ASP.NET MVC

Celem ćwiczenia jest zapoznanie studenta z technologią ASP.NET MVC. Omówione zostaną poszczególne warstwy architektury MVC oraz podstawowe zagadnienia związane z konfiguracją, walidacją danych wprowadzanych przez użytkownika oraz komunikacją asynchroniczną.

1. Uruchom środowisko **Visual Studio** i stwórz nowy projekt aplikację webową w architekturze MVC. Obejrzyj wygenerowaną strukturę projektu.

2. Dodaj nowy kontroler klikając prawym przyciskiem myszy (PPM) na folderze **Controllers** i wybierając **Add -> Controller…**. Nadaj mu nazwę `SongsController` i wybierz szablon **Empty MVC Controller**.

3. Jak widzisz spowodowało to wygenerowanie nowej klasy dziedziczącej z klasy `Controller` zawierającej pojedynczą metodę `Index()`. Metody w kontrolerze to tzw. akcje i są one wywoływane przy pomocy żądań HTTP skierowanych pod odpowiednie adresy. Akcja na razie zawiera tylko jedną linijkę, w której zwraca wynik działania funkcji `View()`. Funkcja ta zwraca w odpowiedzi na żądanie stronę wygenerowaną na podstawie widoku odpowiadającemu danej akcji. Jest to jeden z wielu możliwych wyników działania akcji kontrolera. Do innych należą np. **Content** – dowolne dane; **Empty** – pusta odpowiedź; **File** – plik; **Json** – dane w formacie JSON; **JavaScript** – skrypt; różnego rodzaju przekierowania i błędy.

4. Zamień wywołanie funkcji `View` na `Content` i jako parametr podaj ciąg znaków `"Hello World!"`.

5. Uruchom aplikację i przejdź pod adres *<adres_witryny>/Songs*.

6. Wyświetl źródło strony. Jak widzisz faktycznie znajduje się tam jedynie przesłana treść i nic poza tym. Zaszło jeszcze jedno ciekawe zdarzenie. Otóż skąd wiadomo, że akurat taki adres wywołuje metodę `Index()` w utworzonym przez nas kontrolerze? Dzieje się tak dzięki domyślnej regule dodanej w pliku *RouteConfig.cs* znajdującym się w katalogu **App_Start**. Kolekcja `RouteCollection`, przekazana jako parametr metody `RegisterRoutes()`, służy do przechowywania reguł mapujących adresy na akcje kontrolerów. Metoda `MapRoute` dodaje pojedynczą regułę do kolekcji. Parametr `name` służy do podania nazwy reguły, przez parametr `url` przekazany jest wzorzec adresu, a parametr `defaults` zawiera przypisanie wartości domyślnych.

7. Dodaj do kontrolera `SongsController` nową metodę, która na żądanie *<adres_witryny>/Songs/Square/<id>* odpowie wartością parametru *<id>* podniesioną do kwadratu. Parametr przekaż jako argument funkcji reprezentującej akcję kontrolera.

8. Reguł mapujących może być oczywiście wiele i są one dopasowywane w kolejności ich dodania. Dodaj teraz regułę, która dla adresu witryny wywoła utworzoną przed chwilą akcję z kontrolera `SongsController` z parametrem `23`.

9. Uruchom aplikację i sprawdź czy reguła działa.

10. Zamień kolejność dodawania reguł. Ponownie przetestuj działanie strony. Przywróć poprawną kolejność dodawania reguł.

11. Przywróć poprzednią wersję akcji `Index()`, tak aby zwracała wynik działania funkcji `View()`, następnie uruchom aplikację i wywołaj tę akcję. Co obserwujesz?

12. Jak widzisz na serwerze nie odnaleziono widoku odpowiadającego danej akcji kontrolera. Zauważ jakie lokalizacje są przeszukiwane. Folder **Songs** jest przeznaczony dla widoków odpowiadających metodom kontrolera `SongsController`, natomiast folder **Shared** jest przeznaczony dla widoków wspólnych dla całej aplikacji.

13. Istnieje kilka sposobów na dodanie widoku do projektu. My skorzystamy z kreatora dostępnego przez kontroler. Pozostając kursorem wewnątrz akcji `Index()` kliknij PPM i wybierz z menu opcję **Add View…**. Pozostaw wszystkie opcje ustawione domyślnie i naciśnij przycisk **Add**.

14. Jak widzisz plik *Index.cshtml* został wygenerowany razem z folderem **Songs**. Obejrzyj wygenerowany kod. Jak widzisz nie ma go zbyt wiele… Wykorzystywany w skrypcie obiekt typu `dynamic` – `ViewBag` – jest podobny do zmiennej `Page`, omawianej przy okazji technologii ASP.NET Web Pages, która również tutaj występuje. Jedyna różnica jest taka, że zmienna `ViewBag` jest widoczna po stronie kontrolera, a `Page` nie. Zmienna `ViewBag` może zatem służyć do przekazywania danych pomiędzy kontrolerem a widokiem.

15. Uruchom aplikację i przejdź pod adres *<adres_witryny>/Songs*. Obejrzyj źródło strony.

16. Jak widzisz strona jest poprawnym dokumentem HTML5, mimo że w widoku zdefiniowaliśmy tylko nagłówek drugiego poziomu. Spróbuj to uzasadnić.

17. Sekret tkwi w pliku *_ViewStart.cshtml*. W ćwiczeniu dotyczącym technologii ASP.NET Web Pages mówiliśmy o specjalnym pliku *_PageStart.cshtml*, którego kod wykonywany był przed wykonaniem kodu każdej strony. Tutaj identyczną rolę pełni właśnie plik *_ViewStart.cshtml*. W tym pliku również wykorzystany jest znany już Tobie mechanizm układów, pozwalający zapewnić spójny wygląd całej aplikacji. Wyświetl plik układu.

18. Poza znanymi już Tobie funkcjami `RenderBody` oraz `RenderSection` znajdują się tutaj również wywołania metod `Styles.Render` oraz `Scripts.Render`. Sprawdź jaki kod generują te metody ponownie wyświetlając źródło strony generowane przez widok *Index.cshtml*. Zwróć uwagę, że ścieżki podane jako parametry tych metod tak naprawdę nie istnieją, a mimo to na stronie pojawiają się już właściwe ścieżki do odpowiednich plików. Odpowiedź na tę zagadkę znajduje się w pliku *BundleConfig.cs* znajdującym się w folderze **App_Start**. Wypełniany tutaj obiekt klasy `BundleCollection` definiuje paczki skryptów oraz styli. Dzięki temu na stronie wystarczy wskazać tylko paczkę, z której chcemy skorzystać, a wszystkie wchodzące w niej skład pliki zostaną odpowiednio dołączone. Dodatkowo, w trybie produkcyjnym, wszystkie pliki wchodzące w skład pojedynczej paczki są łączone w jeden i zmniejszane poprzez usunięcie nadmiarowych znaków oraz komentarzy. Aby zaobserwować ten efekt zmień w pliku *Web.config* wartość atrybutu `debug` na `false` po czym ponownie wyświetl źródło strony.

19. Pozostaje jeszcze jedna niewyjaśniona kwestia. Kiedy wywoływane są omówione czynności rejestrujące mapowania ścieżek oraz paczki skryptów? Odpowiedź tkwi w pliku *Global.asax.cs*. Znajduje się tam metoda `Application_Start`, która (jak zapewne się domyślasz) wywoływana jest w momencie uruchomienia aplikacji na serwerze. Jak widzisz to w tym miejscu wywoływane są wszystkie akcje konfiguracyjne. Wrócimy jeszcze do omówienia pozostałych akcji.

20. Mamy już podstawowy kontroler oraz widok. Dodamy teraz model. Istnieje kilka sposobów aby to zrobić: możemy najpierw utworzyć schemat bazy danych i na jego podstawie wygenerować schemat modelu w aplikacji; można utworzyć schemat modelu w aplikacji i na tej podstawie wygenerować schemat bazy danych. My skorzystamy z kolejnej opcji – napiszemy potrzebne nam klasy, a następnie na ich podstawie wygenerujemy schemat bazy danych.

21. Kliknij PMM na folderze **Models** i dodaj nowy plik z klasą o nazwie `Song`. Klasa będzie reprezentowała utwór muzyczny, dodaj więc trzy właściwości typu `string`: `Name`, `Artist`, `Genre` oraz dodatkowe pole `Id` typu `int`, które posłuży jako identyfikator.

22. Na razie pobawimy się modelem bez wykorzystania bazy danych. Przejdź teraz do kontrolera utworów i w metodzie `Index()` stwórz nowy obiekt klasy `Song` i uzupełnij jego pola.

23. Podaj następnie utworzony obiekt jako parametr metody `View()`. Spowoduje to przekazanie obiektu danej klasy jako modelu dla widoku. Aby jednak móc skorzystać z tego modelu należy zdefiniować silnie typowany widok.

24. Aby zdefiniować silnie typowany widok wystarczy skorzystać w widoku z klauzuli `@model`, po której należy podać klasę, która ma posłużyć jako model.

25. Jako pierwszą linijkę w widoku *Index.cshtml* wpisz `@model ASP.NET_MVC.Models.Song`. Zbuduj projekt.

26. Teraz, poprzez zmienną `Model` w widoku, masz dostęp do przekazanego z kontrolera obiektu modelu. Podmień zawartość nagłówka drugiego stopnia na nazwę przekazanego utworu.

27. Poznaliśmy już dwie metody przekazywania danych z kontrolera do widoku: poprzez model oraz zmienną `ViewBag`. Dodamy teraz komunikację z bazą danych. Dodaj do folderu **Models** nową klasę o nazwie `MusicDbContext` dziedziczącą z klasy `DbContext`.

28. Jedyne co teraz musimy zrobić, to zadeklarować kolekcję `DbSet` obiektów typu `Song`. Dodaj do klasy poniższą linijkę kodu.

```
public DbSet<Song> Songs { get; set; }
```

29. To wystarczy! Przy pierwszym odwołaniu do modelu zostanie automatycznie wygenerowany schemat bazy danych odpowiadający wszystkim kolekcjom `DbSet` znajdującym się w klasach dziedziczących z `DbContext`. W naszym przypadku będzie to na razie tylko jedna tabelka. Pytanie brzmi: gdzie zostanie ona utworzona? Odpowiedź na to pytanie można odnaleźć w pliku *Web.config*. Znajduje się tam domyślny ciąg połączenia z bazą danych o nazwie `DefaultConnection`. Zostanie on domyślnie wykorzystany do utworzenia połączenia z bazą danych. Zamień wpis dodający ciąg połączenia na poniższy.

```
<add name="DefaultConnection" providerName="System.Data.SqlClient" connectionString="Data
Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=MusicDb;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\MusicDb.mdf" />
```

30. Możesz również jawnie wskazać ciąg połączenia, z którego aplikacja ma skorzystać, przekazując jego nazwę w parametrze konstruktora klasy `DbContext`. Dodaj bezparametrowy konstruktor do klasy `MusicDbContext` i wywołaj konstruktor nadklasy przekazując jako parametr nazwę ciągu połączenia `DefaultConnection`.

```
public MusicDbContext() : base("DefaultConnection") { }
```

31. Dodamy teraz w naszej aplikacji logikę zarządzania utworami. Usuń widok *Index.cshtml* z folderu **Songs** oraz kontroler `SongsController`. Dodamy go raz jeszcze, tym razem skorzystamy jednak z gotowego szablonu. Kliknij PPM na folderze **Controllers** i wybierz **Add -> Controller…**. W oknie kreatora ponownie podaj nazwę `SongsController`, tym razem jednak wybierz szablon

z kontrolerem i widokami. Wskaż klasę modelu oraz kontekstu połączenia z bazą danych i naciśnij **Add**.

32. Zmodyfikuj reguły mapujące adresy URL w taki sposób, aby domyślnie wywoływana była akcja **Index** kontrolera **Songs**.

33. Uruchom aplikację, przetestuj jej działanie, dodaj kilka ulubionych utworów.

34. Jak widzisz mamy kompletną aplikację realizującą podstawowe czynności CRUD! Przeanalizujemy teraz wygenerowany kod. Zacznijmy od kontrolera. Jak widzisz niektóre akcje występują podwójnie, np. akcja `Create`. Przeznaczeniem bezparametrowej wersji akcji `Create` jest jedynie wyświetlenie pustego widoku *Create.cshtml*, który umożliwi wprowadzenie informacji o utworze i przesłanie ich do kontrolera. Sparametryzowana akcja `Create` przyjmuje obiekt klasy `Song`.

35. Wyświetl źródło strony generowanej przez widok *Create.cshtml*. Jak widzisz pola formularza mają takie same nazwy jak pola klasy `Song`. Framework na tej podstawie uzupełnia pola obiektu przekazanego jako parametr akcji. Pole `ModelState.IsValid` zawiera informację o poprawności przesłanych danych (o tym później). Dalej następuje jedynie dodanie przekazanego utworu do kolekcji i utrwalenie danych w bazie.

36. Jak widzisz w przypadku powodzenia zapisu akcja zwraca przekierowanie do akcji `Index`. W przeciwnym przypadku ponownie wyświetlany jest widok z przesłanymi danymi.

37. Należy tutaj jeszcze zwrócić uwagę na jedną bardzo istotną kwestię. Sparametryzowana akcja `Create` posiada adnotację `HttpPost`, która powoduje, że metodę tę można wykonać jedynie korzystając z metody POST. Można w taki sposób wskazać dowolne metody protokołu HTTP.

38. Jak widzisz analogicznie wyglądają wszystkie pozostałe akcje kontrolera. Znajdująca się na końcu klasy metoda `Dispose` jest odpowiedzialna za zwolnienie zasobów, w tym wypadku zamknięcie połączenia z bazą danych.

39. Teraz przejdziemy do analizy widoków. Zacznijmy od *Create.cshtml*. Jak widzisz widok jest silnie typowany klasą `Song`. Nagminnie wykorzystywany jest tutaj również helper `Html`, który w połączeniu z silnym typowaniem jest bardzo istotnym narzędziem znacznie ułatwiającym tworzenie widoków! Zastosowanie konstrukcji `…For` pozwala w bardzo łatwy sposób powiązać model z poszczególnymi polami. Wyświetl źródło strony aby zobaczyć w jaki sposób wyniki zastosowanych funkcji tłumaczone są na elementy HTML.

40. Obejrzyj widoki *Edit.cshtml* oraz *Delete.cshtml*. Jak widzisz wszystkie wykorzystują te same mechanizmy oraz ten sam schemat budowy.

41. Przejdź teraz do widoku *Index.cshtml*. Różnica pomiędzy tym widokiem a pozostałymi jest taka, że tutaj model to nie klasa `Song`, tylko kolekcja obiektów tej klasy.

42. Zmodyfikujemy teraz trochę wygenerowany szkielet. Pierwszą zmianą, którą wprowadzimy, będzie zmiana operacji usuwania utworów. W aktualnej postaci jest na to przeznaczony osobny widok. My chcemy, żeby czynność ta odbywała się asynchronicznie wyświetlając jedynie komunikat z prośbą o potwierdzenie.

43. Najpierw stworzymy coś w stylu kontrolki, która będzie odpowiedzialna za wyświetlanie samej listy utworów. Kliknij PPM na folderze **Songs** i wybierz **Add -> View…**. Nazwij widok *_SongsList*, wybierz pusty szablon i zaznacz opcję **Create as a partial view**. Wytnij całą tabelkę z widoku *Index* i wklej do kontrolki. Popraw również klauzulę modelu.

44. Teraz wystarczy wstawić kontrolkę na stronę widoku. Aby to osiągnąć skorzystamy z funkcji `Partial` helpera `Html`. Dodaj poniższą linijkę do pliku *Index.cshtml*.

```
@Html.Partial("_SongList", Model)
```

45. Uruchom aplikację.

46. Jak widzisz nic się nie zmieniło i dokładnie o to chodziło☺ Dodamy teraz wspomniane już wcześniej usuwanie asynchroniczne. Najpierw pobierz dodatkową bibliotekę do komunikacji asynchronicznej. W tym celu, uruchom konsolę **NuGet Package Manager** i wykonaj polecenie:

```
Install-Package Microsoft.jQuery.Unobtrusive.Ajax
```

    Następnie, w pliku *BundleConfig.cs* zamień wpis

```
bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                                        "~/Scripts/jquery.validate*"));
```

    na

```
bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                                        "~/Scripts/jquery.unobtrusive*",
                                        "~/Scripts/jquery.validate*"));
```

    Na końcu, w pliku układu strony wstaw następującą linijkę.

```
@Scripts.Render("~/bundles/jqueryval")
```

    Dodaje ona skrypty umożliwiające wywołania asynchroniczne. Ponieważ kolejność ładowania skryptów ma znaczenie, dodaj ją po wszystkich pozostałych.

47. Najpierw zajmiemy się kontrolerem. Usuń akcje `Delete` i `Details`, a następnie zmień nazwę akcji `DeleteCompleted` na `Delete`. Ponieważ chcemy być zgodni z protokołem HTTP akcja ma być możliwa do wywołania jedynie metodą DELETE protokołu HTTP. OK! To wystarczy!

48. Przejdź teraz do edycji kontrolki i usuń przyciski służące do usuwania i wyświetlania szczegółów utworów. Skorzystamy z funkcji `ActionLink` helpera `Ajax`. W wykorzystanej przez nas wersji funkcja przyjmuje cztery parametry: nazwę odnośnika, nazwę wywoływanej w kontrolerze akcji, parametry akcji oraz opcje wywołania asynchronicznego. Wstaw w miejsce odnośnika poniższy kod.

```
@Ajax.ActionLink("Delete", "Delete", new { id = item.Id }, new AjaxOptions()
    {
        HttpMethod = "Delete",
        Confirm = "Are you sure you want to delete this song?",
        UpdateTargetId = "songs"
    })
```

49. Parametr `UpdateTargetId` pozwala wskazać kontrolkę, do której ma zostać załadowana odpowiedź na wysłane żądanie. My wskazujemy na kontrolkę o identyfikatorze `songs`. Przypisz ten identyfikator do elementu `<table>`.

50. Uruchom aplikację i przetestuj usuwanie utworów.

51. Hmm… No chyba nie do końca o to chodziło. Czy potrafisz uzasadnić działanie strony?

52. Dzieje się tak, ponieważ akcja usunięcia powoduje przekierowanie do akcji `Index`, która z kolei zwraca cały odpowiadający jej widok. My jednak chcemy przeładować tylko kontrolkę. Aby to osiągnąć lekko zmodyfikujemy akcję `Index`. Zamiast zawsze zwracać widok wykonamy następujące czynności:
    - Sprawdź, czy akcja została wywołana asynchronicznie (`Request.IsAjaxRequest()`)
    - Jeśli tak, zwróć tylko widok częściowy (kontrolkę) korzystając z funkcji `PartialView`, przekazując jej jako parametry nazwę kontrolki oraz model.
    - W przeciwnym wypadku zwracamy widok z modelem, tak jak we wcześniejszej wersji funkcji.

53. Ponownie uruchom i przetestuj aplikację.

54. Dodamy teraz kolejną klasę modelu, która będzie przechowywała gatunki muzyczne, tak aby przy tworzeniu nowego utworu możliwy był wybór gatunku z rozwijanej listy.

55. Dodaj do modelu nową klasę `Genre`. Klasa ma jedynie przechowywać identyfikator `Id`, nazwę gatunku `Name` oraz kolekcję (`ICollection`) piosenek `Songs`. Zmień również pole `Genre` klasy `Song` na `GenreId` typu `int`. Utrzymanie takiej konwencji spowoduje automatyczne dodanie klucza obcego w bazie.

56. Wygeneruj dla utworzonej klasy kontroler wraz ze wszystkimi widokami. Analogicznie jak w przypadku utworów zamień przycisk usuwający na wywołanie asynchroniczne bez potwierdzenia.

57. Uruchom aplikację.

58. Jak widzisz coś jest nie tak. Przyczyną błędu jest fakt, że zmieniliśmy model. Wypadałoby więc również przegenerować bazę, jednak może to skutkować usunięciem danych, więc narzędzie samo nie podejmie się takiego działania. Aby zarządzać wersjami modelu w ASP.NET MVC 4 należy skorzystać z migracji. My jednak na potrzeby naszego zadania będziemy przy każdej zmianie modelu tworzyli cały schemat na nowo. Aby to osiągnąć dodaj do funkcji `Application_Start` poniższą linijkę.

```
Database.SetInitializer<MusicDbContext>(new
DropCreateDatabaseIfModelChanges<MusicDbContext>());
```

59. Uruchom aplikację i dodaj kilka ulubionych gatunków muzycznych.

60. Musimy teraz naprawić widok dodawania i edycji utworów. Chcemy, aby w miejscu pola tekstowego dla gatunku pojawiła się lista wyboru. Aby to osiągnąć musimy przede wszystkim jakoś przekazać listę z kontrolera do widoku. Znamy dwie metody, które to umożliwiają. Pierwszą opcją jest przekazanie dodatkowych danych przez model. Czasami stosuje się to rozwiązanie tworząc tzw. modele widoków (**ViewModel**), czyli specjalne klasy modelu, które zawierają wszystkie pola niezbędne do przechowania danych zasilających widok. Obiekt takiej klasy tworzy się w kontrolerze na podstawie modeli z bazy oraz innych ewentualnych danych i przekazuje jako model widoku. My jednak skorzystamy z drugiej, prostszej w tym przypadku opcji – ze zmiennej `ViewBag`.

61. Przejdź do kontrolera utworów i dodaj do obu akcji `Create` poniższy fragment kodu.

```
ViewBag.Genres = db.Genres.ToList();
```

pole do wprowadzania gatunku na poniższy kod.

```
@Html.DropDownListFor(
    model => model.Genre,
    ((IEnumerable<ASP.NET_MVC.Models.Genre>)ViewBag.Genres)
        .Select(x => new SelectListItem() { Text = x.Name, Value = x.Id.ToString() }),
    "Select a genre"
)
```

63. Pierwszy parametr to wskazanie dla jakiego pola ma zostać utworzone pole wyboru. Trzeci parametr to tekst zerowego elementu. W drugim parametrze przekazujemy elementy zasilające listę. Funkcja `Select`, analogicznie jak w języku SQL, pozwala dokonać projekcji. W tym przypadku zamieniamy listę elementów typu `Genre` na listę elementów typu `SelectListItem`, które zasilają pole wyboru.

64. Funkcja `Select` jest zaledwie wierzchołkiem góry lodowej, jaką jest język LINQ. Poza wskazaną dokumentacją bardzo przydanym źródłem informacji o tym języku jest również strona 101 LINQ Samples.

65. Uruchom aplikację. Przetestuj dodawanie gatunków muzycznych oraz utworów.

66. Analogicznie jak w przypadku dodawania nowych utworów wprowadź zmianę do edycji.

67. Ostatnim zagadnieniem które poruszymy jest walidacja danych. Jak zapewne zauważyłeś w metodach zapisu danych po stronie kontrolera automatycznie dodany został warunek `if (ModelState.IsValid)`. Pojawia się jednak pytanie – kiedy model jest poprawny? Poprawność modelu można zdefiniować w… modelu☺ Do tego celu wykorzystuje się adnotacje.

68. Otwórz plik modelu utworu. Dodaj do pola `Name` poniższe adnotacje

```
[Required(ErrorMessage = "Name is required!")]
[StringLength(100, ErrorMessage = "Maximal length of the name of a song is 100 characters!")]
```

69. Dodaj stosowne reguły do pola `Artist` raz pola `Name` w klasie `Genre`.

70. Co istotne, adnotacje nie wpływają jedynie na walidację danych, ale również wpływają na definicje samej tabeli w bazie.

71. Uruchom i przetestuj aplikację pod kątem walidacji.

72. To już jest koniec. Jeśli zainteresowała Cię omawiana technologia polecam doskonały darmowy kurs wideo dostępny pod tym adresem, który porusza omówiony tutaj materiał znacznie szerzej. Do nieporuszonych w tym tutorialu zagadnień należą przede wszystkim: bezpieczeństwo, zaawansowane zagadnienia konfiguracyjne, publikowanie aplikacji, testowanie, niestandardowa walidacja, filtry i selektory akcji, alternatywne metody generacji modelu oraz język zapytań LINQ.

    Poniżej znajduje się jeszcze kilka zadań dodatkowych, do których wykonania oczywiście zachęcam☺

Zadania dodatkowe

1. Na liście utworów wyświetlany jest identyfikator gatunku, jednak o wiele lepsze byłoby wyświetlenie jego nazwy. Wprowadź stosowną modyfikację.

2. Wypadałoby również dla gatunku utworu wyświetlać etykietę `Genre`, a nie `GenreId`. Skorzystaj z adnotacji `Display` w modelu.

3. Spróbuj dodać w widoku szczegółów gatunku listę należących do niego utworów.