Client Software Manual

324827328, 213713522

Contents

1 Introduction

When we started building the client-side application for the Movie Database, our goal was to create something that felt alive. We didn't want just a static table of data; we wanted an interface where users could explore, search, and interact with the movies. The result is a responsive web application built with Angular that feels modern and fast. We focused heavily on user experience, ensuring that feedback—like when you rate a movie—is immediate, even if the server takes a moment to catch up.

2 Technologies

Choosing the right tools was the first major decision we faced. Here is what we went with and why:

2.1 TypeScript

We knew from the start that a project of this size could get messy without strict types. JavaScript is great, but TypeScript gave us the safety net we needed. It meant that if we changed the structure of a ‘Movie‘ object in one file, the compiler would immediately tell us which other five files broke. This saved us countless hours of debugging "undefined is not a function" errors.

2.2 Angular

We chose Angular because we wanted a framework that included everything out of the box. Unlike other libraries where you have to hunt for a router or a state management solution, Angular gave us a complete toolkit. We were particularly excited to use the new **Signals** feature. It allowed us to manage the complexity of our application state—like keeping the search results in sync with the selected genres—without writing complex boilerplate code.

2.3 Chart.js

For the analytics dashboard, we needed a library that could handle heavy lifting. We tried a few options, but Chart.js stood out because of its performance with the canvas element. When you're rendering multiple bar charts showing thousands of data points (like budget vs. revenue for flops), you need something that won't freeze the browser.

2.4 RxJS

Asynchronous programming is hard, but RxJS made it manageable. Since almost every action in our app involves talking to the backend, we utilized Observables everywhere. The real power came when we needed to do complex things, like waiting for multiple API calls to finish before showing the dashboard (using ‘forkJoin‘) or cancelling an old search request when the user types a new character (using ‘switchMap‘).

3 Components

We broke the application down into three main areas, each addressing a specific user need.

3.1 The Home Page (HomeComponent)

This is where the journey begins. We designed the Home component (‘client/src/app/components/home-/home.ts‘) to be the discovery engine. The biggest technical challenge here was the filtering logic. We wanted users to be able to type a name AND select a genre, and have the list update instantly.

Instead of writing a complex chain of event listeners, we used a reactive approach. We created a "computed" signal that listens to both the search bar and the genre checkboxes. Whenever either changes, it automatically re-runs the filter logic.

```
filteredMovies = computed(() => {
  const query = this.searchQuery();
  const selectedGenres = this.selectedGenres();

```

```
  // This re-runs automatically whenever query or selectedGenres changes
  return this.movies().filter(movie => {
    const matchesSearch = movie.title.toLowerCase().startsWith(query.toLowerCase());
    const matchesGenre = selectedGenres.size === 0 ||
      movie.genres.some(genre => selectedGenres.has(genre));
    return matchesSearch && matchesGenre;
  });
});
```

Listing 1: Our reactive filtering solution

3.2 Movie Details (MovieDetailComponent)

Once a user clicks a movie, they are taken to the Detail view (‘client/src/app/components/movie-detail/movie-detail.ts‘). This is where we show the rich metadata like cast, crew, and financial stats.

The most interesting part of this component is the rating system. We faced a dilemma: how do we identify users without forcing them to log in? We decided to generate a unique ID for them in the browser.

3.2.1 Shadow Identity

When a user visits for the first time, we check their local storage. If they don’t have an ID, we generate a random string and save it. This acts as their "shadow identity," allowing us to track their votes across sessions.

```
private getUserId(): string {
  let userId = localStorage.getItem('cinelearn_user_id');
  if (!userId) {
    // Generate a sufficiently random ID
    userId = 'user_' + Math.random().toString(36).substring(2, 15) +
             Math.random().toString(36).substring(2, 15);
    localStorage.setItem('cinelearn_user_id', userId);
  }
  return userId;
}
```

Listing 2: Creating a persistent user identity

3.2.2 Optimistic UI

When a user submits a rating, we don’t just wait for the server. We optimistically update the UI to say "Success!" and update the vote count immediately using the server’s response. This makes the app feel incredibly responsive.

```
submitRating(): void {
  const movie = this.movie();
  const rating = Number(this.userRating());

  if (movie && rating >= 0 && rating <= 10) {
    this.movieService.rateMovie(movie.id, rating, this.userId)
      .pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe({
        next: (response) => {
          this.message.set('Rating submitted successfully!');
          // We save it locally so the user sees their own vote next time
          localStorage.setItem(`rating_${movie.id}_${this.userId}`, rating.toString());

          // And we update the global stats with the fresh data from the server
          const updated = {
              ...movie,
              vote_average: response.vote_average,
              vote_count: response.vote_count
          };
          this.movie.set(updated);
        },
        error: (e) => {
          this.message.set('Error submitting rating.');
        }
      });
```

```
  }
}
```

Listing 3: Handling the rating submission

3.3 Analytics Dashboard (AnalyticsComponent)

The Analytics component (‘client/src/app/components/analytics/analytics.ts‘) is where we show the data story.

3.3.1 The Performance Headache

Our initial implementation was sluggish. We were waiting for every single chart data connection to finish before showing anything. The screen would stay blank for seconds.

We solved this by decoupling the data loading. Instead of one giant request, we made each chart independent. We used Angular’s ‘toSignal‘ to transform each HTTP request into a signal that updates its specific chart whenever it arrives. This means the "Top Actors" chart might pop in first, followed by "Genres", keeping the user engaged while the heavier data loads.

```
// This doesn't block the rest of the application
readonly popularGenres = toSignal(
  this.movieService.getPopularGenres(),
  { initialValue: [] as any[] }
);

// This runs in parallel
readonly topActors = toSignal(
  this.movieService.getTopActors(),
  { initialValue: [] as any[] }
);
```

Listing 4: Decoupled data loading

4 Caveats & Usage Notes

4.1 The "Soft Session" Trade-off

You might notice there isn’t a login screen. We consciously chose a "soft session" approach. The user ID is stored in the browser’s Local Storage. This lowers the barrier to entry—anyone can just start rating movies immediately. The trade-off is that if you clear your cache, you become a new user. For this kind of public workshop application, we felt this was the right call.

4.2 One Vote Rule

To prevent spam, we enforced a rule: one vote per movie per user. However, we didn’t want to lock users in. If you change your mind, you can rate again, and your new vote overwrites the old one. We handle this logic on the backend, but the frontend reflects it by loading your previous rating whenever you revisit a movie page.