

## 1. Opis systemu aplikacji zarządzającej filmami oraz serialami „Film”

Aplikacja jest webowym systemem do zarządzania bazą filmów oraz moderowanego rozbudowywania katalogu przez społeczność użytkowników. Główny problem, który rozwiązuje, to brak jednego, spójnego miejsca, w którym użytkownicy mogą nie tylko przeglądać filmy, ale też proponować dodanie nowych pozycji, oceniać je i komentować z jednoczesnym zachowaniem kontroli jakości danych przez administratora (weryfikacja propozycji, nadzór nad treściami i statystykami).

System udostępnia dwa główne typy ról: użytkownika standardowego oraz administratora. Użytkownik standardowy korzysta z funkcji związanych z katalogiem filmów: może przeglądać listę filmów i szczegóły pojedynczych pozycji, dodawać własne propozycje filmów do zatwierdzenia oraz wchodzić w interakcje społeczne z treściami (np. polubienia, oceny, komentarze, rating). System przewiduje również obszar raportów widocznych dla użytkownika, które pozwalają w prosty sposób sprawdzić statystyki dotyczące jego aktywności (np. polubione filmy, rozkład ocen, podstawowe metryki). Administrator pełni funkcję nadzorczą, czyli zatwierdza lub odrzuca propozycje dodania filmów, zarządza słownikami (kategorie) oraz ma dostęp do raportów administracyjnych prezentujących statystyki całego serwisu.

Kluczową funkcjonalnością aplikacji jest obsługa procesu biznesowego „propozycja filmu”. Z perspektywy użytkownika proces zaczyna się od zgłoszenia propozycji dodania nowego filmu wraz z wymaganymi danymi. Następnie propozycja trafia do panelu administracyjnego, gdzie administrator może ją zweryfikować (np. sprawdzić poprawność danych) i podjąć decyzję o akceptacji lub odrzuceniu. Po akceptacji film staje się elementem głównego katalogu i jest dostępny dla wszystkich użytkowników; po odrzuceniu propozycja pozostaje w historii, co wspiera transparentność i pozwala użytkownikowi zrozumieć decyzję lub poprawić zgłoszenie w przyszłości. Taki obieg zapewnia równowagę między otwartością systemu na wkład społeczności a spójnością i jakością danych w bazie.

W obszarze katalogu filmów system pozwala na podstawowe operacje zarządzania filmami (w zależności od uprawnień) oraz ich klasyfikację. Kategoryzacja umożliwia filtrowanie i porządkowanie zawartości, co przekłada się na lepszą nawigację i bardziej czytelne wyszukiwanie. Elementem zwiększającym zaangażowanie użytkowników są interakcje społeczne: oceny filmów (rating), polubienia oraz komentarze, a także możliwość reagowania na komentarze (np. polubienie komentarza). Dodatkowo system przewiduje mechanizm oznaczania/zarządzania statusem wybranych elementów (np. filmów), co może wspierać moderację, publikację lub cykl życia danych w aplikacji.

Istotną częścią rozwiązania są raporty i zestawienia danych. Aplikacja udostępnia raporty administracyjne, takie jak statystyki ocen filmów, zestawienie liczby propozycji w poszczególnych statusach czy ogólne metryki systemowe, co pomaga ocenić aktywność użytkowników i popularność treści. Równolegle dostępne są raporty „moje”, kierowane do zwykłych użytkowników, pozwalające im monitorować własne działania (np. polubione filmy, rozkład ocen). Dzięki temu system nie jest wyłącznie katalogiem, ale też narzędziem do analizy danych powstających w wyniku aktywności społeczności.

Aplikacja posiada moduł uwierzytelniania i autoryzacji, obejmujący rejestrację i logowanie, a następnie ograniczanie dostępu do funkcji zależnie od roli. Takie podejście zabezpiecza operacje administracyjne (np. akceptacja/odrzucanie propozycji, zarządzanie kategoriami i rapportami administracyjnymi), jednocześnie udostępniając użytkownikom standardowym funkcje typowe dla codziennego korzystania z serwisu. Całość tworzy spójny system, w którym dane są porządkowane w relacyjnej bazie, a logika biznesowa (proces propozycji i moderacji) odróżnia projekt od prostego CRUD i realizuje wymaganie pełnego procesu biznesowego oraz generowania raportów.

## 2. Role User/Admin

### Autentykacja: Działanie

- Użytkownik rejestruje konto endpointem POST /api/auth/register, a dane logowania są zapisywane przez ASP.NET Identity (UserManager) w bazie.
- Po rejestracji konto dostaje domyślną rolę „User”, dzięki czemu od razu można rozróżnić uprawnienia między zwykłym użytkownikiem a administratorem.
- Logowanie odbywa się przez POST /api/auth/login: aplikacja weryfikuje hasło przez Identity, pobiera role użytkownika i generuje token JWT zawierający m.in. identyfikator oraz role w postaci claims.
- Kolejne requesty do chronionych endpointów przesyłają token w nagłówku Authorization: Bearer <token>, a middleware UseAuthentication() i UseAuthorization() sprawdza token i buduje obiekt User na podstawie claims.
- Dzięki ustawieniu RoleClaimType = ClaimTypes.Role role zapisane w tokenie są używane przez atrybuty [Authorize] / [Authorize(Roles="Admin")] w kontrolerach.

### Seed ról i konta administratora

Podczas uruchomienia aplikacji wykonywany jest seeder Identity, który tworzy role „Admin” i „User” (jeśli nie istnieją), a także zakłada konto administracyjne admin@filmapp.local i przypisuje mu rolę Admin.

To daje gotowego użytkownika do testów i demonstracji funkcji administracyjnych (np. akceptacja/odrzucanie propozycji, raporty).

- 1) Konfiguracja JWT + middleware

```

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = jwt["Issuer"],
        ValidAudience = jwt["Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(key),
        RoleClaimType = ClaimTypes.Role
    };
});

app.UseAuthentication();
app.UseAuthorization();

```

## Rejestracja i nadanie roli „User”

```

[HttpPost("register")]
Odwołanie: 0
public async Task<IActionResult> Register([FromBody] RegisterInput input)
{
    var existing = await _users.FindByEmailAsync(input.Email);
    if (existing is not null) return Conflict("User already exists.");

    var user = new ApplicationUser { UserName = input.Email, Email = input.Email };
    var result = await _users.CreateAsync(user, input.Password);

    if (!result.Succeeded)
        return BadRequest(result.Errors);

    await _users.AddToRoleAsync(user, "User");
    return NoContent();
}

```

## Logowanie i generowanie tokenu JWT z rolami

```

var roles = await _users.GetRolesAsync(user);

var jwt = _config.GetSection("Jwt");
var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwt["Key"]!));

var claims = new List<Claim>
{
    new Claim(JwtRegisteredClaimNames.Sub, user.Id),
    new Claim(ClaimTypes.NameIdentifier, user.Id),
    new Claim(ClaimTypes.Name, user.UserName ?? user.Email ?? "")
};

foreach (var r in roles)
    claims.Add(new Claim(ClaimTypes.Role, r));

```

## Seeder ról + konto administratora (dev)

```
// roles
if (!await roleManager.RoleExistsAsync("Admin"))
    await roleManager.CreateAsync(new IdentityRole("Admin"));

if (!await roleManager.RoleExistsAsync("User"))
    await roleManager.CreateAsync(new IdentityRole("User"));

// admin user
var adminEmail = "admin@filmapp.local";
var adminPassword = "Admin123!";

var admin = await userManager.FindByEmailAsync(adminEmail);
if (admin is null)
{
    admin = new ApplicationUser { UserName = adminEmail, Email = adminEmail };
    var created = await userManager.CreateAsync(admin, adminPassword);
    if (!created.Succeeded)
        throw new Exception(string.Join("; ", created.Errors.Select(e => e.Description)));
}

if (!await userManager.IsInRoleAsync(admin, "Admin"))
    await userManager.AddToRoleAsync(admin, "Admin");
```

## GET /api/auth/whoami

Endpoint debug GET /api/auth/whoami służy do szybkiej weryfikacji, czy autentykacja JWT działa poprawnie oraz jakie role zostały przypisane do aktualnie zalogowanego użytkownika.

Jest zabezpieczony atrybutem [Authorize], więc zadziała tylko wtedy, gdy klient wyśle poprawny token w nagłówku Authorization: Bearer <token> (w przeciwnym razie API zwróci brak autoryzacji).

Po wywołaniu endpoint zwraca w odpowiedzi m.in. nazwę użytkownika (User.Identity?.Name) oraz listę ról odczytaną z claims (filtrowaną po typie kończącym się na /claims/role).

Dzięki temu można łatwo przetestować różnice między kontem administratora i zwykłego użytkownika: po zalogowaniu jako administrator odpowiedź powinna zawierać rolę Admin, a po zalogowaniu jako zwykły użytkownik rolę User

```
[ApiController]
[Route("api/auth")]
Odwołania: 0
public class AuthDebugController : ControllerBase
{
    [Authorize]
    [HttpGet("whoami")]
    Odwołania: 0
    public IActionResult WhoAmI()
        => Ok(new
    {
        Name = User.Identity?.Name,
        Roles = User.Claims
            .Where(c => c.Type.EndsWith("/claims/role"))
            .Select(c => c.Value)
    });
}
```

### 3. Funkcjonalności aplikacji Film

#### Proces biznesowy

Proces biznesowy w „Film” to moderowane dodawanie nowych pozycji do katalogu: użytkownik zgłasza propozycję filmu/serialu, a administrator ją weryfikuje i akceptuje lub odrzuca. To spełnia wymaganie pełnego procesu biznesowego (dodanie, weryfikacja, zatwierdzenie) oraz pokazuje logikę wykraczającą poza prosty CRUD.

Celem jest utrzymanie jakości danych w katalogu przy jednoczesnym umożliwieniu społeczności współtworzenia bazy filmów. Dzięki temu do głównej tabeli Movies trafiają tylko zweryfikowane propozycje, a sam proces ma jawne statusy: Pending, Approved, Rejected.

#### Uczestnicy i role

- User: tworzy propozycję i przegląda własne zgłoszenia; endpointy propozycji wymagają zalogowania ([Authorize]).
- Administrator: przegląda propozycje wszystkich użytkowników, filtry je po statusie oraz wykonuje decyzję approve/reject; kontroler administracyjny jest ograniczony do roli Admin.

Propozycja (MovieProposal) przechowuje dane wejściowe użytkownika: Title, Year, Type, Reason, Status, CreatedAt oraz UserId, co pozwala śledzić autora i etap procesu.

Kategorie propozycji są trzymane w relacji N:M poprzez MovieProposalCategory, a końcowy katalog filmów wykorzystuje analogiczną relację N:M MovieCategory pomiędzy Movie a Category.

Dostęp do szczegółów propozycji jest ograniczony do jej autora: użytkownik może przeglądać jedynie własne zgłoszenia (lista „mine” oraz szczegóły konkretnej propozycji), co chroni dane innych użytkowników i wspiera prywatność.

## *Tworzenie propozycji (User)*

```
[HttpPost]
Odwołania: 0
public async Task<IActionResult> Create(CreateMovieProposalDto dto)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (string.IsNullOrWhiteSpace(userId)) return Unauthorized();

    var distinctCategoryIds = dto.CategoryIds.Distinct().ToList();
    var existingCategories = await _db.Categories
        .Where(c => distinctCategoryIds.Contains(c.Id))
        .ToListAsync();

    var proposal = new MovieProposal
    {
        Title = dto.Title,
        Year = dto.Year,
        Type = dto.Type,
        Reason = dto.Reason,
        Status = ProposalStatus.Pending,
        CreatedAt = DateTime.UtcNow,
        UserId = userId
    };

    _db.MovieProposals.Add(proposal);
}
```

## *Akceptacja propozycji (Admin)*

```
[HttpPost("{id:int}/approve")]
Odwołania: 0
public async Task<IActionResult> Approve(int id)
{
    var p = await _db.MovieProposals
        .Include(x => x.Categories)
        .FirstOrDefaultAsync(x => x.Id == id);

    if (p is null) return NotFound();

    if (p.Status != ProposalStatus.Pending)
        return Conflict("Only pending proposals can be approved.");

    if (!Enum.IsDefined(typeof(ContentType), p.Type))
        return BadRequest("Invalid content type.");

    if (p.Year < 1888 || p.Year > 2100)
        return BadRequest("Invalid year.");

    var exists = await _db.Movies.AnyAsync(m => m.Title == p.Title && m.Year == p.Year);
    if (exists)
        return Conflict("Movie with this title and year already exists.");

    var movie = new Movie
    {
        Title = p.Title,
        Year = p.Year,
        Type = p.Type,
        Description = null,
        CoverUrl = null
    };

    _db.Movies.Add(movie);
    await _db.SaveChangesAsync();

    if (p.Categories.Count > 0)
    {
        _db.MovieCategories.AddRange(
            p.Categories.Select(pc => new MovieCategory
            {
                MovieId = movie.Id,
                CategoryId = pc.CategoryId
            })
        );
    }

    p.Status = ProposalStatus.Approved;
    await _db.SaveChangesAsync();

    return NoContent();
}
```

## Funkcjonalność: rejestracja użytkownika

Rejestracja odbywa się przez endpoint POST /api/auth/register, gdzie tworzony jest nowy użytkownik w systemie ASP.NET Identity na podstawie adresu e-mail i hasła.

Po poprawnym utworzeniu konta użytkownik dostaje domyślną rolę User, co jest podstawą do dalszej autoryzacji i rozdzielenia uprawnień w aplikacji.

Dodatkowo przy starcie aplikacji działa seeder, który tworzy role Admin i User oraz zakłada konto administratora (środowisko developerskie), co ułatwia testowanie panelu administracyjnego.

## Funkcjonalność: panel administracyjny (zarządzanie filmami i kategoriami)

Panel administracyjny w aplikacji „Film” służy do utrzymania jakości i kompletności katalogu filmów/seriali oraz wykonywania operacji, które nie powinny być dostępne dla zwykłych użytkowników. Dostęp do tej części systemu jest ograniczony do roli Admin poprzez mechanizm autoryzacji oparty o role (endpoints administracyjne wymagają uprawnień administratora).

Administrator może zarządzać zasobem „Movie” w sposób zbliżony do CRUD, z naciskiem na edycję i moderację danych. Edycja jest realizowana m.in. przez częściową aktualizację PATCH /api/movies/{id}, która pozwala zmieniać wybrane pola filmu (np. tytuł, opis, okładkę, rok, typ treści oraz parametry zależne od typu). W trakcie modyfikacji wykonywana jest walidacja spójności danych zależna od ContentType (np. inne wymagania dla filmu niż dla serialu), a w przypadku błędów system zwraca odpowiedź walidacyjną zamiast zapisywać niepoprawne dane.

Istotnym elementem panelu administratora jest także zarządzanie powiązaniami filmu z kategoriami (relacja N:M realizowana przez encję pośrednią). Administrator może usuwać przypisanie kategorii z filmu endpointem DELETE /api/movies/{movieId}/categories/{categoryId}, co umożliwia utrzymanie poprawnej klasyfikacji treści bez usuwania całego filmu. Dodatkowo logika zapobiega tworzeniu duplikatów powiązań (jeżeli dana kategoria jest już przypisana do filmu, system zgłasza konflikt).

## Częściowa edycja filmu (Admin) – PATCH

```
[Authorize(Roles = "Admin")]
[HttpPatch("{id:int}")]
Odwolanie: 0
public async Task<ActionResult<MovieOutput>> Patch(int id, [FromBody] PatchMovieInput input)
{
    var movie = await _db.Movies.FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null) return NotFound();

    if (input.Title is not null) movie.Title = input.Title;
    if (input.Description is not null) movie.Description = input.Description;
    if (input.CoverUrl is not null) movie.CoverUrl = input.CoverUrl;
    if (input.Year.HasValue) movie.Year = input.Year;
    if (input.Type.HasValue) movie.Type = input.Type.Value;
    if (input.DurationMinutes.HasValue) movie.DurationMinutes = input.DurationMinutes.Value;
    if (input.SeasonsCount.HasValue) movie.SeasonsCount = input.SeasonsCount.Value;
    if (input.EpisodesCount.HasValue) movie.EpisodesCount = input.EpisodesCount.Value;

    var errors = MovieTypeValidation.Validate(
        movie.Type,
        movie.DurationMinutes,
        movie.SeasonsCount,
        movie.EpisodesCount
    ).ToList();

    foreach (var e in errors)
    {
        var key = e.MemberNames.FirstOrDefault() ?? "";
        ModelState.AddModelError(key, e.ErrorMessage ?? "Validation error");
    }

    if (!ModelState.IsValid)
        return ValidationProblem(ModelState);

    await _db.SaveChangesAsync();

    return Ok(new MovieOutput
    {
        Id = movie.Id,
        Title = movie.Title,
        Type = movie.Type,
        Description = movie.Description,
        CoverUrl = movie.CoverUrl,
        Year = movie.Year,
        Categories = new()
    });
}
```

## Przypięcie kategorii do filmu – zabezpieczenie przed duplikatem

```
[Authorize(Roles = "Admin")]
[HttpPost("{movieId:int}/categories/{categoryId:int}")]
Odwolanie: 0
public async Task<IActionResult> AddCategoryToMovie(int movieId, int categoryId)
{
    var movieExists = await _db.Movies.AnyAsync(m => m.Id == movieId);
    if (!movieExists) return NotFound("Movie not found.");

    var categoryExists = await _db.Categories.AnyAsync(c => c.Id == categoryId);
    if (!categoryExists) return NotFound("Category not found.");

    var alreadyLinked = await _db.MovieCategories.AnyAsync(mc =>
        mc.MovieId == movieId && mc.CategoryId == categoryId);

    if (alreadyLinked) return Conflict("This category is already assigned to this movie.");

    _db.MovieCategories.Add(new MovieCategory
    {
        MovieId = movieId,
        CategoryId = categoryId
    });

    await _db.SaveChangesAsync();
    return NoContent();
}
```

## Odpinanie kategorii od filmu (Admin) – DELETE

```
[Authorize(Roles = "Admin")]
[HttpDelete("{movieId:int}/categories/{categoryId:int}")]
Odwołanie: 0
public async Task<IActionResult> RemoveCategoryFromMovie(int movieId, int categoryId)
{
    var link = await _db.MovieCategories
        .FirstOrDefaultAsync(mc => mc.MovieId == movieId && mc.CategoryId == categoryId);

    if (link is null) return NotFound("This category is not assigned to this movie.");

    _db.MovieCategories.Remove(link);
    await _db.SaveChangesAsync();

    return NoContent();
}
```

## Walidacja przy tworzeniu/pełnej edycji (Create/Update)

```
using System.ComponentModel.DataAnnotations;
namespace FilmApp.Api.Dtos;

1 odwołanie
public class CreateMovieInput : IValidatableObject
{
    [Required]
    [MaxLength(200)]
    1 odwołanie
    public string Title { get; set; } = string.Empty;

    [MaxLength(4000)]
    1 odwołanie
    public string? Description { get; set; }

    [MaxLength(500)]
    1 odwołanie
    public string? CoverUrl { get; set; }

    1 odwołanie
    public int? Year { get; set; }

    [Required]
    Odwołanie: 2
    public ContentType? Type { get; set; }

    [Range(1, 1000)]
    Odwołanie: 2
    public int? DurationMinutes { get; set; }

    [Range(1, 200)]
    Odwołanie: 2
    public int? SeasonsCount { get; set; }

    [Range(1, 10000)]
    Odwołanie: 2
    public int? EpisodesCount { get; set; }

    Odwołanie: 0
    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
        => MovieTypeValidation.Validate(Type, DurationMinutes, SeasonsCount, EpisodesCount);
}
```

## Funkcjonalność: zarządzanie kategoriami (Admin)

Administrator ma pełny dostęp do tworzenia, edycji i usuwania kategorii w słowniku, co pozwala na elastyczne dostosowywanie klasyfikacji filmów i seriali. Operacje są dostępne pod endpointami POST /api/categories, PUT /api/categories/{id} oraz DELETE /api/categories/{id}, wszystkie zabezpieczone [Authorize(Roles = "Admin")].

Logika zawiera ochronę przed duplikatami nazw (Conflict przy CREATE/UPDATE, gdy nazwa już istnieje) oraz przy usuwaniu kategorii dodatkowo czyści powiązania N:M z filmami (usuwa rekordy z MovieCategories).

### *Tworzenie kategorii (POST)*

```
[Authorize(Roles = "Admin")]
[HttpPost]
Odwolanie: 0
public async Task<ActionResult<CategoryOutput>> Create([FromBody] CreateCategoryInput input)
{
    var categoryExists = await _db.Categories.AnyAsync(c => c.Name == input.Name);
    if (categoryExists) return Conflict("Category with this name already exists.");

    var category = new Entities.Category { Name = input.Name };
    _db.Categories.Add(category);
    await _db.SaveChangesAsync();

    return Ok(new CategoryOutput { Id = category.Id, Name = category.Name });
}
```

### *Edycja kategorii (PUT)*

```
[Authorize(Roles = "Admin")]
[HttpPut("{id:int}")]
Odwolanie: 0
public async Task<ActionResult<CategoryOutput>> Update(int id, [FromBody] CreateCategoryInput input)
{
    var category = await _db.Categories.FirstOrDefaultAsync(c => c.Id == id);
    if (category is null) return NotFound();

    var nameExists = await _db.Categories.AnyAsync(c => c.Id != id && c.Name == input.Name);
    if (nameExists) return Conflict("Category with this name already exists.");

    category.Name = input.Name;
    await _db.SaveChangesAsync();

    return Ok(new CategoryOutput { Id = category.Id, Name = category.Name });
}
```

### *Usuwanie kategorii z czyszczeniem relacji N:M (DELETE)*

```
[Authorize(Roles = "Admin")]
[HttpDelete("{id:int}")]
Odwolanie: 0
public async Task<IActionResult> Delete(int id)
{
    var category = await _db.Categories.FirstOrDefaultAsync(c => c.Id == id);
    if (category is null) return NotFound();

    var links = await _db.MovieCategories
        .Where(mc => mc.CategoryId == id)
        .ToListAsync();

    if (links.Count > 0)
        _db.MovieCategories.RemoveRange(links);

    _db.Categories.Remove(category);
    await _db.SaveChangesAsync();

    return NoContent();
}
```

## **Generowanie raportów**

Aplikacja „Film” spełnia wymaganie generowania raportów poprzez dedykowane kontrolery: AdminReportsController (raporty globalne dla administratora) oraz MeReportsController (raporty osobiste dla użytkownika).

Raporty wykorzystują agregacje LINQ (Count, Average, GroupBy) i są ograniczone rolami autoryzacji: administrator ma dostęp do statystyk całego systemu, użytkownik do swoich danych.

### 1) Raporty administracyjne (AdminReports)

Dostępne tylko dla roli Admin, pod ścieżką /api/admin/reports.

- GET /api/admin/reports/movies-ratings: lista filmów z ocenami, posortowana malejąco po liczbie ocen (średnia ocena, liczba ocen); limit 50 dla wydajności.
  - GET /api/admin/reports/dashboard-metrics: podstawowe metryki systemu (liczba filmów, oczekujące propozycje, użytkownicy, oceny).
  - GET /api/admin/reports/proposals-count-by-status: liczba propozycji filmów w podziale na statusy (Pending, Approved, Rejected).
- 2) Raporty osobiste użytkownika (MeReports)

Dostępne po zalogowaniu ([Authorize]), pod ścieżką /api/me/reports.

- GET /api/me/reports/liked-movies: lista filmów polubionych przez użytkownika (Id + tytuł).
- GET /api/me/reports/movie-statuses: statusy oglądania filmów użytkownika (MovieId, tytuł, status, data aktualizacji).
- GET /api/me/reports/user-metrics: osobiste statystyki (liczba polubionych filmów, obejrzanych, liczba zgłoszonych propozycji)

*Admin: średnie oceny filmów*

```
[HttpGet("movies-ratings")]
Odwolania: 0
public async Task<ActionResult> MoviesRatings()
{
    var data = await _db.Movies
        .AsNoTracking()
        .Where(m => m.Ratings.Any())
        .Select(m => new
        {
            name = m.Title,
            avgScore = m.Ratings.Average(r => (double?)r.Score) ?? 0,
            ratingCount = m.Ratings.Count()
        })
        .OrderByDescending(x => x.ratingCount)
        .Take(50) // Limit dla wydajności
        .ToListAsync();

    return Ok(data);
}
```

*Admin: liczba propozycji po statusach*

```
[HttpGet("proposals-count-by-status")]
Odwolania: 0
public async Task<ActionResult> ProposalsCountByStatus()
{
    var data = await _db.MovieProposals
        .GroupBy(p => p.Status)
        .Select(g => new { status = g.Key.ToString(), count = g.Count() })
        .ToDictionaryAsync(g => g.status, g => g.count);

    return Ok(data);
}
```

*User: metryki osobiste*

```
[HttpGet("user-metrics")]
Odwolania: 0
public async Task<ActionResult<object>> UserMetrics()
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier)!;

    return Ok(new
    {
        likedMovies = await _db.Movies.CountAsync(m => m.Likes.Any(l => l.UserId == userId)),
        watchedMovies = await _db.UserMovieStatuses.CountAsync(s => s.UserId == userId && s.Status == WatchStatus.Watched),
        proposalsCount = await _db.MovieProposals.CountAsync(p => p.UserId == userId)
    });
}
```

## Funkcjonalność: logowanie działań użytkownika

W aplikacji „Film” działania użytkownika są utrwalane w bazie danych w postaci rekordów powiązanych z kontem (UserId) i konkretnym zasobem (np. film, komentarz). Przykłady rejestrowanych akcji to: wystawienie oceny filmu/serialu, polubienie filmu, dodanie komentarza, polubienie komentarza oraz ustawienie statusu oglądania (np. „Watched”).

Wszystkie interakcje użytkownika są zapisywane z kluczem UserId (z tokenu JWT) oraz danymi kontekstowymi (np. CreatedAt/UpdatedAt, powiązanie z filmem/komentarzem).

### Ocena filmu (rating)

Endpoint POST /api/movies/{id}/ratings tworzy nową ocenę, a jeśli użytkownik już oceniał dany film, aktualizuje istniejący rekord (czyli aktywność użytkownika jest „logowana” jako dane w tabeli ocen).

```
[Authorize]
[HttpPost("{id:int}/ratings")]
Odwolania: 0
public async Task<IActionResult> Rate(int id, [FromBody] RateMovieInput input)
{
    var movieExists = await _db.Movies.AnyAsync(m => m.Id == id);
    if (!movieExists) return NotFound();

    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (string.IsNullOrEmpty(userId)) return Unauthorized();

    var existing = await _db.Ratings
        .FirstOrDefaultAsync(r => r.MovieId == id && r.UserId == userId);

    if (existing is null)
    {
        _db.Ratings.Add(new Rating { MovieId = id, Score = input.Score, UserId = userId });
    }
    else
    {
        existing.Score = input.Score;
    }

    await _db.SaveChangesAsync();
    return NoContent();
}
```

### Polubienie filmu

Endpoint POST /api/movies/{id}/like dopisuje rekord polubienia przypisany do użytkownika i filmu, a ponowne kliknięcie nie tworzy duplikatu (idempotentność).

```
[Authorize]
[HttpPost("{id:int}/like")]
Odwolania: 0
public async Task<IActionResult> LikeMovie(int id)
{
    var movieExists = await _db.Movies.AnyAsync(m => m.Id == id);
    if (!movieExists) return NotFound();

    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (string.IsNullOrEmpty(userId)) return Unauthorized();

    var exists = await _db.Likes.AnyAsync(l => l.MovieId == id && l.UserId == userId);
    if (exists) return NoContent();

    _db.Likes.Add(new Like { MovieId = id, UserId = userId });
    await _db.SaveChangesAsync();
    return NoContent();
}
```

### *Komentarze i polubienia komentarzy*

Dodanie komentarza (POST /api/movies/{id}/comments) i polubienie komentarza (POST /api/comments/{commentId}/like) również zapisują aktywność użytkownika w bazie poprzez utworzenie odpowiednich rekordów.

Dodatkowo usuwanie komentarza jest kontrolowane uprawnieniami: użytkownik może usuwać tylko własne komentarze (Forbid gdy próbuje usunąć cudzy), a administrator ma osobny endpoint do moderacji.

### *Status*

### **Funkcjonalność: obsługa błędów systemowych**

Aplikacja implementuje centralną obsługę błędów poprzez GlobalExceptionHandler, który loguje wyjątki (ILogger) i zwraca spójne ProblemDetails z kodem statusu, tytułem oraz szczegółami. Obsługuje specyficzne wyjątki: NotFoundException (404 "Not found"), ConflictException (409 "Conflict") i ValidationException (400 "Validation error"), fallback na 500 dla reszty.

Walidacja domenowa w MovieTypeValidation sprawdza spójność danych (np. DurationMinutes wymagane dla Movie, SeasonsCount dla Series), zwracając ValidationResult do ModelState. W MoviesController błędy jak brak rekordu (NotFound), duplikat kategorii (Conflict) czy walidacja (ValidationProblem) są obsługiwane lokalnie przed middleware'em.

- Brak filmu; NotFoundException; GlobalHandler; 404 ProblemDetails
- Duplikat kategorii; ConflictException; GlobalHandler 409
- Serial bez SeasonsCount; MovieTypeValidation; ModelState 400 ValidationProblem
- Nieautoryzowany; Unauthorized; 401 (standard ASP.NET)

### *Rejestracja globalnego handlера*

```
builder.Services.AddProblemDetails();
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
```

```
app.UseExceptionHandler();
app.UseStatusCodePages();
```

## Globalna obsługa wyjątków : ProblemDetails (GlobalExceptionHandler.cs)

```
Odwolania: 4
public sealed class GlobalExceptionHandler : IExceptionHandler
{
    private readonly ILogger<GlobalExceptionHandler> _logger;

    Odwołania: 0
    public GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger)
        => _logger = logger;

    Odwołania: 0
    public async ValueTask<bool> TryHandleAsync(
        HttpContext httpContext,
        Exception exception,
        CancellationToken cancellationToken)
    {
        _logger.LogError(exception, "Unhandled exception");

        var (status, title) = exception switch
        {
            NotFoundException => (StatusCodes.Status404NotFound, "Not found"),
            ConflictException => (StatusCodes.Status409Conflict, "Conflict"),
            ValidationException => (StatusCodes.Status400BadRequest, "Validation error"),
            _ => (StatusCodes.Status500InternalServerError, "Server error")
        };

        var problem = new ProblemDetails
        {
            Status = status,
            Title = title,
            Detail = exception.Message
        };

        httpContext.Response.StatusCode = status;
        await httpContext.Response.WriteAsJsonAsync(problem, cancellationToken);
        return true;
    }
}
```

## Własne wyjątki domenowe (NotFoundException / ConflictException)

```
public sealed class NotFoundException : Exception
{
    Odwołania: 0
    public NotFoundException() { }

    Odwołania: 0
    public NotFoundException(string message) : base(message) { }

    Odwołania: 0
    public NotFoundException(string message, Exception inner) : base(message, inner) { }
}
```

### Walidacja domenowa zależna od typu (MovieTypeValidation.cs)

```
public static IEnumerable<ValidationResult> Validate(
    ContentType? type,
    int? durationMinutes,
    int? seasonsCount,
    int? episodesCount)
{
    if (type is null)
        yield break;

    if (type == ContentType.Movie)
    {
        if (durationMinutes is null)
            yield return new ValidationResult("DurationMinutes jest wymagane dla filmu.", new[] { "DurationMinutes" });

        if (seasonsCount is not null)
            yield return new ValidationResult("SeasonsCount nie może być ustawione dla filmu.", new[] { "SeasonsCount" });

        if (episodesCount is not null)
            yield return new ValidationResult("EpisodesCount nie może być ustawione dla filmu.", new[] { "EpisodesCount" });
    }
    else if (type == ContentType.Series)
    {
        if (seasonsCount is null)
            yield return new ValidationResult("SeasonsCount jest wymagane dla serialu.", new[] { "SeasonsCount" });

        if (episodesCount is null)
            yield return new ValidationResult("EpisodesCount jest wymagane dla serialu.", new[] { "EpisodesCount" });

        if (durationMinutes is not null)
            yield return new ValidationResult("DurationMinutes nie może być ustawione dla serialu.", new[] { "DurationMinutes" });
    }
}
```

### Zwracanie błędu walidacji w kontrolerze (MoviesController : PATCH)

```
var errors = MovieTypeValidation.Validate(
    movie.Type,
    movie.DurationMinutes,
    movie.SeasonsCount,
    movie.EpisodesCount
).ToList();

foreach (var e in errors)
{
    var key = e.MemberNames.FirstOrDefault() ?? "";
    ModelState.AddModelError(key, e.ErrorMessage ?? "Validation error");
}

if (!ModelState.IsValid)
    return ValidationProblem(ModelState);
```

## 4. CRUD w aplikacji Film

### Movies (Admin — pełny CRUD)

- Create: POST /movies: tworzy nowy film z danymi (tytuł, rok, typ film/serial, okładka)
- Read: GET /movies (lista z paginacją, wyszukiwanie, filtry), GET /movies/{id} (szczegóły z kategoriami/ratingami)
- Update: PUT /movies/{id} (pełna edycja), PATCH /movies/{id} (walidacja typu)
- Delete: DELETE /movies/{id}

Movies	
GET	/api/Movies
POST	/api/Movies
GET	/api/Movies/{id}
PUT	/api/Movies/{id}
DELETE	/api/Movies/{id}
PATCH	/api/Movies/{id}

## Categories (Admin — pełny CRUD)

- Create: POST /categories (nazwa)
- Read: GET /categories (publiczne)
- Update: PUT /categories/{id}
- Delete: DELETE /categories/{id} (czyści relacje z filmami)

Categories	
GET	/api/Categories
POST	/api/Categories
PUT	/api/Categories
DELETE	/api/Categories

## Relacje Movies-Categories (Admin)

- Dodaj: POST /movies/{moviedId}/categories/{catId} (blokada duplikatów)
- Usuń: DELETE /movies/{moviedId}/categories/{catId}

## User interakcje

- Ratings/Likes/Comments/Statusy: upsert/read/delete pod konkretnym filmem
- MovieProposals :User Create/Read swoich, Admin Approve/Reject (przekształca w Movie)

## 5. Walidacja danych wejściowych

Wielowarstwowa walidacja zapewnia integralność i czytelność błędów:

1. Warstwa 1: Adnotacje DataAnnotations (automat ASP.NET Core)

W klasach DTO ([CreateMovieInput], [RegisterInput], [AddCommentInput]) stosowane są:

- [Required]: pola obowiązkowe (Title, Email, Password, Type)
  - [MaxLength(n)]: limity długości (Title:200, Comment:2000, Email:254)
  - [Range(min,max)]: zakresy liczbowe (Year:1900-2100, DurationMinutes:1-1000, Rating:1-10)
  - [EmailAddress]: poprawny format email
  - [MinLength(6)]: hasło minimum 6 znaków
  - ASP.NET automatycznie sprawdza przed kontrolerem, błędy w ModelState: 400 Bad Request.
2. Warstwa 2: Walidacja domenowa (IValidatableObject)

Dla złożonych reguł zależnych od typu treści (Movie vs Series):

CreateMovieInput implementuje IValidatableObject.Validate(), dalej wywołuje MovieTypeValidation.Validate(Type, DurationMinutes, SeasonsCount, EpisodesCount).

MovieTypeValidation:

- Film: Wymaga DurationMinutes, blokuje SeasonsCount/EpisodesCount
  - Serial: Wymaga SeasonsCount + EpisodesCount, blokuje DurationMinutes błędy zwracane jako ValidationResult do ModelState.
3. Warstwa 3: Walidacja biznesowa (kontrolery)

Przed zapisem do bazy:

- Istnienie encji (!movieExists: NotFound)
- Unikalność/duplikaty (alreadyLinked: Conflict)
- Stan obiektu (ProposalStatus.Pending: Conflict)

Obsługa błędów: GlobalExceptionHandler: ProblemDetails (404 NotFound, 409 Conflict, 400 Validation).

## 5. Testy jednostkowe

### Metody pomocnicze (wspólne dla wszystkich testów)

CreateDbContext(string dbName)

- Tworzy InMemory bazę danych EF Core z unikalną nazwą dla każdego testu.
- Izoluje testy: każdy ma czystą bazę danych.
- Dlaczego? Unika konfliktów między testami i nie brudzi prawdziwej bazy.

CreateMoviesController(AppDbContext db, string? userId)

- Tworzy instancję MoviesController z podanym kontekstem DB.
- Symuluje zalogowanego użytkownika poprzez ClaimsPrincipal z userId.
- Ustawia ControllerContext.HttpContext.User: kontroler myśli, że request przyszedł od użytkownika.

## Test1

Sprawdza endpoint GET /api/movies/{id} gdy film nie istnieje.

1. Tworzy pustą bazę danych
2. Wywołuje controller.GetById(999)
3. Oczekuje HTTP 404 (NotFoundResult)

## Test 2

Sprawdza endpoint POST /api/movies (tworzenie nowego filmu przez admina).

1. Tworzy pustą bazę danych
2. Wysyła CreateMovieInput z danymi filmu "Test Movie"
3. Sprawdza czy kontroler:
  - o Zwrócił HTTP 201 (CreatedAtActionResult)
  - o Dodał 1 rekord do db.Movies
  - o Tytuł w bazie = "Test Movie"

## Test 3

1. Sprawdza endpoint POST /api/movies/{id}/ratings (ocenianie filmu).
2. Tworzy film w bazie (movie.Id automatycznie = 1)
3. Zalogowany jako "user1" wystawia ocenę 8
4. Sprawdza czy:
  - o Dodano 1 rekord do db.Ratings
  - o Score = 8, MovieId = 1, UserId = "user1"

## Test 4

Sprawdza endpoint POST /api/movies/{id}/like (lajkowanie filmu).

1. Tworzy film w bazie
2. Zalogowany "user1" lajkuje film
3. Sprawdza czy:
  - o Dodano 1 rekord do db.Likes
  - o MovieId i UserId są poprawne

## 6. Bazy danych

### Diagram

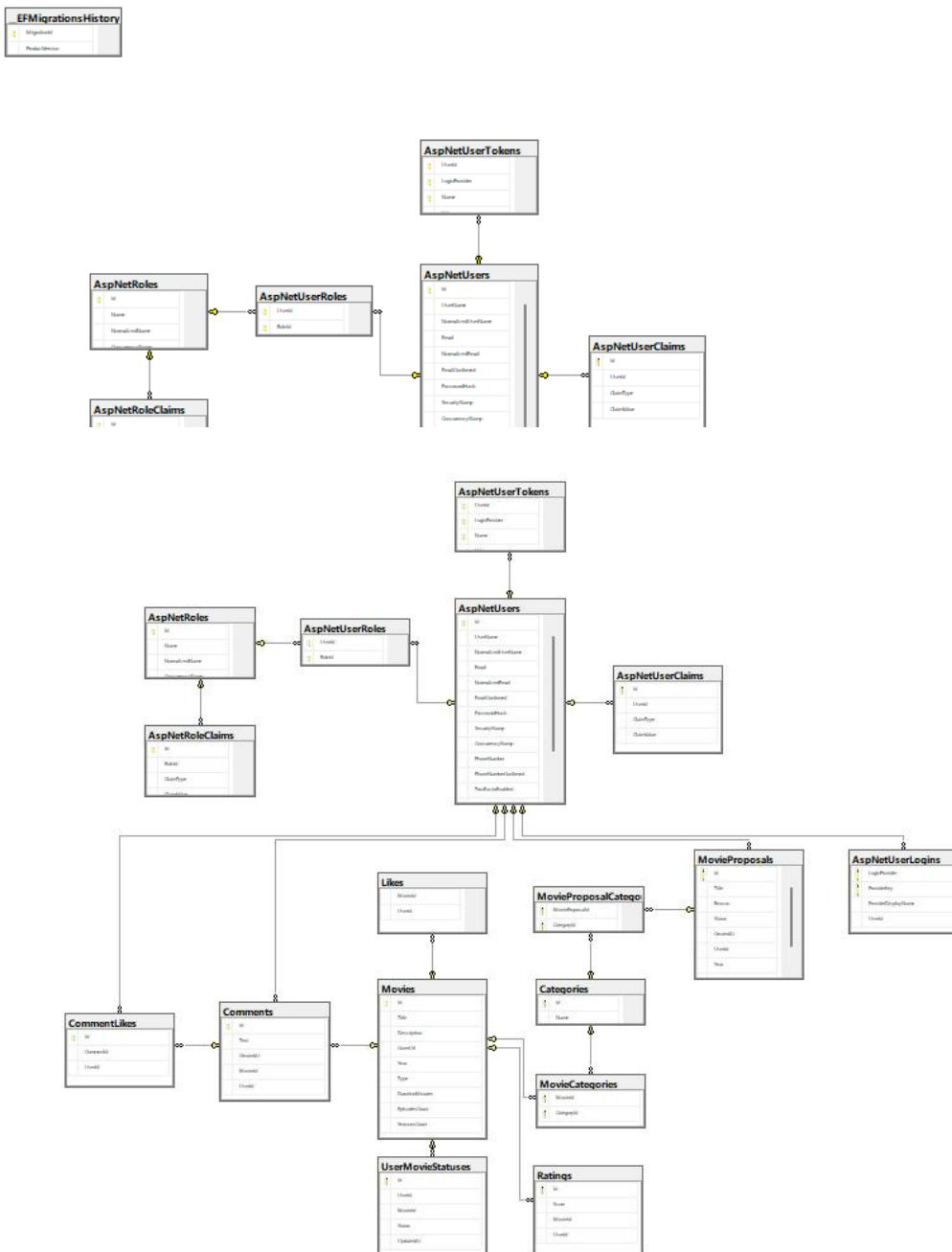


Diagram ER przedstawia strukturę relacyjnej bazy danych aplikacji związanej z filmami oraz użytkownikami, opartej na ASP.NET Identity. Baza danych składa się z tabel systemowych (użytkownicy, role, logowanie) oraz tabel aplikacyjnych (filmy, komentarze, oceny itp.), połączonych relacjami 1:N oraz N:M.

## *1. Tabele i ich przeznaczenie*

### Użytkownicy i autoryzacja (ASP.NET Identity)

#### AspNetUsers

Główna tabela przechowująca dane użytkowników.

- PK: Id
- Atrybuty: UserName, Email, PasswordHash, PhoneNumber, LockoutEnabled itd.
- Tabela centralna: powiązana z wieloma innymi tabelami aplikacyjnymi.

#### AspNetRoles

Przechowuje role systemowe (np. Admin, User).

- PK: Id
- Atrybuty: Name, NormalizedName

#### AspNetUserRoles

Tabela pośrednia realizująca relację N:M między użytkownikami a rolami.

- PK (złożony): UserId, RoleId
- FK:
  - UserId AspNetUsers.Id
  - RoleId AspNetRoles.Id

#### AspNetUserClaims, AspNetRoleClaims, AspNetUserLogins, AspNetUserTokens

Tabele pomocnicze obsługujące mechanizmy:

- roszczeń (claims),
- zewnętrznych logowań,
- tokenów uwierzytelniających.

## *2. Część aplikacyjna filmy*

### 2.1. Filmy i kategorie

#### Movies

- PK: Id
- Atrybuty: Title, Description, CoverUrl, Year, Type, DurationMinutes, EpisodesCount
- Reprezentuje filmy lub seriale.

#### Categories

- PK: Id
- Atrybuty: Name

- Przechowuje kategorie filmów (np. Akcja, Dramat).

## MovieCategories

Tabela pośrednia realizująca relację N:M między filmami a kategoriami.

- PK (złożony): MovieId, CategoryId
- FK:
  - MovieId Movies.Id
  - CategoryId Categories.Id

## 3. Interakcje użytkowników

### 3.1. Komentarze i polubienia

#### Comments

- PK: Id
- Atrybuty: Text, CreatedAt
- FK:
  - MovieId Movies.Id
  - UserId AspNetUsers.Id
- Relacja 1:N: jeden film: wiele komentarzy.

#### CommentLikes

- PK: Id
- FK:
  - CommentId Comments.Id
  - UserId AspNetUsers.Id
- Umożliwia polubienia komentarzy.

#### Likes

- PK: Id
- FK:
  - MovieId Movies.Id
  - UserId AspNetUsers.Id
- Reprezentuje polubienia filmów przez użytkowników.

### 3.2. Oceny i statusy

#### Ratings

- PK: Id
- Atrybuty: Score
- FK:
  - MovieId Movies.Id

- UserId AspNetUsers.Id
- Umożliwia wystawianie ocen filmom.

#### UserMovieStatuses

- PK: Id
- Atrybuty: Status, UpdatedAt
- FK:
  - MovieId Movies.Id
  - UserId AspNetUsers.Id
- Przechowuje status filmu dla użytkownika (np. „obejrzany”, „w trakcie”).

#### 4. Propozycje filmów

##### MovieProposals

- PK: Id
- Atrybuty: Title, Reason, Status, CreatedAt, Year, Type
- FK:
  - UserId AspNetUsers.Id
- Użytkownicy mogą proponować nowe filmy do dodania.

##### MovieProposalCategories

- Tabela pośrednia N:M między propozycjami a kategoriami.
- FK:
  - MovieProposalId MovieProposals.Id
  - CategoryId Categories.Id

#### 5. Klucze, indeksy i integralność danych

- Klucze główne (PK) zapewniają unikalność rekordów w każdej tabeli.
- Klucze obce (FK) gwarantują integralność referencyjną pomiędzy tabelami.
- Relacje 1:N (np. Movies Comments) oraz N:M (np. Movies Categories) są poprawnie odwzorowane.
- Indeksy (np. na UserId, MovieId) poprawiają wydajność zapytań.
- Ograniczenia integralności zapobiegają istnieniu „osieroconych” rekordów.

## 7. Poprawna integracja z Entity Framework

Integracja z Entity Framework Core w FilApp realizuje wzorzec **Code First** z migracjami. Konfiguracja DbContext znajduje się w Program.cs, gdzie rejestrowany jest AppDbContext z SQL Serverem:

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(connectionString));
```

## Fluent API w AppDbContext

W AppDbContext.OnModelCreating() definiowane są relacje N:M, indeksy i ograniczenia, które nie mieścią się w adnotacjach DataAnnotations:

```
// Relacja N:M Movies
modelBuilder.Entity<MovieCategory>()
    .HasKey(mc => new { mc.MovieId, mc.CategoryId });

// Validacja biznesowa
modelBuilder.Entity<Movie>()
    .ToTable(t =>
    {
        t.HasCheckConstraint("CK_Movies_Year", "[Year] IS NULL OR ([Year] >= 1888 AND [Year] <= 2100)");
        t.HasCheckConstraint("CK_Movies_Duration", "[DurationMinutes] IS NULL OR ([DurationMinutes] > 0");
    });

// Indeksy
modelBuilder.Entity<Movie>()
    .HasIndex(i => i.Property(m => m.Title));
modelBuilder.Entity<MovieCategory>()
    .HasIndex(i => i.Property(mc => mc.MovieId));
modelBuilder.Entity<MovieCategory>()
    .HasIndex(i => i.Property(mc => mc.CategoryId));
```

## Migracje

Baza danych jest zarządzana przez migracje Entity Framework Core za pomocą Package Manager Console w Visual Studio, co zapewnia wersjonowanie schematu i automatyczne aktualizacje.

- PM> Add-Migration InitialCreate: Generuje plik migracji na podstawie encji (Movies, Categories, relacje N:M).
- PM> Update-Database: Tworzy/aktualizuje tabele, klucze obce, indeksy w SQL Server.

## 8. Dane testowe (Seed Data)

Dane testowe generowane przez seeder'y EF Core: DbSeeder.cs (kategorie, filmy) i IdentitySeeder.cs (role, admin) – uruchamiane po migracjach.

### 1. DbSeeder.cs

Seeder wypełnia bazę przykładowymi danymi do demonstracji Swagger/CRUD:

- Kategorie (9 szt.): Sci-Fi, Dramat, Akcja, Komedia, Drama, Romans, Anime, Horror, Thriller.
- Filmy/Seriale (4 szt.):
  - Incepca (2010, Movie, 148 min, Sci-Fi/Thriller).
  - Matrix (1999, Movie, 136 min, Sci-Fi).
  - Ojciec chrzestny (1972, Movie, 175 min, Dramat).
  - Breaking Bad (2008, Series, 5 sezonów, 62 odc.).

Relacje N:M MovieCategory seedowane (Incepca: Sci-Fi+Thriller).

### 2. IdentitySeeder.cs

Tworzy role i konto admina do testów autoryzacji:

- Role: User, Admin.
- Admin: admin@filmapp.local / Admin123! (rola Admin).

### 3. Uruchomienie seedera

W Program.cs: DbSeeder.Seed(appDbContext, services); po Update-Database.

## 9. Procedura, Wyzwalacz, Funkcja użytkowa

### Procedura sp\_MoviePopularityReport

Procedura składowana: sp\_MoviePopularityReport.

Raportuje popularność filmu z TOP kategoriami:

- SELECT TOP (@TopCategories) Category, Likes, AvgRating, Comments
- INNER JOIN MovieCategories + Categories (relacja N:M)
- GROUP BY + ORDER BY Likes DESC
- Użycie: EXEC sp\_MoviePopularityReport 1, 5

```
SQLQuery2.s...\\gosia (74)*  X  DESKTOP-OL3... - Diagram_0
1      CREATE PROCEDURE sp_MoviePopularityReport
2          @MovieId INT,
3          @TopCategories INT = 5
4      AS
5      BEGIN
6          SELECT TOP (@TopCategories)
7              c.Name AS Category,
8                  COUNT(l.Id) AS Likes,
9                  AVG(r.Score) AS AvgRating,
10                 COUNT(c.Id) AS Comments
11             FROM Movies m
12             INNER JOIN MovieCategories mc ON m.Id = mc.MovieId
13             INNER JOIN Categories c ON mc.CategoryId = c.Id
14             LEFT JOIN Likes l ON m.Id = l.MovieId
15             LEFT JOIN Ratings r ON m.Id = r.MovieId
16             LEFT JOIN Comments com ON m.Id = com.MovieId
17             WHERE m.Id = @MovieId
18             GROUP BY c.Id, c.Name
19             ORDER BY Likes DESC;
20
21      END;
```

100 % ✓ Nie znaleziono żadnych problemów

Komunikaty

Polecenia zostały ukończone pomyślnie.

Czas ukończenia: 2026-01-26T06:26:14.3789680+01:00

100 % ✓ Nie znaleziono żadnych problemów

## Trigger

Wyzwalacz: trg\_NoDuplicateLikes

Blokuje duplikaty lajków (AFTER INSERT):

- Sprawdza EXISTS duplikat MovieId+UserId
- RAISERROR + ROLLBACK przy próbie powtórkii
- Zapewnia integralność danych Likes (unikalność User-Film)

The screenshot shows a SQL Server Management Studio window with the following details:

- Title Bar:** SQLQuery2.s... \ gosia (74)\* DESKTOP-OL3... - Diagram\_0
- Code Area:** The code for the trigger is displayed, numbered from 1 to 21.

```
1 CREATE TRIGGER trg_NoDuplicateLikes
2     ON Likes
3     AFTER INSERT
4     AS
5     BEGIN
6         DECLARE @MovieId INT, @UserId NVARCHAR(450)
7
8         SELECT @MovieId = MovieId, @UserId = UserId
9         FROM inserted
10
11        IF EXISTS (
12            SELECT 1 FROM Likes
13            WHERE MovieId = @MovieId AND UserId = @UserId
14            AND Id != (SELECT Id FROM inserted WHERE MovieId = @MovieId AND UserId = @UserId)
15        )
16        BEGIN
17            RAISERROR('Duplicate like detected!', 16, 1);
18            ROLLBACK TRANSACTION;
19        END
20    END;
21
```
- Status Bar:** 100 % ✓ Nie znaleziono żadnych problemów
- Messages Tab:** Komunikaty  
Polecenia zostały ukończone pomyślnie.  
Czas ukończenia: 2026-01-26T06:25:51.2454289+01:00

## Funkcja

Funkcja użytkownika: fn.GetUserStats(@UserId)

Statystyki użytkownika (TABLE VALUED FUNCTION):

- LikesCount, AvgRating, CommentsCount, ProposalsCount
- LEFT JOIN wszystkich encji powiązanych z UserId
- GROUP BY u.Id
- Użycie: SELECT \* FROM fn.GetUserStats('user-guid')

The screenshot shows the SQL Server Management Studio interface with a query window titled 'SQLQuery2.s...\\gosia (74)\*'. The code listed is a T-SQL script for creating a function:

```
1 CREATE FUNCTION fn.GetUserStats(@UserId NVARCHAR(450))
2 RETURNS TABLE
3 AS
4 RETURN
5 (
6     SELECT
7         COUNT(DISTINCT l.MovieId) AS LikesCount,
8         AVG(r.Score) AS AvgRating,
9         COUNT(c.Id) AS CommentsCount,
10        COUNT(p.Id) AS ProposalsCount
11    FROM AspNetUsers u
12    LEFT JOIN Likes l ON u.Id = l.UserId
13    LEFT JOIN Ratings r ON u.Id = r.UserId
14    LEFT JOIN Comments c ON u.Id = c.UserId
15    LEFT JOIN MovieProposals p ON u.Id = p.UserId
16    WHERE u.Id = @UserId
17    GROUP BY u.Id
18 );
19
```

The status bar at the bottom indicates '100 %' completion and 'Nie znaleziono żadnych problemów' (No problems found). The 'Komunikaty' (Messages) pane shows the message 'Polecenia zostały ukończone pomyślnie.' (The command was successfully completed) and the execution time 'Czas ukończenia: 2026-01-26T06:25:05.7175911+01:00'.

At the bottom of the screenshot, there is a yellow banner with the message 'Zapytanie zostało wykonane pomyślnie.' (The query was executed successfully).

## 10. Frontend aplikacji Film

Frontend to React 18 SPA z TypeScript + Tailwind CSS: pełna obsługa CRUD, autoryzacja JWT i routing z guards (RequireAuth/Admin).

Architektura: Vite + React Router + Axios wrapper (apiFetch.ts). Dostęp via npm run dev localhost:5173.

### Technologie i struktura

Główna technologia: React + TypeScript z bundlerem Vite (szybki reload) i stylingiem Tailwind CSS.

Kluczowe biblioteki i narzędzia:

- React Router DOM: routing z guards (RequireAuth, RequireAdmin).
- Tailwind CSS: responsywne klasy (np. grid-cols-1 md:grid-cols-2, hover:bg-white5).
- TypeScript: pełne typy (Movie, CreateMovie, MovieUpsertPayload).
- Custom hooks: apiFetch (HTTP client), getAuth (JWT manager).

### Formularz logowania i obsługa autoryzacji

LoginPage.tsx implementuje kompletny formularz logowania z walidacją, komunikacją JWT i automatycznym przekierowaniem po sukcesie.

Główne funkcje:

- Walidacja formularza: pola email/hasło wymagane, format email
- POST /api/auth/login: wysyła credentials, a backend zwraca JWT token
- localStorage: zapis tokena + user data dla sesji
- Auto-redirect: po sukcesie do strony głównej
- Error handling: 401/400

Obsługa ról po stronie użytkownika:

- RequireAuth: blokuje nieautoryzowanych (redirect /login)
- RequireAdmin: blokuje User z paneli admin (403 Forbidden)
- /api/auth/whoami: debug endpoint z rolami (User.Identity.Name + Roles)

Bezpieczeństwo:

- Auto-logout na 401 Unauthorized
- Hasła nigdy nie logowane

## Komunikacja z API poprzez zapytania GET, POST, PUT, DELETE

Komunikacja z backendem odbywa się za pomocą centralnego klienta HTTP apiFetch.ts, który realizuje wszystkie cztery wymagane operacje CRUD (GET, POST, PUT, DELETE) z automatycznym dołączaniem tokena JWT do nagłówków.

### 1. GET pobieranie listy filmów:

Wywołanie `apiFetch('/api/movies?page=1&pageSize=10')` pobiera paginowaną listę filmów (10 na stronę) z pełnymi danymi: tytuł, rok, typ (film/serial), kategorie i średnia ocen. Dane lądują w tabeli z przyciskami edycji/usuwania.

### 2. POST tworzenie nowego filmu:

Użytkownik wypełnia formularz (tytuł, opis, okładka, rok, typ), system automatycznie dostosowuje pola: dla filmu wymagany czas trwania, dla serialu liczba sezonów/odcinków. Dane wysyłane jako JSON na `/api/movies`. Po sukcesie film pojawia się w tabeli.

### 3. PUT edycja filmu:

Kliknięcie "Edytuj" w tabeli ląduje dane do formularza (`setEditId(id)`). Zmiany zapisywane przez `PUT /api/movies/{id}`. Backend waliduje dane (np. serial nie może mieć `DurationMinutes`).

### 4. DELETE usuwanie filmu:

Przycisk "Usuń" pokazuje potwierdzenie (`confirm()`), potem `DELETE /api/movies/{id}`. Film znika z listy po odświeżeniu.

### 5. Dodatkowe operacje relacji N:M (kategorie):

Dla każdego filmu osobne przyciski "Dodaj kategorię" (select + `POST /movies/{id}/categories/{catId}`) i "Usuń kategorię" (`DELETE` tej samej ścieżki). System blokuje duplikaty (409 Conflict).

### 6. Automatyczna obsługa błędów:

- 401 Unauthorized: automatyczne wylogowanie i redirect na `/login`
- 400 Bad Request: komunikat walidacji z backendu (`ProblemDetails`)
- 404 Not Found: "Film nie istnieje"
- 409 Conflict: "Kategoria już przypisana"

7. Stany ładowania: Podczas każdego requestu przyciski pokazują spinner (`busy: true`), a użytkownik nie może wykonać kolejnej operacji.

8. Token JWT: Każdy request automatycznie dostaje nagłówek `Authorization: Bearer ${token}` pobrany z `localStorage` przez `getAuth()`.

9. Przykładowy kompletny flow:

- Admin loguje się: token w localStorage
- Przechodzi do AdminMoviesPage: GET lista filmów
- Klik "Dodaj film": wypełnia form: POST nowy film
- Dodaje kategorie: POST/DELETE relacje N:M
- Edytuje istniejący: PUT zmiany
- Usuwa niepotrzebny: DELETE z potwierdzeniem

## 11. Zrzuty ekranu z gotowej aplikacji pokazujące możliwości

### Pannel Administartora

**Panel admina: Filmy**

Dodaj / Edytuj

Breaking Bad 2008

Serial kryminalny.

[https://m.media-amazon.com/images/M/MV5BMzU5ZGYzMjM0NjIzTg1ZGE4XkEyXkFqcGc@.\\_V1\\_FMjpg\\_UX1000\\_.jpg](https://m.media-amazon.com/images/M/MV5BMzU5ZGYzMjM0NjIzTg1ZGE4XkEyXkFqcGc@._V1_FMjpg_UX1000_.jpg)

Series 120

5 60

+ Dodaj kategorię

Akcja x

Zapisz zmiany Anuluj

**LISTA** ODŚWIĘŻ

TYTUŁ	ROK	TYP	KATEGORIE	AKCJE
aaaaaa	2026	Movie	Anime x Komedie x + Dodaj kategorię Dodać	Edytuj Usuń
aaaaaaaaaa	2026	Movie	Anime x + Dodaj kategorię Dodać	Edytuj Usuń
asdasdaasdadasdad	2026	Movie	Akcja x + Dodaj kategorię Dodać	Edytuj Usuń
asdsf	2026	Movie	Drama x + Dodaj kategorię Dodać	Edytuj Usuń
Breaking Bad	2008	Series	+ Dodaj kategorię Dodać	Edytuj Usuń
fajnyfilm	2020	Movie	Akcja x + Dodaj kategorię Dodać	Edytuj Usuń
Incepja	2010	Movie	Sci-Fi x Thriller x + Dodaj kategorię Dodać	Edytuj Usuń

## Panel admina: Kategorie

DODAJ

Nazwa kategorii

Dodaj

LISTA

DODAJ

Akcja



Anime



Drama



Dramat



Horror



Komedia



Romans



Sci-Fi



Test 1



Test 2



Thriller



Film Profil

STRONA GŁÓWNA | EKSPLORUJ | PROFIL

## Dzień dobry, Administratorze

Oto szybki przegląd systemu

BAZA TYTUŁÓW 15

NOWE PROPOZYCJE 3

UŻYTKOWNICY 4

OCENY DZIŚ

### Szybki dostęp

Zarządzaj filmami Edytuj bibliotekę tytułów

Kategorie Zarządzaj kategoriami

Propozycje Akceptuj propozycje userów

Raporty Statystyki i analizy

Wyloguj się

Film Profil

STRONA GŁÓWNA | EKSPLORUJ | PROFIL

## Raporty Admina

Odśwież Eksportuj CSV

FILM	ŚREDNIA	OCENY
Incepcja	5.0	2
Matrix Test	5.5	2
aaaaaa	7.5	2
aaaaaaaaaa	5.5	2

STRONA 1 Z 1

POPRZEDNIA NASTĘPNA



## Admin: Propozycje

Status: Pending ▾ Odśwież

Tytuł	Rok	Typ	Userid	Status	Akcje
cs	2024	Movie	84c0eddc-be13-4cd4-8177-9ef8613f715b	Pending	<button>Approve</button> <button>Reject</button>
cos	2019	Movie	84c0eddc-be13-4cd4-8177-9ef8613f715b	Pending	<button>Approve</button> <button>Reject</button>
Jakis2	2025	Series	84c0eddc-be13-4cd4-8177-9ef8613f715b	Pending	<button>Approve</button> <button>Reject</button>

## Panel dla każdego (zalogowanych oraz niezalogowanych)

The screenshot shows a dark-themed user interface for a movie and TV show database. At the top, there's a navigation bar with a 'Film' icon and the word 'Film'. On the right side of the bar are links for 'STRONA GŁÓWNA', 'EKSPLORUJ', and 'PROFIL'. In the top right corner, there's a 'Profil' link. Below the navigation bar is a search bar with the placeholder text 'Szukaj filmów, seriali, osób...'. Underneath the search bar are two tabs: 'Filmy' (highlighted in blue) and 'Serials' (highlighted in purple). Below these tabs are sorting options: 'Title ↑' (sorted by title in ascending order), 'Asc' (sort order), 'Filtry' (filters), and a heart icon for saving items. A 'Mode: Exact' button and a 'Wyczyszczać kategorie' (Clear categories) button are also present. A row of genre filters follows, each with a checked checkbox: 'Akcja' (Action), 'Anime', 'Drama', 'Dramat', 'Horror', 'Komedia', 'Romans', 'Sci-Fi', 'Test 1', 'Test 2', and 'Thriller'. The main content area is titled 'Odkrywaj' (Discover). It features a large thumbnail for the TV show 'Breaking Bad', showing a man in a yellow suit sitting in a chair in a warehouse. Below the thumbnail is the show's title 'Breaking Bad'. Underneath the thumbnail, there's a small summary: 'BREAKING BAD', 'Serial 2008', 'Akcja', and a heart icon with the number '0'. At the bottom of the page, there are navigation arrows for the discover section, with the number '1' indicating the current page.

Film

STOSUNKI DŁUGIE | REPREZENTACJA | PROFIL

Najlepiej ocenione filmy

Najlepiej ocenione seriale

Poznaj te filmy

Dekrywaj więcej



Film

Profil

STRONA GŁÓWNA | EKSPLORUJ | PROFIL



## OPIS

Thriller sci-fi.

## Inception

2010 • 148 min • SCI-FI, THRILLER



2

5.7/10 (2)

 W ulubionych

TWÓJ SCENA

1	2	3	4	5
6	7	8	9	10

 Chcę zobaczyć Obejrzać Oglądam

## KOMENTARZ

Dodaj komentarz...

 DODAJ

email@e.e

25.01.2016, 06:26:41

Super film!

 Usuń Zapisz 101

admin@filmappp.local

24.01.2016, 10:10:58

Send comment (admin).



Film

Zaloguj

STRONA GŁÓWNA | EKSPLORUJ | PROFIL

## Zaloguj się

E-mail lub nazwa użytkownika

Hasło

Zaloguj się

Nie masz konta? [Zarejestruj się](#)

O nas Regulamin Prywatność Pomoc



# Panel użytkownika

The screenshot displays the User Panel interface with the following components:

- Header:** Shows the logo "Film" and the word "Profil" in the top right corner.
- User Information:** Greeted with "Witaj, user@test.com!" and shows the user is logged in as "user@test.com" with the role "User".
- User Statistics:** Three cards showing counts: "POLUBIONE" (4), "OBEJRZANE" (1), and "MOJE PROPOZYCJE" (6).
- Twoje panele:** A section listing five items with icons:
  - Moje Propozycje:** Przeglądaj i dodawaj nowe propozycje filmów do bazy.
  - Polubione:** Twój lista ulubionych filmów i seriali.
  - Oglądam:** Zarządzaj listą tytułów, które aktualnie oglądasz.
  - Chęt obiejrzeć:** Lista filmów i seriali do nadrobienia.
  - Obejrzone:** Historia Twoich obejrzanych produkcji.
- Szybkie akcje:** A section with three buttons:
  - Zmień Hasło**
  - Ustawienia Profilu**
  - Wyloguj się**
- Footer:** Includes the "MOVIE Film" logo, "User Support" link, "ver 2.4.0 stable", copyright notice "© 2024 SYSTEM BAZY FILMÓW. PULPIT UŻYTKOWNIKA.", a "Odkrywaj" button with a play icon, and "Profil" link.

## Moje propozycje

## Dodaj propozycję

Tytuł Movie 

Powód (reason)

## Kategorie

- Akcja
- Anime
- Drama
- Dramat
- Horror
- Komedia
- Romans
- Sci-Fi
- Test 1
- Test 2
- Thriller

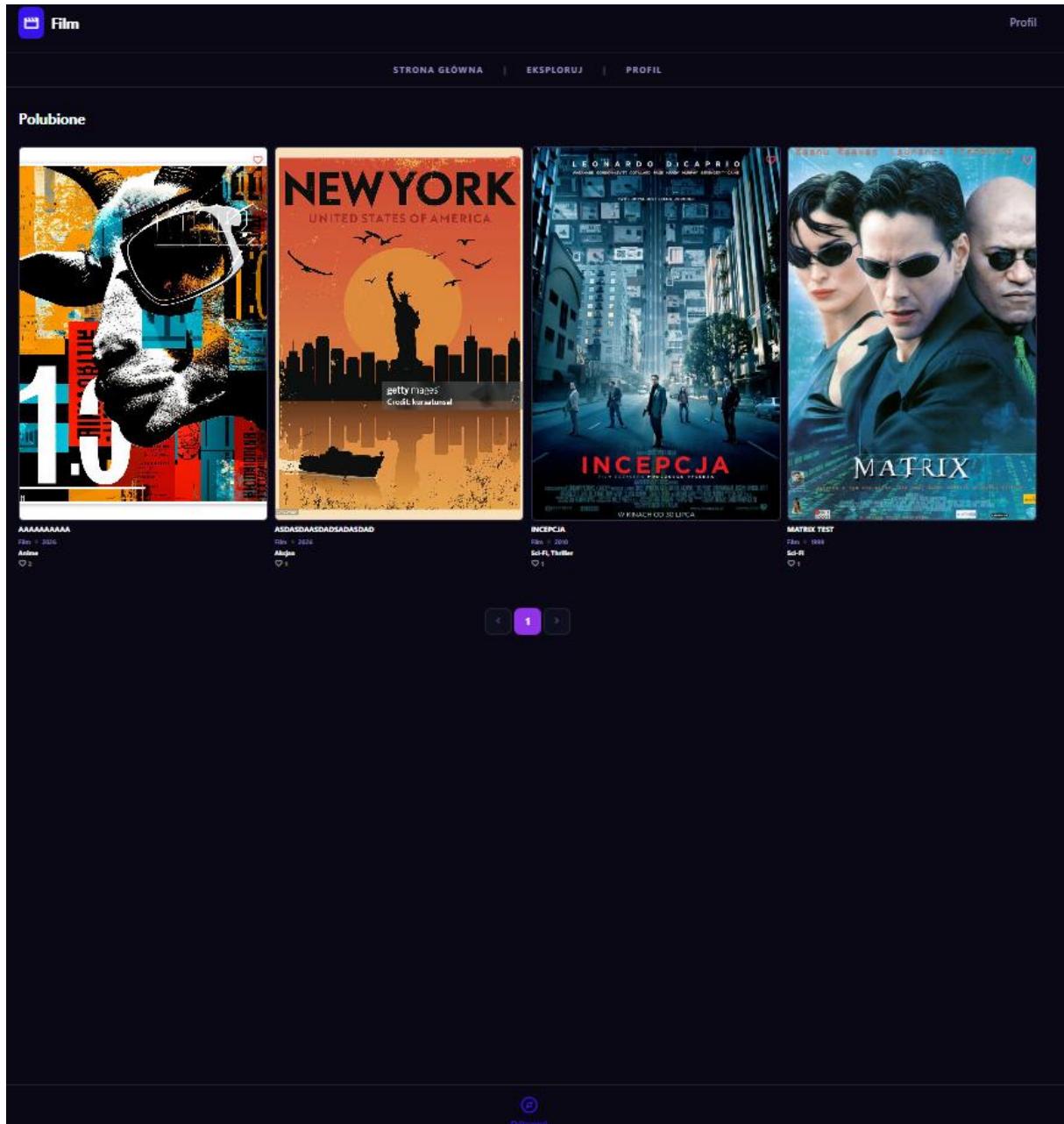
Dodaj

## Lista

Odswież

TYTUŁ	ROK	TYP	STATUS
s	2028	Series	Approved
cs	2024	Movie	Pending
cos	2019	Movie	Pending
Jakis	2025	Movie	Rejected
Jakis2	2025	Series	Pending
JakisUsera	2015	Movie	Approved







Film

Profil

STRONA GŁÓWNA | EKSPLORUJ | PROFIL

## Obejrzane

[Wróć do profilu](#)

fajnyfilm

2020 • Akcja

Likes: 0 Rating: ★ 0.0 (0)

25.01.2026

[Szczegóły](#)[Usuń status](#)

@dienasgal

# Instrukcja uruchomienia aplikacji Film

## Wymagania

- Visual Studio 2022+ (Community OK)
- SQL Server (LocalDB lub instancja)
- Node.js 18+ (dla React frontend)
- Git (do pobrania repo)

## Pobranie i otwarcie projektu

- GitHub/GitLab: Download ZIP lub git clone
- Visual Studio: File: Open: FilmApp.sln
- Solution Explorer: Restore NuGet Packages (prawy przycisk)

## Konfiguracja bazy danych

- Tools: Package Manager Console (PMC)
- Update-Package (odśwież pakiety EF Core)
- PMC: Add-Migration InitialCreate
- PMC: Update-Database
- Baza gotowa

## Uruchomienie Backend (.NET API)

- Solution Explorer: FilmApp (projekt z Program.cs)
- Prawy przycisk: Set as Startup Project
- F5 lub Debug: Start Debugging
- API działa: <https://localhost:7xxx/swagger>

## Uruchomienie Frontend (React)

- VS Code / Terminal: cd FilmApp/ClientApp
- npm install (pierwszy raz)
- npm run dev
- Frontend: <http://localhost:5xxx>

## Konta testowe

- Admin: adminfilmapp.local / Admin123!
- User: usertest@example.com / Test123!