# PoC Angular Web Application: SeriesDiary

**by Andreea Negoițescu**

Babeș-Bolyai University, Master of Applied Computational Intelligence,
Framework Design Course

## 1. Description of Application

SeriesDiary is an intuitive and user-friendly Angular web application designed for TV series enthusiasts who want to keep track of their favorite shows. Acting like a personal diary, the platform allows authenticated users to express their preferences and record thoughts or impressions about the series they want, through reviews.

The application provides:

- User authentication (signup, login, logout) using localStorage
- CRUD-style management of TV series with local caching per user via localStorage: searching by substring of title, adding to favorites, removing from favorites and adding reviews.

The application structure contains:

- **Model:** contains entities classes: UserSignup, UserLogin and Series
- **Routing:** Centralized routing configuration with guard-protected routes
- **Pages:** Login, Signup, Home, Series, Favorites, Header (modular components)
- **Services:** Authentication (auth.service.ts), Series management (series.service.ts)

```ts
export class UserSignup {
    userId: number;
    emailId: string;
    fullName: string;
    password: string;

    constructor(){
        this.userId = 0;
        this.emailId = '';
        this.fullName = '';
        this.password = '';
    }
}
```

```ts
export class UserLogin {
    emailId: string;
    password: string;

    constructor(){
        this.emailId = '';
        this.password = '';
    }
}
```

```ts
export class Series {
    tmdb_id: number;
    name: string;
    keywords?: string | null;
    airing_date?: number | null;
    poster_img_url?: string;
    cast?: string | null;
    genres?: string | null;
    number_of_seasons?: number | null;
    number_of_episodes?: number | null;
    synopsis?: string | null;
    average_rating?: number | null;
    rating_count?: number | null;

    constructor(data: Partial<Series>) {
        this.tmdb_id = data.tmdb_id ?? 0;
        this.name = data.name ?? '';
        this.keywords = data.keywords ?? null;
        this.airing_date = data.airing_date ?? null;
        this.poster_img_url = data.poster_img_url ?? '';
        this.cast = data.cast ?? null;
        this.genres = data.genres ?? null;
        this.number_of_seasons = data.number_of_seasons ?? null;
        this.number_of_episodes = data.number_of_episodes ?? null;
        this.synopsis = data.synopsis ?? null;
        this.average_rating = data.average_rating ?? null;
        this.rating_count = data.rating_count ?? null;
    }
}
```

## 2. Angular Concepts Proof

Angular is a popular open-source web application framework developed and maintained by Google, used to build dynamic and modern single-page client applications with TypeScript and HTML.

### 2.1. Routing and Route Guards

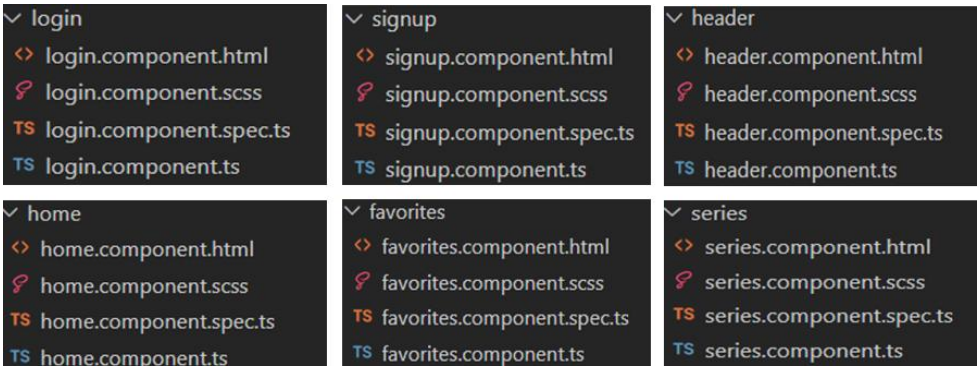
```
export const routes: Routes = [
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'signup', component: SignupComponent },
  { path: 'home', component: HomeComponent, canActivate: [authGuard] },
  { path: 'series', component: SeriesComponent, canActivate: [authGuard] },
  { path: 'favorites', component: FavoritesComponent, canActivate: [authGuard] },
  { path: 'series/:id', component: SeriesComponent, canActivate: [authGuard] },
  { path: '**', redirectTo: 'login' }
];
```

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';

export const authGuard: CanActivateFn = (state) => {
  const auth = inject(AuthService);
  const router = inject(Router);

  if (auth.isLoggedIn()) {
    return true;
  }

  router.navigate(['/login'], { queryParams: { returnUrl: state.url } });
  return false;
};
```

Routing is central to SeriesDiary and is enabled between components using the RouterModule. It provides SPA (Single Page Application) behavior. With every click, just a part of the page gets updated instead of the entire page. The routes are guarded by authGuard to protect the pages destined only to authenticated users: home, favorites, series/:id. It blocks unauthorized access using CanActivateFn and redirects non-authenticated users to /login, preserving attempted URL. AuthGuard uses the modern inject() API. Dynamic route parameters (:id) are supported to display the details of individual series.

### 2.2. Composition with Standalone Components

Components serve as the fundamental building units and each corresponds to a specific section of the web page, contributing to the overall interface. Structuring the application using components promotes better organization, making the codebase more manageable, scalable and easier to maintain as the project evolves. AppComponent is the root standalone component, bootstrapped with Angular's new bootstrapApplication API. Other components can be created from the terminal of the project, using the following command: *ng g c <component_name>*. In SeriesDiary, I have 6 components: login, signup, header, home, favorites and series, each with 4 specific files as in the images below:



Th main parts of each component, put together in the *<component_name>.component.ts* file, are:

- @Component decorator that contains some configuration used by Angular.
- HTML template and CSS/SCSS selector for styling.
- TypeScript class with behaviors, such as handling user input or making requests to a server.

```
@Component({
  selector: 'app-series',
  standalone: true,
  imports: [CommonModule, FormsModule, RouterModule, HeaderComponent],
  templateUrl: './series.component.html',
  styleUrls: ['./series.component.scss']
})
export class SeriesComponent implements OnInit {
  currentUser: any = null;
  series: any;
  reviews: { user: string; rating: number; comment: string; date: string }[] = [];
  userRating: number = 0;
  userComment: string = '';

  constructor(private route: ActivatedRoute, private series_service: SeriesService, private auth_service: AuthService) {}

  ngOnInit(): void {
```

Multiple components can be composed together. Series, home and favorite components all use the header component by importing it (*import { HeaderComponent } from '../header/header.component')* and adding it in the imports list in the Component decorator configurations as in the code above. The components are standalone, so there is no need for the traditional NgModules.

### 2.3. Services with Dependency Injection

Services can be created from the terminal of the project, using the following command: *ng g s <service_name>*. Services are a key part of Angular's dependency injection system, which allows injecting dependencies into components, directives and other services. The AuthService and SeriesService are injected not only in guard, but also across components like HomeComponent, FavoritesComponent and SeriesComponent. They use AuthService to get the current authenticated user and SeriesService to load series in the page, both actions happening at initialization using the **lifecycle hook** ngOnInit(). So, the services need to be specified as injectable:

**@Injectable**({ providedIn: 'root' })
export class SeriesService {…contains **getAllSeries(), getSeriesById(), getFavorites(), addToFavorites(), removeFromFavorites(), isFavorite(), getLocalRating()**…}

SeriesService fetches series data from a JSON file (all-series.json) and caches it locally to reduce HTTP calls. It manages series on a per-user basis stored in localStorage and provides ratings calculations based on stored reviews.

**@Injectable**({ providedIn: 'root' })
export class AuthService {…contains **getCurrentUser(), isLoggedIn(), getAllUsers(), signUp(), login(), logout()** functions…}

```ts
export class HomeComponent implements OnInit {
  currentUser: any = null;
  allSeries: any[] = [];
  filteredSeries: any[] = [];
  selectedSeries: any = null;

  constructor(
    private auth_service: AuthService,
    private series_service: SeriesService
  ) {}

  ngOnInit(): void {
    this.currentUser = this.auth_service.getCurrentUser();
    this.loadAllSeries();
  }

  loadAllSeries(): void {
    this.series_service.getAllSeries().subscribe({
      next: (series) => {
        this.allSeries = series.map(s => {
        const local = this.series_service.getLocalRating(s.tmdb_id);
        return {
          ...s,
          average_rating: local.average_rating,
          rating_count: local.rating_count
        };
      });
      this.filteredSeries = [...this.allSeries];
      },
      error: (err) => console.error('Failed to load all series', err)
    });
  }
```

```ts
export class FavoritesComponent implements OnInit {
  currentUser: any = null;
  allSeries: any[] = [];
  favoriteSeries: any[] = [];
  filteredFavorites: any[] = [];

  constructor( private auth_service: AuthService,
    private series_service: SeriesService) {}

  ngOnInit(): void {
    this.currentUser = this.auth_service.getCurrentUser();
    this.series_service.getAllSeries().subscribe({
      next: (series) => {
        this.allSeries = series;
        this.loadFavorites();
      },
      error: (err) => {console.error('Failed to load series data', err);}
    });
  }

  loadFavorites(): void {
    if (!this.currentUser) return;
    const favoriteIds = this.series_service.getFavorites(this.currentUser.emailId);
    this.favoriteSeries = this.allSeries
      .filter(series => favoriteIds.includes(series.tmdb_id))
      .map(series => {
        const local = this.series_service.getLocalRating(series.tmdb_id);
        return {
          ...series,
          average_rating: local.average_rating,
          rating_count: local.rating_count
        };
      });

    this.filteredFavorites = [...this.favoriteSeries];
  }
```

## 2.4. HttpClient with FetchAPI to allow http calls

```ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideClientHydration(withEventReplay()),
    provideHttpClient(withFetch())
  ]
};
```

## 2.5. Local Storage

It is essential for session persistance, storing all users, their favorite series and their reviews. JSON serialization/deserialization is used for storing complex objects. The current user is retrieved using localStorage.getItem('logData'), all users using localStorage.getItem(' allUsers') and in the following images there are 2 more examples, for setting and getting favorite series ans series reviews items:

```ts
addToFavorites(emailId: string, seriesId: number): void {
  if (!seriesId) return;
  const favorites = JSON.parse(localStorage.getItem('favorites') || '{}');
  const userFav: number[] = favorites[emailId] ? favorites[emailId].map((id: string | number) => Number(id)) : [];
  if (!userFav.includes(seriesId)) {
    userFav.push(seriesId);
  }
  favorites[emailId] = userFav;
  localStorage.setItem('favorites', JSON.stringify(favorites));
}
```

```
submitRating(): void {
  if (!this.isFormValid()) {
    alert('Please enter a valid rating and comment.');
    return;
  }

  const review = {
    user: this.currentUser.fullName || 'Anonymous',
    rating: this.userRating,
    comment: this.userComment,
    date: new Date().toISOString()
  };

  const key = `series-reviews-${this.series.tmdb_id}`;
  this.reviews.unshift(review);
  localStorage.setItem(key, JSON.stringify(this.reviews));

  this.updateAverageRating();
  this.userRating = 0;
  this.userComment = '';
  alert('Review submitted!');
}
```

## 2.6. Reacting Programming with Observables

Angular's reactive architecture relies heavily on RxJS Observables to manage asynchronous data operations, including data fetching, transformation, and side effects. In my application, functions like signUp(), login() and getAllSeries() return Observable streams, enabling reactive data flows throughout the app. Operators such as map(), tap(), and of() are used to transform data, perform side effects like caching, or return mock values. This pattern allows Angular to handle complex asynchronous logic in a clean, declarative, and efficient way. The following function checks if data is cached (this.allSeries). If not, fetches from JSON file via HttpClient. The tap operator caches the fetched series data.

```
getAllSeries(): Observable<Series[]> {
  if (this.seriesCache) {
    return of(this.seriesCache);
  }
  return this.http.get<Series[]>(this.seriesUrl).pipe(
    tap(series => this.seriesCache = series)
  );
}
```

**2.7. Template Syntax and Directives** which enable a dynamic, data-driven user interface.

**EXAMPLE FROM SERIES COMPONENT:** `<textarea [(ngModel)]="userComment"></textarea>`

- **Bidirectional binding:** < textarea [(ngModel)]="userComment"></textarea> here ngModel binds a property from the SeriesComponent (an user comment for a series) and an element from the HTML template of that component. Practically, any change made by the user in the text area will update automatically the value from the SeriesComponent and vice versa. To make this possible, FormsModule needs to be imported in series.component.ts.

**EXAMPLE FROM FAVORITES COMPONENT:**

- **Event binding:** <app-header (searchChanged)="filterFavorites($event)"></app-header> that makes the child component Header emit a searchChanged event. The parent Favorites listens to it and calls filterFavorites($event) with the emitted value. It is important because it makes the components

decoupled and reusable, while maintaining unidirectional data flow. The goal is using the search bar from HeaderComponent to filter favorite series in the FavoritesComponent.

- **Attribute binding:** <div class="clickable-area" [routerLink]="['/series', item.tmdb_id]"> here the [routerLink] binds a value to Angular's routerLink directive and generates a dynamic URL like /series/1. It is important because it helps navigating without full page reload, it respects SPA behavior and Angular Router setup and also makes the div element as a link.

- **Interpolation:** Rating: {{ item.average_rating }}/10 displays dynamic values for the item object helping the templates to be reactive to data changes. It provides a clean and declarative syntax for data binding.

- **Structural directive *ngIf:** <div class="main-container" *ngIf="filteredFavorites.length > 0; else noFavorites"> here *ngIf checks if filteredFavorites has at least one item. If true, the container is rendered. If false, Angular renders the alternative defined by #noFavorites.

- **Structural directive *ngFor:** <div class="series-card" *ngFor="let item of filteredFavorites"> loops over filteredFavorites and for each item it creates a series-card block dynamically.

```
<app-header (searchChanged)="filterFavorites($event)"></app-header>

<div class="main-container" *ngIf="filteredFavorites.length > 0; else noFavorites">
  <div class="series-list">
    <div class="series-card" *ngFor="let item of filteredFavorites">
      <div class="clickable-area" [routerLink]="['/series', item.tmdb_id]">
        <img *ngIf="item.poster_img_url" [src]="item.poster_img_url" alt="{{ item.name }}" class="poster" />
        <div class="card-details">
          <div class="basic-text">{{ item.name }}</div>
          <div class="rating-text">Rating: {{ item.average_rating }}/10</div>
          <div class="basic-text">{{ item.genres }}</div>
          <div class="basic-text">Episodes: {{ item.number_of_episodes }}</div>
        </div>
      </div>
      <button class="remove-btn" (click)="removeFavorite(item.tmdb_id); $event.stopPropagation()">Remove</button>
    </div>
  </div>
</div>

<ng-template #noFavorites>
  <p>No favorites...</p>
</ng-template>
```

## 2.8. Server-Side Rendering (SSR)

Provides integration with Angular's SSR via @angular/ssr and @angular/platform-server. It separates server and client configs (app.config.server.ts, app.config.ts) and offers the express server (server.ts) setup to handle SSR and serve static assets.

```
const serverConfig: ApplicationConfig = {
  providers: [
    provideServerRendering(),
    provideServerRouting(serverRoutes)
  ]
};

export const config = mergeApplicationConfig(appConfig, serverConfig);
```