**Разработка приложения ASP.NET Core, осуществляющего взаимодействие с базой данных, с помощью Visual Studio 2019**

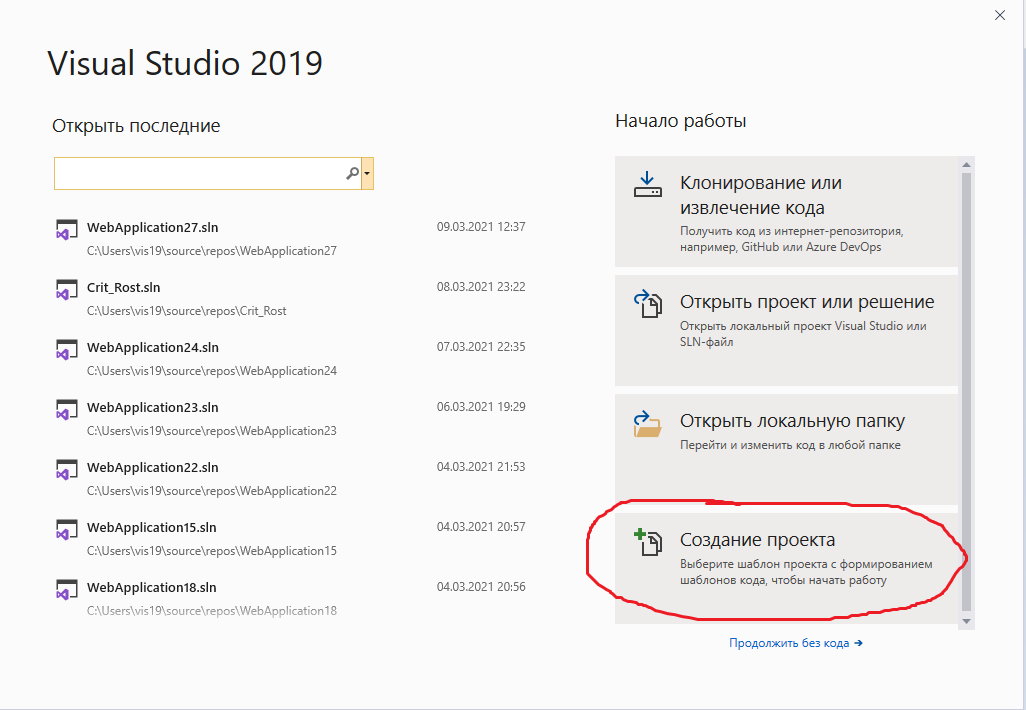
*Иллюстрационные материалы к спецкурсу*

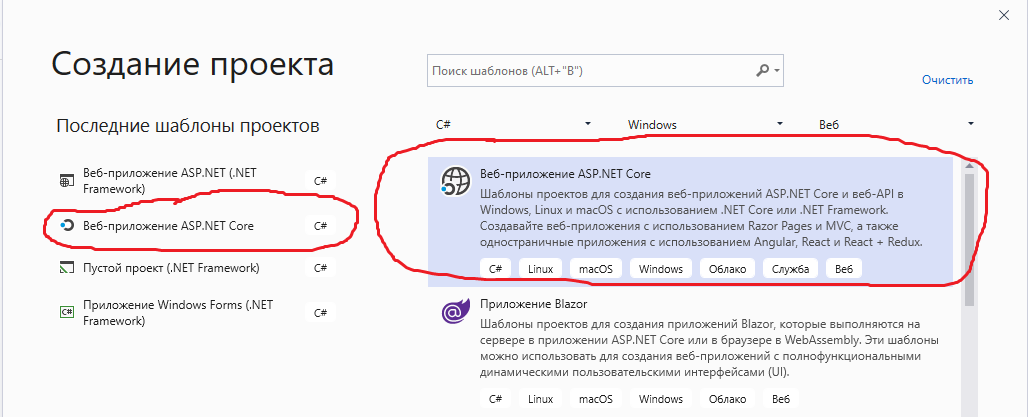
*Доцент Войтешенко И. С., кафедра ТП ФПМИ БГУ*

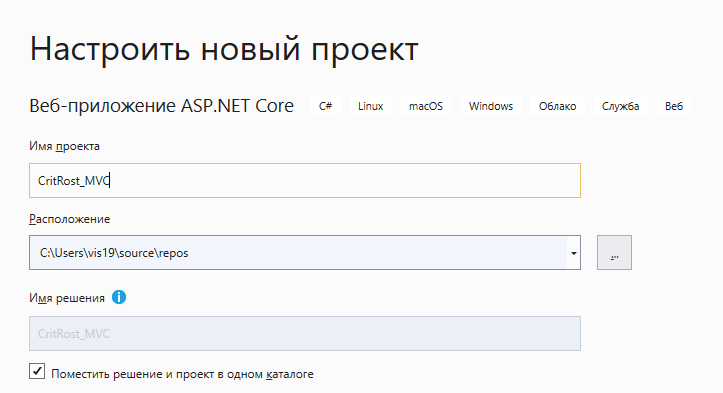
**Задача**. Разработать веб-серверное приложение, позволяющее удаленно работать с БД Critters. Использовать технологию ASP.NET Core, Visual Studio 2019.

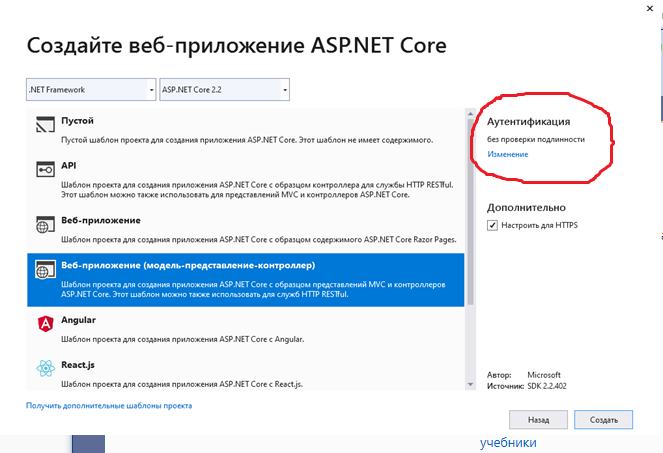
**Решение**.

1. Создадим проект ASP.NET Core

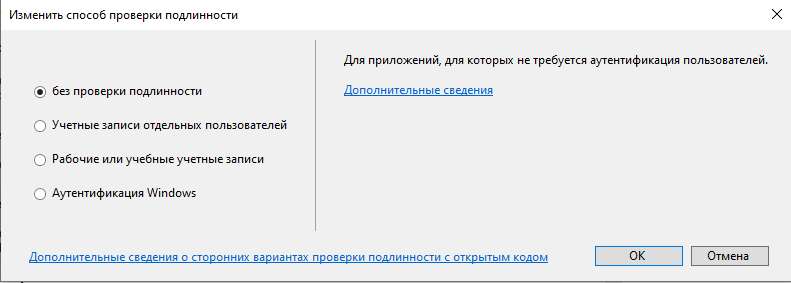




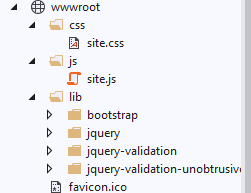
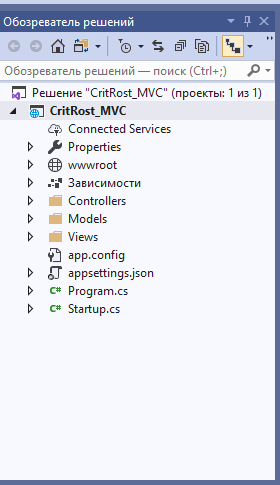




Можно было нажать *Изменить* и выбрать способ аутентификации:

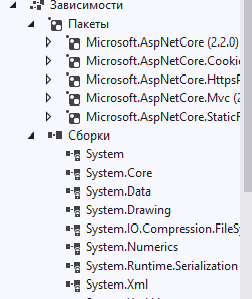


Итак, приложение создано. *Обозреватель решений* показывает нам шаблон созданного приложения:



Папка wwwroot содержит статический контент: CSS-файлы стилей оформления, изображения, JavaScript-сценарии и файл favicon.ico (значок веб-сайта или веб-страницы).

Папка зависимости содержит указание пакеты ASP.NET Core и сборки (пространства имен) ASP.NET Framework, используемые в проекте:



Каталог *Data* включает классы Entity Framework Core, используемые системой ASP.NET Identity для обеспечения аутентификации и авторизации (в нашем проекте пока этого каталога).

Каталог *Models* включает классы C#, представляющие все данные, необходимые для ответа на HTTP-запрос.

Каталог *Controllers* содержит классы C# с методами (известными как действия), которые осуществляют выборку модели и передачу ее представлению.

В каталог *Views* входят CSHTML-файлы, которые объединяют код на языках HTML и C#, реализующий динамическое генерирование HTML-ответа.

Файл *Program.cs* представляет собой консольное приложение, содержащее точку входа *public static void Main* и выполняющее начальную конфигурацию, компиляцию и запуск веб-приложения.

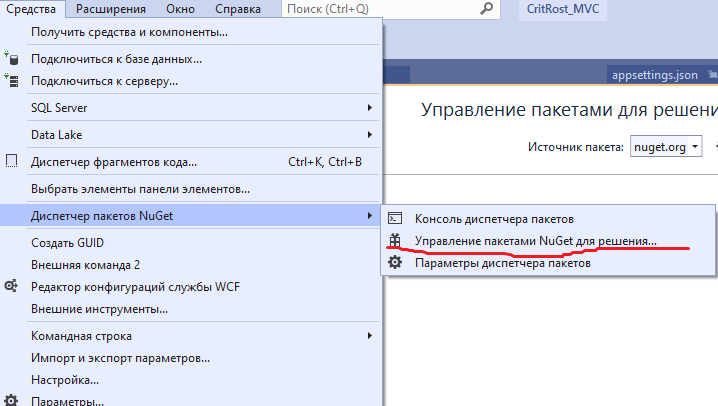
Дополнительный файл *Startup.cs* содержит конфигурационные данные для дополнительной настройки сервисов, к примеру ASP.NET Identity для аутентификации, а также связи между компонентами в вашем приложении.

В файле appsettings.json доступны настройки, которые веб-приложение может загружать во время выполнения, к примеру строка подключения базы данных для системы ASP.NET Identity.

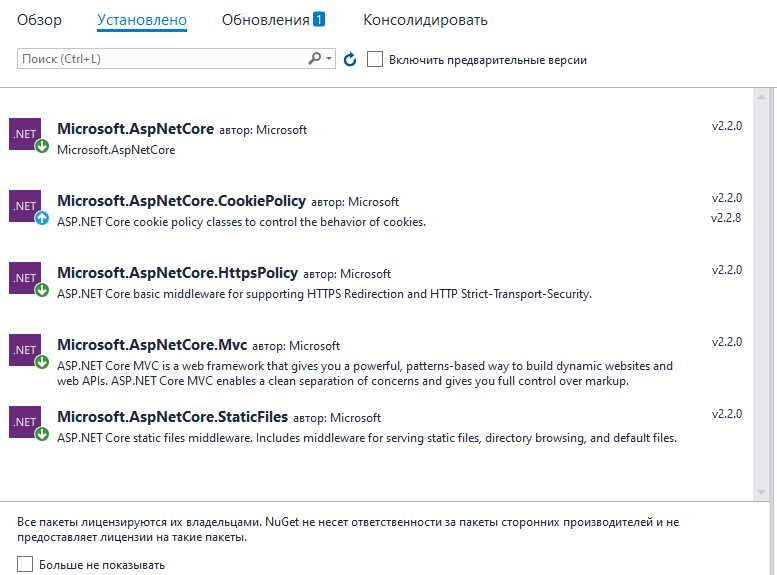
2. Entity Framework

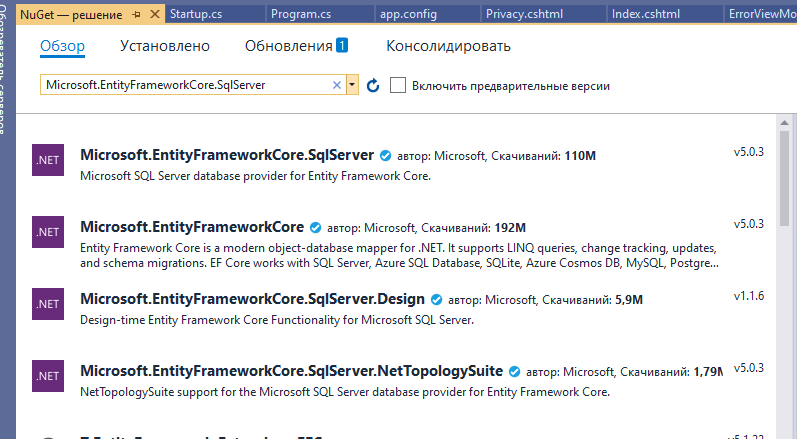
Entity Framework позволяет автоматически связать обычные классы языка C# с таблицами в базе данных. Entity Framework Core нацелен в первую очередь на работу с СУБД MS SQL Server, однако поддерживает также и ряд других СУБД. В данном случае мы будем работать с базами данных в MS SQL Server. Будет использоваться Entity Framework Core - кроссплатформенное решение на базе .NET Core, которое отличается от предыдущих версий, например, от Entity Framework 6.

Для взаимодействия с MS SQL Server через Entity Framework необходим пакет Microsoft.EntityFrameworkCore.SqlServer. По умолчанию он отсутствует в проекте, поэтому его надо добавить, например, через пакетный менеджер Nuget:

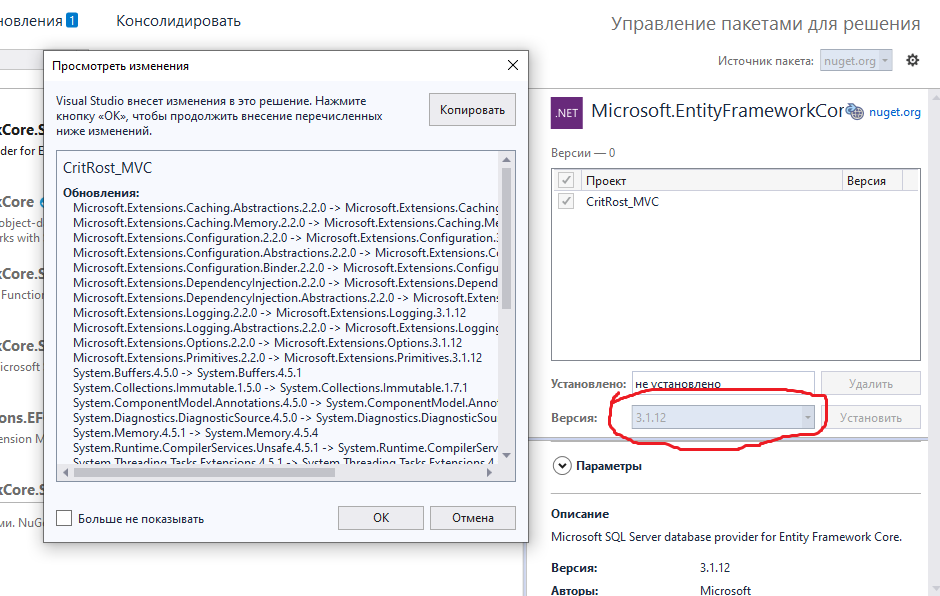


Видим, что среди установленных пакетов требуемого пакета нет. Перейдем на вкладку *Обзор* и найдем требуемый пакет.

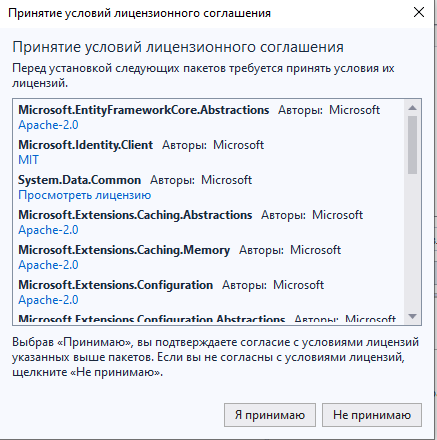




Для моей версии .Net Core не подошли версии пакета 5.\*.\*, поэтому установили версию 3.12:



Принимаем лицензионное соглашение:



Далее добавим в проект класс модели, с которой мы будем работать. По умолчанию в проекте уже есть предназначенная для моделей папка Models. Определим в этой папке определим новый класс roster. Эта модель будет представлять те данные, которые будут храниться в базе данных.

**using** System.ComponentModel.DataAnnotations;

**namespace** **App.Models**

{

**public** **class** **roster**

{

[Key]

**public** **string** playerid { **get**; **set**; }

**public** **string** jersey { **get**; **set**; }

**public** **string** fname { **get**; **set**; }

**public** **string** sname { **get**; **set**; }

**public** **string** position { **get**; **set**; }

**public** **DateTime** birthday { **get**; **set**; }

**public** **string** weight { **get**; **set**; }

**public** **string** height { **get**; **set**; }

**public** **string** birthcity { **get**; **set**; }

**public** **string** birthstate { **get**; **set**; }

**public** **int** ClubId { **get**; **set**; }

}

}

Поместим этот класс в файл roster.cs папки Models. Заметим, как обозначено ключевое поле.

Аналогично, можем создать и другие модели, например:

using System.ComponentModel.DataAnnotations;

namespace App.Models

{

public class Club

{

[Key]

public int ClubId { get; set; }

public string Name { get; set; }

public string Country { get; set; }

}

}

Поместим этот класс в файл clubs.cs папки Models.

*Замечание о System.ComponentModel.DataAnnotations*.

Для валидации вводимых данных фреймворк .NET предлагает гораздо удобный функционал в виде атрибутов из пространства имен System.ComponentModel.DataAnnotations. Этот пакет тоже надо установить через через пакетный менеджер Nuget, так же, как мы устанавливали пакет Microsoft.EntityFrameworkCore.SqlServer. Установилась самая последняя доступная версия пакета, 5.0.0.

Пример валидации:

using System.ComponentModel.DataAnnotations;

namespace HelloApp

{

    public class User

    {

        [Required]

        public string Id { get; set; }

        [Required]

        [StringLength(50, MinimumLength=3)]

        public string Name { get; set; }

        [Required]

        [Range(1, 100)]

        public int Age { get; set; }

    }

}

С учетом возможностей пространства имен *System.ComponentModel.DataAnnotations*, валидацию в классе *roster* можно было бы выполнить более профессионально, например, так:

public class roster

{

[Key]

[Description("Player ID")]

[Required(ErrorMessage = "Please enter player ID")]

public string PlayerId { get; set; }

[Required(ErrorMessage = "Please enter jersey")]

[Description("Jersey")]

public int? Jersey { get; set; }

[Required(ErrorMessage = "Please enter First Name")]

[Description("First Name")]

public string FName { get; set; }

[Required(ErrorMessage = "Please enter Second Name")]

[Description("Second Name")]

public string SName { get; set; }

[Required(ErrorMessage = "Please enter player position")]

[RegularExpression("[a-zA-Z]{2}", ErrorMessage ="Position consists of 2 letters") ]

[Description("Position")]

public string Position { get; set; }

[Required(ErrorMessage = "Please enter player birthday")]

[CustomDate(ErrorMessage ="Birth date should be less than 50 years ago")]

[Description("Birthday")]

public DateTime? Birthday { get; set; }

[Required(ErrorMessage = "Please enter player weight")]

[Range(30, 200, ErrorMessage ="Weight must be in range 30-200 kg")]

[Description("Weight")]

public int? Weight { get; set; }

[Required(ErrorMessage = "Please enter player height")]

[Range(140, 230, ErrorMessage = "Height must be in range 140-230 cm")]

[Description("Height")]

public int? Height { get; set; }

[Required(ErrorMessage = "Please enter player birth city")]

[Description("Birth City")]

public string BirthCity { get; set; }

[Required(ErrorMessage = "Please enter player birth state")]

[RegularExpression("[a-zA-Z]{2}", ErrorMessage = "Birth state consists of 2 letters")]

[Description("Birth State")]

public string BirthState { get; set; }

}

}

Напомним, что int? в C# означает, что переменной данного типа можно присваивать значение null, аналогично для других типов со знаком «?».

Чтобы взаимодействовать с базой данных через Entity Framework, нужен “контекст данных” - класс, унаследованный от класса Microsoft.EntityFrameworkCore.DbContext. Поэтому добавим в папку Models новый класс, который назовем Context (название класса контекста произвольное):

using Microsoft.EntityFrameworkCore;

namespace App.Models

{

public sealed class Context : DbContext

{

public Context(DbContextOptions<Context> options)

: base(options)

{

Database.EnsureCreated(); // создаем базу данных при первом обращении

}

public DbSet<roster> roster { get; set; }

public DbSet<Club> Clubs { get; set; }

}

}

Свойство DbSet представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных.

Через параметр options в конструктор контекста данных будут передаваться настройки контекста.

В конструкторе с помощью вызова Database.EnsureCreated() по определению моделей будет создаваться база данных (если она отсутствует).

Чтобы подключаться к базе данных, нам надо задать параметры подключения. Для этого изменим файл appsettings.json, добавив в него определение строки подключения.

Например

{

  "ConnectionStrings": {

    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=mobilesdb;Trusted\_Connection=True;"

  },

  // остальное содержимое файла

}

В данном случае используется упрощенный движок базы данных LocalDB, который представляет легковесную версию SQL Server Express, предназначенную специально для разработки приложений.

Следующим шагом в настройке проекта является изменение файла Startup.cs. В нем нам надо изменить метод ConfigureServices().

using App.Models;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace App

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

string connection = Configuration.GetConnectionString("DefaultConnection");

services.AddDbContext<Context>(action => action.UseSqlite(connection));

services.AddMvc(options => options.EnableEndpointRouting = false);

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

app.UseMvc();

}

}

}

Изменим имеющийся по умолчанию контроллер HomeController.

using System;

using System.Linq;

using App.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace App.Controllers

{

public class HomeController : Controller

{

private readonly Context \_context;

public HomeController(Context context)

{

\_context = context;

}

[Route("/")]

public IActionResult Index()

{

return View(\_context.Clubs);

}

[Route("/Players")]

public IActionResult Players([FromQuery] int clubId)

{

var club = \_context.Clubs.Find(clubId);

if (club == null)

{

return Content($"No club for id: {clubId}");

}

return View(\_context.roster.Where(x => x.ClubId == clubId));

}

[HttpGet]

[Route("/Player")]

public IActionResult Player([FromQuery] string playerId)

{

var player = \_context.roster.Find(playerId);

if (player == null)

{

return Content($"No player for id: {playerId}");

}

return View(player);

}

[HttpPost]

[Route("/Player")]

public IActionResult EditPlayer(roster roster)

{

try

{

\_context.Entry(roster).State = EntityState.Modified;

\_context.SaveChanges();

return new RedirectResult($"/Players?clubId={roster.ClubId}");

}

catch (Exception e)

{

return Content($"Error while applying changes: {e}");

}

}

}

}

Добавим в папку *Views/Home* файлы представлений.

Файл Players.cshtml

@model IEnumerable<roster>

<style>

table {

border-collapse: collapse;

}

table, th, td {

border: 1px solid black;

}

</style>

<table class="table">

<thead>

<tr>

<th>@Html.DisplayNameFor(model => model.playerid)</th>

<th>@Html.DisplayNameFor(model => model.jersey)</th>

<th>@Html.DisplayNameFor(model => model.fname)</th>

<th>@Html.DisplayNameFor(model => model.sname)</th>

<th>@Html.DisplayNameFor(model => model.position)</th>

<th>@Html.DisplayNameFor(model => model.birthday)</th>

<th>@Html.DisplayNameFor(model => model.weight)</th>

<th>@Html.DisplayNameFor(model => model.height)</th>

<th>@Html.DisplayNameFor(model => model.birthcity)</th>

<th>@Html.DisplayNameFor(model => model.birthstate)</th>

<th>@Html.DisplayNameFor(model => model.ClubId)</th>

<th>Edit page</th>

</tr>

</thead>

<tbody>

@foreach (var item in Model)

{

<tr>

<td>@Html.DisplayFor(modelItem => item.playerid)</td>

<td>@Html.DisplayFor(modelItem => item.jersey)</td>

<td>@Html.DisplayFor(modelItem => item.fname)</td>

<td>@Html.DisplayFor(modelItem => item.sname)</td>

<td>@Html.DisplayFor(modelItem => item.position)</td>

<td>@Html.DisplayFor(modelItem => item.birthday)</td>

<td>@Html.DisplayFor(modelItem => item.weight)</td>

<td>@Html.DisplayFor(modelItem => item.height)</td>

<td>@Html.DisplayFor(modelItem => item.birthcity)</td>

<td>@Html.DisplayFor(modelItem => item.birthstate)</td>

<td>@Html.DisplayFor(modelItem => item.ClubId)</td>

<td>

<a asp-action="Player" asp-controller="Home" asp-route-playerId="@item.playerid">Edit</a>

</td>

</tr>

}

</tbody>

</table>

Файл Player.cshtml

@model roster

@using (Html.BeginForm("EditPlayer", "Home", FormMethod.Post))

{

@Html.HiddenFor(m => m.playerid)

<div>

@Html.LabelFor(m => m.jersey)

@Html.EditorFor(m => m.jersey)

</div>

<div>

@Html.LabelFor(m => m.fname)

@Html.EditorFor(m => m.fname)

</div>

<div>

@Html.LabelFor(m => m.sname)

@Html.EditorFor(m => m.sname)

</div>

<div>

@Html.LabelFor(m => m.position)

@Html.EditorFor(m => m.position)

</div>

<div>

@Html.LabelFor(m => m.birthday)

@Html.EditorFor(m => m.birthday)

</div>

<div>

@Html.LabelFor(m => m.weight)

@Html.EditorFor(m => m.weight)

</div>

<div>

@Html.LabelFor(m => m.height)

@Html.EditorFor(m => m.height)

</div>

<div>

@Html.LabelFor(m => m.birthcity)

@Html.EditorFor(m => m.birthcity)

</div>

<div>

@Html.LabelFor(m => m.birthstate)

@Html.EditorFor(m => m.birthstate)

</div>

<div>

@Html.LabelFor(m => m.ClubId)

@Html.EditorFor(m => m.ClubId)

</div>

<div>

<input type="submit" value="Save"/>

</div>

}

Файл Index.cshtml

@model IEnumerable<Club>

<div>

@{

foreach (var club in Model)

{

<li>

<a asp-action="Players" asp-controller="Home" asp-route-clubId="@club.ClubId">@club.Name</a>

</li>

}

}

</div>