MovieGO

1. Tytuł projektu

Aplikacja internetowa do tworzenia i zarządzania rezerwacjami w kinie.

2. Krótki opis działania projektu

W ramach aplikacji Użytkownik będzie mógł dokonać rejestracji konta, a po zalogowaniu rezerwacji miejsca dla konkretnego seansu na wybranej sali, we wskazanym terminie. Podstrony publiczne, widoczne dla wszystkich będą pozwalały na wyświetlenie dostępnych seansów oraz już zarezerwowanych miejsc w wybranym terminie. Rezerwacje będą podlegać modyfikacji, o ile nie został przekroczony określony termin lub dane miejsce nie będzie już zajęte przez innego użytkownika.

3. Autor/autorzy projektu

Jakub Horyd, Przemysław Drygas

4. Instrukcja uruchamiania projektu

Wykorzystane technologie

- **Język programowania:** C#
- **Framework:** NET 8
- **Wykorzystana baza danych:** MS SQL SERVER
- **ORM:** Enitity Framework Core
- **Frontend:** Razor Pages, HTML + CSS, JavaScript, Bootstrap 5.3

Wymagania

- **System operacyjny**: Windows 10/11 lub Linux z obsługą .NET 8.
- **.NET SDK**: Wersja 8.0.
- **Serwer bazy danych**: MS SQL Server (preferowany) lub SQLite (dostosowanie projektu do SQLite wymaga dodatkowej konfiguracji).
- **IDE**: Visual Studio 2022 (lub nowszy) z ASP .NET 8

Opcja 1: Nowa baza danych

1. Otwórz kod źródłowy projektu w Visual Studio.
2. Otwórz **Package Manager Console** w Visual Studio
    - W menu wybierz **Tools** > **NuGet Package Manager** > **Package Manager Console**.
3. Wykonaj polecenie, aby zastosować migracje i utworzyć nową bazę danych:

    ```
    Update-Database
    ```

4. Uruchom aplikację za pomocą terminala:

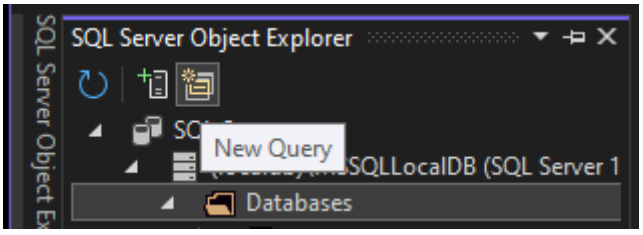
```
dotnet run
```

   - lub bezpośrednio w Visual Studio, klikając przycisk **Uruchom**.
5. Domyślnie aplikacja zostanie uruchomiona pod jednym z poniższych adresów:
   - `https://localhost:7291`
   - `http://localhost:5132`.
6. Otwórz wybrany adres w przeglądarce, aby rozpocząć korzystanie z aplikacji.

Opcja 2: Gotowa baza danych

1. Do projektu dołączona jest wstępnie wypełniona baza danych. Znajdziesz ją w plikach projektu: **MovieGO\Data\MovieGO_DB_TEST.mdf**
2. Aby podłączyć bazę danych:

- Otwórz **SQL Server Object Explorer** w Visual Studio.

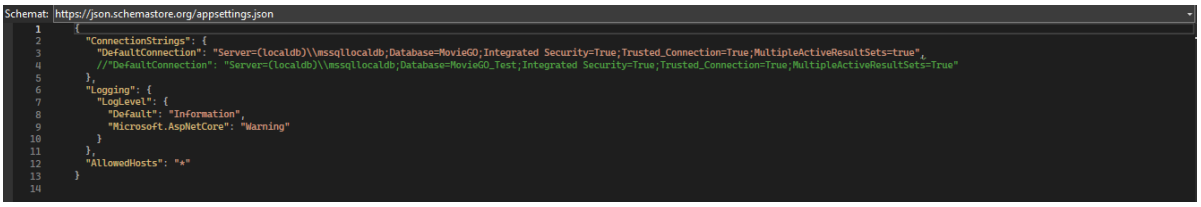


- Wykonaj poniższe zapytanie:

```
CREATE DATABASE MovieGO_Test
ON (FILENAME = '<SCIEŻKA_DO_PROJEKTU>\MovieGO\Data\MovieGO_DB_TEST.mdf')
FOR ATTACH;
```

3. Gdzie "<SCIEŻKA_DO_PROJEKTU>" wskaż lokalizację pliku `.mdf` z kopią bazy danych znajdującego się w folderze projektu: **MovieGO\Data\MovieGO_DB_TEST.mdf**
4. Przejdź do pliku `appsettings.json` i zmień connection string na ten z odkomentowanym parametrem `DefaultConnection`:

```
"ConnectionStrings": {
  "DefaultConnection": "Data Source=
(LocalDB)\\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\\Data\\MovieGO_DB_TEST.m
df;Integrated Security=True;MultipleActiveResultSets=True"
}
```



Konto administratora

Aplikacja automatycznie tworzy konto administratora oraz role użytkowników podczas pierwszego uruchomienia. Dane logowania do konta administratora to:

- **Login**: `admin@admin.com`
- **Hasło**: `Admin123#`

Kod odpowiedzialny za utworzenie konta administratora znajduje się w pliku `Program.cs`:

```
using (var scope = app.Services.CreateScope())
{
    var roleManager =
scope.ServiceProvider.GetRequiredService<RoleManager<IdentityRole>>();
    var userManager =
scope.ServiceProvider.GetRequiredService<UserManager<IdentityUser>>();

    var roles = new[] { "Administrator", "User" };

    foreach (var role in roles)
    {
        if (!await roleManager.RoleExistsAsync(role))
            await roleManager.CreateAsync(new IdentityRole(role));
    }

    var adminEmail = "admin@admin.com";
    var adminPassword = "Admin123#";

    if (await userManager.FindByEmailAsync(adminEmail) == null)
    {
        var adminUser = new IdentityUser
        {
            UserName = adminEmail,
            Email = adminEmail,
            EmailConfirmed = true
        };

        var result = await userManager.CreateAsync(adminUser, adminPassword);
        if (result.Succeeded)
        {
            await userManager.AddToRoleAsync(adminUser, "Administrator");
        }
        else
        {
            var logger =
scope.ServiceProvider.GetRequiredService<ILogger<Program>>();
            foreach (var error in result.Errors)
            {
                logger.LogError($"Nie udało się utworzyć konta:
{error.Description}");
            }
        }
    }
}
```

5. Opis struktury projektu

Projekt jest oparty na wzorcu architektonicznym **Model-View-Controller (MVC)**.

Struktura

- `Controllers`
  - Zawiera logikę obsługi żądań HTTP i przetwarzania danych dla widoków.
- `Models`
  - Odpowiada za dane aplikacji i definicje encji bazy danych.
- `Views`
  - Przechowuje pliki widoków generowanych przez **Razor**.
- `Data`
  - Zawiera plik `ApplicationDbContext`, który definiuje konfigurację bazy danych i powiązania między modelami oraz kopię bazy danych zawartą w pliku `MovieGO_DB_TEST.mdf`
- `Migrations`
  - Zawiera automatycznie wygenerowane pliki migracji dla bazy danych.
- `wwwroot`
  - Przechowuje pliki CSS, JavaScript i obrazy.

6. Modele wykorzystane w projekcie

Category

Model `Category` odpowiada za gatunki filmowe przypisane do filmów

Pola

- `Id` **(int)**: Identyfikator gatunku.
- `Name` **(string)**:
  - Nazwa gatunku filmowego.
  - **Walidacje**:
    - Pole wymagane.
    - Maksymalna długość: 50 znaków.
  - Wyświetlane jako "Nazwa gatunku".
- `Movies` **(kolekcja** `Movie`**)**: Lista filmów przypisanych do danego gatunku.

Cinema

Model `Cinema` przechowuje informacje o kinach, nazwa, adres, ścieżka do zdjęcia podglądowego czy przypisane sale kinowe.

Pola

- `Id` **(int)**: Identyfikator kina.
- `Name` **(string)**:
  - Nazwa kina.

    - **Walidacje**:
        - Pole wymagane.
        - Maksymalna długość: 50 znaków.
    - Wyświetlane jako "Nazwa kina".
- `StreetAddress` **(string)**:
    - Adres ulicy.
    - **Walidacje**:
        - Pole wymagane.
        - Maksymalna długość: 50 znaków.
- `PostalCode` **(string)**:
    - Kod pocztowy.
    - **Walidacje**:
        - Pole wymagane.
        - Format: XX-XXX, gdzie X: cyfra.
- `City` **(string)**:
    - Miasto.
    - **Walidacje**:
        - Pole wymagane.
        - Maksymalna długość: 50 znaków.
- `ImagePath` **(string, opcjonalne)**: Ścieżka do obrazu przedstawiającego kino.
- `Halls` **(kolekcja** `Hall`**)**: Lista sal kinowych przypisanych do kina.

Hall

Model `Hall` odpowiada za sale w danym kinie. Liczba miejsc wynika z liczby rzędów i kolumn.

Pola

- `Id` **(int)**: Identyfikator sali.
- `Name` **(string)**:
    - Nazwa sali.
    - **Walidacje**:
        - Pole wymagane.
        - Maksymalna długość: 50 znaków.
- `RowCount` **(int)**:
    - Liczba rzędów miejsc w sali.
    - **Walidacje**:
        - Pole wymagane.
        - Zakres: 10–20.
- `ColumnCount` **(int)**:

    - Liczba kolumn w sali (miesjca).
    - **Walidacje**:
        - Pole wymagane.
        - Zakres: 10–20.
- `CinemaId` **(int)**: Identyfikator kina do którego należy sala.
- `Cinema` **(referencja do** `Cinema`**)**: Obiekt kina, z którym powiązana jest sala.
- `Screenings` **(kolekcja** `Screening`**)**: Lista seansów przypisanych do sali.

Movie

Model `Movie` odpowiada za filmy w systemie i dane na ich temat, takie jak:

Pola

- `Id` **(int)**: Identyfikator filmu.
- `Title` **(string)**:
    - Tytuł filmu.
    - **Walidacje**:
        - Pole wymagane.
        - Długość: 3–100 znaków.
- `ImagePath` **(string, opcjonalne)**: Ścieżka do obrazu filmu.
- `CategoryId` **(int)**: Identyfikator gatunku, do którego przypisany jest film.
- `Category` **(referencja do** `Category`**)**: Obiekt gatunku filmu.
- `Description` **(string)**:
    - Opis filmu.
    - **Walidacje**:
        - Pole wymagane.
        - Maksymalna długość: 1000 znaków.
- `Duration` **(int)**:
    - Długość filmu w minutach.
    - **Walidacje**:
        - Pole wymagane.
        - Zakres: 20–500 min.
- `Screenings` **(kolekcja** `Screening`**)**: Lista seansów, na których film jest wyswietlany.

Reservation

Opis

Model `Reservation` przechowuje informacje o rezerwacjach miejsc na wybrane seanse.

Pola

- `Id` **(int)**: Identyfikator rezerwacji.
- `ScreeningId` **(int)**: Identyfikator seansu, którego dotyczy rezerwacja.
- `Screening` **(referencja do** `Screening`**)**: Obiekt seansu powiązanego z rezerwacją.
- `Row` **(int)**:
  - Rząd miejsca.
  - **Walidacje**:
    - Zakres: 1–20.
- `Column` **(int)**:
  - Kolumna miejsca.
  - **Walidacje**:
    - Zakres: 1–20.
- `UserId` **(string)**: Identyfikator użytkownika, który zarezerwował miejsce.
- `User` **(referencja do użytkownika)**: Obiekt użytkownika.

Screening

Opis

Model `Screening` odpowiada za seanse. Zawiera informacje o filmie, sali i dacie przypisanej do danego seansu.

Pola

- `Id` **(int)**: Identyfikator seansu.
- `IsHidden` **(bool)**: Flaga określająca, czy seans jest ukryty. Seanse są ukrywane w kontrolerze po 24h od daty seansu.
- `ScreeningDate` **(DateTime)**:
  - Data i godzina seansu.
  - **Walidacje**:
    - Pole wymagane.
    - Seans można utworzyć najwcześniej 24 godziny od teraz.
- `HallId` **(int)**: Identyfikator sali, na której odbywa się seans.
- `Hall` **(referencja do** `Hall`**)**: Obiekt sali kinowej.
- `MovieId` **(int)**: Identyfikator filmu wyświetlanego na seansie.
- `Movie` **(referencja do** `Movie`**)**: Obiekt filmu.

- `Reservations` **(kolekcja** `Reservation`**)**: Lista rezerwacji przypisanych do seansu.

7. Kontrolery wykorzystane w projekcie

MoviesController

Standardowy kontroler CRUD dla filmów wygenerowany z szablonów Entity Framework. Nie zaszły istotne zmiany w logice.

ReservationsController

Index

- **HTTP**: `GET`
- **Parametry**: Brak
- **Opis**: Wyświetla filtrowaną listę rezerwacji w zależności od roli użytkownika:
  - Administrator widzi wszystkie rezerwacje.
  - Użytkownik widzi tylko swoje rezerwacje.
- **Realizacja filtrowania:**

```
var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
var reservations = await _context.Reservations
    .Where(r => User.IsInRole("Administrator") || r.UserId == userId)
    .Include(r => r.Screening)
        .ThenInclude(s => s.Movie)
    .Include(r => r.Screening)
        .ThenInclude(s => s.Hall)
            .ThenInclude(h => h.Cinema)
    .OrderBy(r => r.Screening.ScreeningDate)
    .ToListAsync();
```

```
- **Zwracane dane**: Widok z listą rezerwacji.

#### **Create (GET)**
- **HTTP**: `GET`
- **Parametry**: `int screeningId` - Identyfikator seansu.
- **Opis**:
  - Wyświetla formularz rezerwacji dla konkretnego seansu.
  - Walidacja:
    - Sprawdza, czy seans istnieje.
    - Sprawdza, czy seans już się odbył.
  - **Kod walidacji:**
    ```csharp
    var screening = _context.Screenings
        .Include(s => s.Movie)
        .Include(s => s.Hall)
            .ThenInclude(h => h.Cinema)
        .Include(s => s.Reservations)
        .FirstOrDefault(s => s.Id == screeningId);
```
```

```
if (screening == null || screening.ScreeningDate < DateTime.Now)
{
    ViewBag.ErrorMessage = "Seans nie istnieje lub już się odbył.";
    return View(new Reservation());
}
```

- **Zwracane dane**: Widok formularza lub komunikat błędu.

Create (POST)

- **HTTP**: `POST`
- **Parametry**: `Reservation reservation` - Dane rezerwacji.
- **Opis**:
  - Przypisuje aktualnie zalogowanego użytkownika do rezerwacji:

    ```
    reservation.UserId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    ```

  - Walidacja:
    - Sprawdza, czy seans istnieje.
    - Sprawdza, czy seans już się odbył.
  - **Kod walidacji:**

    ```
    var screening = await
    _context.Screenings.FindAsync(reservation.ScreeningId);
    if (screening == null || screening.ScreeningDate < DateTime.Now)
    {
        ModelState.AddModelError(string.Empty, "Nie można dokonać rezerwacji
    na seans, który już się odbył.");
        return View(reservation);
    }
    ```

- **Zwracane dane**: Przekierowanie do listy rezerwacji lub widok z komunikatem błędu.

Edit (GET)

- **HTTP**: `GET`
- **Parametry**: `int id` - Identyfikator rezerwacji.
- **Opis**:
  - Wyświetla formularz edycji rezerwacji.
  - Walidacja:
    - Sprawdza, czy rezerwacja istnieje oraz czy seans już się odbył.
  - **Kod walidacji:**

    ```
    var reservation = _context.Reservations
        .Include(r => r.Screening)
            .ThenInclude(s => s.Movie)
        .Include(r => r.Screening)
            .ThenInclude(s => s.Hall)
    ```

```
            .ThenInclude(h => h.Cinema)
    .Include(r => r.Screening)
        .ThenInclude(s => s.Reservations)
    .FirstOrDefault(r => r.Id == id);

if (reservation == null || reservation.Screening == null ||
reservation.Screening.ScreeningDate < DateTime.Now)
{
    ViewBag.ErrorMessage = "Rezerwacja nie istnieje lub seans już się
odbył.";
    return View(new Reservation());
}
```

- Przekazuje dodatkowe dane do widoku:

```
ViewBag.HallRowCount = reservation.Screening.Hall.RowCount;
ViewBag.HallColumnCount = reservation.Screening.Hall.ColumnCount;
ViewBag.ExistingReservations = reservation.Screening.Reservations
    .Where(r => r.Id != reservation.Id)
    .ToList();
```

- **Zwracane dane**: Widok formularza lub komunikat błędu.

Edit (POST)

- **HTTP**: `POST`
- **Parametry**: `Reservation reservation` - Dane edytowanej rezerwacji.
- **Opis**:
  - Sprawdza uprawnienia użytkownika:

```
var existingReservation = await _context.Reservations
    .Include(r => r.User)
    .FirstOrDefaultAsync(r => r.Id == id);

if (existingReservation == null || (!User.IsInRole("Administrator") &&
existingReservation.UserId !=
User.FindFirstValue(ClaimTypes.NameIdentifier)))
{
    return Forbid();
}
```

  - Aktualizuje dane rezerwacji:

```
existingReservation.Row = reservation.Row;
existingReservation.Column = reservation.Column;
_context.Update(existingReservation);
await _context.SaveChangesAsync();
```

- **Zwracane dane**: Przekierowanie do listy rezerwacji lub komunikat błędu.

ScreeningsController

Index

- **HTTP**: `GET`
- **Parametry**:
  - `string? city` - Miasto.
  - `string? movieTitle` - Tytuł filmu.
  - `DateTime? screeningDate` - Data seansu.
- **Opis**:
  - Poniższa metoda ukrywa seanse po 24h od daty upłnięcia seansu:

```
private async Task HidePastScreenings()
{
    var pastScreenings = await _context.Screenings
        .Where(s => s.ScreeningDate < DateTime.Now.AddHours(-24) &&
!s.IsHidden)
        .ToListAsync();

    foreach (var screening in pastScreenings)
    {
        screening.IsHidden = true;
    }

    await _context.SaveChangesAsync();
}
```

  - Z kolei ta moteda pozwala na filtrowanie wyników zgodnie z przekazanymi parametrami: Miasto, Tytuł filmu, Data seansu.

```
if (!string.IsNullOrEmpty(city))
{
    screeningsQuery = screeningsQuery.Where(s => s.Hall.Cinema.City ==
city);
}
if (!string.IsNullOrEmpty(movieTitle))
{
    screeningsQuery = screeningsQuery.Where(s =>
s.Movie.Title.Contains(movieTitle));
}
if (screeningDate.HasValue)
{
    screeningsQuery = screeningsQuery.Where(s => s.ScreeningDate.Date ==
screeningDate.Value.Date);
}
```

- **Zwracane dane**: Widok z przefiltrowaną listą seansów.

CategoriesController

Standardowy kontroler CRUD dla gatunkuch filmowych. Nie zaszły istotne zmiany w logice.

CinemasController

Standardowy kontroler CRUD dla kin. Nie zaszły istotne zmiany w logice.

HallsController

Standardowy kontroler CRUD dla sal kinowych. Nie zaszły istotne zmiany w logice.

AdminController

Index

- **HTTP**: `GET`
- **Parametry**: Brak.
- **Opis**: Wyświetla listę użytkowników oraz ich role.

CreateRole (GET)

- **HTTP**: `GET`
- **Parametry**: Brak.
- **Opis**: Wyświetla formularz dodawania nowej roli użytkownika.

CreateRole (POST)

- **HTTP**: `POST`
- **Parametry**: `string roleName` - Nazwa nowej roli.
- **Opis**:
  - Tworzy nową rolę w systemie, sprawdzając, czy już istnieje w bazie.
  - **Kod walidacji:**

    ```
    if (await _roleManager.RoleExistsAsync(roleName))
    {
        ModelState.AddModelError("", "Taka rola już istnieje.");
        return View();
    }
    ```

  - Dodaje role przy uzyciu:

    ```
    await _roleManager.CreateAsync(new IdentityRole(roleName));
    ```

- **Zwracane dane**: Przekierowanie do listy ról lub komunikat błedu.

EditRole (GET)

- **HTTP**: `GET`
- **Parametry**: `string id` - Identyfikator roli.
- **Opis**:
  - Pobiera aktualną rolę oraz użytkowników przypisanych do roli i wyświetla formularz edycji.
  - **Kod pobierania:**

```
var role = await _roleManager.FindByIdAsync(id);
var usersInRole = await _userManager.GetUsersInRoleAsync(role.Name);
```

EditRole (POST)

- **HTTP**: `POST`
- **Parametry**:
  - `string id` - Identyfikator roli.
  - `List<string> users` - Lista użytkowników do przypisania do roli.
- **Opis**:
  - Aktualizuje przypisanie użytkowników do roli.
  - **Kod aktualizacji:**

```
foreach (var user in users)
{
    if (!await _userManager.IsInRoleAsync(user, role.Name))
    {
        await _userManager.AddToRoleAsync(user, role.Name);
    }
}
```

DeleteRole

- **HTTP**: `POST`
- **Parametry**: `string id` - Identyfikator roli.
- **Opis**: Usuwa wybraną rolę, o ile nie jest przypisana do użytkowników.
- **Zwracane dane**: Przekierowanie do listy ról lub komunikat o błędzie.

---

8. System użytkowników

W projekcie wykorzystujemy **.NET Core Identity**, co daje nam obsługę funkcji takich jak rejestracja, logowanie czy zarządzanie rolami. Wybrane widoki zostały przetłumaczone na język polski. Zasadniczo nie wykonano żadnych większych modyfikacji w logice systemu użytkowników.

Główne komponenty systemu:

1. Role użytkowników

System obsługuje dwie podstawowe role:

- **Użytkownik (User):**
  - Może przeglądać seanse.
  - Może rezerwować miejsca.
  - Może przeglądać filmy.
  - Może przeglądać dostępne kina.
  - Może modyfikować swoje rezerwacje.
- **Administrator (Admin):**
  - Posiada wszystkie funkcje użytkownika oraz:
  - Może dodawać i edytować: kina, seanse, filmy, gatunki, sale, role użytkowników.
  - Posiada dostęp do panelu administracyjnego ze statystykami i listą użytkowników oraz ról.

2. Rejestracja i logowanie

- **Rejestracja:**
  - Formularz rejestracji wymaga podania adresu e-mail, hasła oraz potwierdzenia hasła.
  - Rejestracja nie wymaga potwierdzenia mailowego.