What to Watch – Design Document

1. Project Description

- **Elevator pitch:** What to Watch is a full-stack web app that helps streaming fans decide what to queue next. It pairs a curated catalog of movies and TV series with authentication-backed watchlist management and a “Quick Pick” random recommendation.
- **Vision:** Provide an opinionated yet lightweight alternative to overcrowded streaming dashboards by keeping the browsing experience fast, explainable, and centered on the titles a user actually cares about.
- **Primary goals:**
    - Let registered users build, revisit, and manage a personal watchlist that persists in MongoDB.
    - Make discovery fun through paginated browsing and a single-click random suggestion.
    - Keep the tech stack approachable for rapid iteration (Node.js/Express + vanilla JS frontend).

2. User Personas

- **Busy Grad or Professional (Avery, 27):** Time-strapped student/working professional who wants to find something to watch in under two minutes. Often browses late at night from a laptop. Success = fast access to a relevant title and the ability to remember it for later.
- **The Curator (Jordan, 31):** Movie buff who enjoys maintaining themed collections. Curious about catalog depth and wants clear metadata (poster, synopsis, year). Success = an organized, updated watchlist that can be groomed between sessions.
- **Streaming Hopper (Sam, 24):** Casual viewer who jumps between services and appreciates serendipity. Less invested in long-term curation, but wants low-friction sign-in and an easy “surprise me” button.

3. User Stories

3.1 Busy Grad or Professional

- As Avery, I want to sign in securely so that my watchlist is ready when I have a short window to watch.
- As Avery, I want the home page to surface one random recommendation so I can make a quick decision without scrolling endlessly.
- As Avery, I want to add a movie to my watchlist in a single click so I can save it for the weekend.

3.2 The Curator

- As Jordan, I want to page through the movie and series catalogs with posters and synopses so I can evaluate what to add next.
- As Jordan, I want duplicate prevention in my watchlist so I do not have to clean up repeated titles.
- As Jordan, I want to remove titles from my watchlist when plans change so the list always reflects my current priorities.

3.3 Streaming Hopper

- As Sam, I want to create a new account quickly with just email, name, and password so I can dive in without friction.
- As Sam, I want to view my watchlist on a dedicated page with large visuals so I can scan what I felt excited about recently.
- As Sam, I want the interface to highlight which titles are already in my watchlist so I do not add the same thing twice.

4. Product Requirements & Flows

- **Onboarding & Auth:** Users can register (`POST /api/register-user`) and sign in (`POST /api/auth-user`). Successful login stores the user profile in `localStorage`, which downstream features reference.
- **Discovery & Catalog Browsing:** The backend exposes paginated endpoints for movies and series. The frontend presents grid and carousel layouts with consistent card styling and accessible labels.
- **Quick Pick Recommendation:** The home page invokes `/api/get-random-movie` to surface a single MongoDB-sampled title, encouraging exploration and providing instant value.
- **Watchlist Management:** Authenticated users can add/remove titles through `/api/add-to-user-watchlist` and `/api/remove-from-user-watchlist`. Watchlist pages render a carousel with responsive breakpoints and friendly empty/loading states.

5. System Architecture

- **Frontend (Vanilla JS, HTML, CSS):** Static assets served by Express. Modules like `frontend/js/watchlist.js` and `frontend/js/series.js` encapsulate UI logic, call APIs with `fetch`, and manipulate DOM nodes. State is cached in memory and `localStorage`.
- **Backend (Node.js + Express 5):** `backend/server.js` (implied by structure) registers routes defined in `backend/data.js`, handling movie/series retrieval, user authentication, and watchlist mutations. Uses structured logging to understand request flow.
- **Persistence (MongoDB):** `db/mongoDB.js` manages connections. Collections: `Movies`, `Series`, and `users`. User documents embed a `watchlist` array containing denormalized title info plus an `addedAt` timestamp. MongoDB `$addToSet` and `$pull` operators ensure idempotent watchlist updates.
- **Deployment:** Docker Compose orchestrates a Node container and Mongo instance, seeding the database from JSON exports. Local development relies on Node 20+, nodemon, and a running MongoDB instance.

6. Data Model Snapshot

- **User**
  - `_id:ObjectId`
  - `email:string`, `password:string` (plaintext today, slated for hashing)
  - `name:string`
  - `watchlist:Array<WatchlistItem>` (embedded)
  - `createdAt:Date`, `updatedAt:Date`
- **WatchlistItem** (embedded)
  - `id:number` (title identifier from seed data)
  - `title:string`, `poster_path?:string`, `overview?:string`

    - `media_type:'movie'|'series'`
    - `addedAt:Date`
- **Movie / Series**
    - Stored separately in `Movies` and `Series` collections, derived from `movies.json` / `tv_shows.json` seed files. Fields include `id`, `title`/`name`, `overview`, `genre_ids`, `vote_average`, `release_date` or `first_air_date`, and `poster_path`.

7. API Surface (Current)

Method	Route	Purpose
`GET`	`/api/get-random-movie`	Fetch one random movie document for the Quick Pick module.
`GET`	`/api/movies?page=&pageSize=`	Paginated catalog browsing for movies.
`GET`	`/api/series?page=&pageSize=`	Paginated catalog browsing for TV series.
`POST`	`/api/register-user`	Register a new account and initialize an empty watchlist.
`POST`	`/api/auth-user`	Authenticate credentials and return essential profile info.
`GET`	`/api/get-user-watchlist?userId=`	Read the user's current watchlist array.
`POST`	`/api/add-to-user-watchlist`	Append a title to the watchlist with duplicate protection.
`DELETE`	`/api/remove-from-user-watchlist`	Remove an item from the watchlist via title id.

8. UX Notes

- **Responsive layout:** CSS targets multiple breakpoints so catalog cards reflow from grid to carousel on smaller screens.
- **Feedback & State:** Buttons update labels ("Add" ↔ "Remove") and styling via `updateWatchlistButton`. Loading, empty, and not-logged-in states are rendered for the watchlist carousel to reduce confusion.
- **Accessibility considerations:** Semantic elements and ARIA roles on watchlist cards support keyboard navigation; alt text is attached to poster imagery.

9. Non-Functional Requirements

- **Performance:** API pagination defaults to 50 records per page to balance network payloads and Mongo query cost. `$sample` on the movies collection returns a single document with acceptable latency for a dataset of this size.

- **Reliability:** Each route acquires its own MongoDB connection; future iterations should introduce pooling or a singleton connection to reduce overhead.
- **Security:** Passwords are stored in plaintext for the prototype—hashing (bcrypt), validation, and session/token management are priorities for the next release.
- **Maintainability:** The codebase uses ES modules, ESLint, and Prettier. Frontend modules prefer pure functions and data normalization helpers to keep DOM manipulation predictable.

10. Risks & Future Enhancements

- **Security hardening:** Implement password hashing, rate limiting, and JWT/session management before production exposure.
- **Richer discovery:** Layer in filtering by genre, year, or runtime to support curator workflows.
- **Social features:** Enable sharing or exporting watchlists when authentication and authorization are robust.
- **Scalability:** Introduce caching/pooling and refine data access patterns if the catalog or user base grows significantly.