Client Software Manual

324827328, 213713522

Contents

1 Introduction

The development of the client-side application using Angular focused on creating a responsive and interactive user interface.

2 Technologies

Choosing the right tools was the first major decision we faced. We chose technologies based on two reasons, the first one is how much we can learn from it. For example, we really wanted to try out the new Angular Signals API, which we werent familiar with. The second reason is more technical:

2.1 TypeScript

We utilized TypeScript to leverage static typing in our development process. This provided type safety and facilitated refactoring, helping to identify potential errors across the codebase during compilation rather than at runtime.

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

RxJS was employed to manage asynchronous operations. As the application relies heavily on backend communication, Observables were used to handle complex data flows, such as coordinating concurrent API requests (using `forkJoin`) and managing search input streams (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)

The Home component (`client/src/app/components/home/home.ts`) serves as the primary interface for content discovery. A key technical requirement was the implementation of a responsive filtering system that updates results based on both text input and genre selection.

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)

The Detail view (`client/src/app/components/movie-detail/movie-detail.ts`) displays comprehensive movie metadata including cast, crew, and financial statistics.

A user identification mechanism was implemented to allow ratings without requiring account registration.

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

To enhance perceived performance, we implemented an optimistic UI for the voting system. The interface updates the vote count and user rating immediately upon submission, verifying the operation with the server in the background.

```
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`) visualizes statistical data derived from the movie database.

3.3.1 Performance Challenges

Initial testing revealed latency issues during data loading, as the application awaited all data sources before rendering.

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.