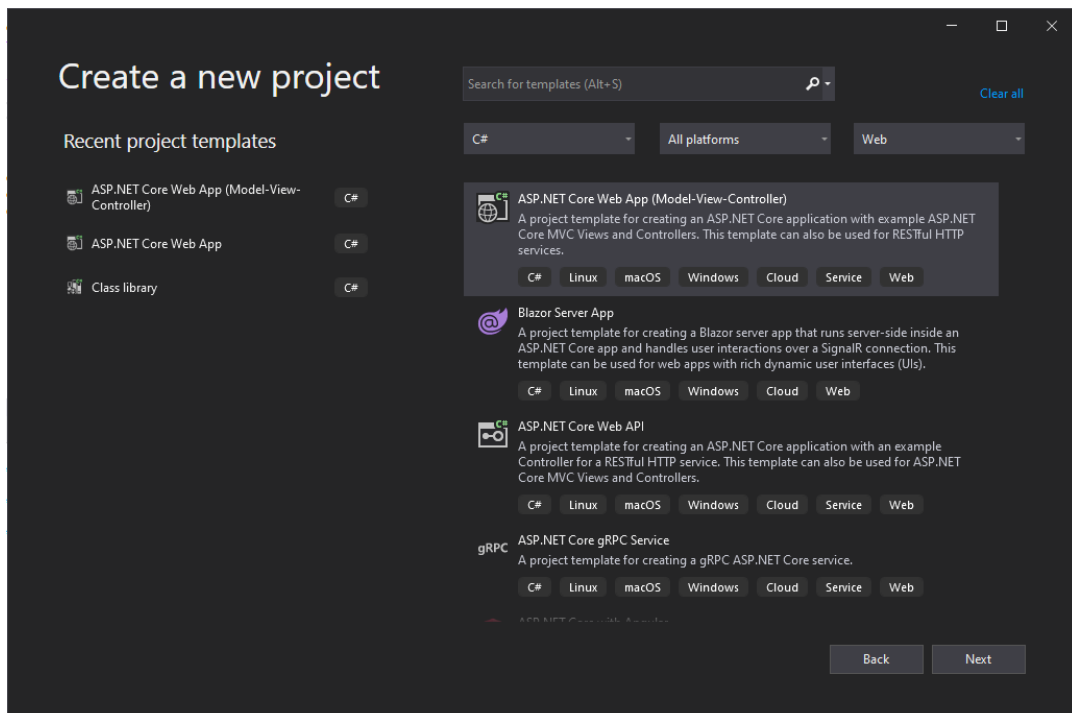




Student Development Program

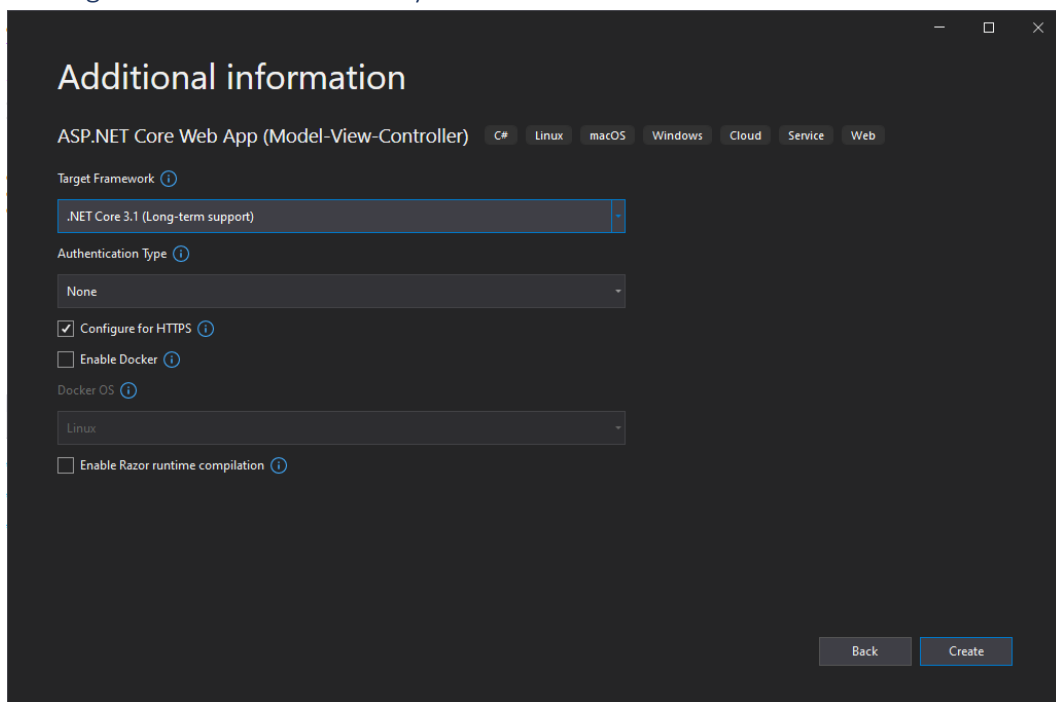
ASP.NET Core MVC

1. Należy utworzyć nowy projekt korzystając z szablonu ASP.NET Core Web App (Model-View-Controller)

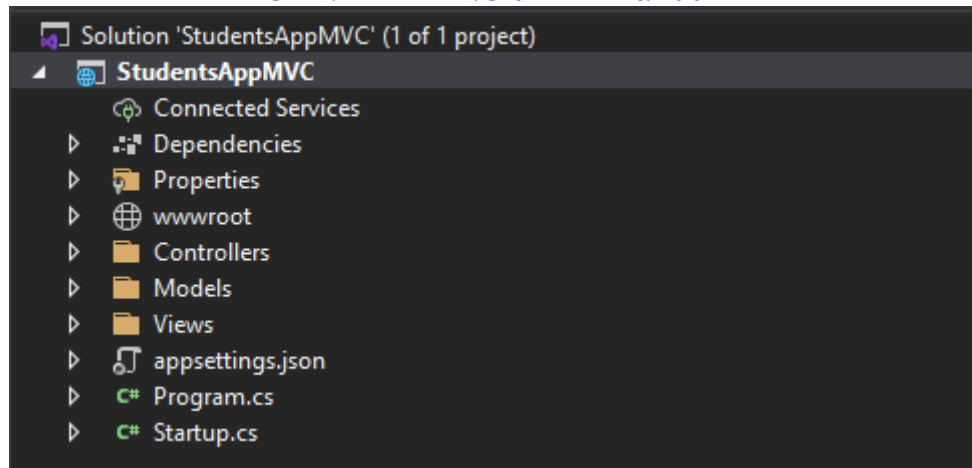


Projekt nazywamy *StudentsAppMVC*

2. Jako Target Framework ustawiamy .NET Core 3.1



3. Startowa struktura katalogów powinna wyglądać następująco

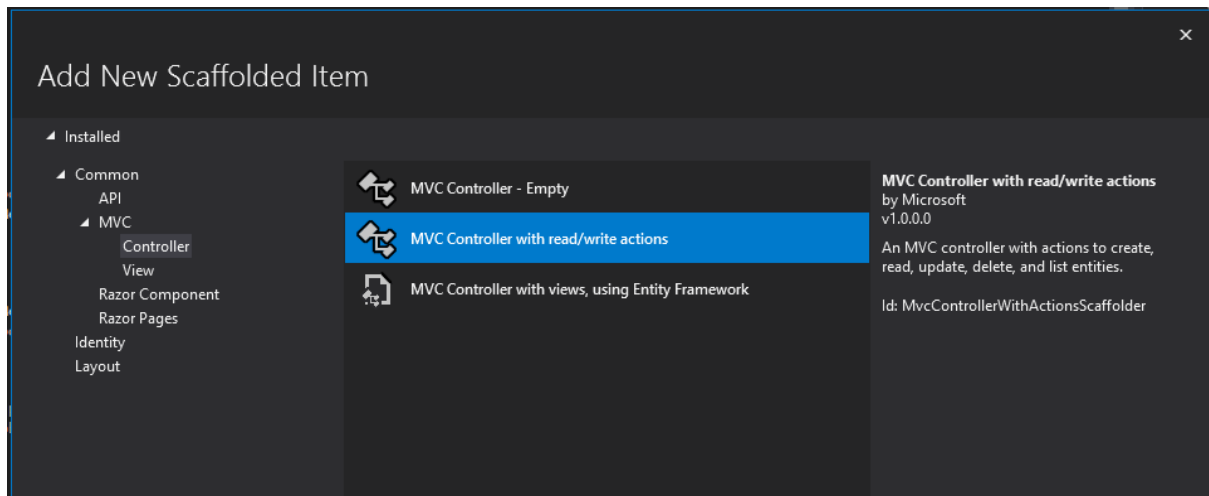


4. Usuwamy domyślnie utworzone pliki, nie będziemy ich używać.
- Plik HomeController.cs z katalogu Controllers
 - Plik ErrorViewModel.cs z katalogu Models
 - Katalog Home z katalogu Views
 - Plik Error.cshtml z katalogu Shared
5. W folderze Models dodajemy nową klasę StudentModel. Uzupełniamy następującymi polami:

```
public int StudentId { get; set; }  
public string Name { get; set; }  
public string LastName { get; set; }  
public string Email { get; set; }  
public int Age { get; set; }  
public bool IsActive { get; set; }
```

6. Następnie w katalogu Controllers dodajemy nową klasę kontrolera

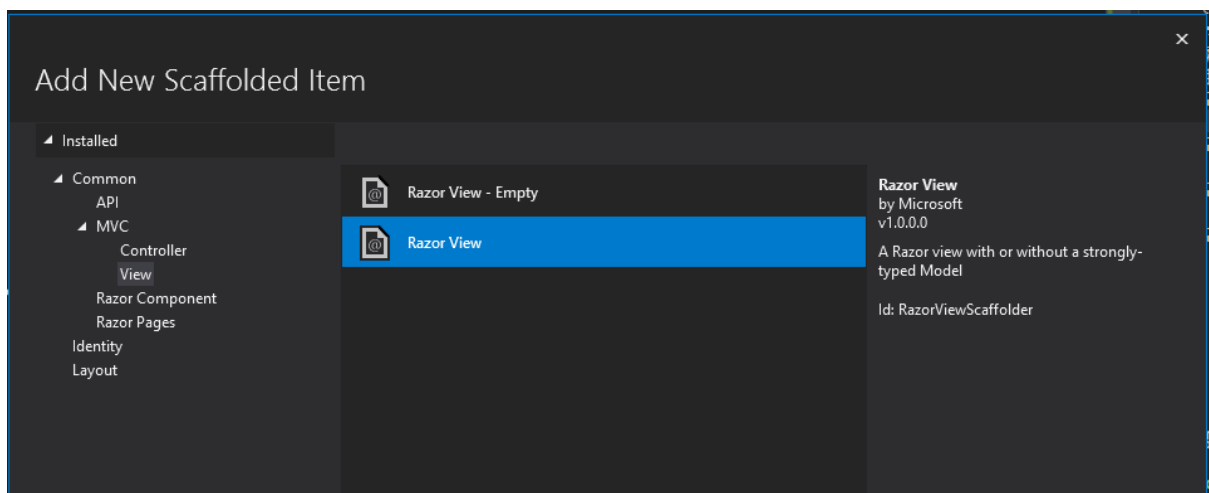
- Prawy przycisk myszy na katalogu Controllers -> Add -> Controller.



- Wybieramy opcję MVC Controller with read/write actions.
- Nowy plik nazywamy StudentController.

7. Dodajemy nasz pierwszy widok

- Otwieramy nowo utworzony kontroler. Widzimy, że Visual Studio wykonało za nas dużo pracy. Wygenerowane zostały podstawowe operacje do pracy na naszym modelu danych. Oczywiście brakuje implementacji, zajmiemy się tym później.
- Znajdujemy metodę Index, klikamy na jej nazwie prawym przyciskiem myszy -> Add View. Wybieramy opcję Razor View.



- Po przejściu dalej *View name* pozostawiamy jako domyślne *Index*, musi on odpowiadać metodzie z kontrolera.
- Możemy wybrać szablon, jaki ma zostać użyty do wygenerowania naszej strony. Strona główna będzie zawierała listę studentów, wybieramy *List*.
- Model wejściowy dla naszej strony głównej to lista studentów. Wybieramy StudentModel.

- Nie działa, dlaczego?

8. Zmieniamy domyślny routing.

- Routing ustawiony w metodzie Startup.cs -> Configure wciąż wskazuje na usunięte przez nas pliki. Domyślny kontroler ustawiamy jako Student, akcję jako Index.

```

53     app.UseEndpoints(endpoints =>
54     {
55         endpoints.MapControllerRoute(
56             name: "default",
57             pattern: "{controller=Student}/{action=Index}/{id?}");
58     });

```

9. Dodajemy tymczasową listę studentów.

- Docelowo utworzymy tabelę w bazie danych i to z nią się będziemy komunikować. W pierwszym kroku w klasie StudentController dodajmy ręcznie listę studentów.

```

private static IList<StudentModel> students = new List<StudentModel>()
{
    new StudentModel(){ StudentId = 1, Name = "Anna", LastName = "Nowak", Age = 19, Email = "test0@wp.pl", IsActive = true },
    new StudentModel(){ StudentId = 2, Name = "Ula", LastName = "Rak", Age = 21, Email = "test1@wp.pl", IsActive = false },
    new StudentModel(){ StudentId = 3, Name = "Ola", LastName = "Kos", Age = 29, Email = "test3@wp.pl", IsActive = false }
};

```

10. Przekazujemy listę studentów do widoku zwracanego w metodzie Index i uruchamiamy aplikację.

```

21     public ActionResult Index()
22     {
23         return View(students);
24     }
25
26     // GET: StudentController/Details/5
27     public ActionResult Details(int id)
28     {
29         return View();
30     }

```

- Nasz dotychczasowy postęp powinien wyglądać następująco:

StudentId	Name	LastName	Email	Age	IsActive	
1	Anna	Nowak	test0@wp.pl	19	<input checked="" type="checkbox"/>	Edit Details Delete
2	Ula	Rak	test1@wp.pl	21	<input type="checkbox"/>	Edit Details Delete
3	Ola	Kos	test3@wp.pl	29	<input type="checkbox"/>	Edit Details Delete

© 2021 - StudentsAppMVC1 - [Privacy](#)

11. Planowanie kolejnych kroków

Docelowo nasza aplikacja będzie się składać z 4 stron:

- Strona główna – tabela wyświetlająca imię, nazwisko oraz status aktywności wszystkich studentów w postaci tabeli.
- Strona dodawania nowego studenta – umożliwia podanie imienia, nazwiska, e-mail, wieku oraz statusu aktywności.
- Strona Edycji – wyświetla imię, nazwisko, e-mail oraz wiek danego studenta. Umożliwia edycję tych danych.
- Strona wyświetlania szczegółowych danych – wyświetla wszystkie dane danej osoby, umożliwia przełączenie statusu aktywności.
- Strona usuwania danych studenta – wyświetla wszystkie dane danej osoby, umożliwia ich usunięcie.

12. Dostosowywanie strony głównej – Index.cshtml

- Otwieramy plik `Views\Student\Index.cshtml`, z definicji tabeli usuwamy zbędne definicje kolumn (zgodnie z punktem 11)
- Nasza aplikacja będzie wyświetlana w języku polskim. Nazwy kolumn w naszej tabeli możemy zmienić dodając adnotację w modelu danych `StudentModel`. Przykładowo:

```
[DisplayName("Imię")]
public string Name { get; set; }
```

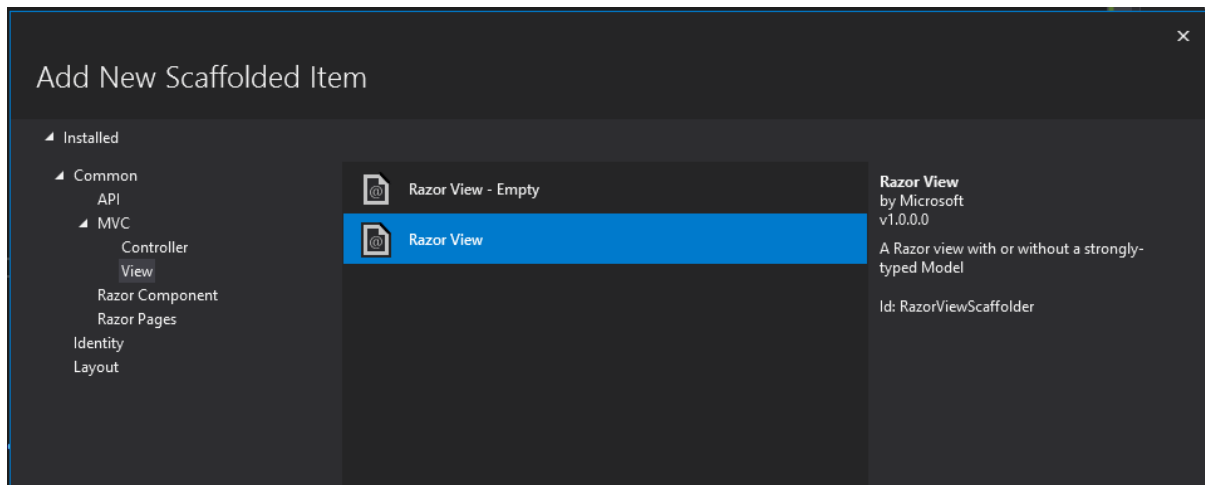
Index

[Create New](#)

Imię	Nazwisko	Aktywny	
Anna	Nowak	<input checked="" type="checkbox"/>	Edit Details Delete
Ula	Rak	<input type="checkbox"/>	Edit Details Delete
Ola	Kos	<input type="checkbox"/>	Edit Details Delete

13. Tworzenie strony dodawania nowego studenta – Create.cshtml

- Definicje metod GET Create oraz POST Create są już dostępne w naszym kontrolerze StudentController.
- GET Create będzie odpowiadać za pobranie danych wejściowych dla strony dodania studenta.
- POST Create będzie przechwytywać podane dane oraz dodawać nowego studenta do listy.
- Dodajemy plik Create.cshtml: Prawy przycisk myszy na nazwie metody GET Create -> Add View



- Jako template używamy Create

Add Razor View

View name:

Template:

Model class:

Options

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

- Otwieramy plik Create.cshtml. Dostosowujemy dane jakie może podać użytkownik zgodnie z punktem 11 – usuwamy możliwość podania ID.

14. Dodawanie walidacji

- Warunki walidacji definiujemy w modelu. Podobnie jak już to zrobiliśmy w przypadku nazw kolumn w tabeli głównej. Wszystkie pola są wymagane, imię i nazwisko musi mieć odpowiednia długość:

```
[DisplayName("Imię")]
[Required(ErrorMessage = "Pole imię jest obowiązkowe")]
[StringLength(20, MinimumLength = 2, ErrorMessage = "Długość imienia musi być w zakresie 2 - 20")]
public string Name { get; set; }
```

- E-mail musi mieć odpowiednią formę

```
[EmailAddress(ErrorMessage = "Proszę podać poprawny adres email")]
[Required(ErrorMessage = "Pole e-mail jest obowiązkowe")]
[DisplayName("Email")]
public string Email { get; set; }
```

- Wiek powinien być liczbą w danych zakresie

```
[Range(6, 100, ErrorMessage = "Proszę podać wiek w zakresie 6 - 100")]
[Required(ErrorMessage = "Musisz podać wiek")]
public int Age { get; set; }
```


Dodaj nowego studenta

Imię

Pole imię jest obowiązkowe

Nazwisko

Pole imię jest obowiązkowe

E-mail

Pole e-mail jest obowiązkowe

Age

Musisz podać wiek

☐ Aktywny

[Powrót do listy studentów](#)

15. Dodanie przekazanych danych nowego studenta do listy

Przechodzimy do kontrolera.

- Metoda GET Create – nie potrzebujemy argumentów, do zwracanego widoku możemy przekazać pusty obiekt StudentModel.

```
// GET: StudentController/Create
public ActionResult Create()
{
    return View(new StudentModel());
}
```

- Metoda POST Create – jako argument przyjmuje obiekt StudentModel, zawierający dane podane przez użytkownika, dodaje je do listy, przekierowuje nasz do strony głównej.

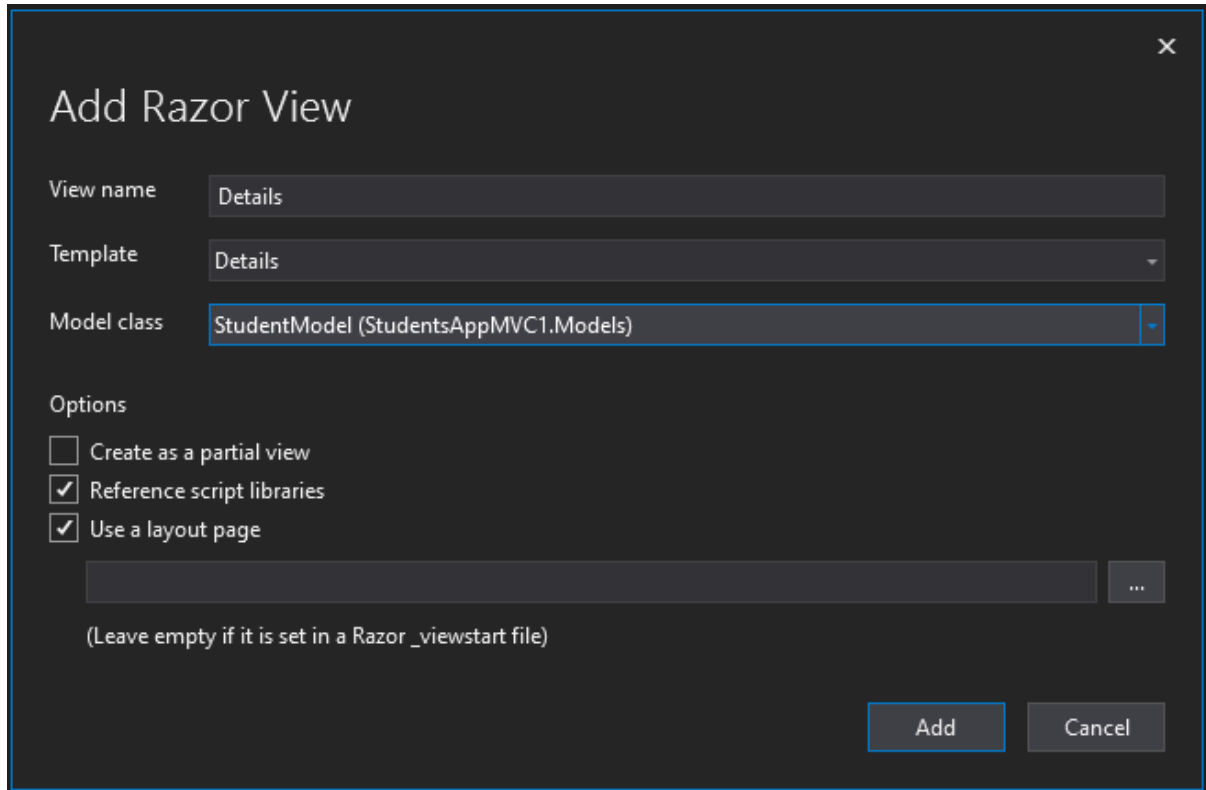
```
// POST: StudentController/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(StudentModel studentModel)
{
    studentModel.StudentId = students.Count + 1;
    students.Add(studentModel);

    return RedirectToAction(nameof(Index));
}
```

- Docelowo połączymy się z bazą danych i przydzielanie kolejnego ID dla nowego studenta będzie się odbywało automatycznie, chwilowo zrobimy to „ręcznie”.

16. Tworzenie strony wyświetlającej dane szczegółowe – Details.cshtml.

- Deklaracja metody GET Details jest już dostępna w StudentController. Jako że tylko wyświetlamy dane, metoda POST jest zbędna i nie została wygenerowana.
- Dodajemy nowy widok Details: prawy przycisk myszy na nazwie metody GET Details.
- Jako szablon wybieramy *Details*



- Aktualizujemy działanie metody GET Details. Jako argument ma przyjmować ID studenta, którego dane chcemy wyświetlić. Na podstawie przekazanego ID pobiera dane z listy wszystkich studentów i przekazuje je do widoku.

```
// GET: StudentController/Details/5
public ActionResult Details(int id)
{
    var specificStudent = students.FirstOrDefault(x => x.StudentId == id);
    return View(specificStudent);
}
```

- Nie działa, dlaczego?

- Przechodzimy do pliku Index.cshtml. Id Studenta musi zostać najpierw przekazane.

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.IsActive)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
            @Html.ActionLink("Szczegóły studenta", "Details", new { id=item.StudentId }) || 
            @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
        </td>
    </tr>
}
```

17. Usuwanie danych studenta

- W kontrolerze znajdujemy metodę Delete -> prawy przycisk myszy i Add View. Jako szablon wybieramy Delete.

Add Razor View

View name: Delete

Template: Delete

Model class: StudentModel (StudentsAppMVC.Models)

Options

- ☐ Create as a partial view
- ☒ Reference script libraries
- ☒ Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

- Przechodzimy do kontrolera
- Metoda GET Delete – przyjmuje Id studenta, do widoku przekazuje model zawierający dane studenta o danym Id.

```
public ActionResult Delete(int id)
{
    var studentToDelete = students.FirstOrDefault(x => x.StudentId == id);
    return View(studentToDelete);
}
```

- Metoda POST Delete – przyjmuje Id studenta oraz model. Usuwa dane studenta z listy wszystkich studentów.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(int id, StudentModel studentModel)
{
    var studentToDelete = students.FirstOrDefault(x => x.StudentId == id);
    students.Remove(studentToDelete);
    return RedirectToAction(nameof(Index));
}
```

18. Tworzenie własnej akcji

- Naturalnie możemy tworzyć własne metody akcji. Użyjemy tego to zrealizowania funkcjonalności przełączania statusu Active dla studenta na stronie Details.cshtml
- W kontrolerze tworzymy metodę ChangeActiveStatus.
- Przyjmuje ona jako parametr Id studenta oraz ustawia stan flagi Active na odwrotny do obecnego.
- Po wykonaniu operacji przekierowuje nas z powrotem na stronę szczegółów dla danego studenta.

```
public ActionResult ChangeActiveStatus(int id)
{
    var specificStudent = students.FirstOrDefault(x => x.StudentId == id);
    specificStudent.IsActive = !specificStudent.IsActive;

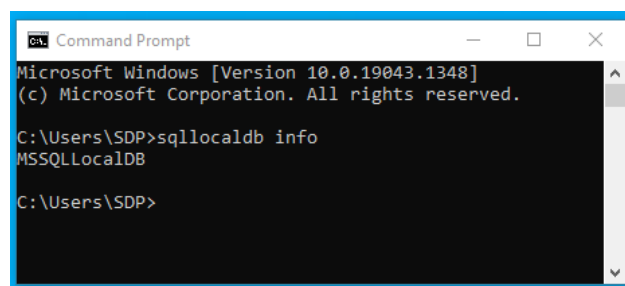
    return RedirectToAction("Details", "Student", new { id = specificStudent.StudentId });
}
```

- W pliku Details.cshtml obok przycisku Edit dodajemy przycisk odwołujący się do metody ChangeActiveStatus.

```
<div>
    @Html.ActionLink("Edytuj", "Edit", new { /* id = Model.PrimaryKey */ }) |
    @Html.ActionLink("Zmień status aktywności", "ChangeActiveStatus", new { id = Model.StudentId }) |
    <a asp-action="Index">Wróć do listy studentów</a>
</div>
```

19. Utworzenie bazy danych

- Do tej pory nasze dane przechowywaliśmy w pamięci, zajmiemy się teraz utworzeniem bazy danych i podłączeniem do niej naszej aplikacji.
- Używając w wierszu poleceń komendy sqllocaldb info możemy sprawdzić listę dostępnych instancji serwera SQL na naszej maszynie. Domyślnie dostępna powinna być MSSQLLocalDB.

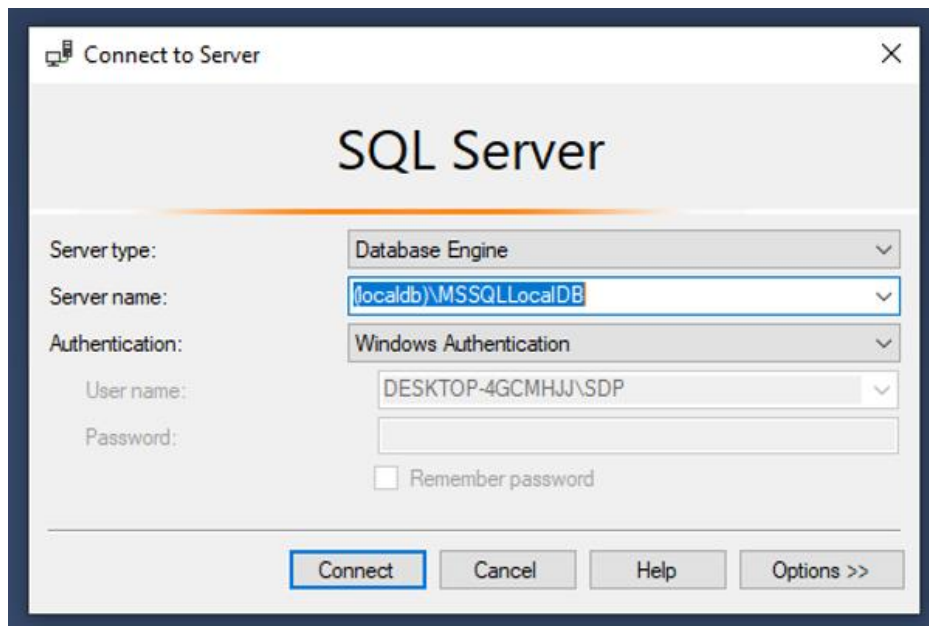


```
ca Command Prompt
Microsoft Windows [Version 10.0.19043.1348]
(c) Microsoft Corporation. All rights reserved.

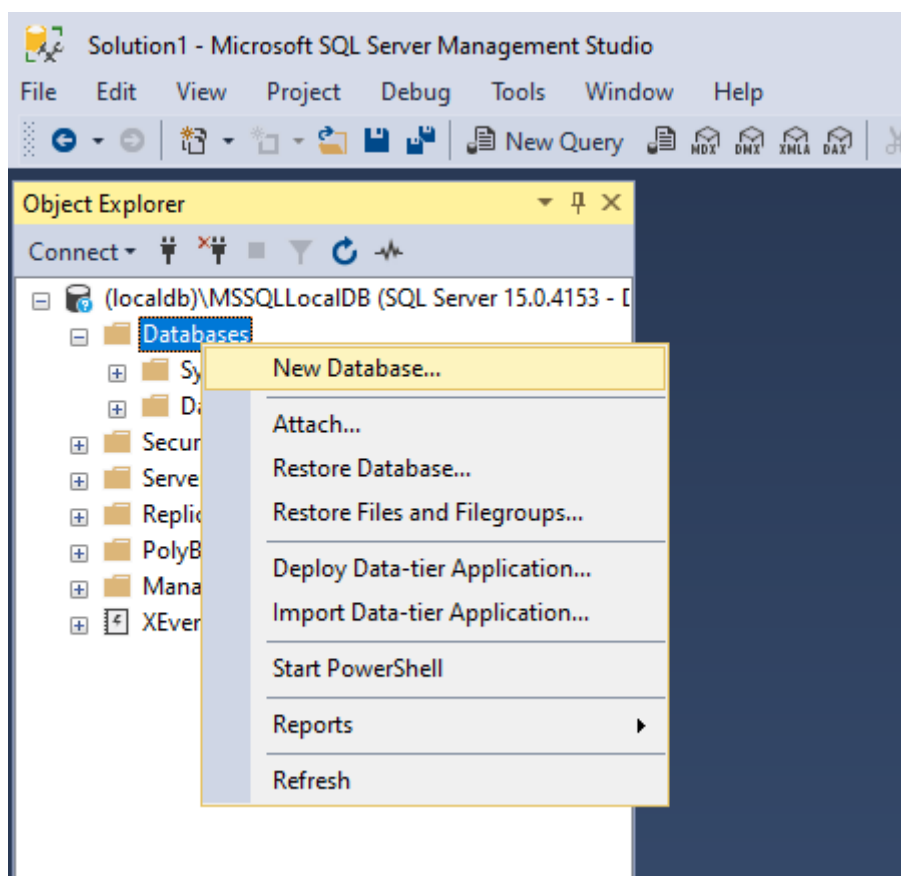
C:\Users\SDP>sqllocaldb info
MSSQLLocalDB

C:\Users\SDP>
```

- Połączmy się do niej za pomocą narzędzia MSSQL Server Management Studio
- Jako Server name podajemy (localdb)\<nazwa_naszej_instancji>
- Jako Authentication – Windows Authentication



- Dodajemy nową bazę danych, nazywamy ją StudentsDB



- Pobieramy plik CreateTableStudents.sql z repozytorium <https://github.com/SDP2021Sharp/MVC>
- Klikamy prawym przyciskiem myszy na bazę danych StudentsDB -> New Query
- Uruchamiamy skrypt z pliku CreateTableStudents.sql
- W bazie danych StudentsDB, w katalogu Tables pojawiła się nowa tabela Students z przykładowymi danymi trzech osób.

The screenshot shows the SQL Server Enterprise Manager interface. On the left, the Object Explorer displays the hierarchy of the StudentsDB database, with the 'dbo.Students' table highlighted. On the right, a SQL query window shows a script for the 'SelectTopNRows' command. The script is as follows:

```

/***** Script for SelectTopNRows command from SSMS *****/
SELECT TOP (1000) [StudentId]
, [Name]
, [LastName]
, [Age]
, [Email]
, [IsActive]
FROM [StudentsDB].[dbo].[Students]

```

Below the script, the 'Results' tab shows the data returned by the query:

	StudentId	Name	LastName	Age	Email	IsActive
1	1	Jan	Kowalski	18	test1@test.pl	1
2	2	Anna	Nowak	19	test2@test.pl	0
3	3	Maciej	Malinowski	20	test3@test.pl	0

20. Dodanie niezbędnych pakietów

- Pierwsze co musimy zrobić, to zainstalować dwa niezbędne pakiety NuGet. Korzystamy z .NET Core 3.1. Instalując je, proszę wybrać obecnie najwyższą wersję 3.1.x
- Microsoft.EntityFrameworkCore

The screenshot shows the NuGet Package Manager window for the 'StudentsAppMVC1' project. The search results for 'Microsoft.EntityFrameworkCore' are displayed. The package list includes:

- Microsoft.EntityFrameworkCore.SqlServer** (6.0.0) - Microsoft SQL Server database provider for Entity Framework Core.
- Microsoft.EntityFrameworkCore** (6.0.0) - Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server...
- Microsoft.EntityFrameworkCore.Sqlite** (6.0.0) - SQLite database provider for Entity Framework Core.
- Microsoft.EntityFrameworkCore.Sqlite.Core** (6.0.0) - SQLite database provider for Entity Framework Core. This package does not include a copy of the native SQLite library.
- Microsoft.EntityFrameworkCore.SqlServer.Design** (1.1.6) - Design-time Entity Framework Core Functionality for Microsoft SQL Server.
- Microsoft.EntityFrameworkCore.Sqlite.Design** (1.1.6) - Design-time Entity Framework Core Functionality for Microsoft SQLite.

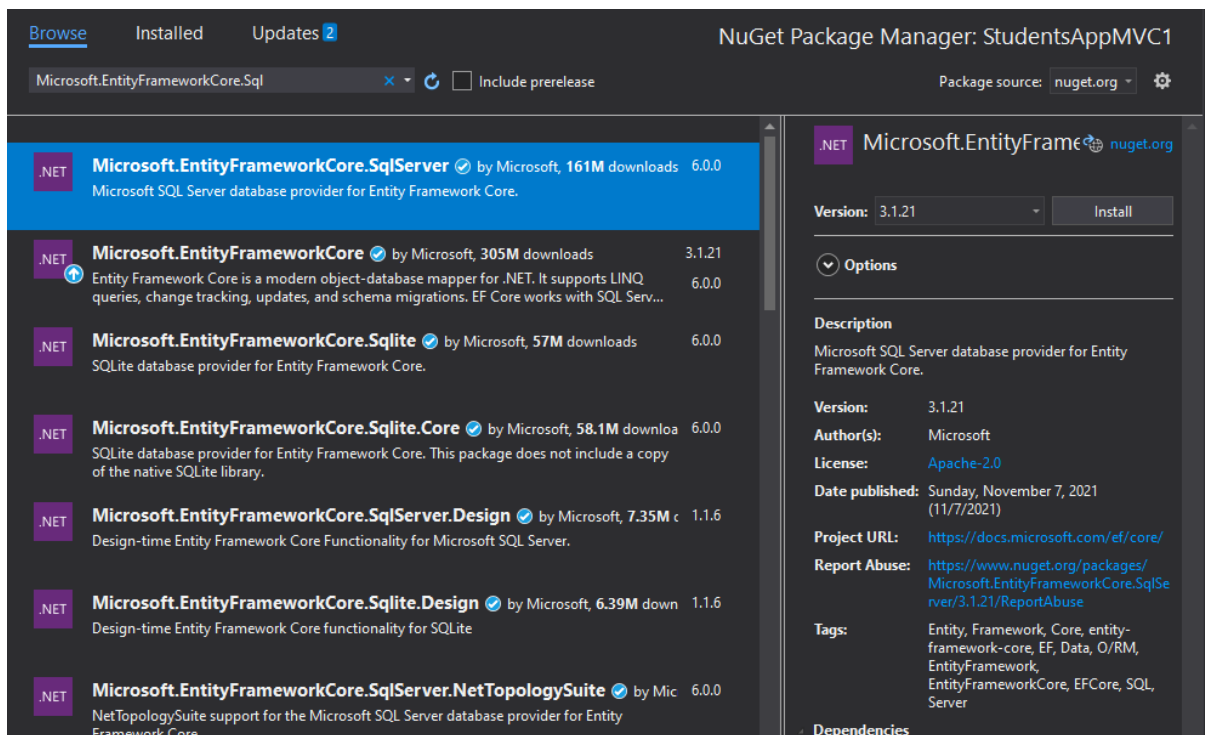
The right-hand pane shows the details for the selected package, **Microsoft.EntityFrameworkCore** (version 3.1.21). It includes the 'Install' button, 'Options', 'Description', and 'Commonly Used Types'.

Description: Entity Framework Core is a lightweight and extensible version of the popular Entity Framework data access technology.

Commonly Used Types: Microsoft.EntityFrameworkCore.DbContext, Microsoft.EntityFrameworkCore.DbSet

Version: 3.1.21
Author(s): Microsoft
License: Apache-2.0
Date published: Sunday, November 7, 2021 (11/7/2021)

- Microsoft.EntityFrameworkCore.SqlServer



21. Dodanie klasy kontekstu

- Do projektu dodajemy katalog *Data* a w nim nowy plik *MvcStudentContext*. Dziedziczy ona po klasie *Microsoft.EntityFrameworkCore.DbContext* i reprezentuje połączenie z bazą danych.

```
public class MvcStudentsContext : DbContext
{
    public MvcStudentsContext(DbContextOptions<MvcStudentsContext> options) : base(options)
    {
    }

    public DbSet<StudentModel> Students { get; set; }
}
```

22. Dodanie ConnectionString do pliku appsettings.json

- Otwieramy plik *appsettings.json* i dodajemy do niego wpis

```
"ConnectionStrings": {
  "StudentsDBConnectionString":
    "Server=(localdb)\\MSSQLLocalDB;Database=StudentsDB;Trusted_Connection=True"
}
```

Podczas rejestrowania naszego kontekstu użyjemy *StudentsDBConnectionString* aby określić:

- Instancję SQL Serwera do którego będziemy się łączyć (*Server=(localdb)\\MSSQLLocalDB*),
- Bazę danych (*Database=StudentsDB*),
- Zaufanie połączenie (*Trusted_Connection=True*), co sprawi, że użyjemy uwierzytelnienia systemu Windows, tak samo jak w przypadku logowania się do MSSQL Server Management Studio.

23. Zarejestrowanie kontekstu bazy danych w kontenerze serwisów

- Przechodzimy do klasy Startup, znajdujemy metodę ConfigureServices i rejestrujemy wcześniej utworzony MvcStudentsContext w konterze zależności naszej aplikacji

```
services.AddDbContext<MvcStudentsContext>(options =>  
    options.UseSqlServer(Configuration.GetConnectionString("StudentsDBConnectionString")));
```

- Od teraz będziemy mogli wstrzyknąć w konstruktorze kontrolera nasz DbContext

24. Użycie kontekstu w kontrolerze

- Przechodzimy do klasy StudentController, tworzymy konstruktor i wstrzykujemy do niego zależność MvcStudentsContext.

```
private readonly MvcStudentsContext _dbContext;  
  
public StudentController(MvcStudentsContext dbContext)  
{  
    _dbContext = dbContext;  
}
```

- _dbContext zawiera DbSet Students (public DbSet<StudentModel> Students { get; set; }) będzie on zawierał stan tabeli Students z bazy danych.
- Aby odczytać dane, musimy wykonać jeszcze jeden krok: przechodzimy do klasy modelu StudentModel i dodajemy atrybuty
- Do klasy StudentModel dodajemy [Table("Students")] aby skonfigurować do jakiej tabeli w bazie danych ma się odwoływać. Do pola StudentId dodajemy atrybut [Key] aby wskazać klucz główny.

```
[Table("Students")]  
24 references  
public class StudentModel  
{  
    [Key]  
    [DisplayName("ID Studenta")]  
    19 references  
    public int StudentId { get; set; }  
}
```

- Uruchamiamy aplikację, w tabeli głównej powinniśmy zobaczyć dane odczytane z bazy danych.

25. Użycie DbContextu do modyfikacji danych.

- Modyfikujemy metodę POST Create. DbSet Students w kontekście _dbContext reprezentuje stan tabeli Students w bazie danych na czas startu aplikacji, chwili w której kontekst został utworzony.
- Możemy ten stan tabeli zmodyfikować zwyczajnie dodając model z danymi nowego studenta a następnie kontekst zapisać, co sprawi, że zmiany zostaną wysłane do bazy danych.

```
[HttpPost]  
[ValidateAntiForgeryToken]  
public ActionResult Create(StudentModel studentModel)  
{  
    _dbContext.Students.Add(studentModel);  
    _dbContext.SaveChanges();  
  
    return RedirectToAction(nameof(Index));  
}
```


- W ramach ćwiczenia proszę zmienić działanie pozostałych metod kontrolera tak aby używały komunikacji z bazą danych.

26. Zadanie domowe

- Proszę utworzyć branch w repozytorium <https://github.com/SDP2021Sharp/MVC>
- Format sdp_mvc_imie_nazwisko na przykład sdp_mvc_krzysztof_krywiak
- Proszę umożliwić edycję danych studenta
 - Dodanie nowego widoku, używając szablonu *Edit*
 - Wyświetlamy imię, nazwisko, e-mail oraz wiek danego studenta.
 - Metoda GET Edit przyjmuje Id studenta
 - Metoda POST Edit przyjmuje Id studenta, obiekt modelu zawierający nowe dane dla studenta o podanym Id
 - Wywołanie okna edycji ma być możliwe z głównej tabeli Index.cshtml oraz z poziomu okna wyświetlającego szczegóły studenta Details.cshtml
- Czas na wykonanie: 1 tydzień

W razie pytań można się komunikować bezpośrednio na GitHub lub wysyłając wiadomość na krzysztof.krywiak@globallogic.com