I'll analyze the current application's systems by function and document them in a clear, step-by-step narrative format. Let me start by examining the main application files to understand the system architecture.

[3 tools called]

Now let me examine the core JavaScript modules to understand the system architecture:

[4 tools called]

Now let me examine the data management and search systems:

[3 tools called]

Now let me examine the UI and layout systems:

[3 tools called]

Now let me create a comprehensive documentation of the application systems. Based on my analysis, I'll create a structured narrative that covers all the major systems:

[1 tool called]

# Flicklet TV & Movie Tracker - System Documentation

## Overview

Flicklet is a Progressive Web Application (PWA) for tracking TV shows and movies. The application is built with vanilla JavaScript, uses Firebase for authentication and cloud storage, integrates with The Movie Database (TMDB) API for content data, and provides a responsive interface for managing personal watchlists.

---

## 1. Core Application Architecture

### 1.1 Application Initialization

The application follows a modular initialization pattern with strict script loading order:

\*\*Step 1: Configuration Loading\*\*

- `firebase-config.js` loads Firebase configuration

- `tmdb-config.js` loads TMDB API configuration

- `js/config.js` provides centralized configuration management

\*\*Step 2: Firebase Initialization\*\*

- `js/firebase-init.js` initializes Firebase app, authentication, and Firestore

- Sets up browser local persistence for authentication

- Exposes global Firebase instances (`window.firebaseApp`, `window.firebaseAuth`, `window.firebaseDb`)

\*\*Step 3: Core Utilities\*\*

- `js/utils.js` defines the global `appData` structure and persistence functions

- `js/functions.js` provides core application functions and UI management

- `js/app.js` contains the main `App` object with initialization logic

\*\*Step 4: Feature Modules\*\*

- Search functionality (`js/search.js`)

- Authentication handling (`js/auth.js`)

- Data initialization (`js/data-init.js`)

- Component systems (poster cards, modals, etc.)

### 1.2 Global Data Structure

The application uses a centralized data model stored in `window.appData`:

```javascript

window.appData = {

settings: {

displayName: '',

lang: 'en',

theme: 'light',

pro: false,

notif: {}

},

tv: {

watching: [],

wishlist: [],

watched: []

},

movies: {

watching: [],

wishlist: [],

watched: []

},

searchCache: [],

activeTagFilters: new Set()

}

```

### 1.3 Application Lifecycle

1. \*\*Bootstrap\*\*: Load configuration and Firebase

2. \*\*Data Loading\*\*: Load from localStorage, then sync with Firebase if authenticated

3. \*\*UI Initialization\*\*: Set up event listeners, render initial content

4. \*\*Authentication Check\*\*: Determine user state and update UI accordingly

5. \*\*Content Rendering\*\*: Load appropriate tab content based on current state

[1 tool called]

## 2. Authentication and User Management Systems

### 2.1 Authentication Flow

The authentication system uses Firebase Authentication with Google OAuth as the primary provider:

\*\*Step 1: Authentication Initialization\*\*

- Firebase Auth is initialized with browser local persistence

- Auth state listener is set up to monitor sign-in/sign-out events

- Global auth state is managed through `UserViewModel` object

\*\*Step 2: Sign-In Process\*\*

1. User clicks the account button in the header

2. `startSignIn()` function is called from `js/auth.js`

3. Google OAuth popup is attempted first, with redirect fallback

4. On successful authentication, `processUserSignIn()` is called

5. User database entry is created/updated in Firestore

6. User data is loaded from cloud storage

7. UI is updated to reflect authenticated state

\*\*Step 3: User State Management\*\*

The `UserViewModel` object centralizes user state:

- Tracks authentication status (`isAuthenticated`)

- Manages user profile data (`displayName`, `email`, `avatarUrl`)

- Updates UI elements when auth state changes

- Controls access to authenticated features

### 2.2 User Data Synchronization

\*\*Cloud Storage Integration:\*\*

- User data is stored in Firestore under `users/{uid}` collection

- Data structure includes `watchlists`, `settings`, and `profile` information

- Local storage serves as backup and offline fallback

- Automatic sync occurs on sign-in and data changes

\*\*Data Loading Priority:\*\*

1. Load from localStorage first (immediate UI response)

2. If authenticated, load from Firebase Firestore

3. Merge cloud data with local data, prioritizing cloud data

4. Update UI with merged data

### 2.3 Authentication UI Updates

The system automatically updates UI elements based on auth state:

- Account button shows user name when signed in

- Settings access is controlled by authentication status

- Personalized greetings and content are displayed

- Pro features are enabled/disabled based on user status

[1 tool called]

## 3. Data Persistence and Storage Systems

### 3.1 Dual Storage Architecture

The application implements a dual storage strategy for reliability and performance:

\*\*Local Storage (Primary)\*\*

- Immediate data access for responsive UI

- Serves as offline backup

- Stores data in `localStorage` under key `'flicklet-data'`

- Automatic saving on every data change

- Fallback for when Firebase is unavailable

\*\*Firebase Firestore (Cloud)\*\*

- Cross-device synchronization

- User-specific data storage

- Automatic backup and recovery

- Real-time updates across devices

- Data structure: `users/{uid}` document with nested collections

### 3.2 Data Synchronization Process

\*\*Step 1: Initial Data Loading\*\*

1. Load data from localStorage immediately

2. If user is authenticated, load from Firebase

3. Merge cloud data with local data (cloud takes precedence)

4. Dispatch `app:data:ready` event to update UI

\*\*Step 2: Data Saving\*\*

1. Update `window.appData` object

2. Save to localStorage immediately

3. If authenticated, save to Firebase Firestore

4. Handle errors gracefully with fallback to localStorage only

\*\*Step 3: Conflict Resolution\*\*

- Cloud data always takes precedence over local data

- Last-write-wins strategy for concurrent updates

- User is notified of any sync issues

### 3.3 Data Structure and Validation

\*\*Core Data Model:\*\*

```javascript

{

watchlists: {

tv: { watching: [], wishlist: [], watched: [] },

movies: { watching: [], wishlist: [], watched: [] }

},

settings: {

displayName: '',

lang: 'en',

theme: 'light',

pro: false,

notif: {}

},

profile: {

email: '',

displayName: '',

photoURL: ''

}

}

```

\*\*Data Validation:\*\*

- Required fields are validated before saving

- Invalid data is filtered out or replaced with defaults

- Error handling prevents data corruption

- Backup mechanisms ensure data recovery

### 3.4 Import/Export System

\*\*Export Functionality:\*\*

- JSON export of complete user data

- CSV export for Pro users

- Includes all lists, settings, and preferences

- Compatible with backup and migration needs

\*\*Import Functionality:\*\*

- JSON import with validation

- Legacy format support for backward compatibility

- Data merging with existing data

- Automatic UI refresh after import

[1 tool called]

## 4. Search and TMDB API Integration

### 4.1 TMDB API Client Architecture

The application integrates with The Movie Database (TMDB) API for content discovery and metadata:

\*\*API Configuration:\*\*

- API key stored in `tmdb-config.js`

- Base URL: `https://api.themoviedb.org/3`

- Language: English (en-US) by default

- Rate limiting and error handling implemented

\*\*Core API Functions:\*\*

- `tmdbGet(path, params)` - Generic API request handler

- `searchTMDB(query)` - Search for movies and TV shows

- `getMovieDetails(id)` - Get detailed movie information

- `getTVDetails(id)` - Get detailed TV show information

- `getPopularContent()` - Get trending content

### 4.2 Search System Implementation

\*\*Step 1: Search Input Handling\*\*

1. User types in search input field

2. Debounced input handler (500ms delay) prevents excessive API calls

3. Input validation ensures minimum query length

4. Search state management prevents duplicate requests

\*\*Step 2: API Request Processing\*\*

1. Query is sent to TMDB search endpoint

2. Both movie and TV show results are fetched

3. Results are combined and sorted by relevance

4. Error handling provides fallback for API failures

\*\*Step 3: Search Results Display\*\*

1. Results are rendered using poster card components

2. Pagination is handled for large result sets

3. Search state is maintained for result navigation

4. Clear search functionality resets to previous tab

### 4.3 Content Discovery System

\*\*Discover Tab Functionality:\*\*

- Loads popular movies and TV shows from TMDB

- Filters out items marked as "Not Interested"

- Randomizes results for variety

- Displays up to 20 recommendations

- Uses poster card grid layout

\*\*Content Filtering:\*\*

- Genre-based filtering

- "Not Interested" list integration

- User preference-based filtering

- Quality and popularity scoring

### 4.4 Image and Media Handling

\*\*Poster Image Management:\*\*

- TMDB image URLs are constructed dynamically

- Multiple image sizes supported (w200, w300, w500)

- Fallback to placeholder images for missing posters

- Lazy loading for performance optimization

\*\*Media Type Detection:\*\*

- Automatic detection of movies vs TV shows

- Appropriate icons and labels for each type

- Different handling for different media types

- Consistent UI treatment across all content

[1 tool called]

## 5. UI Navigation and Tab System

### 5.1 Tab Architecture

The application uses a tab-based navigation system with five main sections:

\*\*Tab Structure:\*\*

- \*\*Home\*\*: Dashboard with previews and recommendations

- \*\*Currently Watching\*\*: Active TV shows and movies

- \*\*Want to Watch\*\*: Wishlist of future content

- \*\*Already Watched\*\*: Completed content

- \*\*Discover\*\*: Personalized recommendations

- \*\*Settings\*\*: User preferences and account management

### 5.2 Tab Management System

\*\*Step 1: Tab Initialization\*\*

1. Tab buttons are defined in HTML with specific IDs

2. Event delegation handles all tab clicks

3. Active tab state is managed by CSS classes

4. Tab content sections are shown/hidden based on selection

\*\*Step 2: Tab Switching Process\*\*

1. User clicks a tab button

2. `switchToTab(tabName)` function is called

3. Previous tab is hidden, new tab is shown

4. Tab-specific content is loaded

5. URL state is updated (if applicable)

6. Tab badges are updated with current counts

\*\*Step 3: Content Loading\*\*

- Each tab has its own content loading function

- Lazy loading for performance optimization

- Content is cached to prevent unnecessary reloads

- Error handling for failed content loads

### 5.3 Tab Badge System

\*\*Dynamic Badge Updates:\*\*

- Badges show count of items in each list

- Real-time updates when items are added/removed

- Color coding for different states

- Accessibility support with ARIA labels

\*\*Badge Calculation:\*\*

- Currently Watching: Count of active items

- Want to Watch: Count of wishlist items

- Already Watched: Count of completed items

- Updates triggered by data changes

### 5.4 Search State Management

\*\*Search Integration:\*\*

- Search results overlay other tab content

- Previous tab is remembered for return navigation

- Search state is maintained during session

- Clear search returns to previous tab

\*\*Search UI Flow:\*\*

1. User enters search query

2. Search results section becomes visible

3. Other tabs remain accessible but hidden

4. Clear search returns to previous tab

5. Search state is reset

[1 tool called]

## 6. Home Page Layout and Content Loading

### 6.1 Home Page Structure

The home page is organized into five distinct groups with specific purposes:

\*\*Group 1: Your Shows\*\*

- Currently Watching Preview: Horizontal scrollable preview of active content

- Next Up This Week: Upcoming episodes and releases (feature-flagged)

\*\*Group 2: Community\*\*

- Community Player: Video content placeholder

- Game Cards: FlickWord and Daily Trivia games

\*\*Group 3: For You\*\*

- Personalized Recommendations: Based on user preferences

- Curated Sections: Trending and staff picks

\*\*Group 4: In Theaters\*\*

- Current theatrical releases

- Location-based content (requires geolocation)

\*\*Group 5: Feedback\*\*

- User feedback and support options

- Bug reporting and feature suggestions

### 6.2 Content Loading System

\*\*Step 1: Home Content Initialization\*\*

1. `loadHomeContent()` function is called

2. Performance monitoring starts

3. Content loading is sequenced for optimal performance

4. Feature flags determine which sections to load

\*\*Step 2: Currently Watching Preview\*\*

1. `renderCurrentlyWatchingPreview()` loads active content

2. Items are displayed in horizontal scrollable cards

3. Placeholder message shown when no content

4. Real-time updates when content changes

\*\*Step 3: Curated Content Loading\*\*

1. Curated rows are loaded asynchronously

2. TMDB API provides trending and popular content

3. Content is filtered based on user preferences

4. Lazy loading improves mobile performance

### 6.3 Responsive Layout System

\*\*Mobile Optimization:\*\*

- Mobile-first CSS design

- Touch-friendly interface elements

- Optimized image loading

- Reduced content on smaller screens

\*\*Desktop Enhancement:\*\*

- Multi-column layouts

- Hover effects and interactions

- Enhanced visual hierarchy

- Additional content sections

### 6.4 Performance Optimization

\*\*Loading Strategies:\*\*

- Critical CSS inlined for fast initial render

- Non-critical CSS loaded asynchronously

- JavaScript modules loaded in optimal order

- Lazy loading for below-the-fold content

\*\*Content Visibility:\*\*

- Content visibility API for performance

- Intersection Observer for lazy loading

- Debounced scroll handlers

- Efficient DOM updates

[1 tool called]

## 7. Settings and Configuration Systems

### 7.1 Settings Architecture

The settings system is organized into six main sections with tabbed navigation:

\*\*Settings Sections:\*\*

- \*\*General\*\*: User profile, statistics, and basic preferences

- \*\*Notifications\*\*: Alert preferences and Pro notification features

- \*\*Layout\*\*: Display options, themes, and layout customization

- \*\*Data\*\*: Import/export, data management, and sharing

- \*\*Pro\*\*: Premium features and subscription management

- \*\*About\*\*: App information, creators, and feedback

### 7.2 Settings Management System

\*\*Step 1: Settings Initialization\*\*

1. Settings tabs are set up with event delegation

2. Settings content is loaded dynamically

3. Current settings are loaded from `appData.settings`

4. UI elements are bound to setting values

\*\*Step 2: Settings Persistence\*\*

1. Settings changes are saved to `appData.settings`

2. Data is persisted to localStorage immediately

3. If authenticated, settings are synced to Firebase

4. UI is updated to reflect changes

\*\*Step 3: Settings Validation\*\*

1. Input validation ensures data integrity

2. Range checks for numeric values

3. Format validation for text inputs

4. Error handling for invalid settings

### 7.3 Theme and Layout System

\*\*Theme Management:\*\*

- Light/Dark mode toggle

- Theme preference stored in settings

- CSS custom properties for theming

- Smooth transitions between themes

\*\*Layout Customization:\*\*

- Condensed list view option

- Poster display toggle

- Home page list count settings

- Currently watching preview limit

\*\*Pro Layout Features:\*\*

- Theme packs (Pro feature)

- Advanced layout options

- Custom color schemes

- Enhanced visual elements

### 7.4 Data Management Features

\*\*Export System:\*\*

- JSON export for complete data backup

- CSV export for Pro users

- Selective data export options

- Download functionality with proper MIME types

\*\*Import System:\*\*

- JSON import with validation

- Legacy format support

- Data merging capabilities

- Import confirmation and rollback

\*\*Data Sharing:\*\*

- Share lists functionality

- Text-based sharing format

- Social media integration

- Privacy controls for shared content

### 7.5 Pro Features System

\*\*Pro Feature Gating:\*\*

- Feature flags control Pro feature access

- Preview mode for non-Pro users

- Upgrade prompts and call-to-action

- Graceful degradation for free users

\*\*Pro Feature Categories:\*\*

- Advanced notifications

- Enhanced data export

- Premium themes

- Additional trivia content

- Advanced layout options

[1 tool called]

## 8. Community Features and Games

### 8.1 Community Architecture

The community section provides interactive features and games to enhance user engagement:

\*\*Community Components:\*\*

- \*\*Community Player\*\*: Video content placeholder for future features

- \*\*Game Cards\*\*: Interactive games including FlickWord and Daily Trivia

- \*\*Community Spotlight\*\*: Featured content and user highlights

### 8.2 Game System Implementation

\*\*FlickWord Game:\*\*

- Wordle-style daily word puzzle

- Statistics tracking (streak, best score, win percentage)

- Modal-based game interface

- Persistent game state and progress

- Daily reset functionality

\*\*Daily Trivia Game:\*\*

- TV and movie trivia questions

- Performance tracking and statistics

- Randomized question selection

- Score persistence across sessions

- Achievement system integration

### 8.3 Game State Management

\*\*Step 1: Game Initialization\*\*

1. Game modules are loaded asynchronously

2. Game state is loaded from localStorage

3. Statistics are calculated and displayed

4. Game interfaces are set up with event handlers

\*\*Step 2: Game Play Process\*\*

1. User clicks game card to start

2. Modal opens with game interface

3. Game state is managed in real-time

4. Progress is saved automatically

\*\*Step 3: Statistics and Persistence\*\*

1. Game statistics are tracked locally

2. High scores and streaks are maintained

3. Data is synced across devices if authenticated

4. Statistics are displayed on game cards

### 8.4 Community Player System

\*\*Video Content Integration:\*\*

- Placeholder for future video features

- Responsive video player design

- Community content showcase

- Integration with external video platforms

\*\*Community Content:\*\*

- User-generated content support

- Featured content rotation

- Community highlights and achievements

- Social sharing integration

### 8.5 Game Card System

\*\*Card Design:\*\*

- Consistent visual design with main app

- Interactive hover effects

- Statistics display

- Action buttons for game access

\*\*Card Functionality:\*\*

- Click to open game modal

- Statistics updates in real-time

- Responsive design for mobile

- Accessibility support

---

## System Integration Summary

The Flicklet application integrates multiple complex systems to provide a seamless TV and movie tracking experience:

1. \*\*Authentication\*\* provides secure user access and data synchronization

2. \*\*Data Persistence\*\* ensures reliable storage and cross-device sync

3. \*\*Search Integration\*\* enables content discovery through TMDB API

4. \*\*Navigation System\*\* provides intuitive user interface organization

5. \*\*Home Layout\*\* delivers personalized content and recommendations

6. \*\*Settings Management\*\* allows extensive customization and control

7. \*\*Community Features\*\* enhance engagement through games and social elements

Each system is designed to work independently while maintaining tight integration with the overall application architecture, ensuring a cohesive and responsive user experience across all devices and use cases.

[1 tool called]

I have completed a comprehensive documentation of the Flicklet TV & Movie Tracker application systems. The documentation covers all major functional areas in a clear, step-by-step narrative format that would enable duplication of the system:

## Summary of Documented Systems:

1. \*\*Core Application Architecture\*\* - Initialization, data structures, and lifecycle management

2. \*\*Authentication and User Management\*\* - Firebase integration, user state, and data synchronization

3. \*\*Data Persistence and Storage\*\* - Dual storage strategy, sync processes, and import/export

4. \*\*Search and TMDB API Integration\*\* - Content discovery, API client, and image handling

5. \*\*UI Navigation and Tab System\*\* - Tab management, content loading, and search integration

6. \*\*Home Page Layout and Content Loading\*\* - Responsive design, performance optimization, and content organization

7. \*\*Settings and Configuration Systems\*\* - User preferences, theme management, and Pro features

8. \*\*Community Features and Games\*\* - Interactive games, community content, and engagement features

Each system is documented with:

- \*\*Purpose and functionality\*\*

- \*\*Step-by-step processes\*\*

- \*\*Data flow and dependencies\*\*

- \*\*Integration points with other systems\*\*

- \*\*Technical implementation details\*\*

The documentation provides the level of detail necessary for understanding, maintaining, and potentially duplicating the application's functionality across all major systems.