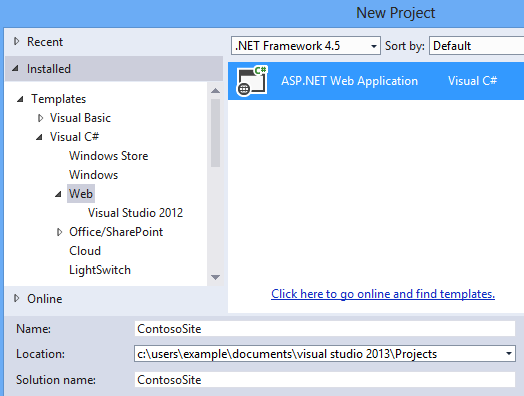
11 – Реализация БД (до создание моделей)

12 – Реализация Основного функционала программы по данной методичке

13 – Реализация индивидуального функционала не прописанного в данном документе

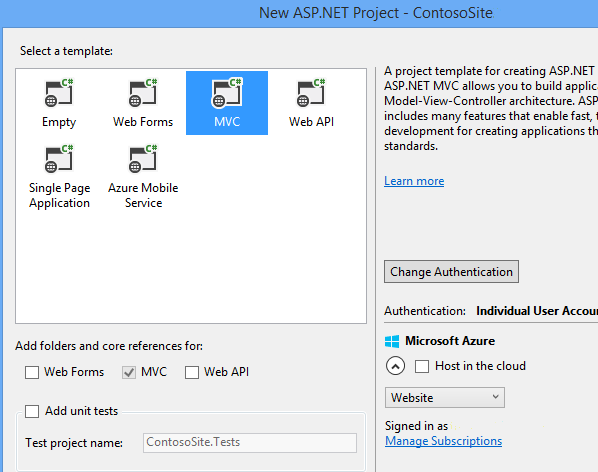
**Создание нового веб-приложения ASP.NET**

В новом решении или в том же решении, что и проект базы данных, создайте новый проект в Visual Studio и выберите шаблон **веб-приложения ASP.NET** . Назовите проект **ContosoSite** .



Нажмите « **ОК»** .

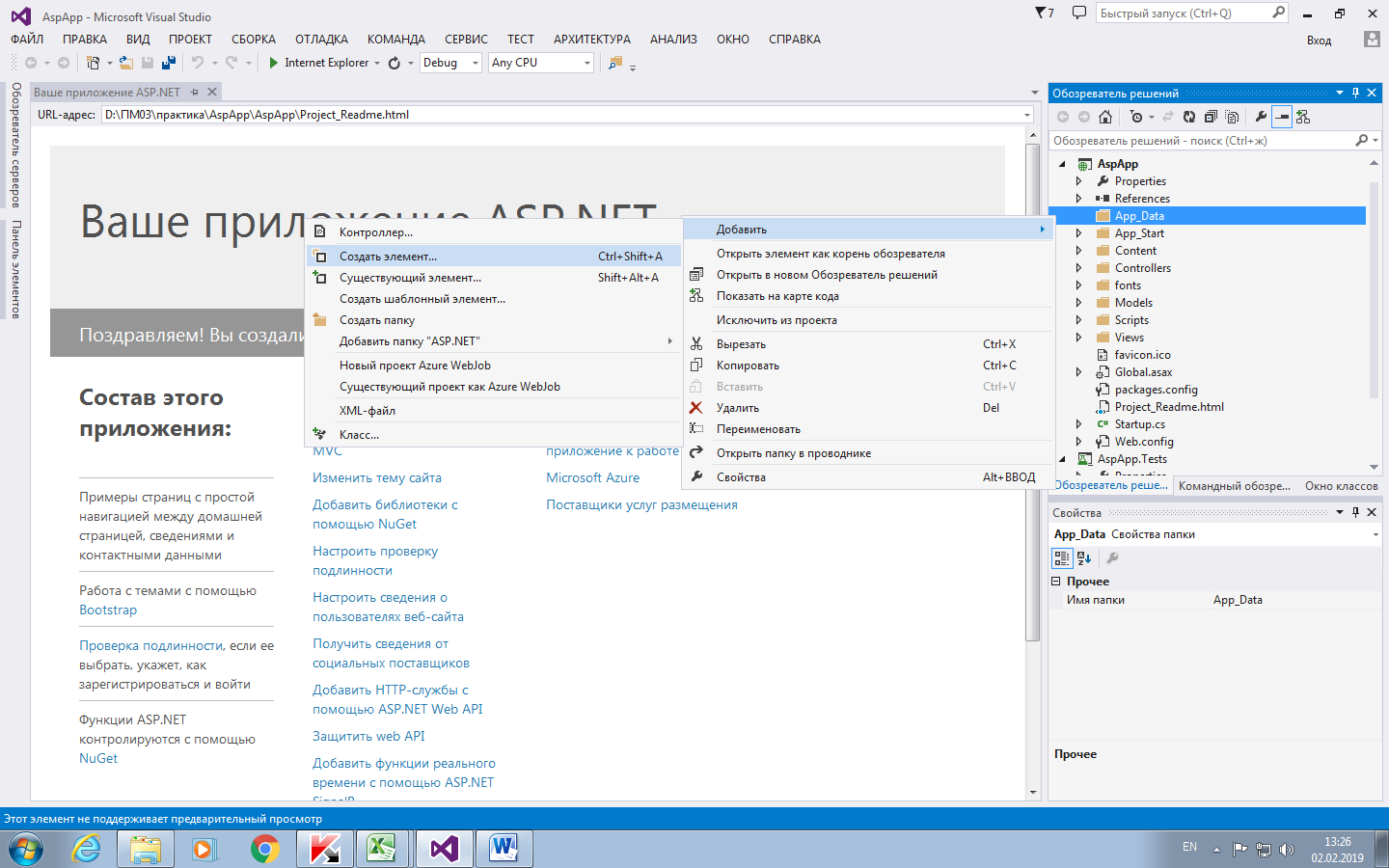
В окне «Новый проект ASP.NET» выберите шаблон MVC . Очистить флажок Узел в облаке. Нажмите « ОК», чтобы создать приложение.



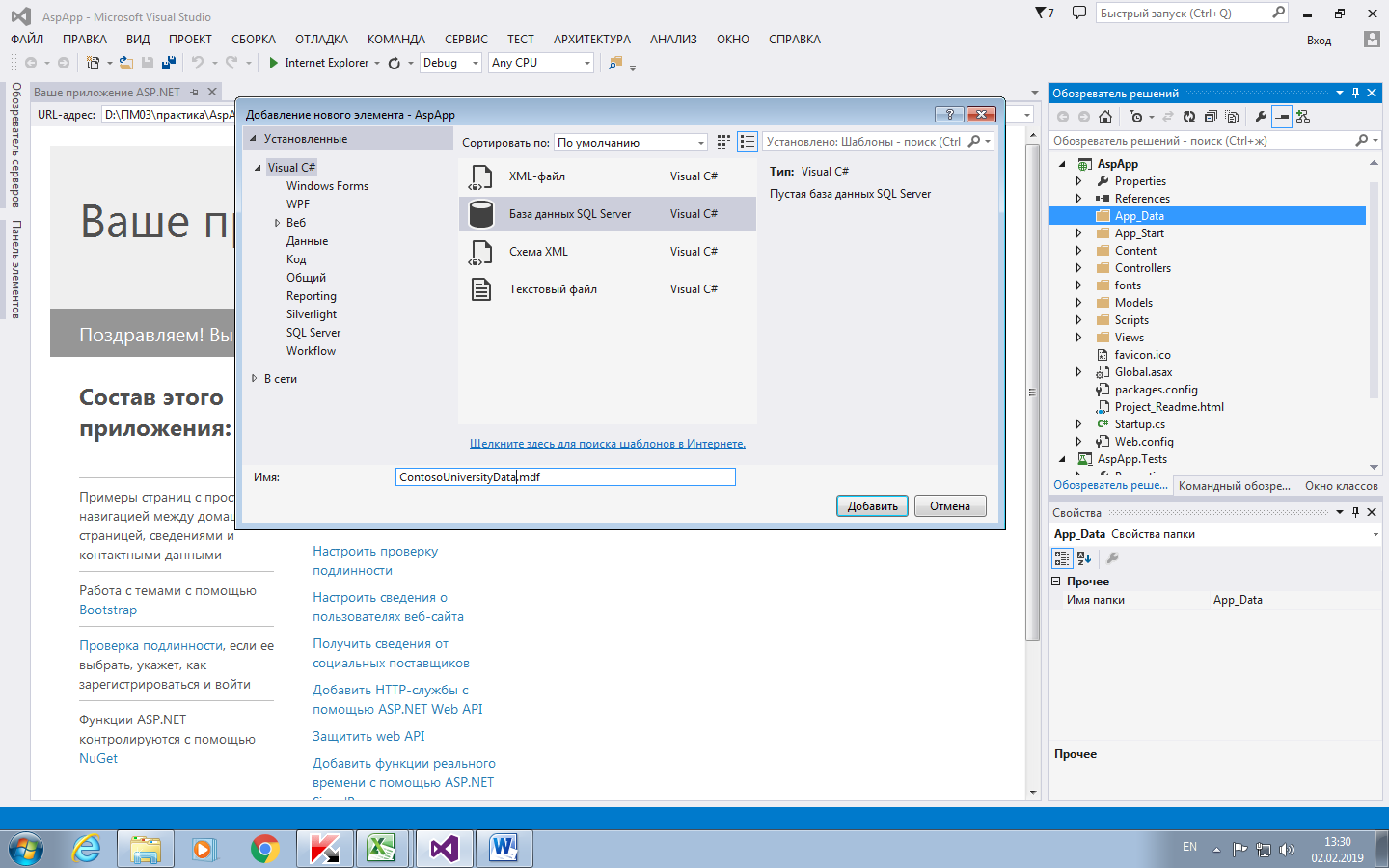
Проект создается с файлами и папками по умолчанию.

## Создание базы данных

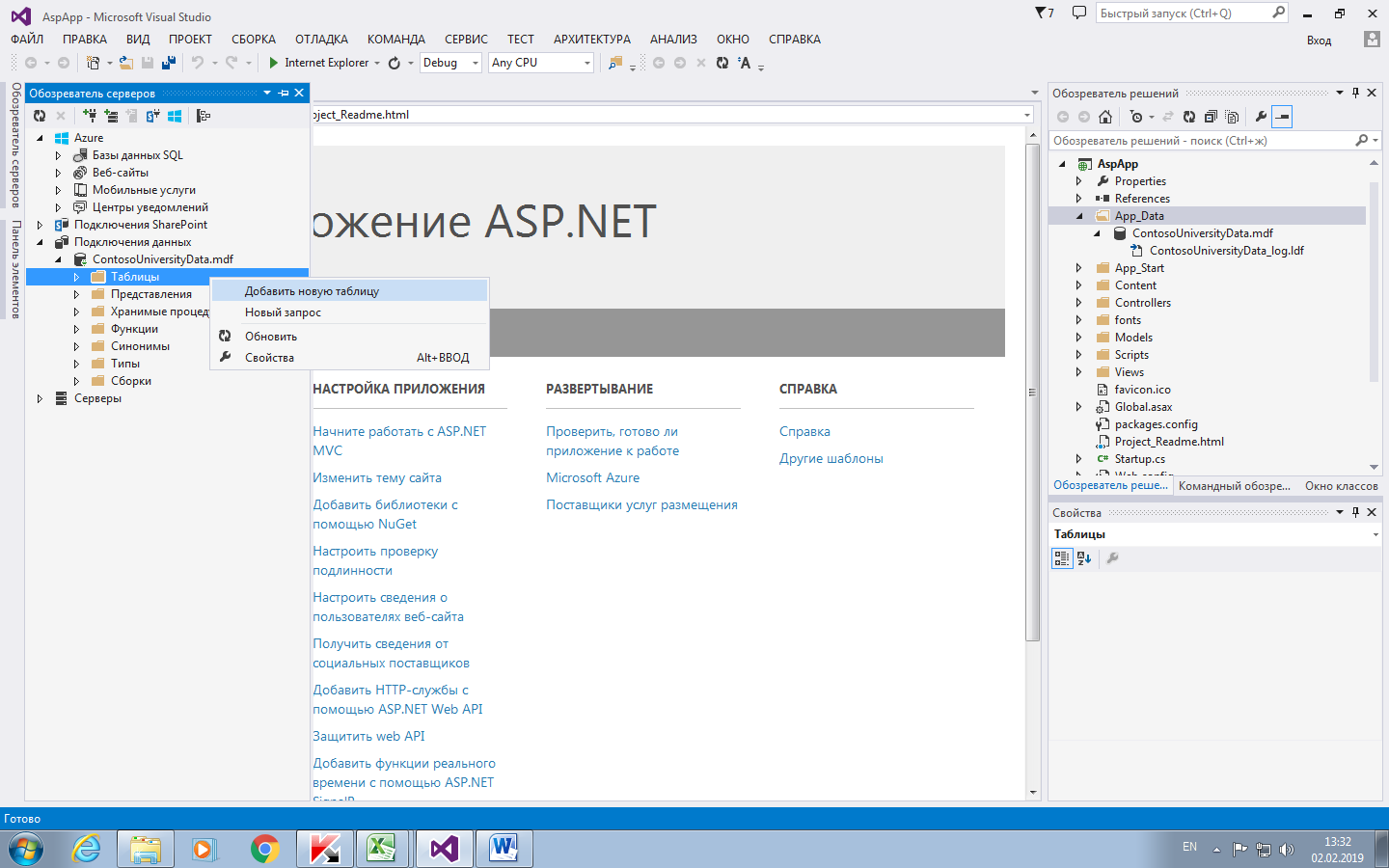
В обозревателе решений правой кнопкой мыши нажать на папку App\_Data и создать новый элемент



Создать базу данных со своим именем

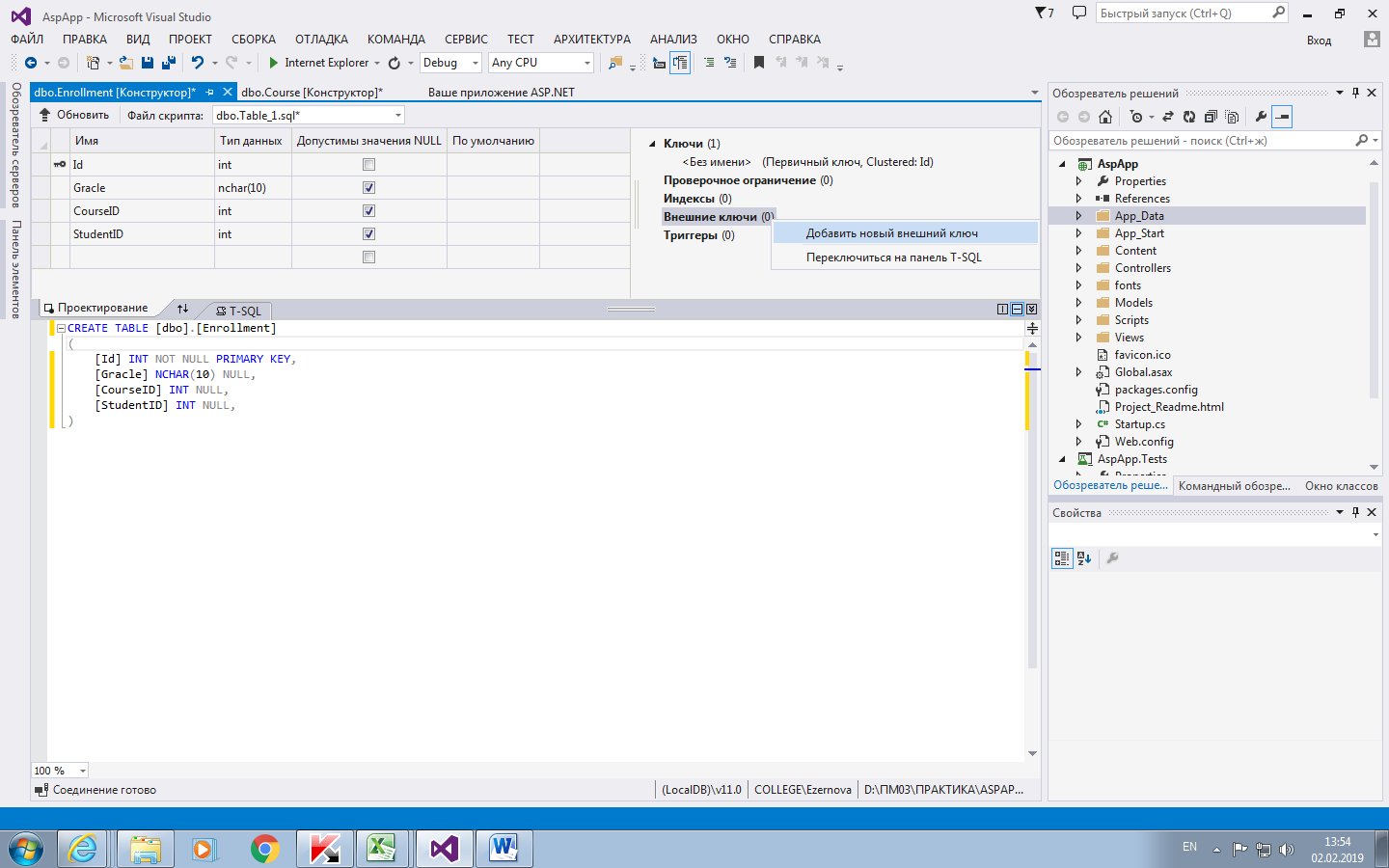


В обозревателе серверов в своей базе данных создать таблицы



Добавьте столбцы в таблицы и определите тип хранимых данных.

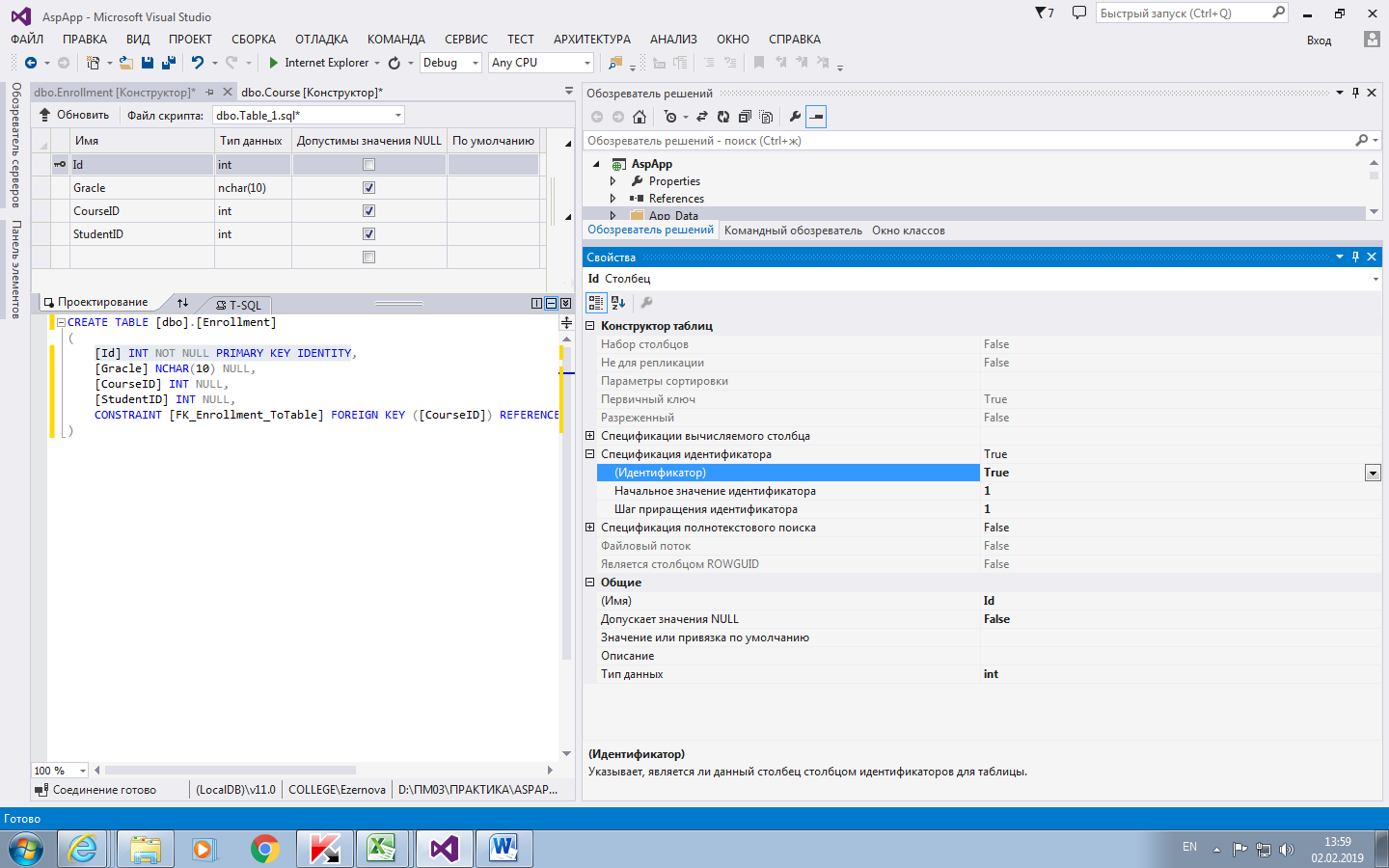
Определите первичные и вторичные ключи.



В появившейся надписи подправить названия полей и таблиц

CONSTRAINT [FK\_Enrollment\_ToTable] FOREIGN KEY ([CourseID]) REFERENCES [Course]([CourseID])

Для полей первичных ключей определите автоинкремент

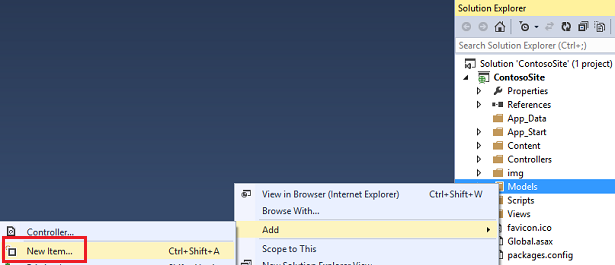


После того как все таблицы были созданы и все связи были определены переходим к созданию модели данных в приложении.

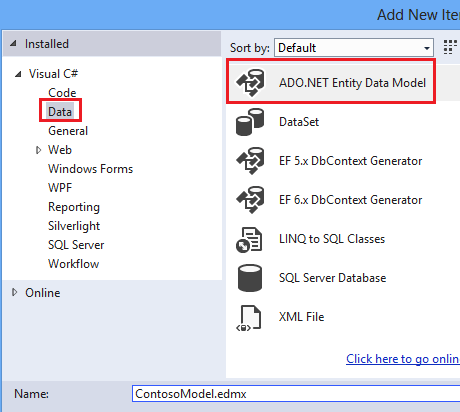
## Создание моделей (Всё после – 12 лаба)

Теперь вы создадите модели Entity Framework из таблиц базы данных. Эти модели являются классами, которые вы будете использовать для работы с данными. Каждая модель отражает таблицу в базе данных и содержит свойства, соответствующие столбцам в таблице.

Щелкните правой кнопкой мыши папку « **Модели** » и выберите « **Добавить»** и « **Создать элемент»** .

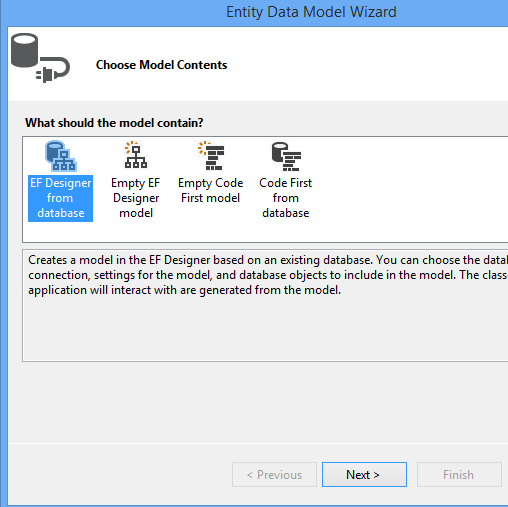


В окне «Добавить новый элемент» выберите « **Данные»** в левой панели и « **Модель ADO.NET EDM»** из параметров в центральной панели. Назовите новый файл модели **ContosoModel** .



Нажмите « **Добавить»** .

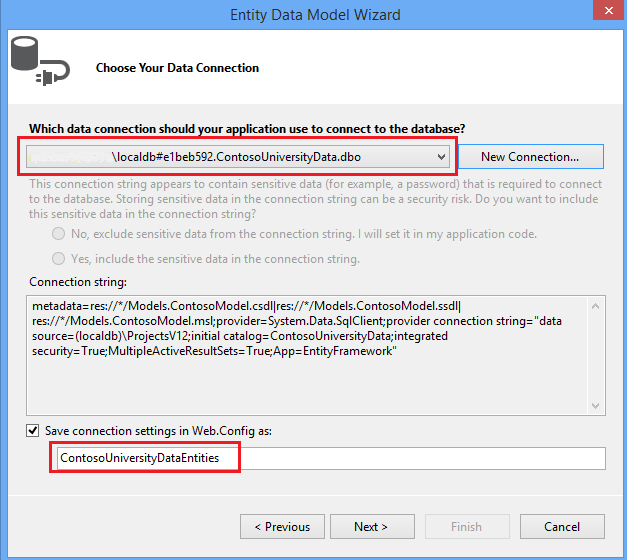
В Мастере моделей данных сущностей выберите **конструктор EF из базы данных** .



Нажмите « **Далее»** .

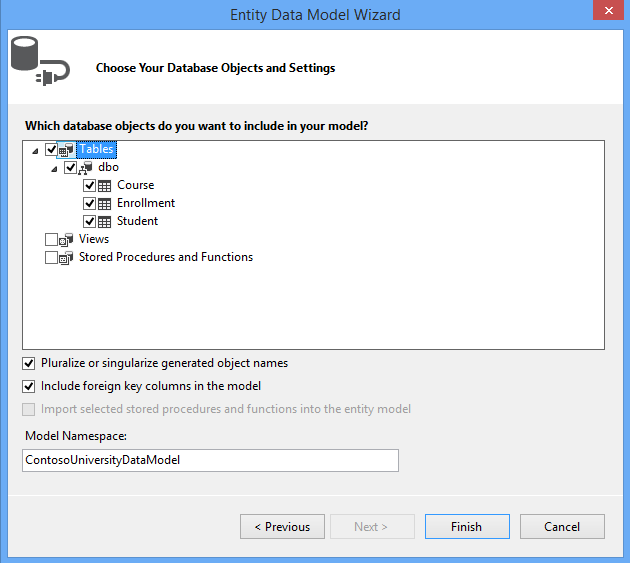
Если у вас есть подключения к базе данных, определенные в вашей среде разработки, вы можете увидеть одно из этих подключений, предварительно выбранных. Однако вы хотите создать новое соединение с базой данных, созданной ранее. Нажмите кнопку « **Новое соединение»** .

Вы можете использовать имя по умолчанию для подключения в файле Web.Config



Нажмите « **Далее»** .

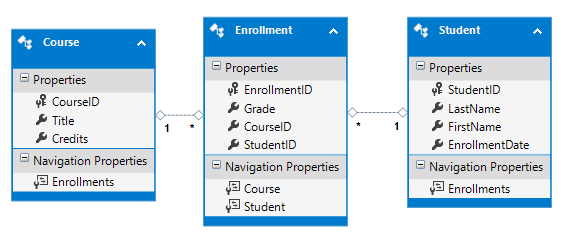
Выберите « **Таблицы»** для создания моделей для всех трех таблиц.



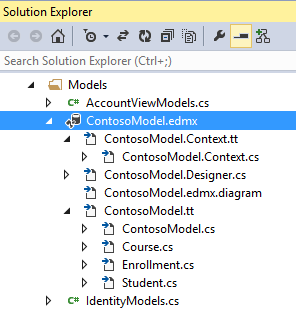
Нажмите « **Готово»** .

Если вы получили предупреждение о безопасности, выберите « **ОК»,** чтобы продолжить запуск шаблона.

Модели создаются из таблиц базы данных, и отображается диаграмма, показывающая свойства и отношения между таблицами.



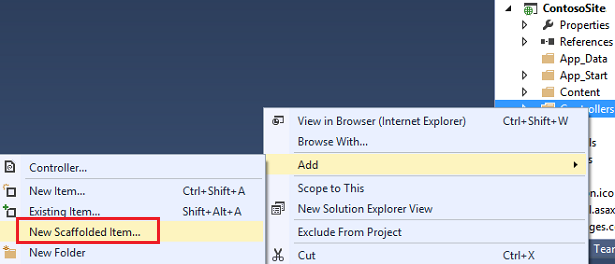
Папка «Модели» теперь включает в себя множество новых файлов, связанных с моделями, которые были созданы из базы данных.



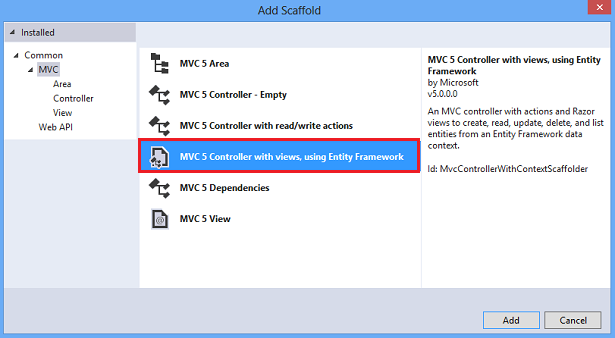
**ContosoModel.Context.cs**  файл содержит класс, производный от **DbContext** класса, а также предоставляет свойство для каждого класса модели , которая соответствует таблице базы данных. В **Course.cs** , **Enrollment.cs** и **Student.cs** файлы содержат классы модели, которые представляют таблицы баз данных. При работе с подклассами вы будете использовать как класс контекста, так и классы модели.

Сейчас необходимо сгенерировать код, который будет обеспечивать стандартные операции с данными для классов модели. Вы добавляете код, добавляя элемент Scaffold.  Scaffold будет включать контроллер и представления, соответствующие моделям учащихся и регистрации, которые вы создали в предыдущем разделе. (Необходимо собрать проект, прежде чем добавлять контроллеры!)

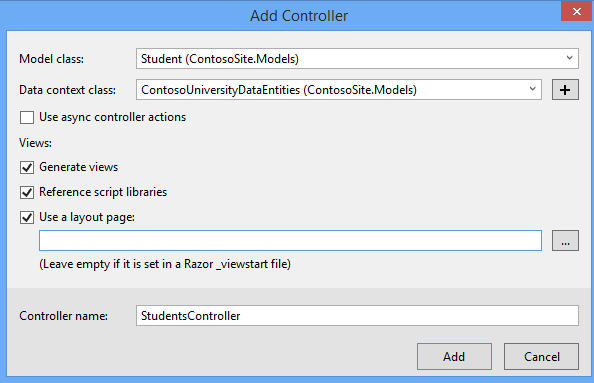
Чтобы поддерживать согласованность в вашем проекте, надо добавить новый контроллер в существующую папку Controllers . Щелкните правой кнопкой мыши Контроллеры папку и выберите Добавить – Новый Шаблонный элемент .



Выберите MVC 5 Controller с представлениями, используя параметр Entity Framework . Эта опция генерирует контроллер и представления для обновления, удаления, создания и отображения данных в вашей модели.



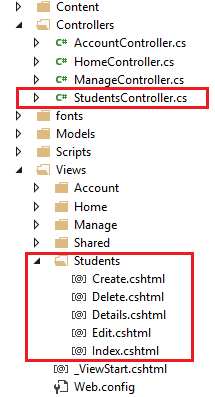
Выберите « Студент» для класса модели и выберите ContosoUniversityEntities для класса контекста. Сохраните имя контроллера как StudentsController ,



Нажмите « Добавить» .

Если вы получили сообщение об ошибке, возможно, это связано с тем, что проект не был создан в предыдущем разделе. Если да, попробуйте создать проект, а затем снова добавьте элемент шаблона.

По завершении процесса генерации кода вы увидите новый контроллер и представления в своем проекте.

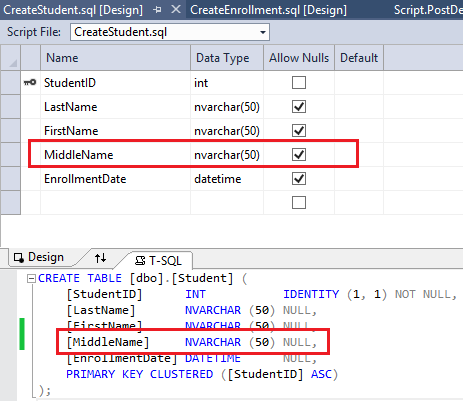


Повторите те же шаги, но добавьте Scaffold для класса регистрации. Когда закончите, у вас должен быть файл EnrollmentsController.cs и папка под заголовками Views named Enrollments с помощью представлений «Создать, удалить, детализировать, редактировать и индексировать».

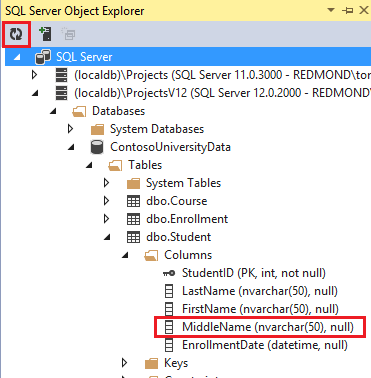
## Добавить столбец

Если вы обновляете структуру таблицы в своей базе данных, вам необходимо убедиться, что ваше изменение распространяется на модели данных, представления и контроллер.

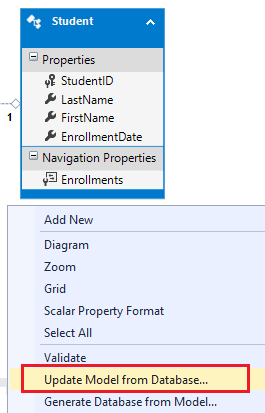
В таблицу Student добавьте столбец с именем **MiddleName,** который имеет тип NVARCHAR (50), и допускает значения NULL.



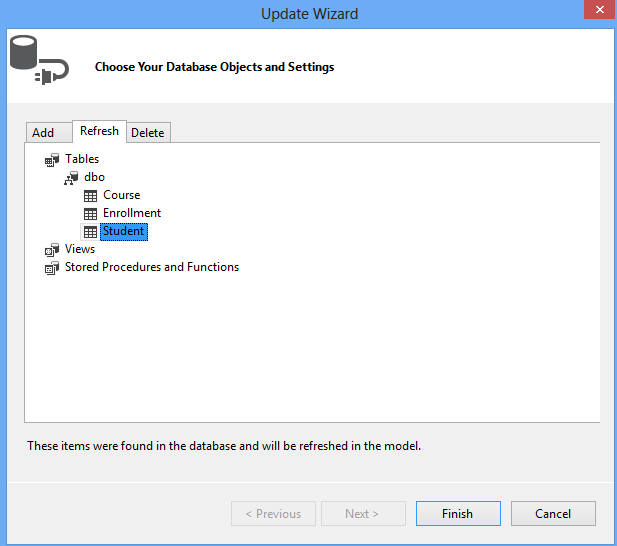
Разверните это изменение в своей локальной базе данных. Новое поле добавляется в таблицу. Если вы не видите его в обозревателе объектов SQL Server, нажмите кнопку «Обновить» на панели.



Новый столбец существует в таблице базы данных, но в настоящее время он не существует в классе модели данных. Вы должны обновить модель, чтобы добавить новый столбец. В папке « Модели » откройте файл ContosoModel.edmx, чтобы отобразить диаграмму модели. Обратите внимание, что модель Student не содержит свойства MiddleName. Щелкните правой кнопкой мыши в любом месте проектной поверхности и выберите « **Обновить модель из базы данных»** .



В мастере обновления выберите вкладку « **Обновить** » и таблицу « **Студент»** .



Нажмите « **Готово»** .

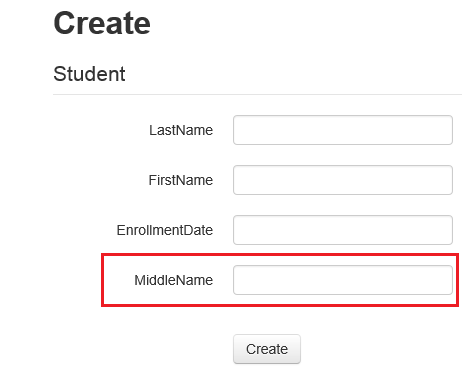
По завершении процесса обновления диаграмма базы данных включает новое свойство **MiddleName** . Сохраните файл **ContosoModel.edmx** . Вы должны сохранить этот файл для нового свойства, которое будет распространено в классе **Student.cs** . Теперь вы обновили базу данных и модель.

Постройте решение.

К сожалению, в представлениях все еще нет нового свойства. Чтобы обновить представления, у вас есть два варианта: вы можете повторно сгенерировать представления, добавив снова шаблоны для класса Student, или вы можете вручную добавить новое свойство к существующим представлениям. В этом уроке вы снова добавите шаблоны, потому что вы не внесли никаких изменений в автоматически создаваемые представления. Вы можете вручную добавить свойство, когда вы внесли изменения в представления, и не хотите терять эти изменения.

Чтобы обеспечить повторное создание представлений, удалите папку « **Студенты** » в разделе « **View»** и удалите « **StudentsController** **»** . Затем щелкните правой кнопкой мыши папку **Controllers** и добавьте шаблон для модели **Student** . Опять же, назовите контроллер **StudentsController** . Выберите **ОК** .

Представления теперь содержат свойство MiddleName.



Вы можете настроить код в соответствии с конкретными требованиями вашего приложения. В настоящее время ваше приложение не отображает зарегистрированные курсы для выбранного студента. В этом разделе вы добавите зарегистрированные курсы для каждого учащегося в представление Details для учащегося.

Open Students / Details.cshtml и ниже последней вкладки </ dl>, но перед закрывающим тегом </ div> добавьте следующий код.

CSHTMLкопия

<table class="table">

<tr>

<th>

Course Title

</th>

<th>

Grade

</th>

<th>

Credits

</th>

</tr>

@foreach (var item in Model.Enrollments)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Course.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.Grade)

</td>

<td>

@Html.DisplayFor(modelItem => item.Course.Credits)

</td>

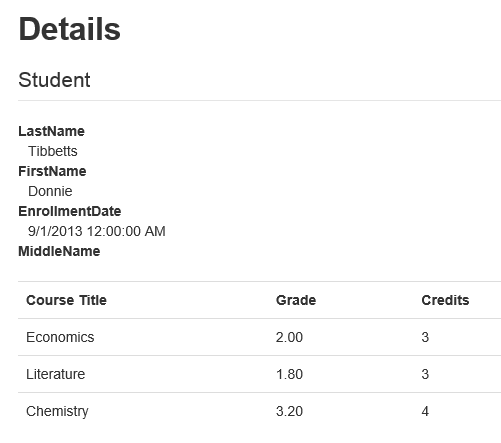
</tr>

}

</table>

Этот код создает таблицу, которая отображает строку для каждой записи в таблице регистрации для выбранного ученика. Метод Display отображает HTML для объекта (modelItem), который представляет выражение. Вы используете метод Display (а не просто вставляете значение свойства в код), чтобы убедиться, что значение отформатировано правильно в зависимости от его типа и шаблона для этого типа. В этом примере каждое выражение возвращает одно свойство из текущей записи в цикле, а значения - это примитивные типы, которые отображаются как текст.

Перейдите к разделу «Студенты / Индекс» и выберите « Подробности» для одного из учеников. Вы увидите, что зарегистрированные курсы включены в представление.



чтобы ограничить длину значений, предоставленных для свойств FirstName, LastName и MiddleName. В базе данных эти значения ограничены 50 символами; однако в вашем веб-приложении ограничение персонажа в настоящее время не применяется. Если пользователь предоставляет более 50 символов для одного из этих значений, страница будет сбой при попытке сохранить значение в базе данных. Вы также ограничите класс до значений от 0 до 4.

Откройте файл Student.cs в папке « Модели ». Добавьте в класс следующий выделенный код.

C #копия

namespace ContosoSite.Models

{

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

public partial class Student

{

public Student()

{

this.Enrollments = new HashSet<Enrollment>();

}

public int StudentID { get; set; }

[StringLength(50)]

public string LastName { get; set; }

[StringLength(50)]

public string FirstName { get; set; }

public Nullable<System.DateTime> EnrollmentDate { get; set; }

[StringLength(50)]

public string MiddleName { get; set; }

public virtual ICollection<Enrollment> Enrollments { get; set; }

}

}

В Enrollment.cs добавьте следующий выделенный код.

C #копия

namespace ContosoSite.Models

{

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

public partial class Enrollment

{

public int EnrollmentID { get; set; }

[Range(0, 4)]

public Nullable<decimal> Grade { get; set; }

public int CourseID { get; set; }

public int StudentID { get; set; }

public virtual Course Course { get; set; }

public virtual Student Student { get; set; }

}

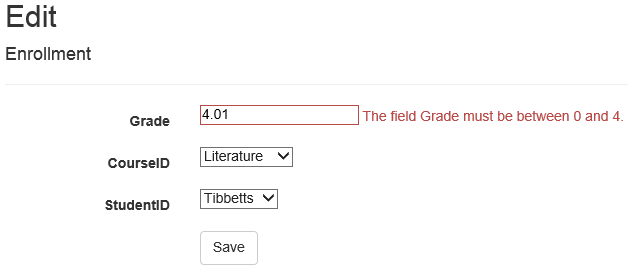
}

Постройте решение.

Перейдите на страницу для редактирования или создания ученика. Если вы попытаетесь ввести более 50 символов, появится сообщение об ошибке.



Перейдите на страницу для редактирования заявок и попытайтесь присвоить оценку выше 4.



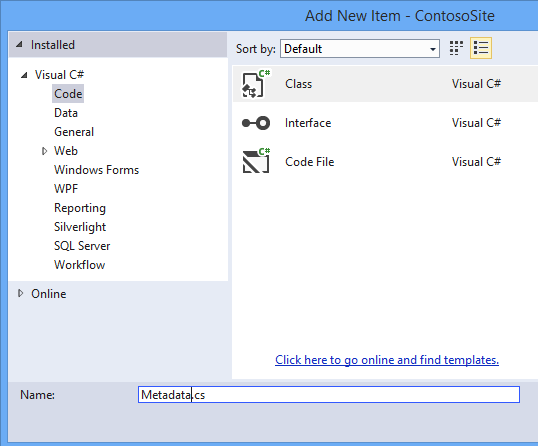
Полный список аннотаций проверки данных, которые вы можете применить к свойствам и классам, см . В разделе [System.ComponentModel.DataAnnotations](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.aspx) .

**Добавление классов метаданных**

Добавление атрибутов проверки непосредственно в класс модели работает, если вы не ожидаете изменения базы данных; однако, если ваша база данных изменяется, и вам необходимо восстановить класс модели, вы потеряете все атрибуты, которые вы применили к классу модели. Такой подход может быть очень неэффективным и подвержен утрате важных правил валидации.

Чтобы избежать эту проблему, вы можете добавить класс метаданных, содержащий атрибуты. Когда вы сопоставляете класс модели с классом метаданных, эти атрибуты применяются к модели. В этом подходе класс модели может быть восстановлен без потери всех атрибутов, которые были применены к классу метаданных.

В папке « Модели » добавьте класс с именем Metadata.cs .



Замените код в Metadata.cs следующим кодом.

using System;

using System.ComponentModel.DataAnnotations;

namespace ContosoSite.Models

{

public class StudentMetadata

{

[StringLength(50)]

[Display(Name="Last Name")]

public string LastName;

[StringLength(50)]

[Display(Name="First Name")]

public string FirstName;

[StringLength(50)]

[Display(Name="Middle Name")]

public string MiddleName;

[Display(Name = "Enrollment Date")]

public Nullable<System.DateTime> EnrollmentDate;

}

public class EnrollmentMetadata

{

[Range(0, 4)]

public Nullable<decimal> Grade;

}

}

Эти классы метаданных содержат все атрибуты проверки, которые вы ранее применяли к классам модели. Display атрибут используется для изменения значения, используемого для текстовых меток.

Теперь вы должны связать классы модели с классами метаданных.

В папке « Модели » добавьте класс с именем PartialClasses.cs .

Замените содержимое файла на следующий код.

using System;

using System.ComponentModel.DataAnnotations;

namespace ContosoSite.Models

{

[MetadataType(typeof(StudentMetadata))]

public partial class Student

{

}

[MetadataType(typeof(EnrollmentMetadata))]

public partial class Enrollment

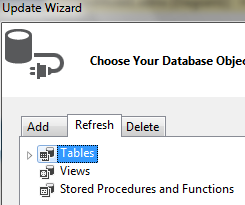
{

}

}

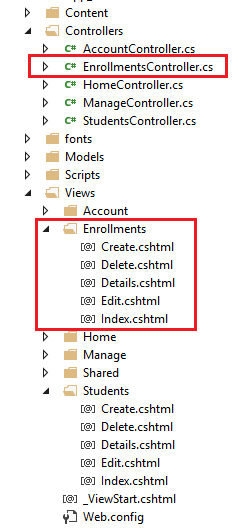
Обратите внимание, что каждый класс помечен как partial класс, и каждый соответствует имени и пространству имен как класс, который автоматически генерируется. Применяя атрибут метаданных к частичному классу, вы гарантируете, что атрибуты проверки данных будут применены к автоматически генерируемому классу. Эти атрибуты не будут потеряны при регенерации классов модели, потому что атрибут метаданных применяется в частичных классах, которые не восстанавливаются.

Чтобы восстановить автоматически созданные классы, откройте файл ContosoModel.edmx. Еще раз щелкните правой кнопкой мыши по поверхности дизайна и выберите « Обновить модель из базы данных» . Несмотря на то, что вы не изменили базу данных, этот процесс будет восстанавливать классы. На вкладке « Обновить » выберите « Таблицы» и « Готово» .



Сохраните файл ContosoModel.edmx, чтобы применить изменения.

Откройте файл Student.cs или файл Enrollment.cs и обратите внимание, что ранее не были добавлены атрибуты проверки данных, которые вы применяли ранее. Однако запустите приложение и обратите внимание, что правила проверки все еще применяются при вводе данных.



**Добавить ссылки на таблицы**

Чтобы облегчить вам переход к новым представлениям, вы можете добавить пару гиперссылок в представления Index для учащихся и учащихся. Откройте файл в Views / Home / Index.cshtml , который является домашней страницей вашего сайта. Добавьте следующий код ниже jumbotron.

CSHTMLкопия

<div>

@Html.ActionLink("List of students", "Index", "Students")<br />

@Html.ActionLink("List of enrollments", "Index", "Enrollments")

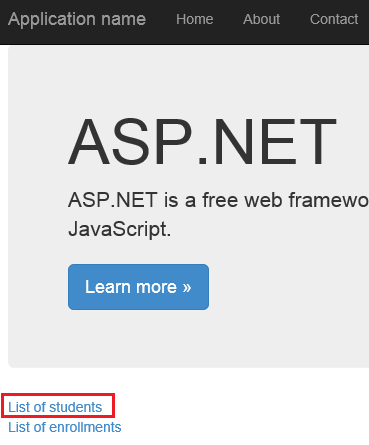
</div>

Для метода ActionLink первым параметром является текст, отображаемый в ссылке. Второй параметр - это действие, а третий параметр - имя контроллера. Например, первая ссылка указывает на действие индекса в StudentsController. Из этих значений строится фактическая гиперссылка. Первая ссылка в конечном итоге включает пользователей в файл Index.cshtml в папке Views / Students .

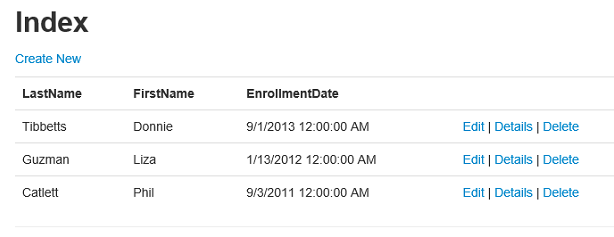
**Просмотр таблиц с данными**

Вы убедитесь, что код, добавленный в ваш проект, правильно отображает список учеников и позволяет пользователям редактировать, создавать или удалять записи учащихся в базе данных.

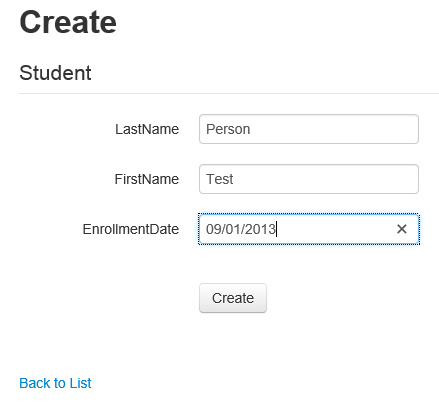
Щелкните правой кнопкой мыши файл Views / Home / Index.cshtml и выберите « Просмотр в браузере» . На этой странице щелкните ссылку для списка учеников.



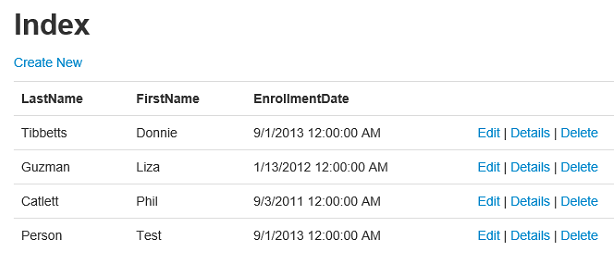
На этой странице обратите внимание на список учащихся и ссылки для изменения этих данных.



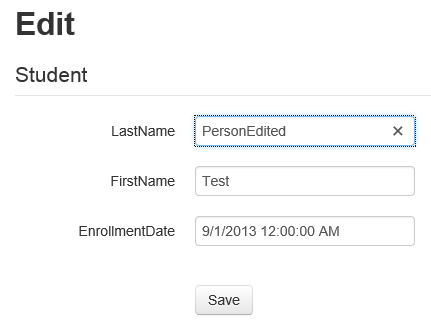
Нажмите ссылку « Создать новую» и укажите некоторые значения для нового ученика.



Нажмите « Создать» и обратите внимание, что новый ученик добавлен в ваш список.

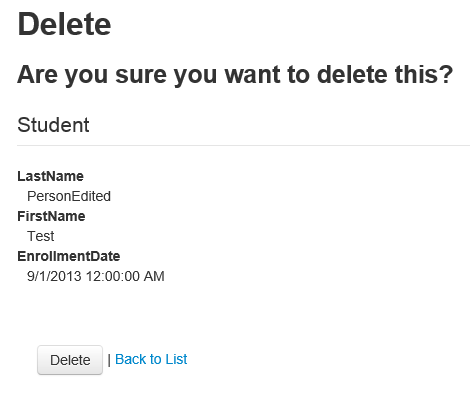


Выберите ссылку « Изменить» и измените некоторые значения для учащегося.



Нажмите « Сохранить» и обратите внимание, что запись студента была изменена.

Наконец, выберите ссылку « Удалить» и подтвердите, что вы хотите удалить запись, нажав кнопку « Удалить» .

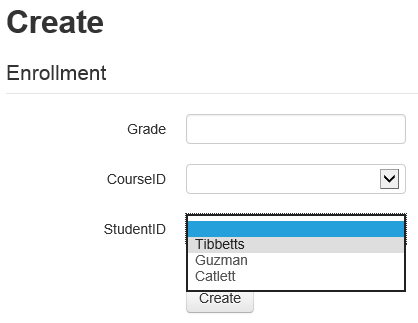


Без написания кода вы добавили представления, которые выполняют общие операции с данными в таблице Student.

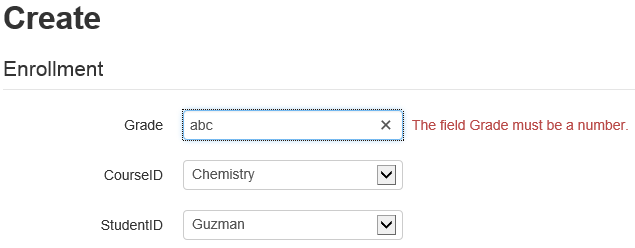
Возможно, вы заметили, что текстовая метка для поля основана на свойстве базы данных (например, LastName ), которое не обязательно означает, что вы хотите отображать на веб-странице. Например, вы можете предпочесть, чтобы метка была Фамилия . Вы устраните эту проблему с отображением позже в учебнике.

**Проверка создания новых элементов**

Ваша база данных включает отношения «один ко многим» между таблицами «Студент» и «Зачисление» и отношения «один ко многим» между таблицами «Курс» и «Зачисление». Представления для регистрации правильно обрабатывают эти отношения. Перейдите на главную страницу своего сайта и выберите ссылку « Список ссылок», а затем ссылку « Создать новое» . В представлении отображается форма для создания новой записи регистрации. В частности, обратите внимание, что форма содержит два раскрывающихся списка, которые заполняются значениями из связанных таблиц.



Кроме того, валидация предоставленных значений автоматически применяется на основе типа данных поля. При попытке предоставить несовместимое значение отображается сообщение об ошибке.



## Добавление ссылки сортировки столбцов на страницу индекса учащихся (нужно на 3)

Добавление сортировки на страницу указателя учащихся изменим Index метод StudentController и добавьте код в Student Индексируйте представление.

### Добавление функции сортировки в метод Index

* В Controllers\StudentController.cs, замените Index метод следующим кодом:

public ActionResult Index(string sortOrder)

{

ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name\_desc" : "";

ViewBag.DateSortParm = sortOrder == "Date" ? "date\_desc" : "Date";

var students = from s in db.Students

select s;

switch (sortOrder)

{

case "name\_desc":

students = students.OrderByDescending(s => s.LastName);

break;

case "Date":

students = students.OrderBy(s => s.EnrollmentDate);

break;

case "date\_desc":

students = students.OrderByDescending(s => s.EnrollmentDate);

break;

default:

students = students.OrderBy(s => s.LastName);

break;

}

return View(students.ToList());

}

Этот код принимает параметр sortOrder из строки запроса в URL. Значение строки запроса, предоставляемые ASP.NET MVC в качестве параметра метода действия. Параметр — это строка, которая является «Name» или «Date» (необязательно) за которым следует символ подчеркивания и строки «desc», чтобы указать порядок по убыванию. По умолчанию используется порядок сортировки по возрастанию.

При первом запросе страницы Index строка запроса отсутствует. Учащиеся отображаются в порядке возрастания значений LastName, который используется по умолчанию, установленные по фамилиям в switch инструкции. Когда пользователь щелкает гиперссылку заголовка столбца, в строку запроса подставляется соответствующее значение параметра sortOrder.

Две  переменные ViewBag  используются, чтобы представление можно настроить гиперссылок в заголовки столбцов с соответствующими значениями строки запроса:

C#Копировать

ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name\_desc" : "";

ViewBag.DateSortParm = sortOrder == "Date" ? "date\_desc" : "Date";

Это тернарные условные операторы. Первый из них указывает, что если sortOrder параметр равен null или пусто, ViewBag.NameSortParm должно быть присвоено «имя\_desc»; в противном случае задается пустая строка. Следующие два оператора устанавливают гиперссылки в заголовках столбцов в представлении следующим образом:

| **Текущий порядок сортировки** | **Гиперссылка "Last Name" (Фамилия)** | **Гиперссылка "Date" (Дата)** |
| --- | --- | --- |
| "Last Name" (Фамилия) по возрастанию | по убыванию | по возрастанию |
| "Last Name" (Фамилия) по убыванию | по возрастанию | по возрастанию |
| "Date" (Дата) по возрастанию | по возрастанию | по убыванию |
| "Date" (Дата) по убыванию | по возрастанию | по возрастанию |

Данный метод использует [LINQ to Entities](https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities) позволяет выбрать столбец для сортировки. Код создает <xref:System.Linq.IQueryable%601> переменной перед switch инструкция, изменяет его в switchинструкции и вызовы ToList метод после switch инструкции. После создания и изменения переменных IQueryable запрос в базу данных не отправляется. Запрос не выполняется, пока вы не преобразуете IQueryable объект в коллекцию путем вызова метода, например ToList. Таким образом, этот код вызывает в одном запросе, не выполняется до return View инструкции.

Вместо написания различных операторов LINQ для каждого порядка сортировки можно динамически создать оператор LINQ.

### Добавление гиперссылок в заголовки столбцов в представление Student index

1. В Views\Student\Index.cshtml, замените <tr> и <th> элементов для строки заголовков на следующий код:

<p>

@Html.ActionLink("Create New", "Create")

</p>

<table class="table">

<tr>

<th>

@Html.ActionLink("Last Name", "Index", new { sortOrder = ViewBag.NameSortParm })

</th>

<th>First Name

</th>

<th>

@Html.ActionLink("Enrollment Date", "Index", new { sortOrder = ViewBag.DateSortParm })

</th>

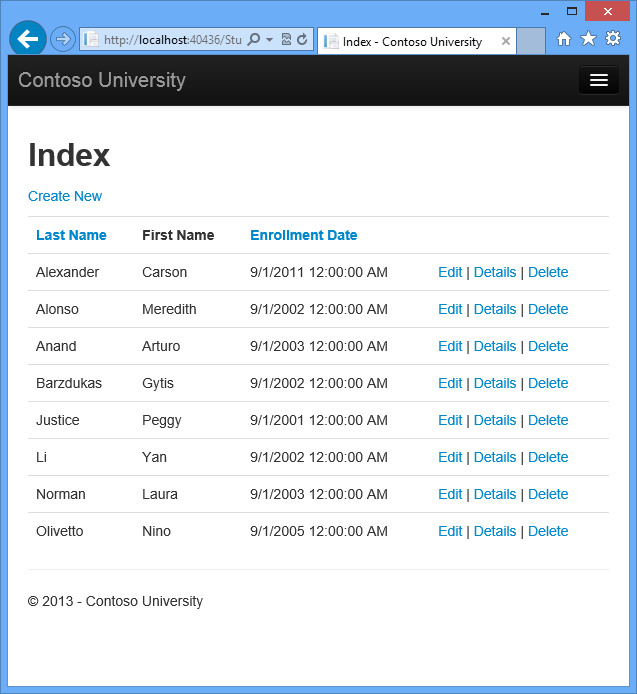
<th></th>

</tr>

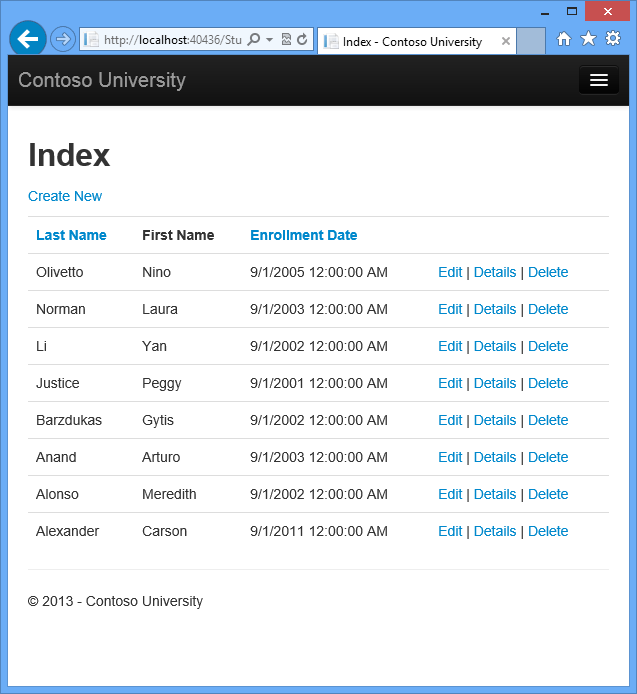
@foreach (var item in Model) {

Этот код использует сведения в ViewBag свойства для настройки гиперссылок с использованием соответствующего запроса строковые значения.

1. Откройте страницу и щелкните **Фамилия** и **Дата регистрации** заголовки столбцов, чтобы убедится, что Сортировка работает.



После того как вы щелкнете заголовок **Фамилия**, учащиеся отображаются по убыванию по фамилии.



## Добавление поля поиска на страницу индекса учащихся (нужно на 3)

Для добавления фильтра на страницу индекса учащихся, добавьте в представление текстовое поле и кнопку отправки и внести соответствующие изменения в Index метод. Текстовое поле позволяет ввести строку для поиска в имена полей имени и фамилии.

### Добавление функций фильтрации в метод Index

* В Controllers\StudentController.cs, замените Index метод (изменения выделены) следующим кодом:

public ViewResult Index(string sortOrder, string searchString)

{

ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name\_desc" : "";

ViewBag.DateSortParm = sortOrder == "Date" ? "date\_desc" : "Date";

var students = from s in db.Students

select s;

if (!String.IsNullOrEmpty(searchString))

{

students = students.Where(s => s.LastName.Contains(searchString)

|| s.FirstMidName.Contains(searchString));

}

switch (sortOrder)

{

case "name\_desc":

students = students.OrderByDescending(s => s.LastName);

break;

case "Date":

students = students.OrderBy(s => s.EnrollmentDate);

break;

case "date\_desc":

students = students.OrderByDescending(s => s.EnrollmentDate);

break;

default:

students = students.OrderBy(s => s.LastName);

break;

}

return View(students.ToList());

}

Код добавляет searchString параметр в Index метод. Значение строки поиска получается из текстового поля, которое мы добавим в представление Index. Он также добавляет where в оператор LINQ, которое отбирает только студентов, чье имя или Фамилия содержат строку поиска. Оператор, который добавляет <xref:System.Linq.Queryable.Where%2A> выполняется только в том случае, если отсутствует значение для поиска.

**Примечание**

Во многих случаях можно вызвать тот же метод, либо набор сущностей Entity Framework, либо как метода расширения для коллекции в памяти. Обычно получаются такие же результаты, но в некоторых случаях результат может отличаться.

Например, реализация .NET Framework Contains метод возвращает все строки, передает пустую строку, когда поставщик Entity Framework для SQL Server Compact 4.0 не возвращает пустые строки. Поэтому код в примере (размещение Where инструкции внутри if инструкции) гарантирует, что вы получите те же результаты для всех версий SQL Server. Кроме того, реализация .NET Framework Contains метод по умолчанию выполняет сравнение с учетом регистра, но поставщики Entity Framework SQL Server по умолчанию выполняют сравнения без учета регистра. Таким образом, вызовом ToUpper  производится проверка явно регистронезависимым способом и гарантирует, что результаты не изменятся при изменении кода позднее, которое будет возвращать IEnumerable , а не IQueryable объекта. (При вызове метода Contains коллекции IEnumerable выполняется реализация .NET Framework; при вызове этого же метода у объекта IQueryable выполняется реализация поставщика базы данных.)

Обработка NULL также может быть разным для разных поставщиков баз данных или при использовании IQueryable сравниваемый объект при использовании IEnumerable коллекции. Например, в некоторых сценариях Where условия, такие как table.Column != 0 не может возвращать столбцы, имеющие null как значение.

### Добавление поля поиска в представление Student index

1. В Views\Student\Index.cshtml, добавьте выделенный код непосредственно перед открытием table тега для создания заголовка, текстового поля и кнопки **поиска**.

<p>

@Html.ActionLink("Create New", "Create")

</p>

@using (Html.BeginForm())

{

<p>

Find by name: @Html.TextBox("SearchString")

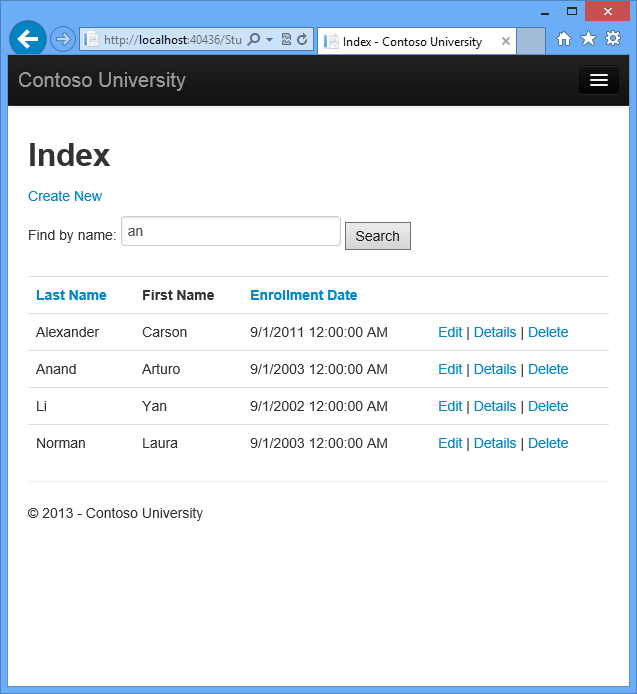
<input type="submit" value="Search" /></p>

}

<table>

<tr>

1. Запустить эту страницу, введите строку поиска и нажмите кнопку **поиска** для проверки работы фильтра.



## Создание страницы About со статистикой студентов (не нужно)

Веб сайт для университета Contoso на странице About должен отображать количество поданных заявок учащихся на каждую дату регистрации. Для этого понадобится группировка и выполнение простых расчетов в группах. Для выполнения этой задачи нам потребуется следующее:

* Создать класс модели представления для данных, которые необходимо передать в представление.
* Изменить метод About  в контроллере Home .
* Изменить About представления.

### Создание модели представления

Создайте папку *ViewModels*  в папке проекта. В этой папке добавьте файл класса *EnrollmentDateGroup.cs* и замените код шаблона следующим кодом:

using System;

using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.ViewModels

{

public class EnrollmentDateGroup

{

[DataType(DataType.Date)]

public DateTime? EnrollmentDate { get; set; }

public int StudentCount { get; set; }

}

}

### Изменение контроллера Home

1. В HomeController.cs, добавьте следующий using инструкций в верхней части файла:

using ContosoUniversity.DAL;

using ContosoUniversity.ViewModels;

1. Добавьте переменную класса для контекста базы данных сразу после открывающей фигурной скобки для класса:

public class HomeController : Controller

{

private SchoolContext db = new SchoolContext();

1. Замените метод About следующим кодом:

public ActionResult About()

{

IQueryable<EnrollmentDateGroup> data = from student in db.Students

group student by student.EnrollmentDate into dateGroup

select new EnrollmentDateGroup()

{

EnrollmentDate = dateGroup.Key,

StudentCount = dateGroup.Count()

};

return View(data.ToList());

}

Запрос LINQ группирует записи из таблицы студентов по дате зачисления, вычисляет число записей в каждой группе и сохраняет результаты в коллекцию объектов моделей представления EnrollmentDateGroup.

1. Добавление Dispose метод:

protected override void Dispose(bool disposing)

{

db.Dispose();

base.Dispose(disposing);

}

### Изменение представления About

1. Замените код в Views\Home\About.cshtml файла следующим кодом:

@model IEnumerable<ContosoUniversity.ViewModels.EnrollmentDateGroup>

@{

ViewBag.Title = "Student Body Statistics";

}

<h2>Student Body Statistics</h2>

<table>

<tr>

<th>

Enrollment Date

</th>

<th>

Students

</th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.EnrollmentDate)

</td>

<td>

@item.StudentCount

</td>

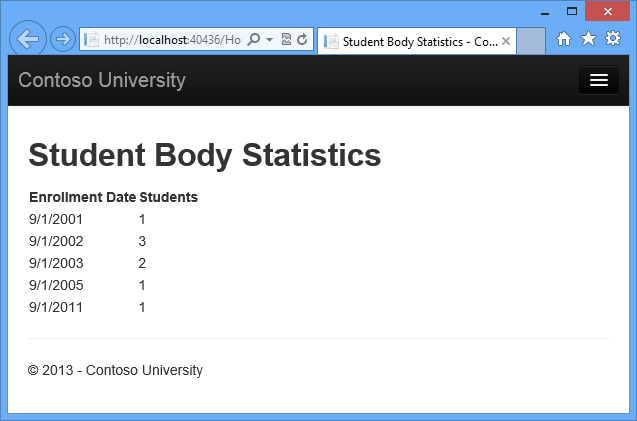
</tr>

}

</table>

1. Запустите приложение и нажмите меню About.

Количество студентов, зачисленных отображает в таблице.



**Работа с изображениями (нужно на 4)**

1. с помощью запросов sql На update обновить поле таблицы с image

update Buses

set bus\_picture = (SELECT MyImage.\* from Openrowset(Bulk 'D:\3.png', Single\_Blob) MyImage) where id\_bus = 3

2. в представления таблиц (INDEX.CSHTML, Details.cshtml), имеющей image, добавить элемент управления:

<td>

@{ byte[] photo = item.bus\_picture;

string imageSrc = null;

if (photo != null)

{

MemoryStream ms = new MemoryStream();

ms.Write(photo, 0, photo.Length);

string imageBase64 = Convert.ToBase64String(ms.ToArray());

imageSrc = string.Format("data:image/jpg;base64,{0}", imageBase64);

string base64String = Convert.ToBase64String(photo, 0, photo.Length);

}

}

<img src="@imageSrc" alt="Image" width="150" height="100" />

</td>

3. В Create.cshtml

Добавить

@model WebExASP\_Full.Buses

@{

ViewBag.Title = "Create";

}

<h2>Новый атвобус</h2>

@using (Html.BeginForm(Html.BeginForm("Create", "Buses", null, FormMethod.Post, new {enctype = "multipart/form-data"})))

И элементы управления:

<div class="form-group">

@Html.LabelFor(model=>model.bus\_picture, new { @class = "control-label col-md-2" })

<div class="col-md-10">

<input type="file" name="upload" value="Обзор..." />

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Добавить" class="btn btn-default" />

</div>

</div>

В контроллере этого представления изменить метод public ActionResult Create([Bind(….

public ActionResult Create([Bind(Include="id\_bus,name\_bus,id\_model,date\_bus,quant,bus\_picture,bus\_price")] Buses buses, HttpPostedFileBase upload)

{

if (ModelState.IsValid)

{

if (upload != null && upload.ContentLength > 0)

{

using (var reader = new System.IO.BinaryReader(upload.InputStream))

{

buses.bus\_picture = reader.ReadBytes(upload.ContentLength);

}

}

db.Buses.Add(buses);

db.SaveChanges();

return RedirectToAction("Index");

}

ViewBag.id\_model = new SelectList(db.ModelsofBuses, "id\_model", "name\_model", buses.id\_model);

return View(buses);

}

4. в Edit.cshtml

@model WebExASP\_Full.Buses

@{

ViewBag.Title = "Edit";

}

<h2>Edit</h2>

@using (Html.BeginForm(Html.BeginForm("Edit", "Buses", null, FormMethod.Post, new { enctype = "multipart/form-data" })))

Элемент управления:

<div class="editor-label">Фото</div>

<div class="editor-field">

@if (Model.bus\_picture == null)

{

@:None

}

else

{

byte[] photo = Model.bus\_picture;

string imageSrc = null;

if (photo != null)

{

MemoryStream ms = new MemoryStream();

ms.Write(photo, 0, photo.Length);

string imageBase64 = Convert.ToBase64String(ms.ToArray());

imageSrc = string.Format("data:image/jpg;base64,{0}", imageBase64);

string base64String = Convert.ToBase64String(photo, 0, photo.Length);

}

<img width="250" height="150" src="@imageSrc" />

}

<div>Выбрать новую фотографию: <input type="file" name="upload" /></div>

</div>

В контроллере этого представления изменить метод public ActionResult Edit([Bind(Include =…

try

{

if (ModelState.IsValid)

{

db.Entry(buses).State = EntityState.Modified;

if (upload != null && upload.ContentLength > 0)

{

using (var reader = new System.IO.BinaryReader(upload.InputStream))

{

buses.bus\_picture = reader.ReadBytes(upload.ContentLength);

}

db.SaveChanges();

}

else

{

db.Entry(buses).Property(m => m.bus\_picture).IsModified = false;

db.SaveChanges();

}

return RedirectToAction("Index");

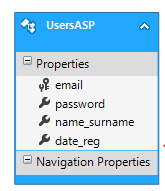
}

return View(buses);

}

catch(Exception e) {}

**Регистрация (нужно на 5)**



1. создаем ХП к базе на Insert

CREATE PROCEDURE [dbo].[Insert\_User]

@Email NVARCHAR(30) ,

@Password NVARCHAR(20),

@Nickname NVARCHAR(20)

AS

BEGIN

SET NOCOUNT ON;

IF EXISTS(SELECT email FROM UsersASP WHERE email = @Email)

BEGIN

SELECT -1 AS UserId -- Username exists.

END

ELSE

BEGIN

INSERT INTO [UsersASP]

([email]

,[password]

,[name\_surname]

,[date\_reg])

VALUES

(@Email

,@Password

,@Nickname

,GETDATE())

select 1

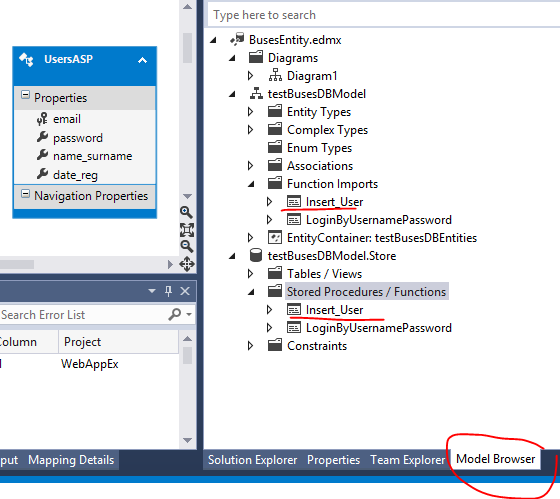
--SELECT SCOPE\_IDENTITY() AS UserId -- UserId

END

END

2. проверяем выполнение ХП на сервере с помощью EXECUTE!!!!

3. Добавляем ХП в модель, она отобразится как функция (а не свойство), ее можно увидеть только в ModelBrowser



4. Редактируем класс RegisterViewModel в AccountViewModels.cs в зависимости от нужных полей

5. редактируем страницу регистрации Register.cshtml в зависимости от нужных полей

6. В AccountController.cs редактируем асинхронную функцию Register например так

try

{

// Verification.

if (ModelState.IsValid)

{

// Initialization.

var regInfo = this.DBTest.Insert\_User(model.UserName, model.Password,model.NameSurname).ToList();

// Verification.

if (regInfo != null && Convert.ToInt32(regInfo[0]) !=-1)

{

//SAVING CHANGES TO DATABASE

**DBTest.SaveChanges();**

// return RedirectToAction("Index", "Home");

return RedirectToAction("Login", "Account");

}

else

{

ModelState.AddModelError(string.Empty, "ПОЛЬЗОВАТЕЛЬ С ТАКИМ ИМЕНЕМ (E-MAIL) УЖЕ СУЩЕСТВУЕТ.");

}

}

}

catch (Exception ex)

{ }

return this.View(model);

**Авторизация (нужно на 5)**

**1. ХП на проверку логина\пароля из базы (имя ХП можете дать ЛЮБОЕ)**

CREATE PROCEDURE [dbo].[LoginByUsernamePassword]

@username varchar(50),

@password varchar(50)

AS

BEGIN

SELECT email, UsersASP.password

FROM UsersASP

WHERE email = @username

AND UsersASP.password = @password

END

**2. Обновили EDM!**

**3. В "RouteConfig.cs" в папке "App\_Start" поменять**

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Account", action = "Login", id = UrlParameter.Optional });

по дефолту будет ВХОД В АКК

**4. В "Startup.Auth.cs"**

app.UseCookieAuthentication(new CookieAuthenticationOptions

{

AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,

LoginPath = new PathString("/Account/Login"),

LogoutPath = new PathString("/Account/LogOff"),

ExpireTimeSpan = TimeSpan.FromMinutes(5.0)

});

**5. В "Global.asax.cs" добавить в конец метода "Application\_Start":**

AntiForgeryConfig.UniqueClaimTypeIdentifier = ClaimTypes.Name;

**6. В "\_Layout.cshtml" все страницы-представления убрать под IF**

@if (User.Identity.IsAuthenticated)

{

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>@Html.ActionLink("About", "About", "Home")</li>

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>

<li>@Html.ActionLink("Buses", "Index", "Buses")</li>}

}

**7. В "Index.cshtml" под "Views->Home" для класса jumbotron**

<div class="jumbotron">

<h1>Приветики</h1>

<p class="lead">Login from "@User.Identity.Name" Account.</p>

</div>

**8. В "HomeController.cs" в папке "**Controller**"  оставить только:**

[Authorize]

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

**9. Открыть AccountViewModels.cs**

Внимательно посмотреть на класс LoginViewModel!!!!

**10. в "AccountController.cs" в папке "Controller"**

**объявить Entity**

testEntities1 DBTest = new testEntities1();

**Изменить метод public ActionResult Login(string returnUrl)**

[AllowAnonymous]

public ActionResult Login(string returnUrl)

{

try

{

// Verification.

if (this.Request.IsAuthenticated)

{

// Info.

return this.RedirectToLocal(returnUrl);

}

}

catch (Exception ex)

{

// Info

}

// Info.

return this.View();

}

**Изменить метод public async Task<ActionResult> Login(LoginViewModel model, string returnUrl) (можно убрать асинхронный)**

public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)

{

try

{

// Verification.

if (ModelState.IsValid)

{

// Initialization.

var loginInfo = this.DBTest.LoginByUsernamePassword(model.UserName, model.Password).ToList();

// Verification.

if (loginInfo != null && loginInfo.Count() > 0)

{

// Initialization.

var logindetails = loginInfo.First();

// Login In.

this.SignInUser(logindetails.email, false);

// Info.

return this.RedirectToLocal(returnUrl);

}

else

{

ModelState.AddModelError(string.Empty, "Invalid username or password.");

}

}

}

catch (Exception ex)

{

}

// If we got this far, something failed, redisplay form

return this.View(model);

}

**Изменить метод public ActionResult LogOff()**

public ActionResult LogOff()

{

try

{

// Setting.

var ctx = Request.GetOwinContext();

var authenticationManager = ctx.Authentication;

// Sign Out.

authenticationManager.SignOut();

}

catch (Exception ex)

{

// Info

}

// Info.

return this.RedirectToAction("Login", "Account");

}

**добавить метод private void SignInUser(string username, bool isPersistent)**

private void SignInUser(string username, bool isPersistent)

{

// Initialization.

var claims = new List<Claim>();

try

{

// Setting

claims.Add(new Claim(ClaimTypes.Name, username));

var claimIdenties = new ClaimsIdentity(claims, DefaultAuthenticationTypes.ApplicationCookie);

var ctx = Request.GetOwinContext();

var authenticationManager = ctx.Authentication;

// Sign In.

authenticationManager.SignIn(new AuthenticationProperties() { IsPersistent = isPersistent }, claimIdenties);

}

catch (Exception ex)

{

// Info

throw ex;

}

}

**Изменить метод private ActionResult RedirectToLocal(string returnUrl)**

private ActionResult RedirectToLocal(string returnUrl)

{

try

{

// Verification.

if (Url.IsLocalUrl(returnUrl))

{

// Info.

return this.Redirect(returnUrl);

}

}

catch (Exception ex)

{

// Info

}

// Info.

return this.RedirectToAction("Index", "Home");

}

