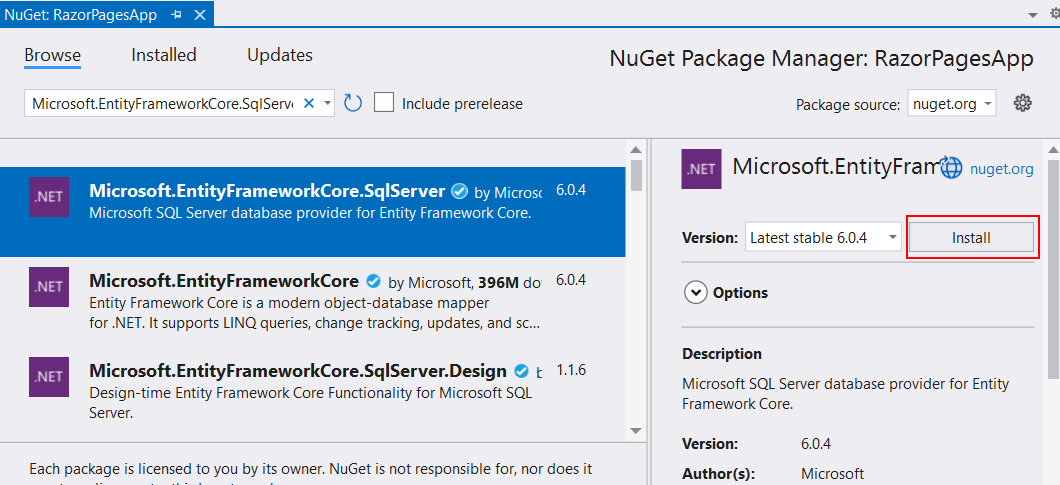
**Работа с базой данных через Entity Framework**

**Подключение к базе данных в Razor Pages**

Для работы с базой данных один из распространенных подходов в ASP.NET Core является использование фреймворка Entity Framework. **Entity Framework** представляет ORM-решение, которое позволяет автоматически связать обычные классы языка C# с таблицами в базе данных. Entity Framework Core поддерживает различные СУБД, но в данном случае мы будем работать с базами данных MS SQL Server. Рассмотрим создание простейшего CRUD-интерфейса и взаимодействие с БД из Razor Pages. Пусть у нас есть пустой проект для ASP.NET Core по типу **ASP.NET Core Empty**.

Прежде всего для взаимодействия с MS SQL Server через Entity Framework добавим в проект через Nuget пакет **Microsoft.EntityFrameworkCore.SqlServer**.



Создадим в проекте новую папку **Models**, в которую затем добавим класс **User** со следующим кодом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | namespace RazorPagesApp.Models  {      public class User      {          public int Id { get; set; }          public string? Name { get; set; } // имя пользователя          public int Age { get; set; } // возраст пользователя      }  } |

Этот класс определяет данные, с которыми мы далее будем работать.

И также добавим в папку **Models** класс контекста данных **ApplicationContext**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Microsoft.EntityFrameworkCore;    namespace RazorPagesApp.Models  {      public class ApplicationContext : DbContext      {          public DbSet<User> Users { get; set; } = null!;          public ApplicationContext(DbContextOptions<ApplicationContext> options)              : base(options)          {              Database.EnsureCreated();   // создаем базу данных при первом обращении          }      }  } |

Свойство **DbSet** представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных. При этом по умолчанию название свойства должно соответствовать множественному числу названию модели в соответствии с правилами английского языка. То есть User - название класса модели представляет единственное число, а Users - множественное число.

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::

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {    "ConnectionStrings": {      "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=razorpagesdb;Trusted\_Connection=True"    },    // остальное содержимое файла  } |

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

И в конце подключим контекст данных ApplicationContext в приложение в виде сервиса и для этого изменим файл **Program.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using Microsoft.EntityFrameworkCore;  using RazorPagesApp.Models;    // пространство имен класса ApplicationContext    var builder = WebApplication.CreateBuilder(args);    // получаем строку подключения из файла конфигурации  string connection = builder.Configuration.GetConnectionString("DefaultConnection");    // добавляем контекст ApplicationContext в качестве сервиса в приложение  builder.Services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(connection));    builder.Services.AddRazorPages();    var app = builder.Build();    app.MapRazorPages();    app.Run(); |

С помощью метода **AddDbContext** класс контекста добавляется в приложение в виде сервиса, и у него устанавливается строка подключения. Добавление контекста данных в виде сервиса позволит затем получать его в конструкторе контроллера через механизм внедрения зависимостей.

## Добавление и получение данных из базы данных

Продолжим работу с проектом из прошлой темы, где уже настроено подключение и создана база данных. Теперь определим в проекте в папке **Pages** новую страницу Razor, которую назовем **Index.cshtml**. После ее добавления в папке **Pages** появятся два файла: **Index.cshtml** и **Index.cshtml.cs**. Эта страница будет предназначена для вывода список пользователей из базы данных.

Изменим файл **Index.cshtml.cs** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using Microsoft.AspNetCore.Mvc.RazorPages;  using Microsoft.EntityFrameworkCore;  using RazorPagesApp.Models;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          ApplicationContext context;          public List<User> Users { get; private set; } = new();          public IndexModel(ApplicationContext db)          {              context = db;          }          public void OnGet()          {              Users = context.Users.AsNoTracking().ToList();          }      }  } |

Поскольку в прошлой статье контекст данных ApplicationContext был добавлен в приложение в качестве сервиса, то в конструкторе модели страницы мы можем получить этот контекст и затем использовать его для взаимодействия с базой данных.

И изменим файл **Index.cshtml** для вывода списка данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @page    @model RazorPagesApp.Pages.IndexModel    <h2>Список пользователей</h2>  <p><a href="/Create">Добавить пользователя</a></p>  <table>      <tr><th>Имя</th><th>Возраст</th></tr>      @foreach (var user in Model.Users)      {          <tr><td>@user.Name</td><td>@user.Age</td></tr>      }  </table> |

Сверху над таблицей эта страница будет содержать ссылку на другую страницу - Create, которая будет предназначена для добавления объектов.

Теперь аналогичным образом добавим в папку **Pages** еще одну страницу Razor - **Create.cshtml**. И вначале изменим файл **Create.cshtml.cs** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;  using RazorPagesApp.Models;    namespace RazorPagesApp.Pages  {      [IgnoreAntiforgeryToken]      public class CreateModel : PageModel      {          ApplicationContext context;          [BindProperty]          public User Person { get; set; } = new();          public CreateModel(ApplicationContext db)          {              context = db;          }          public async Task<IActionResult> OnPostAsync()          {              context.Users.Add(Person);              await context.SaveChangesAsync();              return RedirectToPage("Index");          }      }  } |

Класс CreateModel также через конструктор получает контекст данных для последующего добавления объекта в БД. Сам добавляемый объект определен в виде свойства Person, к которому применяется атрибут **[BindProperty]**. А это значит, что система автоматически будет связывать данные запроса со свойствами этого объекта.

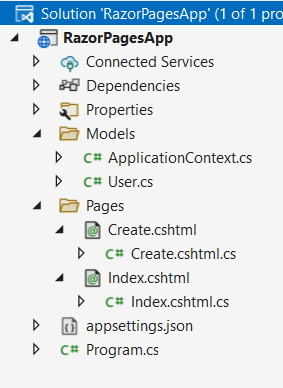
В методе **OnPostAsync()** происходит добавление объекта в базу данных, после чего идет редирект на страницу Index.

И поскольку данные будут отправляться из формы в запросе Post, чтобы избежать необходимости устанавливать antiforgery-токен, к классу CreateModel применяется атрибут **[IgnoreAntiforgeryToken]**

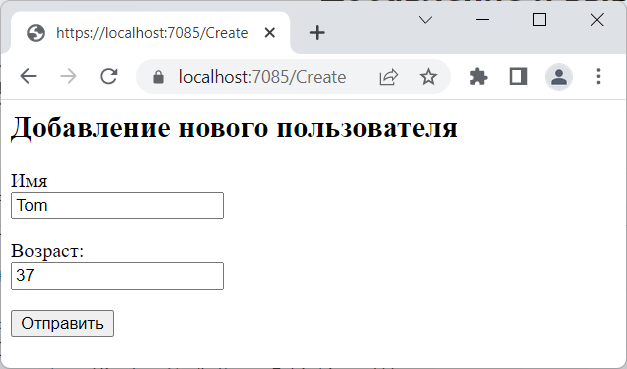
А на странице Create.cshtml определим форму ввода данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @page  @model RazorPagesApp.Pages.CreateModel    <h2>Добавление нового пользователя</h2>  <form method="post">      <p><label>Имя</label><br />          <input name="name" />      </p>      <p>          <label>Возраст:</label><br />          <input name="age" type="number" />      </p>      <input type="submit" value="Отправить" />  </form> |

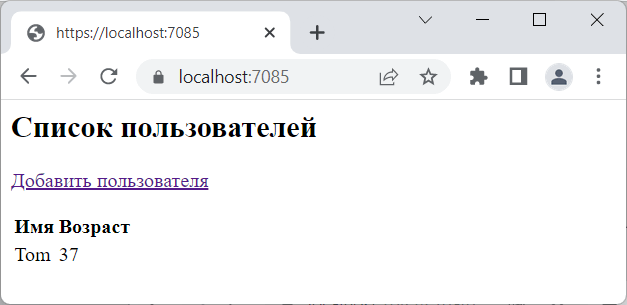
Таким образом, у нас получится следующий проект:



Запустим проект. Вначале у нас нет объектов в бд, поэтому перейдем к странице Create.cshtml и добавим один объект:



И после добавления страница Index.cshtml отобразит добавленный объект:



## Изменение и удаление данных в базе данных

Продолжим работу с проектом из прошлой темы и добавим в него функционал по редактированию и удалению данных. Итак, добавим в папку **Pages** новую страницу Razor **Edit.cshtml**, которая будет предназначена для редактирования данных.

Вначале изменим файл связанного кода **Edit.cshtml.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;  using RazorPagesApp.Models;    namespace RazorPagesApp.Pages  {      [IgnoreAntiforgeryToken]      public class EditModel : PageModel      {          ApplicationContext context;          [BindProperty]          public User? Person { get; set; }            public EditModel(ApplicationContext db)          {              context = db;          }          public async Task<IActionResult> OnGetAsync(int id)          {              Person = await context.Users.FindAsync(id);                if (Person == null) return NotFound();                return Page();          }          public async Task<IActionResult> OnPostAsync()          {              context.Users.Update(Person!);              await context.SaveChangesAsync();              return RedirectToPage("Index");          }      }  } |

В методе OnGetAsync через параметр получаем id пользователя, по этому id получаем из базы данных объект и передаем его в свойство Person.

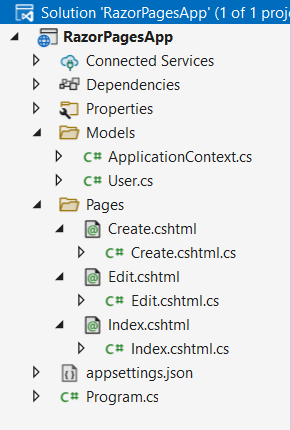
Поскольку к свойству Person применяется атрибут **[BindProperty]**, то данные из запроса будут автоматически связаны со свойствами объекта Person. В методе OnPostAsync сохраняем изменения в базе данных и выполняем переадресацию на страницу Index. Обратите внимание, что в данном случае при обновлении к объекту Person применяется оператор !. Хотя данное свойство может представлять значение null, но поскольку оно автоматически связывается с данными запроса, и даже если в запросе не будет отправлено никаких данных, все равно будет создан объект User, и свойство Person не будет равно null. Поэтому в данном случае мы можем написать context.Users.Update(Person!);, тем самым указывая, что Person не равен null. Однако в других условиях и сценариях может потребоваться проверять объект перед обновлением на null.

Также изменим файл **Edit.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @page "{id:int}"  @model RazorPagesApp.Pages.EditModel    <h2>Редактирование пользователя</h2>    <form method="post">      <input name="id" value="@Model.Person?.Id" type="hidden" />      <p>          <label>Имя</label><br />          <input name="name" value="@Model.Person?.Name" />      </p>      <p>          <label>Возраст</label><br />          <input name="age" value="@Model.Person?.Age" type="number" />      </p>      <input type="submit" value="Сохранить" />  </form> |

Чтобы параметр id передавался как параметр маршрута, после директивы page указан шаблон параметров маршрута "{id:int}". И далее идет форма для редактирования, все поля которой привязаны к свойствам модели EditModel.

В итоге весь проект будет выглядеть следующим образом:



Также изменим код IndexModel в файле **Index.cshtml.cs**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;  using Microsoft.EntityFrameworkCore;  using RazorPagesApp.Models;    namespace RazorPagesApp.Pages  {      public class IndexModel : PageModel      {          ApplicationContext context;          public List<user> Users { get; private set; } = new();          public IndexModel(ApplicationContext db)          {              context = db;          }          public void OnGet()          {              Users = context.Users.AsNoTracking().ToList();          }          public async Task<IActionResult> OnPostDeleteAsync(int id)          {              var user = await context.Users.FindAsync(id);                if (user != null)              {                  context.Users.Remove(user);                  await context.SaveChangesAsync();              }                return RedirectToPage();          }      }  }  </user> |

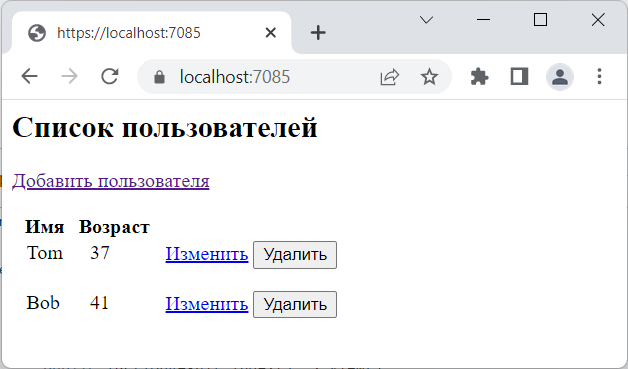
По сравнению с кодом из прошлой темы здесь добавлен обработчик OnPostDeleteAsync, который обрабатывает запросы post и который принимает id объекта на удаление. Затем, если объект найден, производится его удаление. Хотя также можно было бы для удаления создать отдельную страницу, но в данном случае это будет избыточно.

И также изменим файл **Index.cshtml**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | @page  @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers    @model RazorPagesApp.Pages.IndexModel    <h2>Список пользователей</h2>  <p><a href="/Create">Добавить пользователя</a></p>  <table>      <tr><th>Имя</th><th>Возраст</th><th></th></tr>      @foreach (var user in Model.Users)      {          <tr>              <td>@user.Name</td              ><td>@user.Age</td>               <td>                  <form method="post">                      <a asp-page="Edit" asp-route-id="@user.Id">Изменить</a>                      <button type="submit" asp-page-handler="delete" asp-route-id="@user.Id">                          Удалить                      </button>                  </form>              </td>          </tr>      }  </table> |

В форму в отдельном столбце помещены ссылка на редактирование и кнопка, по нажатию на которую будет идти post-запрос к обработчику delete (OnPostDeleteAsync) на текущей странице.

В итоге страница со списком пользователей будет выглядеть следующим образом:



По нажатию на кнопку объект будет удален, а по нажатию на ссылку приложение перейдет к странице редактирования:

