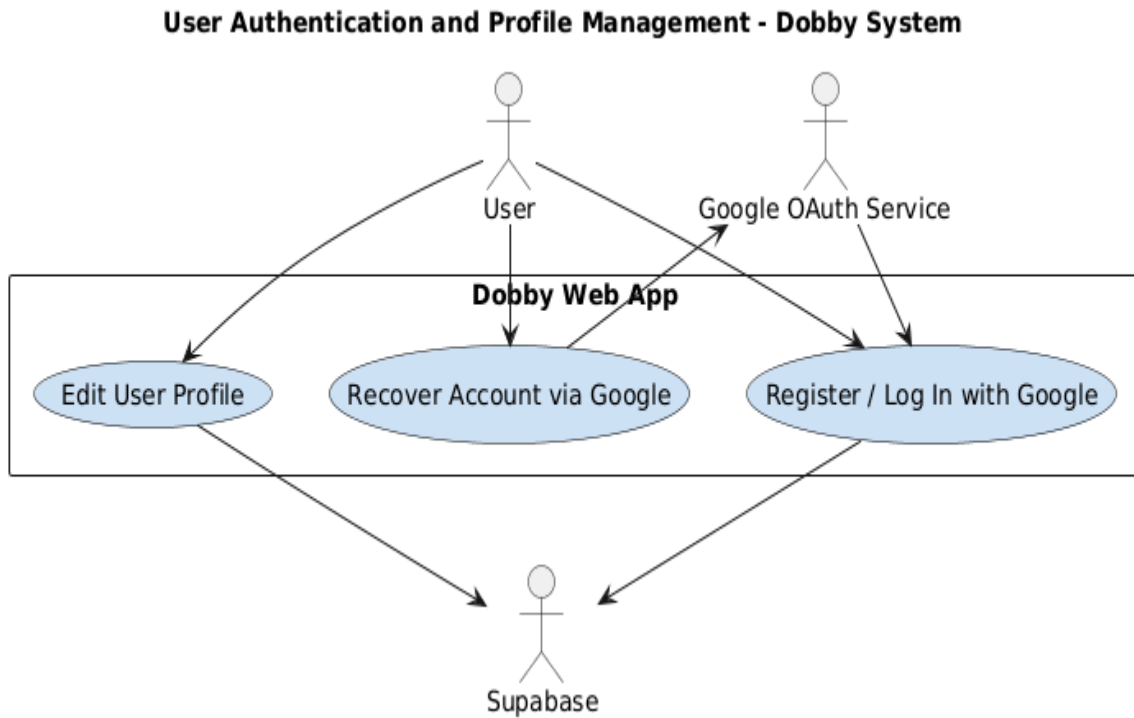
- Browse and Search Media
- Manage Favorites and Watchlist
- Rate and Review Media
- Receive Personalized Recommendations
- Maintain and Monitor System



## 2. Use Case Diagrams for Key Features

These diagrams illustrate the main functional areas of the Dobby system in more detail.

## 2.1 Authentication & User Management

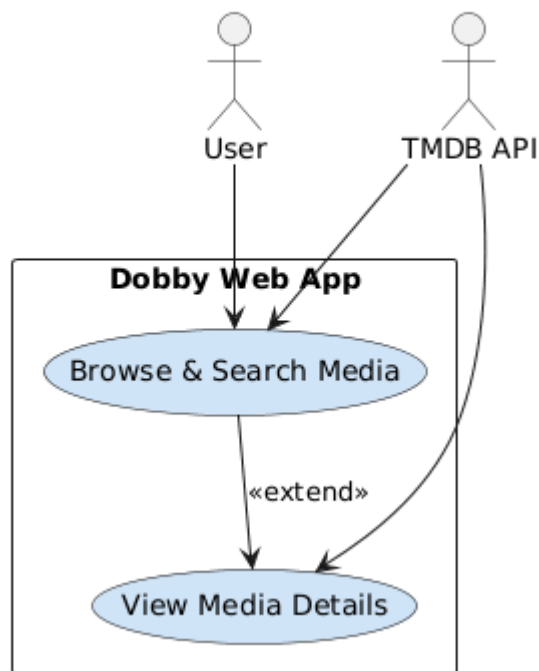


### Description:

This diagram shows user authentication and account management features. Users can register, log in via Google OAuth, recover accounts, and edit their profiles. Interactions with Supabase and Google OAuth are depicted.

## 2.2 Media Discovery

### Media Discovery and Browsing - Dobby System



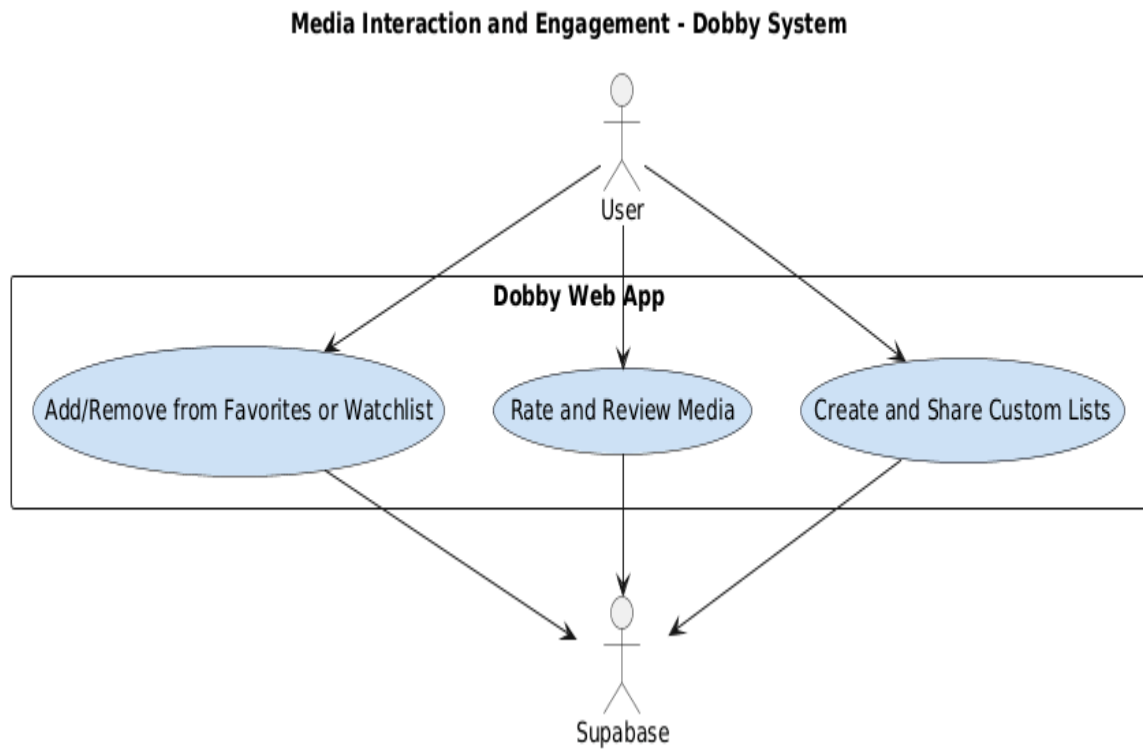
### Description:

Illustrates media browsing, searching, viewing details, managing favorites and watchlists, and submitting ratings or reviews. Shows integration with TMDB API and Supabase for

data persistence.

---

## 2.3 Interaction & Engagement

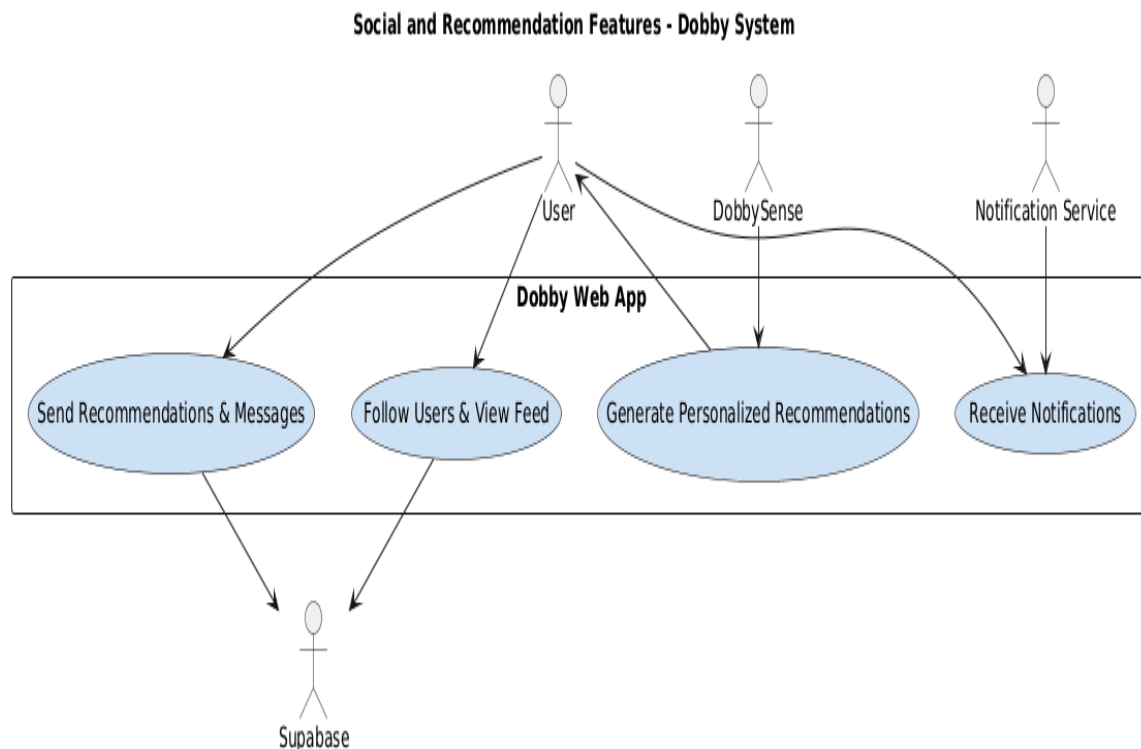


### Description:

Describes how users engage with content and express their personal preferences. Users can rate media, write reviews of arbitrary length, add media to their favorites, create custom lists, and control who can view or access these lists.

---

## 2.4 Social activity & Recommendations



### Description:

Focuses on social interactions and personalized recommendations. Users can follow others, view activity feeds, send messages or recommendations, receive notifications, and get content suggestions via the DobbySense engine.

## 3. Use Cases Description

### UC-01 – User Registration and Login

**Main Actor:** User

**Goal:** To register a new account and log into the Dobby system.

**Participants:** User, Supabase

**Precondition:** User has a valid Google account.

#### Main Flow:

1. The user selects “Register”. (F-001)
2. The system redirects the user to Google for authentication.
3. Google verifies the user’s identity and returns an authentication token.
4. The system creates or retrieves the corresponding user record in Supabase.
5. The user is successfully logged into the Dobby system. (F-001)

#### Alternative Scenarios:

- Invalid email → system displays an error and prompts re-entry.
- Email already registered → user is prompted to log in instead.
- Supabase connection failure → system retries the request or shows an error message.

## UC-02 – Account Access Issues (Google OAuth)

**Main Actor:** User

**Goal:** To regain access to the Dobby system using Google account recovery.

**Participants:** User, Google OAuth Service

**Precondition:** User has an existing Google account linked to Dobby.

**Main Flow:**

1. The user selects “Sign in with Google”. (*F-002*)
2. If authentication fails, the system shows an error message.
3. The user follows Google’s account recovery process (outside the app).
4. After successful recovery, the user retries login via “Continue with Google”.

**Alternative Scenarios:**

- User’s Google account disabled → Dobby access denied.
  - OAuth token expired → system requests re-authentication.
- 

## UC-03 – Edit User Profile

**Main Actor:** User

**Goal:** To edit personal profile details such as username, avatar, or preferences.

**Participants:** User, Supabase

**Precondition:** User is logged in.

**Main Flow:**

1. The user navigates to “Profile Settings.” (*F-003*)
2. The user updates one or more fields (e.g., username, avatar, visibility).
3. The system validates input and updates data in Supabase.
4. The updated profile is displayed to the user.

**Alternative Scenarios:**

- Invalid data → user is prompted to correct fields.
  - Database error → changes are not saved, and an error message is displayed.
- 

## UC-04 – Browse and Search Media

**Main Actor:** User

**Goal:** To browse and search movies or TV shows in the system.

**Participants:** User, TMDB API

**Precondition:** User is logged into the system.

**Main Flow:**

1. The user navigates to the “Browse” page. (*F-004*)
2. The user enters a keyword or selects filters.
3. The system queries TMDB API for matching results. (*F-005*)
4. The system displays movie/TV metadata (title, poster, genre, rating). (*F-005*)
5. The user selects an item to view details.

**Alternative Scenarios:**

- TMDB API unavailable → cached or default content is displayed.
- No results found → “No matches found” message displayed.

## UC-05 – Manage Favorites and Watchlist

**Main Actor:** User

**Goal:** To add or remove media items from favorites or watchlist.

**Participants:** User, Supabase

**Precondition:** User is logged in.

### Main Flow:

1. The user selects a media item. (*F-007*)
2. The user adds or removes the item from favorites/watchlist.
3. The system updates the Supabase record.
4. The updated list is displayed to the user.
5. Changes are synchronized across sessions.

### Alternative Scenarios:

- No internet connection → action is queued for later synchronization.
  - Database error → system shows an error and keeps previous state.
- 

## UC-06 – Rate and Review Media

**Main Actor:** User

**Goal:** To submit a rating and optional review for a selected movie or show.

**Participants:** User, Supabase

**Precondition:** User has previously viewed the media.

### Main Flow:

1. The user opens the media details page. (*F-009*)
2. The user enters a rating and optional review text.
3. The system validates and saves data to Supabase.
4. The system updates the average rating in real time.
5. The user can edit or delete their review.

### Alternative Scenarios:

- Invalid input → system requests correction.
  - Database error → user notified and prompted to retry.
- 

## UC-07 – Create and Share Custom Lists

**Main Actor:** User

**Goal:** To create personalized lists and share them with others.

**Participants:** User, Supabase

**Precondition:** User is logged in.

### Main Flow:

1. The user navigates to “My Lists.” (*F-010*)
2. The user selects “Create New List” and adds media items.
3. The system stores the list in Supabase.
4. The user can mark the list as public or private.
5. Other users can view or follow public lists.

**Alternative Scenarios:**

- Invalid list name → user prompted to rename.
  - Network failure → list creation queued until reconnected.
- 

**UC-08 – Send Recommendations and Messages**

**Main Actor:** User

**Goal:** To send a recommendation or message to another user.

**Participants:** User, Supabase

**Precondition:** Both sender and receiver have active accounts.

**Main Flow:**

1. The user selects a media item or opens a chat. (*F-012*)
2. The user writes a message or attaches a media recommendation.
3. The system saves and delivers the message through Supabase.
4. The recipient receives a notification. (*F-015*)

**Alternative Scenarios:**

- Recipient offline → message delivered later.
  - Message content invalid (empty, spam) → validation error shown.
- 

**UC-09 – Follow Users and View Activity Feed**

**Main Actor:** User

**Goal:** To follow other users and view their activity feed.

**Participants:** User, Supabase

**Precondition:** User has a valid account and is logged in.

**Main Flow:**

1. The user visits another user's profile. (*F-011*)
2. The user clicks "Follow."
3. The system records the relationship in Supabase.
4. The follower's feed updates with new activities from followed users. (*F-013*)

**Alternative Scenarios:**

- User already followed → system shows "Following."
  - Feed service unavailable → cached or partial data displayed.
- 

**UC-10 – Generate Personalized Recommendations**

**Main Actor:** User

**Goal:** To receive personalized movie and TV show recommendations.

**Participants:** User, DobbySense Engine, Supabase

**Precondition:** User has provided ratings or favorites history.

**Main Flow:**

1. The user navigates to the "Recommendations" page. (*F-008*)
2. The system requests user preference data from Supabase.

3. DobbySense analyzes historical interactions and similarity scores.
4. Engine generates a ranked list of recommended items. (*F-008*)
5. System displays the results in the user interface.

**Alternative Scenarios:**

- Insufficient data → system shows trending or popular content instead.
  - Engine timeout → user prompted to retry or check connection.
- 

**UC-11 – Notifications**

**Main Actor:** User

**Goal:** To receive real-time updates about relevant events (messages, followers, reactions).

**Participants:** User, Supabase, Notification Service

**Precondition:** User has notification permissions enabled.

**Main Flow:**

1. An event occurs (new message, follow, or review reaction). (*F-015*)
2. The system creates a notification record in Supabase.
3. Notification is displayed in-app or via email.
4. User can mark notifications as read or clear them.

**Alternative Scenarios:**

- Email service unavailable → notification stored for later delivery.
  - User disabled notifications → system logs event silently.
- 

**UC-12 – Admin Moderation**

**Main Actor:** Administrator

**Goal:** To manage users, content, and reports.

**Participants:** Administrator, Supabase

**Precondition:** Admin is logged in with elevated privileges.

**Main Flow:**

1. Admin opens the “Admin Dashboard.” (*F-016*)
2. Admin views reports or flagged content.
3. Admin can deactivate users or delete inappropriate content.
4. Changes are stored in Supabase and reflected in the UI.

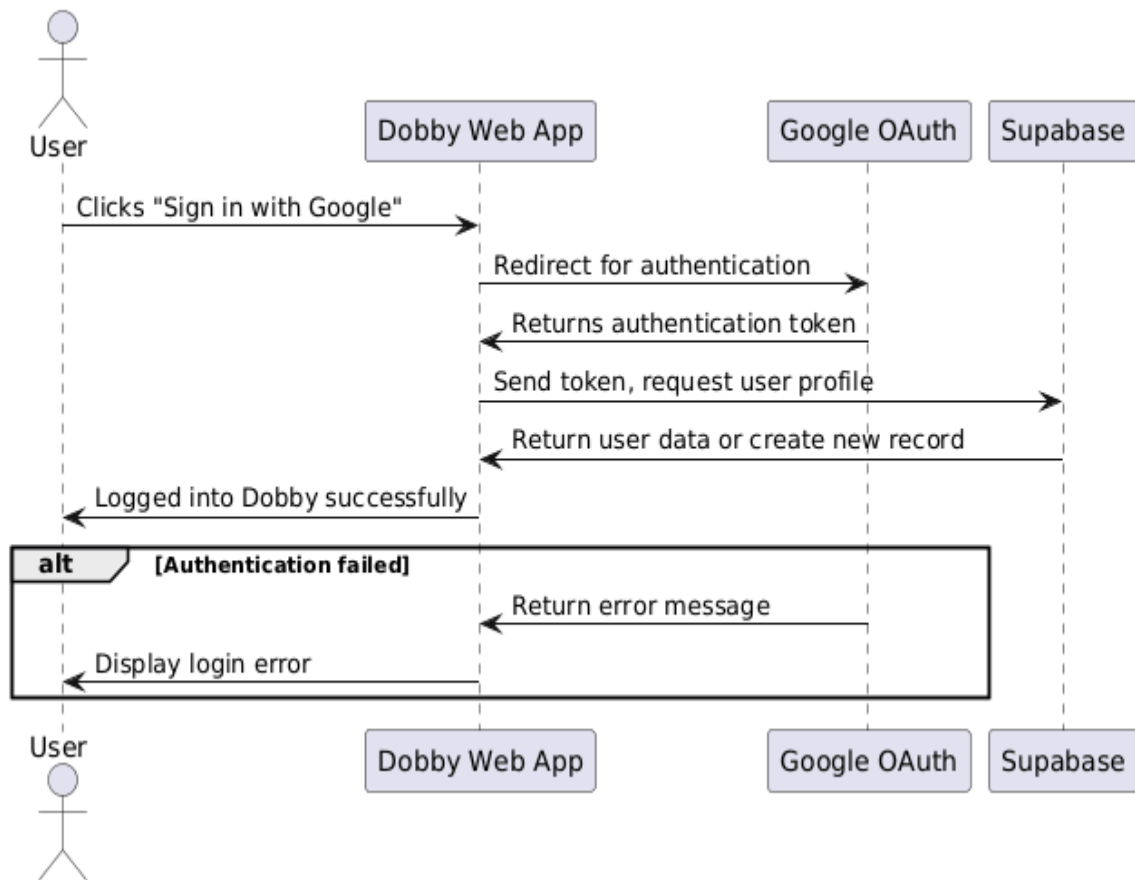
**Alternative Scenarios:**

- Invalid permissions → access denied.
  - Database error → system logs failure and retries.
-

## 4. Sequence Diagrams

### 4.1 User Registration and Login

UC-01 - User Registration and Login

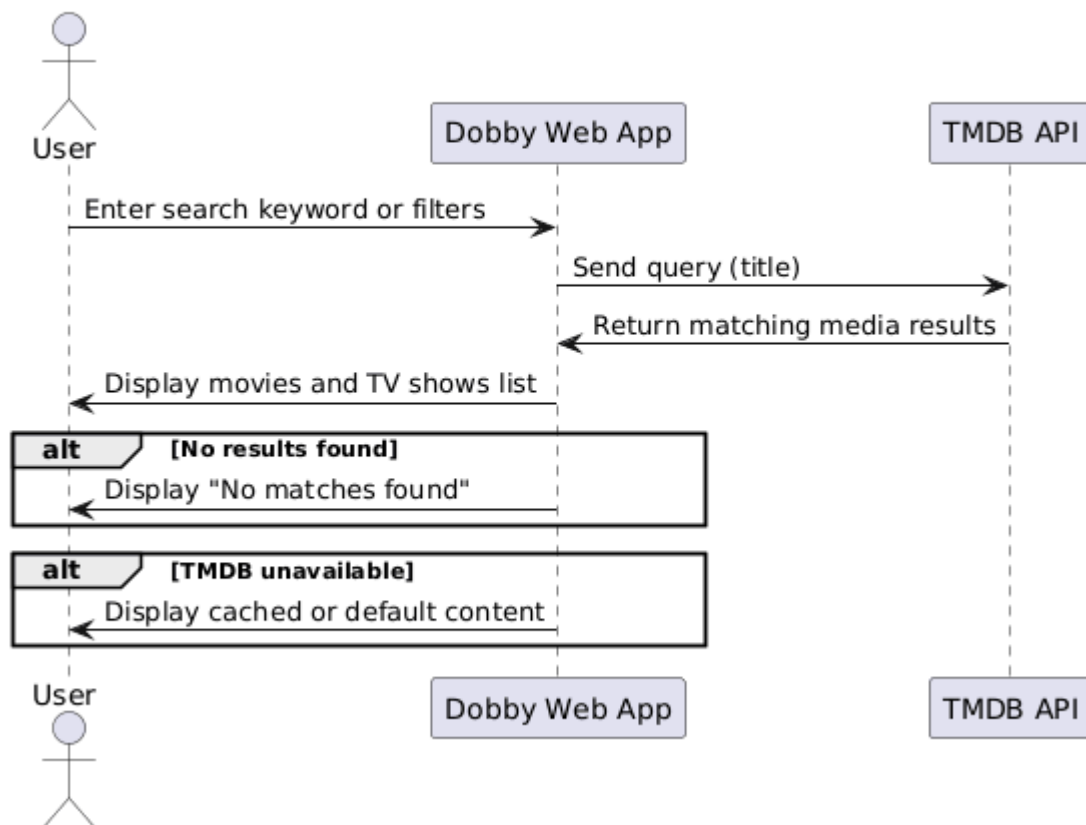


**Description:**

Shows the interaction between the User, Google OAuth, and Supabase during the registration or login process. Highlights authentication flow, account creation/retrieval, and error handling scenarios.

## 4.2 Browse and Search Media

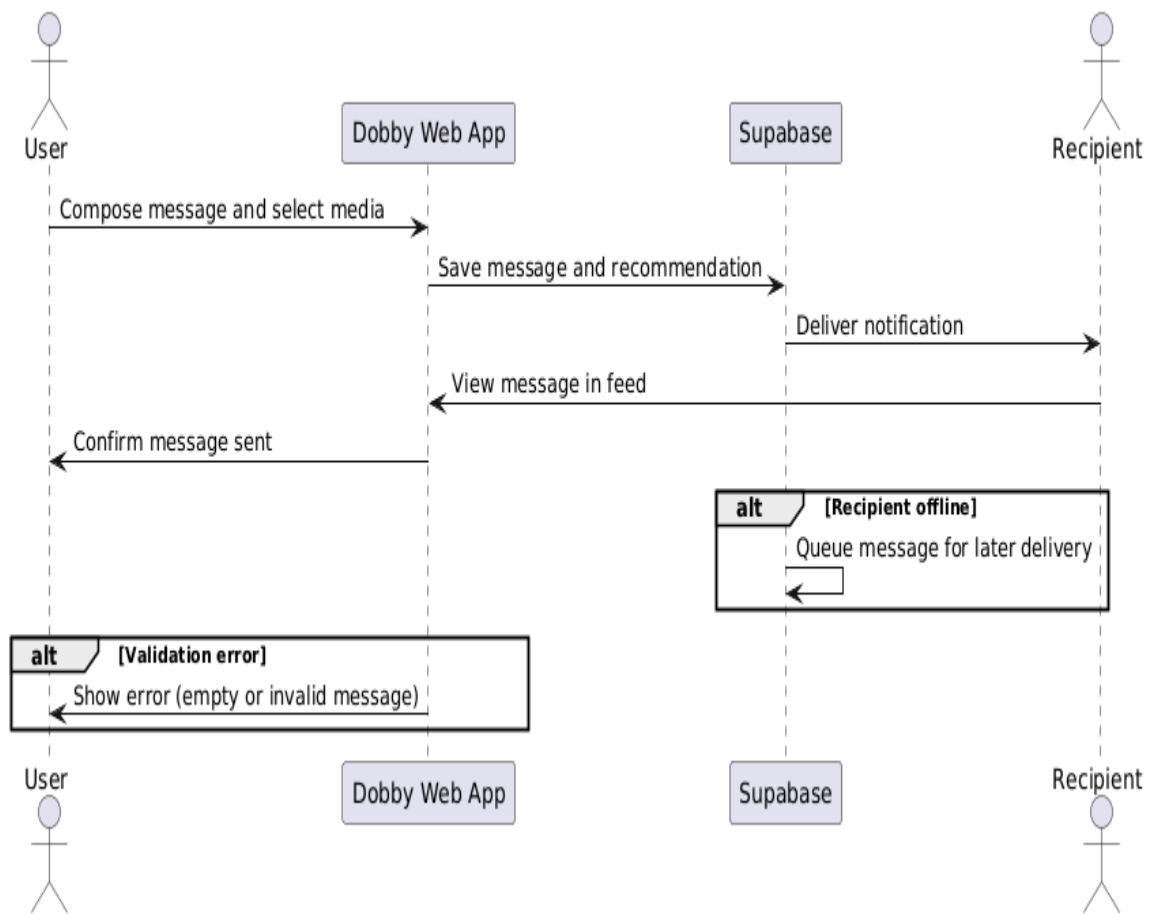
### UC-04 - Browse and Search Media

**Description:**

Illustrates how the User interacts with the Dobby Web App and the TMDB API to browse or search for media. Includes fetching metadata and displaying search results.

## 4.3 Send Recommendations and Messages

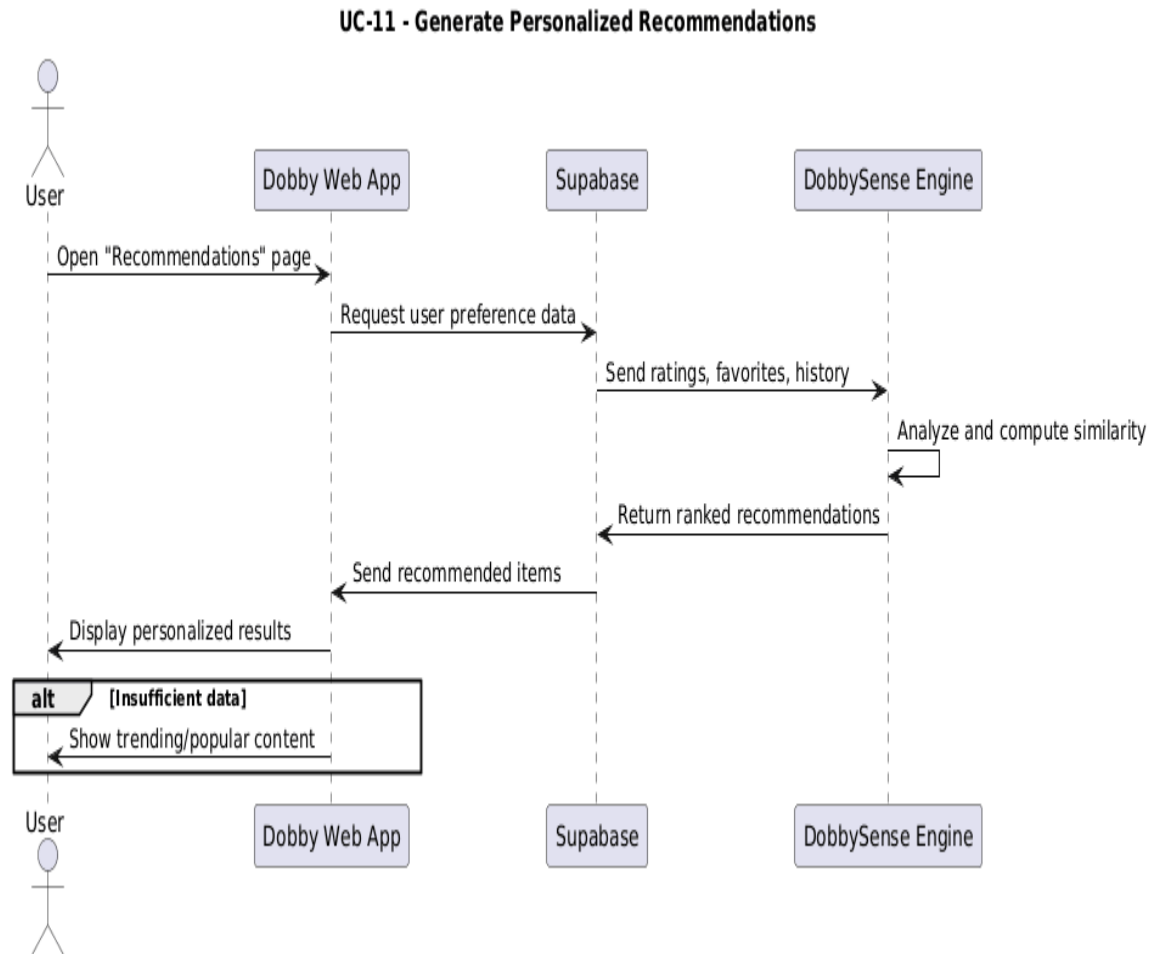
UC-09 - Send Recommendations and Messages



**Description:**

Depicts the flow of sending a recommendation or message from one user to another. Shows how the message is stored in Supabase and how notifications are triggered for the recipient.

## 4.4 Generate Personalized Recommendations



### Description:

Shows the interaction between the User, Supabase, and the DobbySense Engine to generate personalized movie and TV show recommendations. Highlights retrieval of user history, similarity computation, and display of ranked recommendations.

## 5. Key Functionality Coverage in Use Cases

Use Case ID	Use Case Name	Functional Requirements Covered
UC-01	User Registration and Login	F-001
UC-02	Account Access Issues (Google OAuth)	F-002
UC-03	Edit User Profile	F-003
UC-04	Browse and Search Media	F-004, F-005
UC-05	Manage Favorites and Watchlist	F-007
UC-06	Rate and Review Media	F-009
UC-07	Create and Share Custom Lists	F-010
UC-08	Send Recommendations and Messages	F-012
UC-09	Follow Users and View Activity Feed	F-011, F-013
UC-10	Generate Personalized Recommendations	F-008

Use Case ID	Use Case Name	Functional Requirements Covered
UC-11	Notifications	F-015
UC-12	Admin Moderation	F-016

# Routes-&-Functionality

## Frontend Routes

Route	Description	Key Features
/	Landing Page	Public page rerouting depending on if the user is authorized
/auth/login	Login	User login page
/auth/sign-up	Sign Up	User registration page
/home	Home Feed	Authenticated user home; displays feed, “create review”, etc.
/movies	Movies Browser	Browse movies with search and filters
/movies/[id]	Movie Details	Detailed view of a specific movie
/movies/forYou	Recommended Movies	Personalized movie recommendations
/shows	TV Shows Browser	Browse TV shows with search and filters
/shows/[id]	Show Details	Detailed view of a specific TV show
/shows/forYou	Recommended Shows	Personalized show recommendations
/profile	My Profile	Current user’s profile, ratings, and stats
/users/[id]	User Profile	View other users’ profiles and ratings
/watchlist	My Watchlist	Manage personal watchlist(s)
/messages	Messaging	Chat interface for conversations

## API Reference

## Content & Discovery

API Route	Method	Description
/api/movies	GET	Search and list movies (TMDB integration)
/api/shows	GET	Search and list TV shows
/api/recommendation-engine	GET/POST	Core logic for generating recommendations
/api/dobbySenseAPI/fold-in	POST	Recommendation engine “fold-in” operations
/api/dobbySenseAPI/genreLayer	GET/POST	Genre-based recommendation layering
/api/genres	GET	List available genres

## User & Social

API Route	Method	Description
/api/user	GET	Get current authenticated user details
/api/users	GET	List/Search users
/api/friends	GET	Manage user friends
/api/follows	GET	Manage user follows
/api/reviews	GET, POST	CRUD operations for movie/show reviews
/api/watchlist	GET, POST, DELETE	Manage items in the watchlist
/api/watchlists	GET	Retrieve user watchlists (collections)
/api/notifications	GET	Fetch user notifications

## Messaging

API Route	Method	Description
/api/conversations	GET	List conversations for the current user

## System-Architecture-and-Design

## System Architecture and Design – Dobby

# 1. System Architecture

## Architecture Overview

### Architecture Style:

The *Dobby* system is based on a **microservices architecture** combined with a **client–server model**.

The frontend (Next.js) acts as the client that communicates with backend services (FastAPI and Supabase) via REST APIs.

This architecture enables **high modularity, scalability, and independent maintenance** of subsystems.

The choice of a microservices style is guided by the principles of: - **Loose coupling** between components,

- **High cohesion** within services,
- **Independent scalability**, and
- **Ease of future extensions and maintenance**.

---

## Subsystems

Subsystem	Responsibility
Next.js Frontend	Client-side web application built using Next.js (React) with Tailwind CSS and shadcn/ui components. Displays the user interface and communicates with backend services through REST API calls.
FastAPI Microservice (DobbySense)	Python service responsible for processing user data and generating personalized recommendations using collaborative and content-based filtering algorithms.
Supabase Backend	PostgreSQL database and authentication service. Manages user accounts, preferences, watchlists, messages, and reviews.
TMDB API (external service)	Provides movie and TV show metadata (title, genre, synopsis, image, rating).

---

## Deployment Platform Mapping

The application is deployed in a **cloud environment**: - The frontend (Next.js) is hosted on **Vercel**.

- The FastAPI microservice runs on **Render** or **Railway**.
- **Supabase** provides authentication, PostgreSQL storage, and API endpoints.

This “serverless” and cloud-native approach ensures **automatic scaling, high availability**, and **minimal maintenance overhead**.

---

## Data Storage

The system uses a **relational PostgreSQL database** (via Supabase).

Data is organized in normalized tables storing:

- User profiles and preferences,
- Viewing history and ratings,
- Messages and peer recommendations,
- Movie and TV show metadata.

Image assets (user avatars, posters) are stored in **Supabase Storage** (object storage).

---

## Network Protocols

Communication between components is established through:

- **HTTPS** – secure client-server communication,
  - **REST API** – data exchange between Next.js frontend and FastAPI services,
  - **WebSocket (planned)** – for future real-time messaging and notifications.
- 

## Global Control Flow

1. The user authenticates via **Supabase OAuth 2.0**.
  2. The frontend retrieves user data and preferences.
  3. When recommendations are requested, the frontend sends a request to the **DobbySense microservice** via REST API.
  4. FastAPI processes the request, analyzes history and preferences, and returns personalized recommendations.
  5. The frontend displays results to the user.
  6. Interaction data (favorites, ratings, messages) are persisted in Supabase.
- 

## Hardware and Software Requirements

Component	Requirements
Server	Cloud environment (Vercel / Render) with Node.js 18+ and Python 3.11+
Frontend	Modern web browsers (Chrome, Firefox, Safari, Edge); responsive on desktop and mobile devices
Database	PostgreSQL 14+ (Supabase)
RAM/CPU	Minimum 2 GB RAM and 2 vCPUs for the development environment
Local Development OS	Windows, macOS, or Linux with Docker support

---

## 2. Rationale for Architectural Choices

The chosen **microservices architecture** provides:

- **Scalability** – each service (frontend, API, recommendations) can scale independently,
- **Fault tolerance** – a failure in one service (e.g., recommendations) does not crash the entire system,

- **Development flexibility** – frontend and backend can use different technologies (JavaScript + Python),
- **Seamless integration** – easy connection with external APIs such as TMDB.

**Alternative considered:**

A **monolithic architecture** was evaluated but rejected due to limited scalability and maintenance complexity.

Microservices allow easier expansion in the future with parallel development and easier maintenance.

**Design principles applied:** - **High cohesion** – each service has a focused, well-defined role.

- **Loose coupling** – communication occurs via REST APIs without direct dependencies.
  - **Security** – Supabase OAuth ensures safe authentication and authorization.
  - **Flexibility and extensibility** – new microservices and APIs can be added with minimal system impact.
- 

## 3. High-Level System Organization

### Client–Server Model

Dobby follows a **client–server architecture**: - **Client (Next.js)** → sends HTTP requests to FastAPI and Supabase backends.

- **Servers** → process requests, fetch data, and return JSON responses.

### Database

The system uses a **PostgreSQL** relational database provided by Supabase.

It stores normalized and relational data including users, messages, ratings, watchlists, and viewing history.

### File System

User profile images and media posters are stored in **Supabase Storage** (object storage service).

### Graphical User Interface

The web UI is developed using **Next.js**, **Tailwind CSS**, and **shadcn/ui**.

It is **fully responsive**, supports both dark and light themes, and provides intuitive navigation for browsing, reviewing, and viewing recommendations.

---

## 4. Application Organization

### Frontend and Backend Layers

Layer	Description
Frontend (Next.js)	Organized into <i>pages</i> , <i>components</i> , <i>services</i> , and <i>context</i> layers. Responsible for UI rendering, input

Layer	Description
	validation, and API communication.
Backend (FastAPI + Supabase)	FastAPI handles recommendation logic and exposes REST endpoints. Supabase manages authentication, data storage, and user management.

MVC Architecture (within the microservice)

Within the FastAPI microservice, the **Model–View–Controller (MVC)** pattern is applied:

- **Model:** SQLAlchemy models for database tables (User, Movie, Rating, Watchlist).
- **Controller:** FastAPI routes (/recommend, /user, /favorites).
- **View:** JSON responses sent to the frontend.

High-Level Architecture Diagram



Database

This project uses **Supabase PostgreSQL** as the main database solution. Supabase provides authentication, row-level security, and a hosted PostgreSQL instance, which perfectly integrates with the Next.js frontend and FastAPI microservice. The database is normalized and designed to support user management, movie and show data caching, ratings, watchlists, and social interactions.

ER Model

The entity–relationship model defines seven main entities: profiles, movies, shows, ratings, watchlists, watchlist\_items, and follows.

ER Diagram:



##  
 Sect  
 (RL  
 Poli  
 Row  
 Leve  
 Sect  
 (RL  
 enat  
 on a  
 user  
 relat  
 table  
 proto  
 priva  
 data  
 ensu  
 clie  
 side  
 safe  
 Sup  
 # Cl  
 Diag

## Generic Functionality

Classes related to the core functionality (e.g., users, movies, shows, ratings, watchlists, follows):

### Classes with Attributes and Methods:

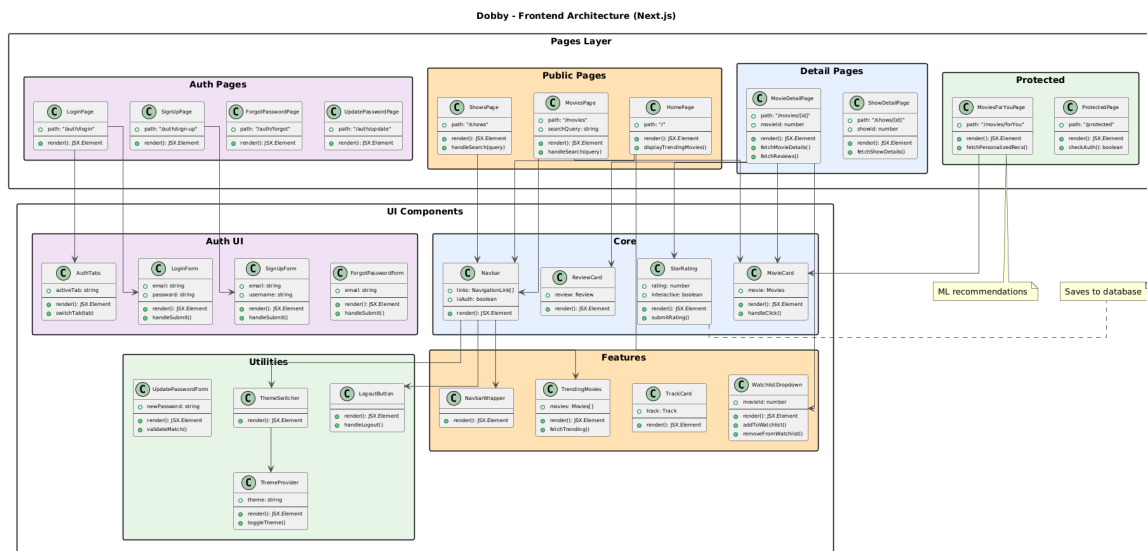
- **User**
  - **Attributes:** id, username, email, created\_at, updated\_at
  - **Methods:** getWatchlist() (public), getRatings() (public)
- **Movie**
  - **Attributes:** id, title, overview, poster\_path, release\_date, vote\_average
  - **Methods:** getMovieDetails(id) (public), getMovieReviews(id) (public)
- **Show**
  - **Attributes:** id, title, poster\_path, release\_date, vote\_average, overview
  - **Methods:** getShowDetails() (public)
- **Rating**
  - **Attributes:** id, user\_id, movie\_id, show\_id, rating, created\_at
  - **Methods:** addRating(userId, movieId, rating) (public), updateRating(ratingId, rating) (public)
- **Watchlist**
  - **Attributes:** id, user\_id, movie\_id, show\_id, type, added\_at
  - **Methods:** createWatchlist() (public), addToWatchlist(userId, movieId) (public)
- **Follow**
  - **Attributes:** follower\_id, following\_id, followed\_at
  - **Methods:** followUser() (public), unfollowUser() (public)

### Relationships for core functionalities:

- User 1 — \* Rating
  - Movie 1 — \* Rating
  - Show 1 — \* Rating
  - User 1 — \* Watchlist
  - Watchlist \* — \* Movie
  - Watchlist \* — \* Show
  - User \* — \* User (Follow)
- 

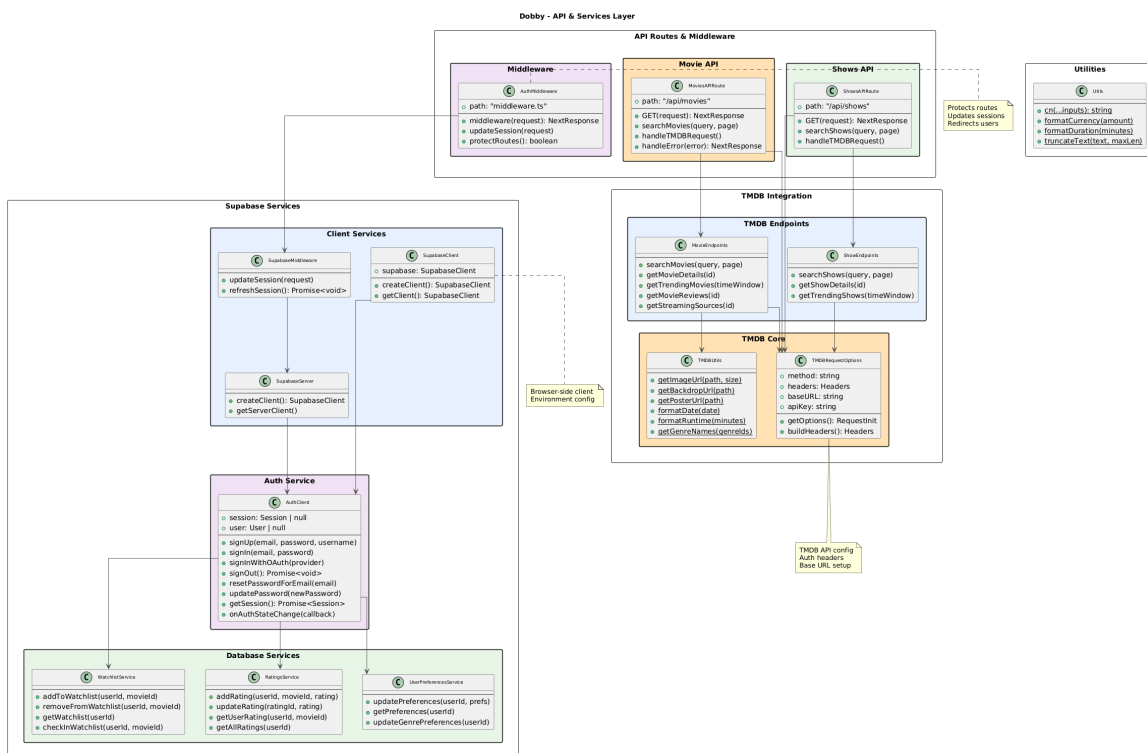
## Fronted Architecture (Pages layer)

This diagram models the presentation layer of the application. It includes classes representing pages, UI components, and navigation flows. The focus is on how the user interacts with the system, and how data from backend services is consumed and displayed. This layer communicates with the API layer to fetch and update information like movies, shows, ratings, and watchlists.



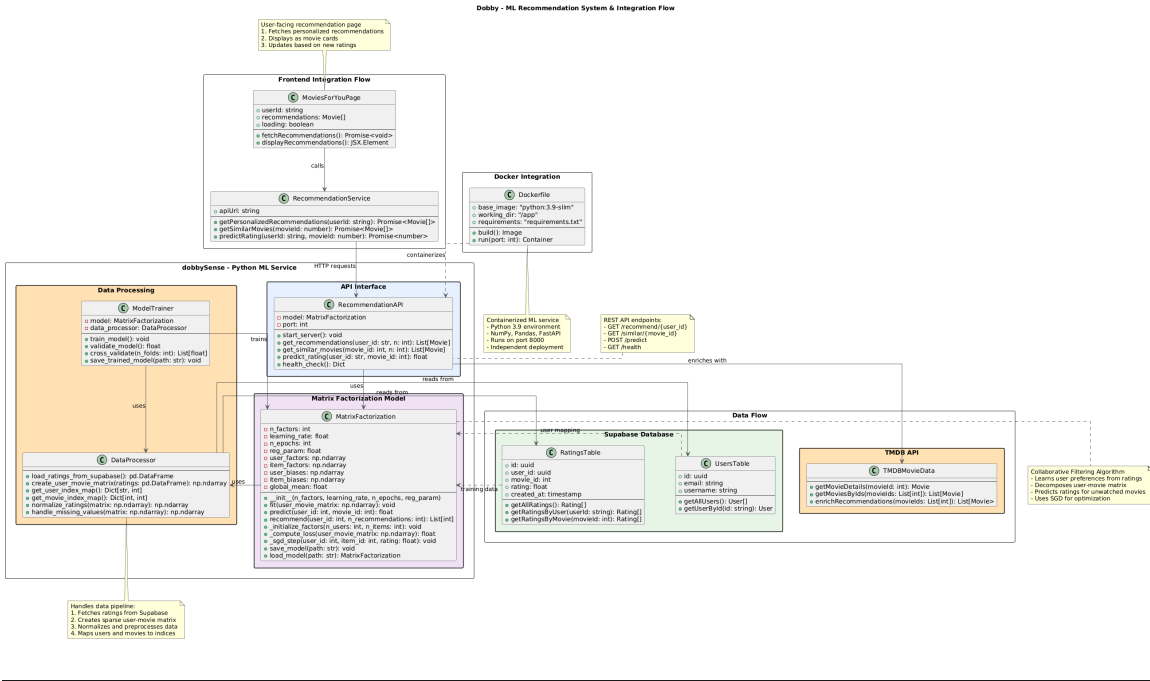
## API & Services layer

This diagram represents the backend services and business logic. It shows classes responsible for handling requests, processing data, and interacting with the database. The layer abstracts the internal workings of the system from the frontend, exposing services like fetching movie/show details, submitting ratings, managing watchlists, and handling user follow relationships.



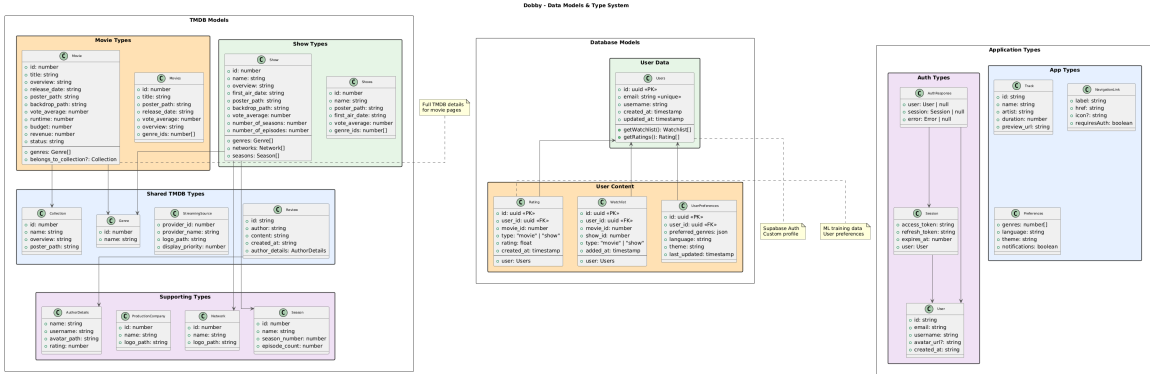
## ML Recommendation System & Integration Flow

This diagram illustrates the machine learning recommendation engine and its integration with the system. It includes classes responsible for data processing, feature extraction, model prediction, and generating personalized recommendations. Relationships with other layers, like API and Data Models, are shown to highlight how recommendations are retrieved and served to users.



# Data Models & Type System

This diagram focuses on core entities and their relationships. It captures attributes, methods, and associations, representing the structure of the underlying database and type system. This layer ensures data integrity and supports both the frontend and backend operations.



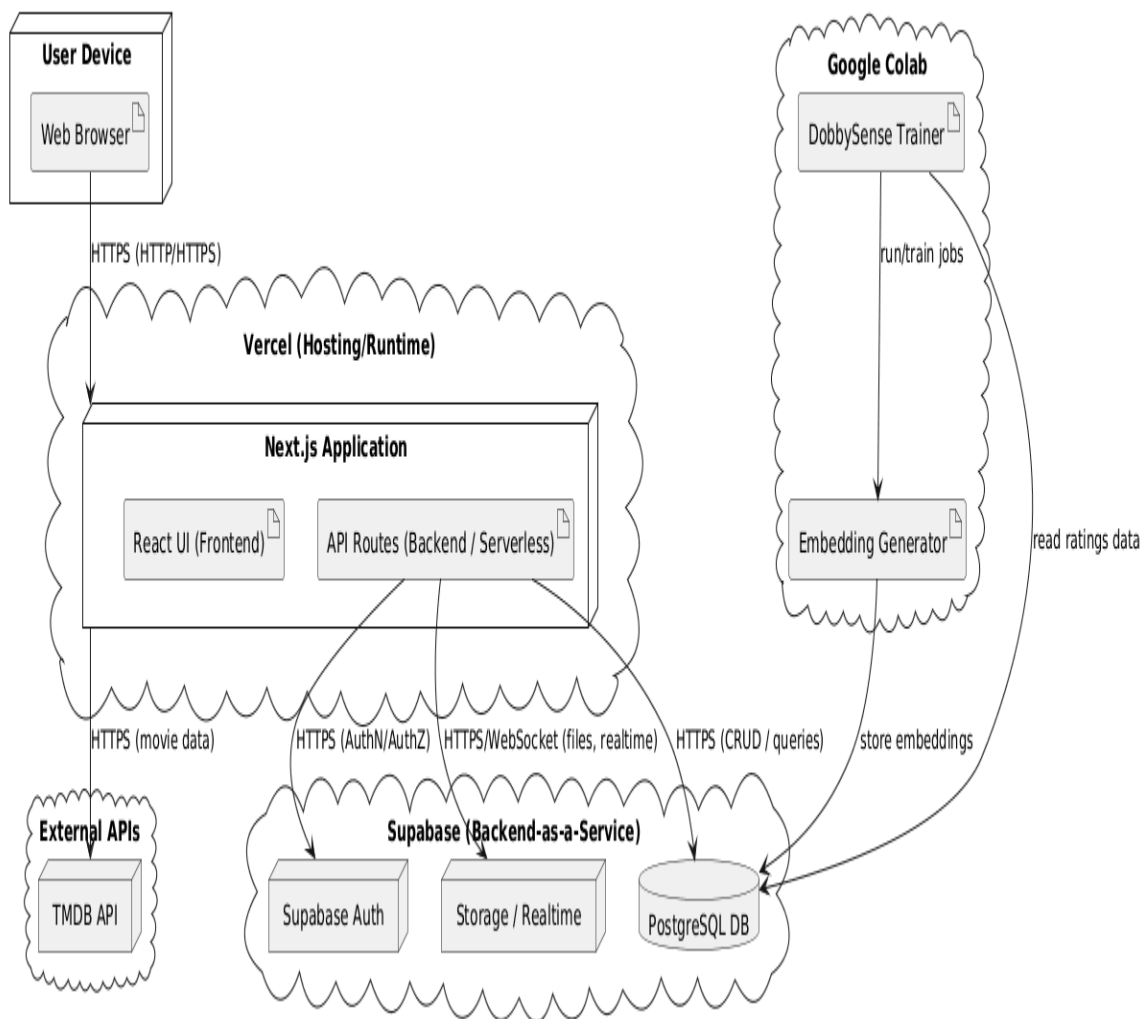
# Component-and-Deployment-Architecture

## Component Diagram

Component diagram  
Component diagram

# Deployment Diagram

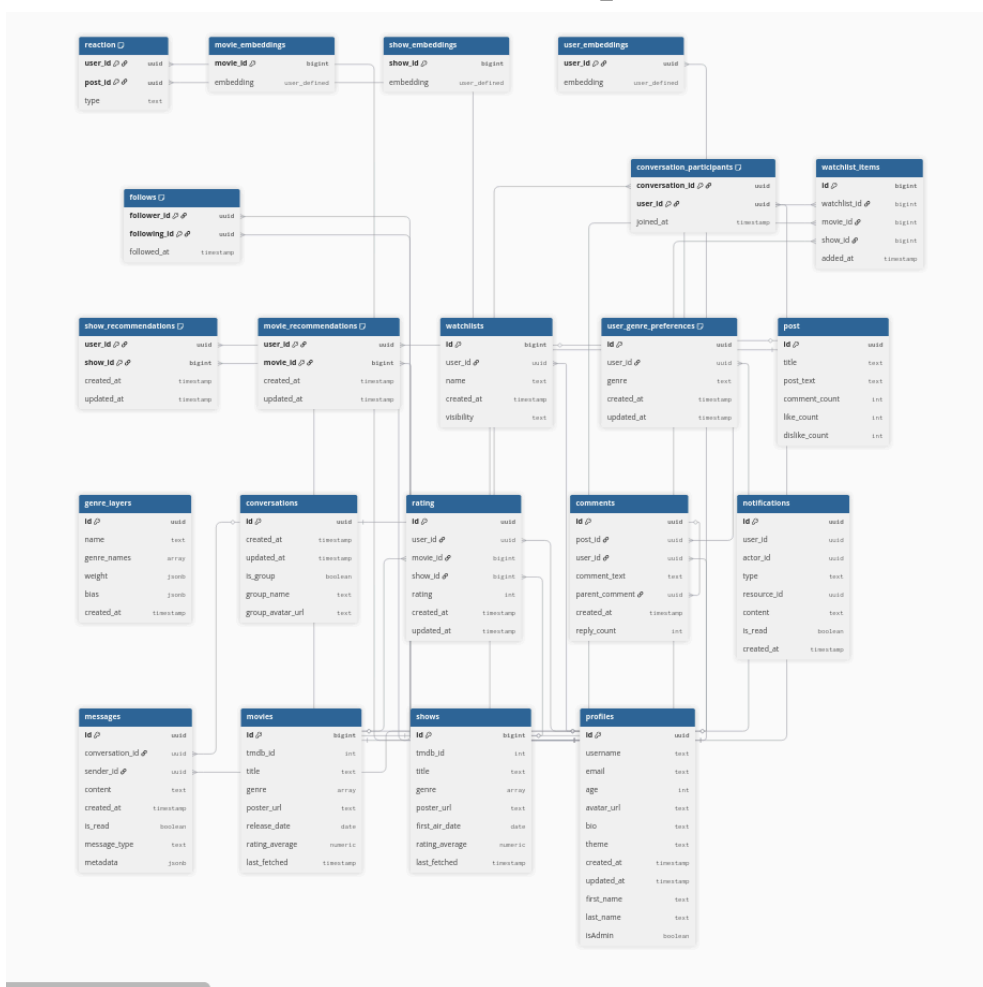
Dobby - Deployment Diagram



## Database

This is a comprehensive explanation of our Supabase PostgreSQL database.

## ER Model



er-model-png

## Profiles table

Stores user profile information linked to auth.users. Includes personal details (username, email, name, age, avatar, bio), theme preference, and timestamps. Automatically updated when modified.

## Movies table

Caches minimal movie data from TMDB (ID, title, genres, poster, release date, rating). Used for quick reference and local storage of movie info.

## Shows table

Caches minimal TV show data from TMDB (ID, title, genres, poster, first air date, rating). Used similarly to movies for local show references.

## Movie Ratings table

Stores user-submitted ratings and optional reviews for movies. Each user can rate a movie only once. Includes timestamps and updates automatically on edit.

## Show Ratings table

Stores user-submitted ratings and optional reviews for shows. Each user can rate a show only once. Includes timestamps and auto-updates on edit.

## **Watchlists table**

Holds collections of movies/shows created by users. Each watchlist has a name, visibility level (public, private, followers), and creation timestamp. Users can have multiple watchlists.

## **Watchlist Items table**

Connects movies or shows to a specific watchlist. Each item represents either a movie or a show (never both). Ensures users can only modify items in their own watchlists.

## **Follows table**

Defines the social graph — tracks which users follow others. Prevents self-following and cascades deletions. Used to manage access to “followers-only” watchlists.

## **Movie Embeddings table**

Holds vector embeddings (64-dim) representing movie features for AI similarity or recommendation tasks.

## **Show Embeddings table**

Holds vector embeddings (64-dim) representing show features for recommendations.

## **User Embeddings table**

Stores vector embeddings for users, used to match user preferences with movies/shows in recommendations.

## **Movie Recommendations table**

Stores movie recommendations generated for each user. Includes timestamps and updates automatically on edit.

## **Show Recommendations table**

Stores show recommendations for each user, with timestamps and auto-update on changes.

## **Messages table**

Stores messages sent in conversations. Includes sender, content, timestamps, and read status. Supports optional metadata and message types.

## **Conversations table**

Holds conversation metadata for chat. Can be a group or direct conversation and stores creation/update timestamps.

## Conversation Participants table

Links users to conversations, defining who is in each chat. Tracks when users joined the conversation.

## Comments table

Stores user comments on posts. Supports parent-child relationships for replies and tracks creation timestamps and reply counts.

## Posts table

Stores posts created by users. Includes title, content, and counters for likes, dislikes, and comments.

## Notifications table

Tracks notifications for users, including type, actor, target resource, content, read status, and timestamps.

## Reactions table

Stores user reactions to posts. Tracks type of reaction and prevents multiple reactions per user per post.

## Rating table

Stores general ratings for movies and shows. Includes user ID, content ID, rating value, and timestamps.

## Functions & Triggers

`update_updated_at_column()` → Automatically updates `updated_at` timestamp on row changes.

`handle_new_user()` → Creates default profile and starter watchlists (“Recently Searched”, “Favourites”) when a new user is registered.

`get_top_movies_for_user()` / `get_top_shows_for_user()` → Return top recommended items for a user using vector similarity (<#> operator).

## Row Level Security (RLS) & Policies

RLS is enabled for all sensitive tables (profiles, watchlists, watchlist\_items, ratings, follows). Policies enforce that:

Users can only modify or delete their own data.

Public and followers-only content is shared according to visibility.

All profiles are viewable but only self-editable.

---

# Database-Migrations

## Database Version Control with Supabase

To have a successful project, we need a good **database version control and migrations system**. We chose to use the built-in Supabase migration system, which works off **pure SQL syntax**, unlike some popular ORMs.

### Motivation

Let's say we have a set database. We have tables, functions, indexes, and other elements, but we decide we need to change the profiles table and add a new attribute. We could simply run an `ALTER TABLE` query and add the attribute, or we could use the Supabase Dashboard.

### Problem

Our database gets **extremely cluttered** this way. We don't know exactly what we have until we perform a DB dump, which is too verbose for a human to read (it can be over 100,000+ lines). If we use standard migrations (writing our own queries and pushing them to the DB), we won't have a **readable database**, and a person joining the project will have a hard time understanding the database structure.

### Fix: Using Supabase Declarative Schema

We use the **Supabase declarative schema**. We have a single SQL file that is the **“ONLY source of truth.”** This means the declarative schema and the production Supabase DB will always be **1:1**.

For example, if we want to add an attribute to a table, we don't use `ALTER TABLE` queries. We go to our source-of-truth SQL file, find the `CREATE TABLE` query for the table we want to change, and then we simply add another row, just as if we were creating it initially. Supabase will then **diff** the file and the remote Supabase DB and calculate exactly what it needs to do to make it **1:1 WITHOUT LOSING DATA!** Then, we simply push those Supabase-created queries, and that's it.

### How to Implement the Declarative Schema

#### 1. Install the Supabase CLI tool:

- <https://github.com/supabase/cli>

#### 2. Initialize your project:

- Go to your project directory and call `supabase init`.
- This will create a `supabase` folder with a `config.toml` file where you can edit migration settings (we're keeping it default for now).

- **Note:** Instead of a single source-of-truth file, you can split them into multiple files (e.g., one file per table). If you run the Supabase query, the order of files scanned is alphabetical. Because one file can reference a table that doesn't yet exist, you can set up the exact scan order in `config.toml`, which is extremely useful.

#### 3. Log in to Supabase:

- From the project root directory (you will always use this directory for calling Supabase commands), call `supabase login` and enter the verification number you get from the browser.
  - 4. Link your project:**
    - Call `supabase link` and select your project.
    - This will create a `.temp` folder inside the `supabase` folder containing the credentials to your Supabase project (it's included in `.gitignore` by default when you call `supabase init`).
  - 5. Pull, Diff, and Push:**
    - Call `supabase db pull`. This creates a migration file inside a new `migrations` folder. That file contains every single command needed to build up your remote project from start to finish.
    - Next, create a new folder named `schemas` next to the `migrations` folder and insert your source-of-truth file there.
    - Call `supabase db diff -f <name_of_the_migration>`. This creates a new migration that includes everything needed to sync the recently pulled migration with your source-of-truth schema.
    - Review that file to ensure everything is correct, and then call `supabase db push`.
    - **Note:** We already had an upstream Supabase DB, which is why we needed to call `pull` first, then insert our source-of-truth file, and call `diff`. If you had the `schemas` folder before pulling, it would mess up the migrations by mixing the schema file with the initial remote migration file. If you did **not** have an upstream Supabase DB, you don't need to call `pull`.
  - 6. Check the Dashboard:**
    - You can now check the Supabase Dashboard. All changes will be applied, and your data will be saved!
    - To make future changes, just go into the source-of-truth file, make the change, call `supabase db diff -f <migration_name>`, and then `supabase db push`.
- 

## Migrations Management Strategy

You can always keep the migration files, as it's good practice. However, if you prefer a cleaner look, you can delete them and call the `pull` command every time before making changes (this is the main advantage of the declarative schema).

- **Keeping Migration Files:** They will pile up, and each new one will be an **add-on** to the previous one.
- **Deleting Migration Files:** Deleting them all and pulling the DB will create one migration file that contains everything. Then, with `diff`, you create the add-on migration you want.

Before the declarative schema, if you lost one migration file, it would be extremely hard to keep track of your database without performing a DB dump. This way, you always have the source of truth exactly how you want it.

---

## Initial-Phase-&-Authentication

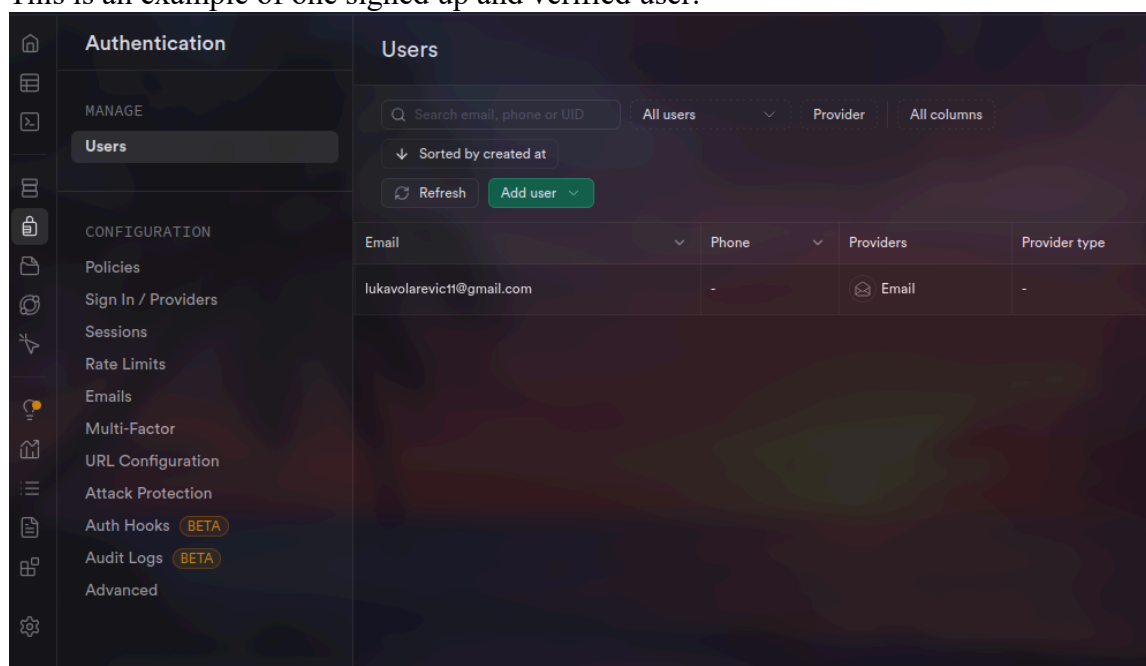
# Stack

The stack we are going to use for this project will be Next.js, Supabase and Vercel. To initiate the project we called the `npx create-next-app` where we chose the default settings (App router, Typescript, Tailwind, eslint, Turbopack). Then we called `npx shadcn@latest init` to get the shadcn UI Library in our code base. Then we called `npx shadcn@latest add https://supabase.com/ui/r/password-based-auth-nextjs.json`, this command completely sets up Supabase for our project including server/client side rendering, middleware and complete auth routing that we are gonna customize later. Now we can see the `/auth` route in our codebase.

# Auth

Authentication has been handled with Supabase Auth. Previously we called `npx shadcn@latest add https://supabase.com/ui/r/password-based-auth-nextjs.json` which initialized our Auth routes and we can now enter the login page together with “forgot password” page, “sign up” page etc. Auth settings are managed over Supabase Dashboard where we can choose the password security level, fake email filtering, etc.

This is an example of one signed up and verified user:



We can directly edit the Auth routes in `page.tsx` per endpoint. This will be our starting Auth for our app.

Supabase also allows us to include Google as our login provider, so after creating the Google Cloud login client, we just import the Secret Key into Supabase Dashboard and we can now use Google’s one-click login.

Important thing to note are the redirects. After sign-up or login, our goal is to get the user redirected to `/` page, that means we need to do some backend work to allow that redirects (since all redirects are blocked by default). In GCP console we need to allow redirects to pages that start with `https://dobby-sand.vercel.app/` (`http://localhost:3000/` is there by default), and in Supabase console we have to allow these redirects: `https://dobby-sand.vercel.app/` and `http://localhost:3000/`. Now we have working redirects for both production and development stages.

# Dynamic-Behavior-of-the-Application

## Dynamic Behavior of the Application

---

The dynamic behavior of an application refers to how objects within the system evolve over time, including transitions between different states. This encompasses activities, events, decisions, and interactions within the application. UML state diagrams allow visualization of these changes and make it easier to understand the system's dynamics.

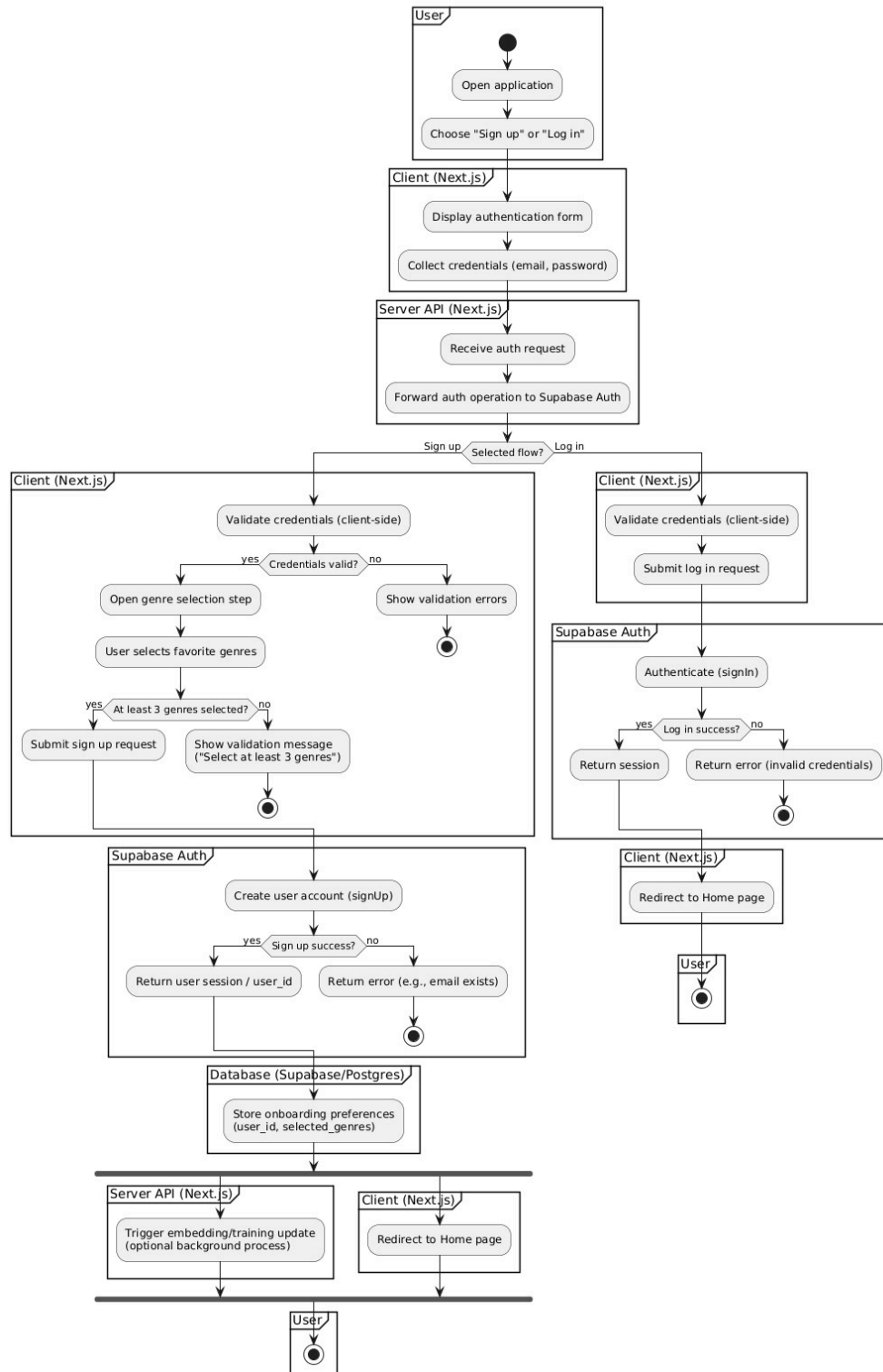
Understanding state changes is essential for the proper functioning of the application as it provides insight into interactions between objects, components, and users during the system's operation. By using UML state and activity diagrams, it is possible to visualize transitions and states of objects, identify potential issues, ensure accurate implementation, and improve communication among team members.

### UML State Diagrams

UML state diagrams are necessary for understanding the dynamic behavior of the system. They clearly show the changes in the states of objects over time, depending on events and conditions.

### UML Activity Diagrams

Activity diagrams depict the flow of execution of a particular process. In addition to understanding the flow of data within the application, they are used for business process analysis.

**Business Process: Authentication & Onboarding (Sign up with Genre Selection / Log in)**

ActivityDiagram

# DobbySense

## DobbySense: Advanced Hybrid Recommender System

**DobbySense** is the algorithmic heart of the Dobby platform. It is a sophisticated recommendation engine built on PyTorch that moves beyond simple Collaborative Filtering by integrating content-based features (Genres) directly into the latent factor learning process. This hybrid approach solves the classic “Cold Start” problem while maintaining the high accuracy of Matrix Factorization.

---

# 1. Mathematical Foundation

## 1.1 Standard Matrix Factorization (The Baseline)

At its core, DobbySense builds upon Probabilistic Matrix Factorization (PMF). In a standard MF model, we approximate the rating matrix  $R$  as the product of two lower-rank matrices: User Factors  $P$  and Item Factors  $Q$ .

The predicted rating  $\hat{r}_{ui}$  for user  $u$  and item  $i$  is given by:

$$\hat{r}_{ui} = \mathbf{p}_u \cdot \mathbf{q}_i + b_u + b_i + \mu$$

Where:

- $\mathbf{p}_u \in \mathbb{R}^k$ : Latent vector for user  $u$ .
- $\mathbf{q}_i \in \mathbb{R}^k$ : Latent vector for item  $i$ .
- $b_u, b_i$ : Bias terms for user and item (capturing individual tendencies, e.g., a harsh critic or a universally acclaimed movie).
- $\mu$ : Global average rating.
- $k$ : The number of latent factors (Dimension of the embedding space).

## 1.2 The Hybrid Extension (Genre-Aware Embeddings)

Standard MF fails when an item has few ratings (item cold start) or when we want to capture semantic relationships explicitly. DobbySense introduces a **Genre Projector Layer**.

We represent the genres of movie  $i$  as a multi-hot vector  $\mathbf{g}_i \in \{0, 1\}^N$ , where  $N$  is the number of total genres. The model learns a linear transformation to map these discrete logical features into the continuous latent space of the model.

Let  $f_{\text{genre}}(\cdot)$  be our learned linear mapping:

$$f_{\text{genre}}(\mathbf{x}) = \mathbf{W}_g \mathbf{x} + \mathbf{b}_g$$

The model modifies the effective item representation used for prediction. The effective item vector  $\mathbf{q}_i'$  becomes a weighted sum of its unique identity vector and its semantic content vector:

$$\mathbf{q}_i' = \mathbf{q}_i + \lambda \cdot f_{\text{genre}}(\mathbf{g}_i)$$

Substituting this back into the prediction equation:

$$\hat{r}_{ui} = \mathbf{p}_u \cdot (\mathbf{q}_i + \lambda(\mathbf{W}_g \mathbf{g}_i + \mathbf{b}_g)) + b_u + b_i + \mu$$

- $\mathbf{W}_g$ : A learnable weight matrix of shape  $(k \times N)$  representing genre embeddings.

- $\lambda$ : A hyperparameter (GENRE\_WEIGHT) controlling how much genres influence the recommendation versus pure user behavior.

## 2. Neural Network Architecture

The system is implemented as a PyTorch `nn.Module` (`HybridMatrixFactorization`) with the following components:

1. **User Embedding Layer:** `nn.Embedding(num_users, latent_dim)`
  - Learns a specialized vector for every known user ID.
2. **Movie Identity Layer:** `nn.Embedding(num_movies, latent_dim)`
  - Learns unique characteristics of a movie that *aren't* explained by its genres (e.g., acting quality, direction style).
3. **Genre Projection Layer:** `nn.Linear(num_genres, latent_dim)`
  - Learns what “Action” or “Romance” looks like in the abstract vector space.
4. **Bias Layers:**
  - User and Movie biases are learned as scalar embeddings (`latent_dim=1`).
5. **Regularization:**
  - **Dropout:** Applied to user and item vectors during training to prevent overfitting.
  - **L2 Regularization** (Weight Decay): Applied via the Adam optimizer to constrain the magnitude of the latent vectors.

## 3. The “Fold-In” Algorithm (Solving User Cold Start)

A critical limitation of Matrix Factorization is that it cannot make predictions for a user who has never rated anything ( $\mathbf{p}_u$  is unlearned). DobbySense solves this with an algebraic **Fold-In** technique.

### 3.1 Concept

Since we have learned a mapping between **Genres** and the **Latent Space** ( $\mathbf{W}_g$ ), we can reverse-engineer a user vector for a new user based on their stated preferences.

If a new user selects “Sci-Fi” and “Thriller” during onboarding, we don’t need to retrain the model. We can synthesize their vector immediately.

### 3.2 The Algorithm

1. **Export:** The trained Genre Layer weights  $\mathbf{W}_g$  and bias  $\mathbf{b}_g$  are exported to the database.
2. **User Input:** User selects a set of preferred genres  $S_{prefs}$ .
3. **Synthesis:** We calculate the “Center of Gravity” for those genres in the latent space.

$$\mathbf{p}_{new} \approx \frac{1}{|S_{prefs}|} \sum_{j \in S_{prefs}} (\mathbf{W}_{g}[j] + \mathbf{b}_g)$$

This synthesized vector  $\mathbf{p}_{new}$  is mathematically compatible with all existing movie vectors  $\mathbf{q}_i'$ . We can immediately perform Cosine Similarity searches to generate recommendations:

$$\text{Score}(u_{new}, i) = \cos(\mathbf{p}_{new}, \mathbf{q}_i')$$

This allows Dobby to provide highly personalized recommendations **milliseconds** after account creation.

## 4. Training Pipeline

The training process is orchestrated via Jupyter Notebooks (`matrixFactorization.ipynb`) rather than scripts, allowing for interactive analysis and visualization.

### 4.1 Datasets

The model is trained on a massive merge of movie and TV show data:

- 1. **The Movies Dataset (Kaggle):**
  - Source: [Rounak Banik on Kaggle](#)
  - Content: Metadata and **26M+ ratings** from 270,000 users for all movies listed in the Full MovieLens Dataset. This provides the deep historical behavior data needed for collaborative filtering.
- 2. **Full TMDb TV Shows Dataset (Kaggle):**
  - Source: [Asaniczka on Kaggle](#)
  - Content: Comprehensive metadata for over 150,000 TV shows. This allows Dobby to recommend shows alongside movies using the same genre-based vector space.
- 3. **Dynamic Data:** Real-time user ratings fetched from Supabase (`movie_ratings` table) are merged with the Kaggle data at training time, ensuring the model adapts to current Dobby users.

### 4.2 Hyperparameters Explained

Parameter	Default	Explanation
N_FACTORS	64	The size of the embedding vector (e.g., length of the array representing a user/movie). Higher = more nuance, but harder to train and more memory. 64 is a sweet spot for 20M ratings.
EPOCHS	5	How many times the model sees the entire dataset. Since we have 26M ratings, 5 epochs is sufficient for convergence without overfitting.
LEARNING_RATE	1e-3	The step size for the Adam optimizer. 0.001 is standard for Adam; too high = diverge, too low = slow training.
BATCH_SIZE	2048	Number of rating samples processed at once. Large batch sizes (2048+) are crucial for GPU efficiency.
GENRE_WEIGHT (λ)	0.5	<b>Critical Hybrid Parameter.</b> Controls how much “Genre” matters vs. “Specific Movie Identity”. <ul style="list-style-type: none"><li>• If 0.0: Pure Collaborative Filtering (ignores genres).</li><li>• If 1.0: Strong genre smoothing (good for cold start items).</li></ul>

Parameter	Default	Explanation
MODEL_DROPOUT	<b>0.1</b>	Randomly zeroes out 10% of the embedding elements during training. This forces the model to learn robust representations and prevents memorizing the training data.
WEIGHT_DECAY	<b>1e-6</b>	L2 Regularization. Adds a penalty for large weights, encouraging simpler models and preventing “exploding” embeddings.
CLIP_GRAD_NORM	<b>None</b>	Gradient Clipping threshold. If set (e.g., 1.0), limits the maximum size of parameter updates to prevent instability during training bursts.
EMBEDDING_INIT_STD	<b>0.01</b>	The standard deviation for the normal distribution used to initialize the embedding matrices. Small random values are crucial for breaking symmetry at the start of learning.
TEST_SIZE	<b>0.0</b>	Percentage of data held back for validation. Set to <b>0.0</b> for production training to use maximal data for the best final model.
RANDOM_STATE	<b>42</b>	A fixed seed for the random number generator, ensuring that data splitting and weight initialization are reproducible across runs.
CHUNK_SIZE	<b>500</b>	The number of records sent to Supabase in a single batch insert. Tuned to avoid HTTP 500/Timeout errors during the export phase.

## 4.3 Training Strategy

1. **Data Ingestion:** Load CSVs from GCS + Fetch Supabase table.
2. **Preprocessing:**
  - **Re-indexing:** Maps raw user/movie IDs to continuous 0..N integers. This is crucial for embedding lookup efficiency.
  - **Multi-Hot Encoding:** Converts raw genre strings (e.g., “Action|Sci-Fi”) into a binary matrix where each column represents a genre.
  - **Alignment:** Ensures that the `genre_matrix` is perfectly aligned with the `movie_idx` used in the PyTorch model.
3. **Training:** Run Adam optimizer on `MSELoss`.
4. **Export:** Save artifacts to Supabase.

## 5. Inference & Recommendation Engine (Serving Phase)

Once the model has finished training and exported the embeddings to Supabase, the **Next.js** application takes over. The serving phase isn’t just a simple database query; it is a multi-stage pipeline designed to balance mathematical relevance with human quality standards.

### 5.1 The Vector Search Strategy

Since both Users and Movies now inhabit the same 64-dimensional latent space, the relevance of a movie  $m$  to a user  $u$  is the cosine of the angle between their vectors:

$$\text{Similarity}(u, m) = \frac{\|\mathbf{u}\| \cdot \|\mathbf{m}\|}{\|\mathbf{u}\| \cdot \|\mathbf{m}\|}$$

The system calls a PostgreSQL RPC function (`get_top_movies_for_user`) which performs a fast approximate nearest neighbor search (ANN) using the `pgvector` extension.

- **Input:** User Embedding Vector (fetched or computed via Fold-In).
- **Process:** Index scan on the vectors column.
- **Output:** Top  $K$  raw ID candidates (e.g., top 20 movies) closest to the user's taste.

## 5.2 The FBS (Filter, Better-Similar) Algorithm

A common issue with pure Collaborative Filtering is that it may recommend “mathematically correct” but “qualitatively poor” items. For example, it might recommend a C-grade Sci-Fi movie simply because it has the perfect vector coordinates for a Sci-Fi fan.

To solve this, DobbySense uses the **FBS Algorithm** (Filter & Better-Similar) defined in `fbs.functions.ts`.

### Step A: Quality Filtering (Filter)

Every candidate item is passed through a heuristic filter (`isBadMovie / isBadShow`) that checks against a configuration config (e.g., `minVoteAverage`, `minYear`).

An item is flagged as “**Bad**” if:

1. **Low Rating:** `vote_average < minVoteAverage` (e.g., 5.0).
2. **Too Old:** `year < minYear` (e.g., 1980).
3. **Mid-Tier Trap:** `vote_average < midTierRating AND year <= midTierYear`.
  - *Rationale:* Older movies (e.g., from 1990) are acceptable if they are classics (high rating), but mediocre old movies are usually irrelevant to modern users.

### Step B: Smart Replacement (Better-Similar)

Instead of simply discarding a “Bad” item (which would shrink the recommendation list), the system attempts to **upgrade** it.

1. **Fetch Similar:** The system queries TMDb for items similar to the “Bad” candidate.
2. **Filter & Sort:** The similar items are themselves filtered (removing bad ones) and sorted by **Popularity**.
3. **Stochastic Selection:** We pick a random item from the **Top 5** best alternatives.
  - *Why Top 5?* Picking #1 every time reduces variety. Randomness ensures the feed feels fresh.

## 5.3 Fuzzy Logic Layer: Cold Start & Ranking

To further improve recommendation quality—especially for new users or users with sparse history—DobbySense incorporates a fuzzy logic layer for ranking and cold start scenarios.

### Fuzzy Membership Functions

For both genre match ratio and popularity, fuzzy membership functions are used to assign a degree of membership to three categories: low, mid, and high. For a value  $x$  in  $[0, 1]$ :

- **Low:** 1 if  $x \leq 0.2$ , linearly decreasing to 0 at  $x = 0.5$
- **Mid:** 0 if  $x \leq 0.2$ , peaks at  $x = 0.5$ , 0 again at  $x = 0.8$
- **High:** 0 if  $x \leq 0.5$ , linearly increasing to 1 at  $x = 0.8$

## Fuzzy Inference and Scoring

A 3x3 rule matrix combines genre and popularity memberships to produce a fuzzy score:

- High genre & High popularity: 1.0 (Excellent)
- High genre & Mid popularity: 0.9 (Very Good)
- Mid genre & High popularity: 0.8 (Good)
- Mid genre & Mid popularity: 0.6 (Okay)
- High genre & Low popularity: 0.5 (Hidden gem)
- Low genre & High popularity: 0.4 (Fair)
- Mid genre & Low popularity: 0.3 (Poor)
- Low genre & Mid popularity: 0.2 (Very Poor)
- Low genre & Low popularity: 0.1 (Lowest priority)

The final fuzzy score is a weighted sum of all rule outputs, normalized by the sum of rule weights.

## Application in Ranking

- **For users with genre preferences but little/no rating history (cold start):**
  - The fuzzy score is weighted heavily (e.g., 85%) in the final ranking, with a small base score for list position.
- **For users with rating history:**
  - The fuzzy score is blended with the embedding-based rank and a small random jitter for diversity.

This approach ensures that recommendations are both mathematically relevant and intuitively appealing, even for new users.

## 5.4 Pipeline Execution Flow

The DobbySense recommendation pipeline consists of three main stages, executed in order:

**1. Embedding-based Candidate Selection** - Retrieve top candidate movies and shows for the user using vector search in the shared embedding space (via `get_top_movies_for_user` and `get_top_shows_for_user`).

**2. FBS (Filter, Better-Similar) Post-Processing** - Remove or upgrade low-quality items using the FBS algorithm: filter out “bad” candidates and, where possible, replace them with better, similar alternatives.

**3. Fuzzy Logic Scoring and Ranking** - Apply fuzzy logic to score and rank the remaining candidates, taking into account genre match, popularity, and user preferences. This ensures recommendations are both mathematically relevant and intuitively appealing, especially for new or cold-start users.

### Summary Flow:

Embeddings (vector search)

↓

FBS (Filter, Better-Similar)



## Fuzzy Logic Scoring & Ranking

This logic lives in valid Next.js API routes (e.g., `api/recommendation-engine`):

1. **Trigger:** User opens “For You” page (`/home`).
  2. **Check Cache:** System checks `movie_recommendations` table for fresh (  $< 24\text{h}$  old) rows.
  3. **Compute:** If cache miss:
    - Calculate Vector Logic (`get_top_movies_for_user`).
    - Run **FBS Post-Processing** (Javascript Layer).
    - Apply fuzzy logic scoring and ranking.
    - Fetch metadata for final list.
  4. **Cache & Return:** Result is saved to `movie_recommendations` and returned to UI.
- 

## 6. References & Further Reading

- Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems.
  - He, X., et al. (2017). Neural Collaborative Filtering.
  - PyTorch `nn.Embedding` documentation.
- 

# Installation

## Dobby Project Setup Guide

This guide covers the installation, configuration, and running procedures for the **Dobby** project. The project consists of a Next.js frontend (`dobby-next.js`) and a Matrix Factorization recommendation engine (`dobbySense`).

## Prerequisites

Ensure you have the following installed on your machine:

- **Node.js** (v18.17 or later recommended) - [Download](#)
  - **npm** (comes with Node.js)
  - **Python** (v3.9 or later) - [Download](#)
  - **Docker** (Optional, for containerizing the recommendation engine) - [Download](#)
  - **Supabase CLI** (Optional, for local backend development) - [Installation Guide](#)
- 

## 1. Web Application (`dobby-next.js`)

This is the main application containing the frontend and API routes.

## Installation

1. Navigate to the project directory:

```
cd dooby-next.js
```

2. Install dependencies:

```
npm install
```

## Environment Configuration

Create a `.env` (or `.env.local`) file in the `dooby-next.js` directory with the following variables. You will need to obtain API keys for Supabase, TMDB, and Watchmode.

```
# Supabase Configuration
NEXT_PUBLIC_SUPABASE_URL=your_supabase_url
NEXT_PUBLIC_SUPABASE_PUBLISHABLE_OR_ANON_KEY=your_supabase_anon_key

# The Movie Database (TMDB) API
TMDB_API_KEY=your_tmdb_api_key
TMDB_ACCESS_TOKEN=your_tmdb_access_token

# Watchmode API
WATCHMODE_API=your_watchmode_api_key
```

## Running Development Server

To start the local development server:

```
npm run dev
```

Open <http://localhost:3000> with your browser to see the result.

## Building for Production

To build the application for production:

```
npm run build
```

To start the production server after building:

```
npm start
```

## Linting

To check for code issues:

```
npm run lint
```

---

## 2. Recommendation Engine (dobbySense)

The doobbySense folder contains the Machine Learning model for movie recommendations. We recommend running it in Google Colab as it is very compute intensive.

### Setup (Python)

1. Navigate to the directory:

```
cd doobbySense
```

2. Create and activate a virtual environment:

```
python -m venv venv  
source venv/bin/activate # On Windows: venv\Scripts\activate
```

3. Install dependencies:

```
pip install -r trainer/requirements.txt
```

### Running the Model

The main training logic is located in `trainer/`. You can explore the `matrixFactorization.ipynb` notebook using Jupyter or VS Code, or run Python scripts if available.

### Docker Support

Refer to the `doobbySense/README.md` for detailed Docker instructions. Typically, to build the container:

```
docker build -t doobby-recommender .
```

---

## 3. Database (Supabase)

The supabase folder contains database migrations and configuration. We suggest using the supabase dashboard.

### Deployment

If you are using a hosted Supabase project, you can link it and push migrations:

```
supabase link --project-ref your-project-ref  
supabase db push
```

---

## Project Structure Overview

- `/dobby-next.js` - Main Next.js application (Frontend & API).
- `/dobbySense` - Python based recommendation engine (ML Model).

- /supabase - Database migrations and configuration.
  - /assets - Static assets for the project documentation/repo.
- 

# Deployment

## Deployment Guide

This project uses **Vercel** for continuous deployment.

### Automatic Deployment

The deployment pipeline is fully automated and connected to the GitHub repository.

- **Trigger:** Every time code is pushed to the main branch.
- **Action:** Vercel automatically builds and deploys the latest version of the application.
- **URL:** The live application is updated to reflect the changes in the main branch.

### How it Works

1. **Development:** Developers work on feature branches.
2. **Pull Request:** Changes are reviewed via Pull Requests.
3. **Merge:** Once a Pull Request is merged into main, the deployment hook is triggered.
4. **Build:** Vercel runs the `npm run build` command defined in `package.json`.
5. **Live:** Upon successful build, the new version replaces the production instance.

### Environment Variables

For the deployment to succeed, the necessary environment variables (matching those in `.env.local` but tailored for production) must be configured in the **Vercel Project Settings**.

Required variables include: - NEXT\_PUBLIC\_SUPABASE\_URL -  
NEXT\_PUBLIC\_SUPABASE\_PUBLISHABLE\_OR\_ANON\_KEY - TMDB\_API\_KEY - TMDB\_ACCESS\_TOKEN  
- WATCHMODE\_API

### Access the Deployment

You can access the deployed application at the production URL provided in the Vercel dashboard, assuming you have authorized access.

---