Министерство образования и науки Республики Казахстан

ВОСТОЧНО-КАЗАХСТАНСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. Д. СЕРИКБАЕВА

Г. Жомартқызы

Конспект лекции

Проектирование и разработка распределенных Web-приложений

Специальность: 5В070400 «Вычислительная техника и программное обеспечение»

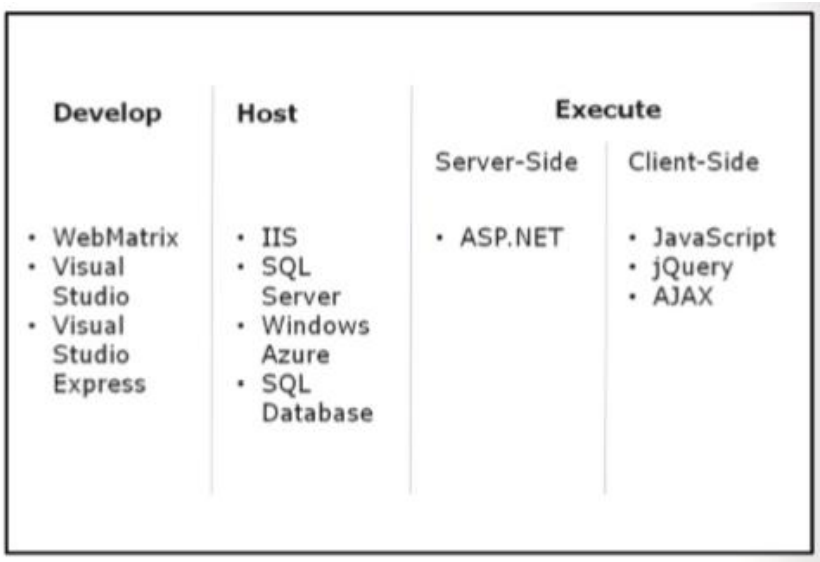
Усть-Каменогорск

2017

1 ЛЕКЦИЯ. ВВЕДЕНИЕ В ASP.NET MVC

Microsoft Web Technologies

Microsoft предоставляет широкий спектр технологий для создания богатых веб-приложений и публикации их на локальных сетях или в Интернете.



1)программные средства разработки

- WebMatrix 2,
- Microsoft Visual Studio
- Microsoft Visual Studio Express 2012 for Web

2)размещение веб-приложения.

- Microsoft Internet Information Server,
- Windows Azure

Windows Azure - облачная платформа, обеспечивающая услуги по созданию, развертыванию, и управлению веб-приложений через центры обработки данных Microsoft.

Кроме того, не нужно беспокоиться о создании масштабируемой инфраструктуры, потому что Windows Azure автоматически добавляет ресурсы, необходимые для расширения веб-сайта.

3) технология выполнения кода (Code Execution Technologies)

- на стороне веб-сервера
- на стороне веб-браузера пользователя

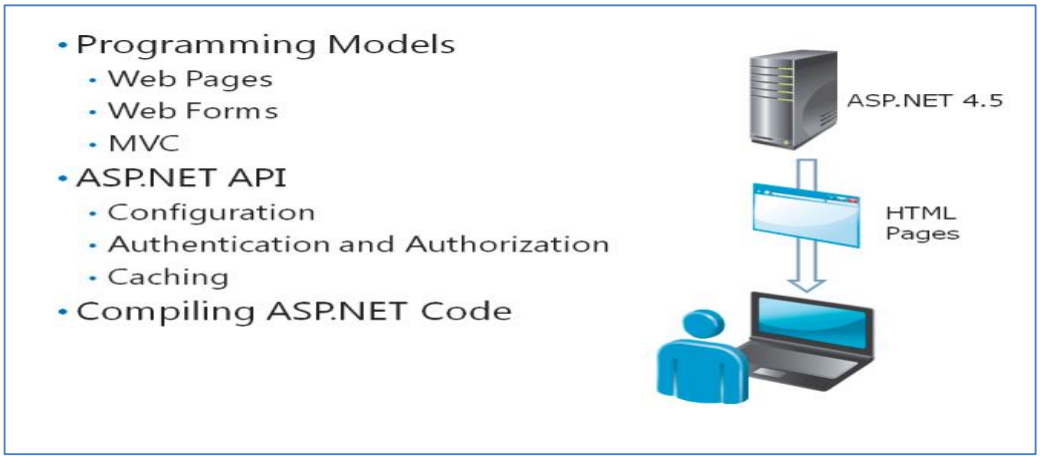
1.1 Введение

Платформа ASP.NET MVC представляет собой фреймворк для создания сайтов и веб-приложений с помощью реализации паттерна MVC.

Обзор ASP.NET

Вы можете использовать ASP.NET 4.5 для разработки на основе баз данных, высоко-функциональных, масштабируемых и динамических веб-приложений (использующие код стороне клиента и на стороне сервера).

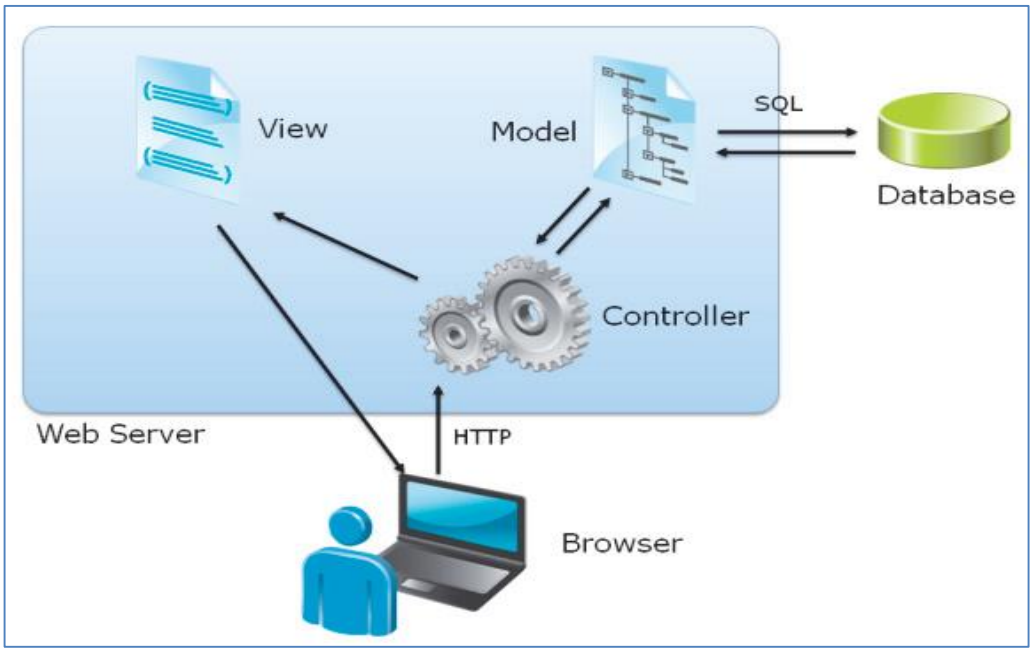
Вы можете создать множество видов веб-приложений ASP.NET, например, веб-порталы, онлайн торговые сайты, блоги, вики.



Модели программирования:

- Web Pages
- Web Forms
- MVC

Концепция паттерна (шаблона) MVC (model - view - controller) предполагает разделение приложения на три компонента:



Контроллер (controller) представляет класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления.

Представление (view) - это собственно визуальная часть или пользовательский интерфейс приложения. Как правило, html-страница, которую пользователь видит, зайдя на сайт.

Модель (model) представляет класс, описывающий логику используемых данных.

Общую схему взаимодействия этих компонентов можно представить следующим образом:

В этой схеме модель является независимым компонентом - любые изменения контроллера или представления не затрагивают модель. Контроллер и представление являются относительно независимыми компонентами, и нередко их можно изменять независимо друг от друга.

Благодаря этому реализуется концепция **разделение ответственности,** в связи с чем легче построить работу над отдельными компонентами. Кроме того, вследствие этого приложение обладает лучшей тестируемостью. И если нам, допустим, важная визуальная часть или фронтэнд, то мы можем тестировать представление независимо от контроллера. Либо мы можем сосредоточиться на бэкэнде и тестировать контроллер.

Конкретные реализации и определения данного паттерна могут отличаться, но в силу своей гибкости и простоты он стал очень популярным в последнее время, особенно в сфере веб-разработки.

Свою реализацию паттерна представляет платформа ASP.NET MVC. 2013 год ознаменовался выходом новой версии ASP.NET MVC - MVC 5, а также релизом Visual Studio 2013, которая предоставляет инструментарий для работы с MVC5.

Хотя во многих аспектах MVC 5 не слишком сильно будет отличаться от MVC 4, многое из одной версии вполне применимо к другой, но в то же время есть и существенные отличия:

- В MVC 5 изменилась концепция аутентификации и авторизации. Вместо SimpleMembershipProvider была внедрена система ASP.NET Identity, которая использует компоненты OWIN и Katana.
- Для создания адаптивного и расширяемого интерфейса в MVC 5 используется css-фреймворк Bootstrap
- Добавлены фильтры аутентификации, а также появилась функциональность переопределения фильтров
- В MVC 5 также добавлены атрибуты маршрутизации

Это наиболее важные нововведения в MVC 5. Кроме того, есть еще ряд менее значимых, например, использование по умолчанию Entity Framework 6,

некоторые изменения при создании проекта (концепция One ASP.NET), дополнительные компоненты и т.д.
В любом случае все полученные при работе с MVC 4 навыки можно успешно применять при использовании MVC 5, учитывая, конечно, нововведения.

1.2 Структура проекта MVC 5

Весь этот функционал обеспечивается следующей структурой проекта:

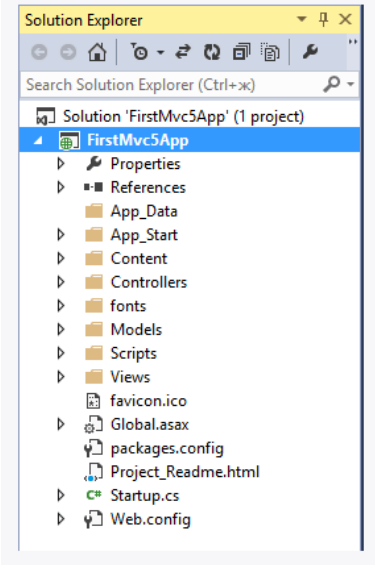


Рисунок 1

Вкратце рассмотрим, для чего нужны все эти папки и файлы.

- **App_Data**: содержит файлы, ресурсы и базы данных, используемые приложением
- **App_Start**: хранит ряд статических файлов, которые содержат логику инициализации приложения при запуске
- **Content**: содержит вспомогательные файлы, которые не включают код на c# или javascript, и которые развертываются вместе с приложением, например, файлы стилей css
- **Controllers**: содежит файлы классов контроллеров. По умолчанию в эту папку добавляются два контроллера - HomeController и AccountController
- **fonts**: хранит дополнительные файлы шрифтов, используемых приложением
- **Models**: содержит файлы моделей. По умолчанию Visual Studio добавляет пару моделей, описывающих учетную запись и служащих для аутентификации пользователя
- **Scripts**: каталог со скриптами и библиотеками на языке javascript
- **Views**: здесь хранятся представления. Все представления группируются по папкам, каждая из которых соответствует одному контроллеру. После обработки запроса контроллер отправляет одно из этих представлений клиенту. Также здесь имеется каталог Shared, который содержит общие для всех представления

- **Global.asax**: файл, запускающийся при старте приложения и выполняющий начальную инициализацию. Как правило, здесь срабатывают методы классов, определенных в папке App_Start
- **Startup.cs**: поскольку в приложении MVC 5 используются библиотеки, применяющие спецификацию OWIN, то данный файл организует связь между OWIN и приложением. (OWIN представляет спецификацию, описывающую взаимодействие между компонентами приложения)
- **Web.config**: файл конфигурации приложения

Конкретная структура каждого отдельного приложения, естественно, будет отличаться, а гибкость MVC позволяет изменять структуру, приспосабливая, ее к своим потребностям. Но описанные выше моменты будут общими для большинства проектов.

1.3 Жизненный цикл приложения ASP.NET MVC

Отправляя из браузера запрос к приложению, нередко мы практически моментально получаем нужный контент. Однако в реальности обработка запроса проходит кучу различных этапов. И в этой статье мы рассмотрим основные этапы жизненного цикла приложения на ASP.NET MVC 5.

1. После получения IIS запроса на обработку, при первом обращении к ресурсу, работающему под управлением CLR, создается объект класса **ApplicationManager**, который представляет домен приложения, внутри которого обрабатывается запрос. Домены приложений изолируют выполняющиеся приложения друг от друга. Уже внутри домена приложения создается объект класса**HostingEnvironment**, который предоставляет доступ к информации о приложении, в частности, он сообщает имя и каталог приложения.

2. После создания домена приложения также создаются и инициализируются такие объекты, как **HttpContext, HttpRequest** и **HttpResponse**, которые инкапсулируют всю информацию, связанную с текущим запросом к приложению.

3. На следующей стадии уже непосредственно запускается приложение, которое представляет экземпляр класса **HttpApplication**. Если в приложении определен файл *Global.asax*, то среда ASP.NET в качестве приложения создает объект класса **Global.asax**, который в свою очередь наследуется от класса HttpApplication. На этой же стадии происходит начальная инициализация приложения в методе Application_Start, который находится в файле Global.asax.cs.

4. После этого запрос начинает обрабатываться в конвейере класса HttpApplication - начинается собственно обработка запроса приложением. И первым шагом здесь является установка маршрута. Запрос перехватывается специальным HTTP-модулем под

названием **UrlRoutingModule**. Этот модуль выбирает маршрут, который соответствует входящему запросу.

Весь набор маршрутов определяется в файле RouteConfig.cs:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute( name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
        );
    }
}
```

А сама регистрация маршрутов происходит в методе Application_Start строкой RouteConfig.RegisterRoutes(RouteTable.Routes);. Для каждого приложения имеется только один объект коллекции маршрутов RouteTable.

5. Если модулю UrlRoutingModule удастся сопоставить запрос с одним из маршрутов в коллекции RouteTable, то затем UrlRoutingModule выбирает обработчик маршрутов сопоставленного маршрута - объект IRouteHandler. По умолчанию объект IRouteHandler представляет экземпляр класса MvcRouteHandler.

6. Затем у объекта IRouteHandler вызывается метод GetHandler, который возвращает объект интерфейса **IHttpHandler**, используемый для обработки запроса. По умолчанию в качестве IHttpHandlera используется объект класса MvcHandler.

7. У обработчика IHttpHandler вызывается метод ProcessRequest для обработки запросов:

```
protected internal virtual void ProcessRequest(HttpContextBase
httpContext)
{
    SecurityUtil.ProcessInApplicationTrust(delegate {
        IController controller;
        IControllerFactory factory;
        this.ProcessRequestInit(httpContext, out controller, out factory);
        try
        {
            controller.Execute(this.RequestContext);
        }
        finally
```

```
13      {
14          factory.ReleaseController(controller);
15      }
16  });
    }
```

8. На этом этапе уже происходит непосредственно создание контроллера, который представляет объект интерфейса IController. За создание контроллера отвечает объект интерфейса IControllerFactory, представляющий фабрику контроллеров. В качестве класса по умолчанию для фабрики контроллеров выступает
класс **System.Web.Mvc.DefaultControllerFactory**

9. После этого начинается собственно выполнение кода контроллера. После инициализации для запуска метода для обработки запроса контроллер вызывает метод InvokeAction():

```
1public virtual bool InvokeAction(ControllerContext
controllerContext, string actionName)
```

10. Далее привязчик модели (по умолчанию
класс **System.Web.Mvc.DefaultModelBinder**) извлекает данные из запроса, производит из преобразование, форматирование, валидацию и связывает их с определенными параметрами вызываемого метода.

11. При вызове метода в ASP.NET MVC 5 запускается фильтр аутентификации, представляющий объект интерфейса IAuthenticationFilter. Он аутентифицирует пользователя

12. После фильтра аутентификации в ASP.NET MVC 5 запускается фильтр авторизации, представляющий реализацию интерфейса IAuthorizationFilter. До MVC 5 фильтры аутентификации и авторизации объединялись в один фильтр и срабатывали вместе. Фильтр авторизации управляет доступом пользователя к определенным ресурсам на основе его учетных данных.

13. Перед непосредственным выполнением метода контроллера запускается метод OnActionExecuting фильтра действий. Фильтр действий представляет объект интерфейса IActionFilter. Кроме того, также после выполнения метода контроллера запускается другой метод фильтра действий - метод OnActionExecuted

14. Собственно выполнение метода контроллера. Он выполняет определенную логику и на выходе генерирует результат обработки в виде объекта ActionResult.

15. При обработке результата срабатывает другой фильтр - фильтр результатов - объект интерфейса IResultFilter. Его методOnResultExecuting срабатывает до обработки результата, а
метод OnResultExecuted после.

16. Генерация результата представляет создание объекта одного из классов результатов действий - ViewResult, ContentResult, FileResult, RedirectResult и др.

На заключительном этапе у каждого объекта ActionResult вызывается метод ExecuteResult, который обрабатывает результат действия. Для объектов ViewResult и PartialViewResult это выражается в поиске необходимого представления и его рендеринге движком представлений (как правило, движком Razor), который представляет объект интерфейса IViewEngine. Результат обработки в виде html-страницы посылается пользователю.

Для других объектов ActionResult (ContentResult, RedirectRefult и др.) происходит простая отправка результата в выходной поток.

17. И в конце пользователь получает результат обработки своего запроса.

Схематично весь процесс конвейера приложения на ASP.NET MVC можно представить так:

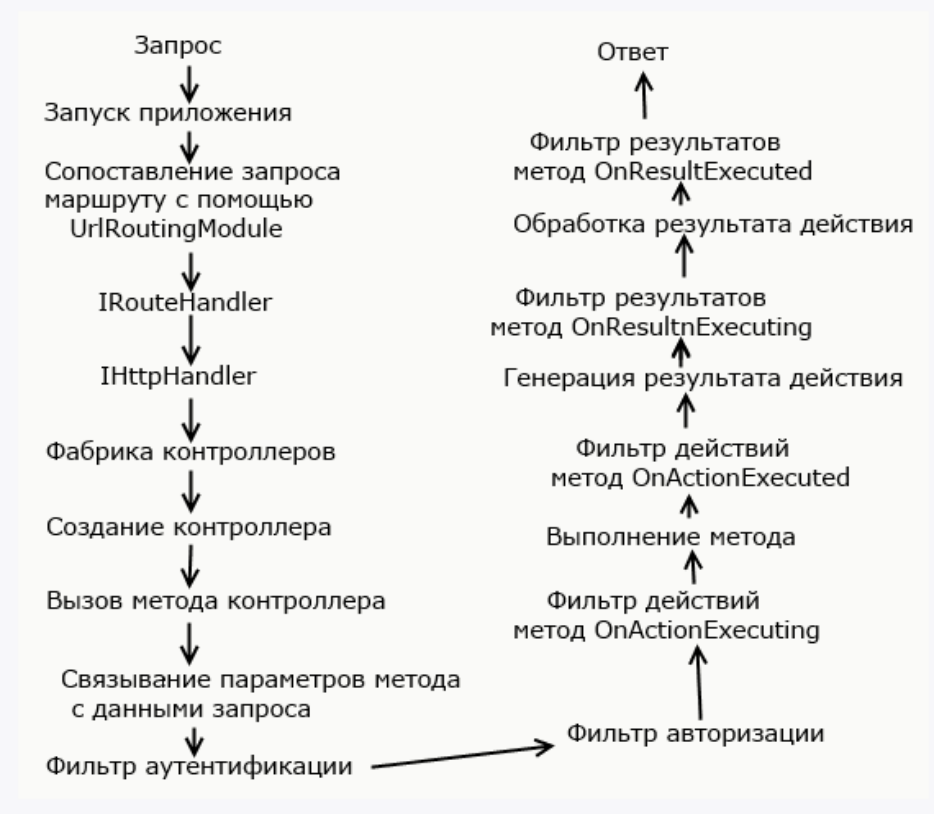


Рисунок 2

Литература

1. Фримен А., Сандерсон С. ASP.NET MVC 4 Framework с примерами на C# для профессионалов. 4-е издание, 2014. – 688 с.
2. Official microsoft learning product. Developing ASP.NET MVC 4 Web Applications, 2013, p 598.
3. Палермо Д., Богард Д. и др. ASP.NET MVC 4 в Действии, 2012. – 408 с.
4. Freeman A. Pro ASP.NET MVC 5, ISBN13: 978-1-4302-6529-0, 2013, p 832.
5. Введение в ASP.NET MVC 5, url: http://metanit.com/sharp/mvc5/1.1.php.
6. Трей Н. C# 2008: ускоренный курс для профессионалов. Пер с анг. - М.: ООО, “И.Д. Вильямс”, 2008. -576 с.
7. Official microsoft learning product. Programming in HTML5 with JavaScript and CSS3, 2012, p 492.
8. ASP.NET MVC 4 Content Map.
9. url:https://msdn.microsoft.com/en-us/library/gg416514(v=vs.108).aspx.

2 ЛЕКЦИЯ. ПРОЕКТИРОВАНИЕ ASP.NET MVC ВЕБ-ПРИЛОЖЕНИЙ

2.1 Введение

Microsoft ASP.NET MVC предлагает модель программирования для создания мощных и сложных веб-приложений. Тем не менее, все комплексные и, в частности, крупные проекты могут быть сложными и запутанными, для понимания. Без полного понимания целей проекта, невозможно разработать эффективное решение проблемы клиента.

Поэтому необходимо предварительно знать определение набора бизнес-требований и планирование MVC веб-приложений, удовлетворяющих указанных требований.

Подробный и точный план проекта гарантирует, что мощные возможности MVC используются наиболее эффективно для решения бизнес требований клиентов.

Целю проектирования asp.net mvc веб-приложений является:

- планирование общей архитектуры веб-приложения MVC;
- планирование моделей, контроллеров и представлений, необходимые для реализации набора функциональных требований.

2.2 Планирование этапа разработки проектов/ проектирования проекта

2.2.1 Методология разработки проекта

Разработка веб приложений зачастую является сложным процессом, который включает в себя множество разработчиков в разных командах, выполняющих различные роли.

Для организации процесса разработки и обеспечения гарантии совместной работы каждого в проекте используется широкий спектр методологии разработки.

Методология разработки описывает этапы разработки проекта, роли участников, и другие аспекты проекта.

Методологию разработки следует определить на ранней стадии проекта. Многими организациями часто для разработки проектов используются стандартные методологии.

Перечислим некоторые методологии разработки проектов:

- водопадная модель (the waterfall model),
- итерационная модель (the iterative development model), модель прототипов (the prototyping model),
- методология гибкой разработки программного обеспечения (the agile software development model),

- экстремальное программирование (extreme programming),
- разработка через тестирование (test-driven development).

Водопадная модель (the waterfall model)

Водопад модель является ранней методологией, определяющая следующие этапы проекта:

Анализ осуществимости (Feasibility analysis). На этом этапе, проектировщики и разработчики изучают и определят подходы и технологии, которые могут быть использованы для создания программного приложения.

Анализ потребностей (Requirement analysis.).

На этом этапе, планировщики и аналитики интервью у пользователей, менеджеров, администраторов и других заинтересованных сторон в приложении, чтобы определить их потребности.

Проектирования приложения (Application design). На этом этапе, планировщики, аналитики и разработчики записывают предлагаемое решение.

Кодирование и модульное тестирование (Coding and unit testing.). На этом этапе, разработчики создают код и тестируют компоненты, составляющте систему.

Интеграция и тестирование системы (Integration and system testing). На этом этапе, разработчики интегрируют созданные компоненты и проводят испытание системы в целом.

Развертывание и обслуживание. На этом этапе, разработчики и администраторы проводят развертывание решения, на данном этапе пользователи могут начинать использовать программное приложение.

Водопадная модель разбивает разрабатываемый проект в различные фазы с четким определением результатов для каждой фазы. Модель также подчеркивает важность тестирования. Тем не менее, клиент не сможет получить функциональную программу для обзора до конца разработки проекта. Поэтому по данной методологии при разработке сложно проводить изменения.

2.2.2 Сбор Требования

Для разработки проекта необходимо видение его решения. Видение может часто быть расплывчатым и требует подробного изучения до этапа разработки, чтобы учесть все требования заинтересованных сторон, разрабатываемого проекта. Эти требования могут быть двух типов:

1) *Функциональные требования*. Эти требования описывают: работу и реакции приложения на действия пользователей. Функциональные требования часто называют поведенческими требованиями. Они включают:

- *требования пользовательского интерфейса*. Эти требования описывают, как пользователь взаимодействует с приложением.
- *требования по применению*. Эти требования содержат описания работы пользователя с приложением.

- *бизнес-требования*. Эти требования описывают, как приложение будет выполнять бизнес-функций.

2) *Технические требования*. Эти требования описывают технические особенности применения и относятся к доступности, безопасности или производительности. Эти требования иногда называют нефункциональные (или не поведенческие) требования.

Как правило, проводится сбор требований через опрос заинтересованных сторон, таких, как пользователи, администраторы (других разработчиков, членов совета директоров, владельцев бюджетных и руководителей команд). Каждая из этих групп будет иметь различный набор приоритетов, которые приложение должно выполнять.

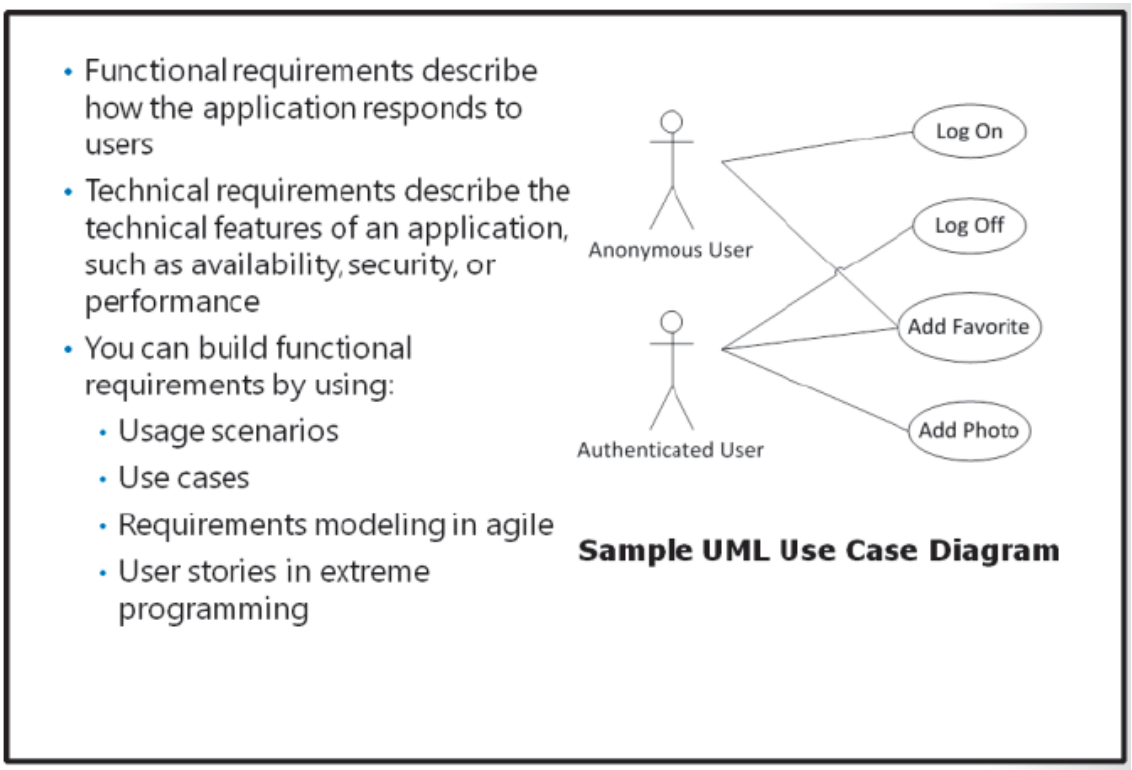


Рисунок 3

2.2.2 Сценарии использования и варианты использования.

Вариант использования похож на сценарии использования, но более в обобщенном виде. Варианты использования не включают имена пользователей или значения передаваемых данных. Они описывают несколько путей взаимодействия, которые зависят от того, что пользователь предоставляет в качестве входных данных. Ниже приведен простой пример:

1. Пользователь нажимает на ссылку Добавить фото на главном меню сайта.

2. Если пользователь является анонимным, выводится страница входа в систему, и пользователь предоставляет учетные данные.

3. Если учетные данные введены правильно, отображается вид CreatePhoto.

4. Пользователь вводит название.

5. Пользователь задает файл для загрузки фотографий.

7. Пользователь нажимает кнопку Загрузить.

8. Веб-приложение сохраняет новое фото и отображает Фотогалерея к пользователю.

Подобно словесным описаниям используются UML-диаграммы вариантов использования (Use Case diagrams) для записи вариантов использования (прецедентов) разрабатываемого веб-приложения.

Анализируя сценарии использования и варианты использования (прецеденты), вы можете определить функциональные требования всех типов.

Например, с помощью рассмотренных выше прецедентов можно выделить следующие требования пользовательского интерфейса: на веб-странице, которая позволяет пользователям создавать новую фотографию необходимо включить: название и описание текстовых полей, контроль входного файла для файла фотографии и кнопку загрузки, чтобы сохранить фотографию.

Анализируя вариант использования (сценарии использования) и прецеденты, вы можете определить функциональные требования всех типов. Например, из предыдущего варианта использования, можно определить следующее требования пользовательского интерфейса:

> Веб-страница, которая позволяет пользователям создавать новую фотографию необходимо включить название и описание текстовых полей, контроль входного файла для файла с фотографией, и кнопка загрузки, чтобы сохранить фотографию.

2.2.3 Планирование проектирования БД

После полного понимания функциональных и технических требований, разрабатываемого веб-приложения можно переходить к проектированию физической реализации приложения (physical implementation of the application).

Одним из самых важных физических объектов планирования является база данных (одна или несколько). Хотя не все веб-приложения используют базы данных для хранения информаций, они являются базовым объектом для большинства сайтов и используются обычно в большинстве проектов.

Этапы проектирования БД:

- логическое моделирование;
- физическая структура базы данных (физическая модель данных).

Логическое моделирования. Проектирование структуры данных на высоком уровне выполняется с помощью следующих диаграмм:

- диаграмма доменной модели UML или модель предметной области (UML Domain Model)
- диаграмма логической модели данных (Logical Data Model, LDM).

Диаграмма *доменной модели* или *концептуальная модель* данных описывает концептуальные объекты на высоком уровне, управляемые веб-

приложением. Например, в веб-приложении электронной коммерции, модель предметной области включает в себя понятия как клиенты, корзина покупок и продукты. Доменная модель не включает в себя подробные информаций о свойствах каждого понятия, но описывает отношения между понятиями. Доменная модель используется обычно для записи первоначальной беседы с заинтересованными сторонами.

В сущности, логическая модель ПО является моделью предметной области с дополнительными деталями. Диаграмма логической модели используется, чтобы заполнить более подробную информацию, например, свойств и типов данных, для понятий, которые вы определили в доменной модели.

Внимание: объекты в логической модели данных не соответствуют таблицам в базе данных.

Например, объект “виртуальная корзина” может отображать данные таблиц как из базы данных Клиенты, так из базы данных Продукты.

Физическая структура базы данных. Рассматриваются следующие объекты базы данных в плане проекта:

- таблицы
- просмотры.
- хранимые процедуры
- безопасность.

В UML схеме физической модели данных отображаются таблицы, столбцы, типы данных и отношения между таблицами.

2.2.4 Работа с администраторами баз данных.

Иногда, команда разработчиков имеет полный контроль над базой данных, расположенный в основе веб-приложения. Это происходит, например, когда организация небольшая или когда веб-приложение имеет отдельный сервер базы данных без каких-либо бизнес-критичных данных. Тем не менее, в крупных организациях или в проектах, где база данных хранит важную деловую информацию, может быть специальная группа администраторов баз данных (АБД). Администраторы БД, как правило, это высококвалифицированные специалисты в области проектирования и администрирования баз данных, их работа заключается в обеспечении целостности данных на основе политики хранения данных организации.

Если БД проекта находится под управлением команды администраторов БД, становится важным работать в сотрудничестве с ними. Поэтому разработчики должны проконсультироваться с администраторами БД для уточнения их требований. Администраторы БД часто накладывают список технических требований, которые другие заинтересованные стороны,

возможно, не понимают. Разработчиками выполняется создание и развертывание веб-приложения. Администраторы БД отвечают за создание баз данных на серверах или кластерах и назначение разрешений. Они существенно влияют на функционирование, предоставляемого веб-приложения.

2.2.5 Планирование распределенных приложений

Для небольшого веб-приложения с небольшим уровнем пользовательского трафика можно выбрать вариант размещения всех компонентов веб-приложения на одном сервере. Тем не менее, с ростом размера веб-приложения, часто используется **распределенное развертывание**, в котором различные серверы принимают отдельные компоненты приложения. **Распределенные веб-приложения** часто используют многоуровневую (многослойную) архитектуру:

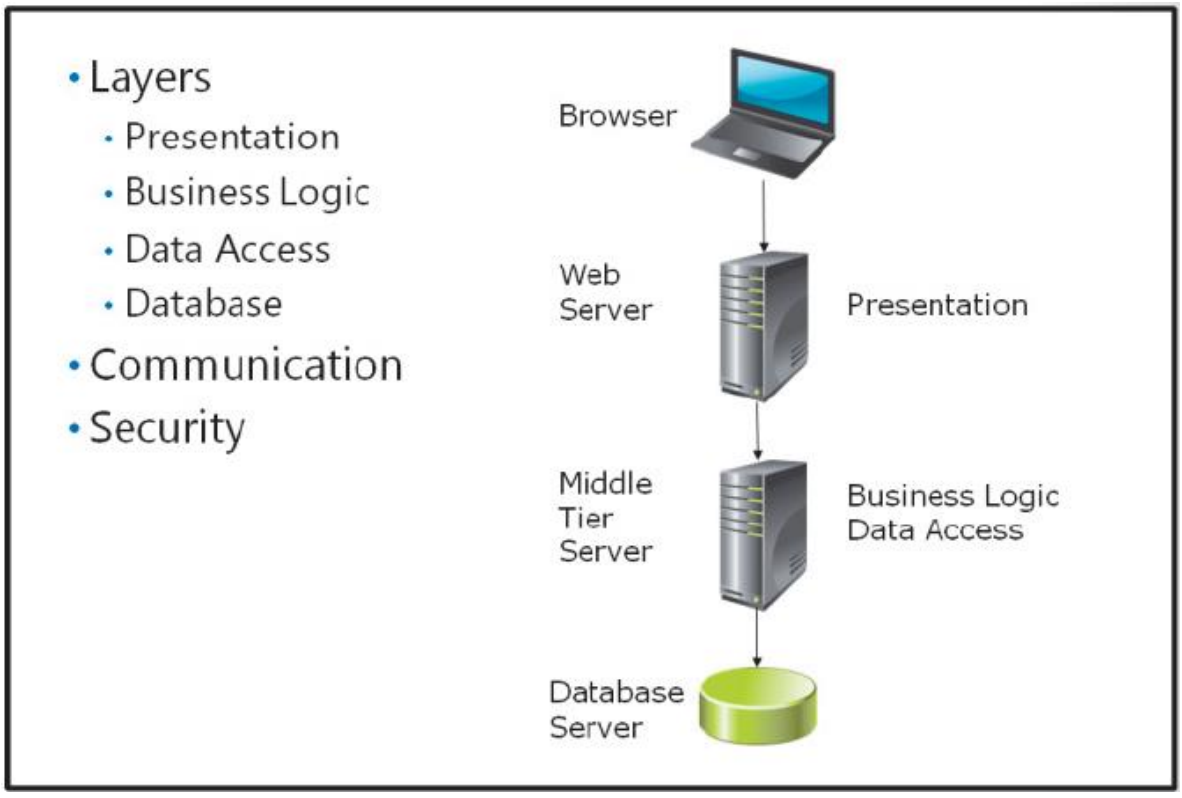


Рисунок 4 - Многоуровневая архитектура веб-приложения

1) *Уровень представления*. Компоненты в этом слое обеспечивают выполнение пользовательского интерфейса и логику представления. В веб-приложениях MVC Представления и Контроллеры выполняют презентационный слой.

2) Слой бизнес-логики. На этом уровне компонентами служат *высокоуровневые бизнес-объекты*, такие как продукты, или клиенты (классы). Если строится веб-приложение MVC, *Модели* веб-приложения составят слой *бизнес-логики*.

3) *Уровень доступа к данным*. Компоненты в этом слое реализуют:

- операции доступ к базе данных;

-абстрактные объекты базы данных (экземпляры классов), такие как таблицы из бизнес-объектов.

Например, бизнес-объект продукт может включать в себя данные из таблиц баз данных Products и StockLevels. Если вы строите веб-приложение MVC, модели часто выполняют как слой бизнес-логики, так и слои доступ к данным.

Однако, с практикой тщательной разработки и программирования можно реорганизовать код, чтобы отделить эти слои.

4) *Слой базы данных*. Этот слой представляет саму базу данных.

При реализации такой многоуровневой архитектуры веб-приложения, вы можете разместить каждый слой на отдельных серверах. Часто, например, слой презентации размещается на одном или нескольких серверах IIS, слои бизнес-логики и доступа к данным размещаются на выделенных серверах среднего уровня, а база данных размещается на выделенном SQL сервере. **Этот подход имеет следующие преимущества:**

- вы можете указать серверное оборудования, что позволяет точно соответствовать каждой роли. Например, сервер, на котором расположены компоненты бизнес-логики требует хороших ресурсов памяти и процессора.
- вы можете выделить несколько серверов для каждой роли для того, чтобы отказ на одном сервере не приводил к сбоям в обслуживании.
- только веб-серверы должны быть в подключены к интернет-зоне. Оба серверы среднего уровня и сервер баз данных могут быть защищены двумя брандмауэрами без прямого доступа в Интернет.
- в качестве альтернативы, вы можете разместить слой среднего уровня и базу данных на облачном сервисе, как WindowsAzure.

Вопрос: Если клиент просит вас обеспечить 95% доступность, является ли это функциональное требование или техническое требование?

3 ЛЕКЦИЯ. ВВЕДЕНИЕ В ASP.NET RAZOR

Синтаксиса Razor в Visual Studio

Razor не является языком программирования. Это язык разметки на стороне сервера.

Веб-страницы ASP.NET с синтаксисом Razor позволяют использовать простой синтаксис программирования для написания кода веб-страниц, при котором серверный код встраивается в HTML-код веб-страницы. Код Razor выполняется на сервере до отправки страницы в браузер. Этот серверный код может динамически создавать клиентское содержимое, т. е. генерировать разметку HTML или другое содержимое "на лету", а затем передавать его в браузер вместе со статическим HTML-кодом страницы.

Веб-страницы с синтаксисом Razor являются альтернативой веб-формам ASP.NET. В основе страниц веб-форм лежат элементы управления веб-сервера, которые автоматически создают HTML-код и эмулируют программную модель на основе событий, используемую в клиентских приложениях.Страницы Razor, напротив, больше похожи на обычные страницы HTML, на которых разработчик создает почти всю разметку HTML, а функциональность добавляется к этой разметке с помощью серверного кода. В общем случае страницы Razor занимают меньше места, чем страницы веб-форм.Из-за этого, а также из-за простоты синтаксиса, технология Razor может оказаться более легкой в освоении разработчиками, а скорость разработки динамических веб-страниц может увеличиться.

У веб-страниц с содержимым Razor имеется особое расширение файла — CSHTML или VBHTML.Сервер распознает эти расширения, запускает код, помеченный синтаксисом Razor, а затем отправляет полученную страницу в браузер.

Технология Razor была впервые применена на сайте Microsoft WebMatrix. В этом пошаговом руководстве показано, как использовать ASP.NET Razor при работе в Visual Studio.

Razor поддерживает как C # так и VB (Visual Basic).

Основные Синтаксические правила Razor в C#

1. кодовые блоки Razor заключаются в **@ {...}**
2. Встроенные выражения (переменные и функции) начинаются с **@**
3. Кодовое выражение заканчивается точкой с запятой
4. Переменные объявляются с помощью ключевого слова **var**
5. Строки заключаются в кавычки
6. C # код чувствителен к регистру
7. C # файлы имеют расширение .cshtml

Пример C#

```
<!-- Однострочный block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of myMessage is: @myMessage</p>

<!—Многострочный  block -->
@{
var greeting = "Welcome to our site!";
var weekDay = DateTime.Now.DayOfWeek;
var greetingMessage = greeting + " Here in Huston it is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

Работа с объектами.

Серверный код часто включает объекты. Объект "DateTime" является типичным встроенный объект ASP.NET, но объекты могут также быть самостоятельной определяется как веб-страница, текстовое поле, файл, запись в базе данных, и т.д.

Объекты могут иметь методы которые они могут выполнять. Запись из БД может иметь метод "Сохранить" и так далее.

Объекты также обладают свойствами, которые описывают их характеристики. Объект (запись) в БД может иметь свойства, например FirstName и свойство LastName (среди прочих).

Объект ASP.NET DateTime имеет свойство Now (как DateTime.Now), а свойство Now имеет свойство Day (как as DateTime.Now.Day). В приведенном ниже примере показано, как получить доступ к некоторым свойствам объекта DateTime:

```
<table border="1">
<tr>
<th width="100px">Name</th>
<td width="100px">Value</td>
</tr>
<tr>
<td>Day</td><td>@DateTime.Now.Day</td>
</tr>
<tr>
<td>Hour</td><td>@DateTime.Now.Hour</td>
</tr>
<tr>
<td>Minute</td><td>@DateTime.Now.Minute</td>
</tr>
<tr>
<td>Second</td><td>@DateTime.Now.Second</td>
</tr>
</td>
</table>
```

Условия if-else.

Важной особенностью динамических веб-страниц является возможность определения действий с помощью условий. Пример

```
@{
var txt = "";
if(DateTime.Now.Hour > 12)
  {txt = "Good Evening";}
else
  {txt = "Good Morning";}
}
<html>
<body>
<p>The message is @txt</p>
</body>
</html>
```

Чтение пользовательских данных

Другой важной особенностью динамических веб-страниц является возможность чтение пользовательских данных.

Пользовательских данные считываются функцией Request[], и размещение (вход) проходят проверку

Отправление (пользовательских данные) проверяется условием IsPost

```
@{
var totalMessage = "";
if(IsPost)
    {
    var num1 = Request["text1"];
    var num2 = Request["text2"];
    var total = num1.AsInt() + num2.AsInt();
    totalMessage = "Total = " + total;
    }
}
<html>
<body style="background-color: beige; font-family: Verdana, Arial;">
<form action="" method="post">
<p><label for="text1">First Number:</label><br>
< input type="text" name="text1" /></p>
<p><label for="text2">Second Number:</label><br>
<input type="text" name="text2" /></p>
<p><input type="submit" value=" Add " /></p>
</form>
<p>@tot<html>
<body>
@for(var i = 10; i < 21; i++)
    {<p>Line @i</p>}
</body>
</html>alMessage</p>
< /body>
</html>
```

Переменные. Переменные объявляются с помощью ключевого слова var, или с помощью типа (если вы хотите объявить тип), ASP.NET как правило, может автоматически определить типы данных.

```
// Using the var keyword:
var greeting = "Welcome to W3Schools";
var counter = 103;
var today = DateTime.Today;

// Using data types:
string greeting = "Welcome to W3Schools";
int counter = 103;
DateTime today = DateTime.Today;
```

Циклы.

Оператор For. Пример:

```
<html>
<body>
@for(var i = 10; i < 21; i++)
    {<p>Line @i</p>}
</body>
</html>
```

Оператор For Each. Пример:

```
        <html>
<body>
<ul>
@foreach (var x in Request.ServerVariables)
    {<li>@x</li>}
</ul>
</body>
</html>
```

Оператор While. Пример:

```
<html>
<body>
@{
var i = 0;
while (i < 5)
    {
    i += 1;
    <p>Line @i</p>
    }
}
</body>
</html>
```

Массивы

```
@{
string[] members = {"Jani", "Hege", "Kai", "Jim"};
int i = Array.IndexOf(members, "Kai")+1;
int len = members.Length;
string x = members[2-1];
```

```
}
<html>
<body>
<h3>Members</h3>
@foreach (var person in members)
{
<p>@person</p>
}
<p>The number of names in Members are @len</p>
<p>The person at position 2 is @x</p>
<p>Kai is now in position @i</p>
</body>
</html>
```

Применение условий

Оператор If. Пример:

```
@{var price=50;}
<html>
<body>
@if (price>30)
    {
    <p>The price is too high.</p>
    }
</body>
</html>
```

Оператор if else. Пример1

```
@{var price=20;}
<html>
<body>
@if (price>30)
  {
  <p>The price is too high.</p>
  }
else
  {
  <p>The price is OK.</p>
  }
</body>
</html>
```

Пример 2:

```
@{var price=25;}
<html>
<body>
@if (price>=30)
  {
  <p>The price is high.</p>
  }
else if (price>20 && price<30)
  {
  <p>The price is OK.</p>
```

```
  }@{
var weekday=DateTime.Now.DayOfWeek;
var day=weekday.ToString();
var message="";
}
<html>
<body>
@switch(day)
{
case "Monday":
    message="This is the first weekday.";
    break;
case "Thursday":
    message="Only one day before weekend.";
    break;
case "Friday":
    message="Tomorrow is weekend!";
    break;
default:
    message="Today is " + day;
    break;
}
<p>@message</p>
</body>
</html>
else
  {
  <p>The price is low.</p>
  }
</body>
</html>
```

Оператор **switch. Пример.**

```
@{
var weekday=DateTime.Now.DayOfWeek;
var day=weekday.ToString();
var message="";
}
<html>
<body>
@switch(day)
{
case "Monday":
    message="This is the first weekday.";
    break;
case "Thursday":
    message="Only one day before weekend.";
    break;
case "Friday":
    message="Tomorrow is weekend!";
    break;
default:
    message="Today is " + day;
    break;
}
<p>@message</p>
</body>
</html>
```

Источник.
http://www.w3schools.com/aspnet/razor_syntax.asp

4 ЛЕКЦИЯ

4.1 Роуты

Также как и модели, представления и контроллеры, MVC приложения используют систему маршрутизации (роутинговую систему) ASP.NET, которая решает, как URL-адреса соответствуют контроллерам и действиям. Когда Visual Studio создает MVC проект, она в начале добавляет некоторые роуты по умолчанию. Вы можете запросить любую из следующих ссылок, и они будут направлены на метод Index контроллера HomeController:

- /
- /Home
- /Home/Index

Поэтому когда браузер запрашивает http://yoursite/ или http://yoursite/Home, он получает выходные данные HomeController метода Index. Вы можете попробовать сделать это самостоятельно, изменив URL в браузере. На данный момент, это будет http://localhost:61982/, за исключением того, что порт может быть другим. Если добавить в URL /Home или /Home/Index и обновить страницу, вы увидите тот же Hello World MVC приложения.

Это хороший пример от MVC соглашений. В данном случае соглашение заключается в том, что у нас есть контроллер HomeController и что он будет отправной точкой для нашего MVC приложения. Роуты по умолчанию, которые Visual Studio создает для нового проекта, предполагают, что мы будем следовать этому соглашению. И так как мы *следовали* соглашению, мы получили поддержку для URL адресов из предыдущего списка.

Примечание. Вы можете просмотреть и отредактировать роутинговые настройки, открыв файл RouteConfig.cs в папке App_Start.

Представление (рендеринг) веб страниц. Чтобы создать на запрос браузера HTML ответ, мы должны создать представление.

Первое, что мы должны сделать, это добавим метод Index, как показано в следующем листинге.

Листинг 4.1.

```
…
public ViewResult Index()
        {   return View();   }
...
```

Когда мы возвращаемся к объекту ViewResult метода действия, мы поручаем MVC сделать представление. Мы создаем ViewResult, вызывая метод View без параметров. Это указывает MVC обрабатывать для метода действия представление по умолчанию.

Если вы сейчас запустите приложение, вы увидите, как MVC Framework пытается найти нужное

представление по умолчанию, и это показано в сообщении об ошибке, которое представлено на рисунке 4.1.

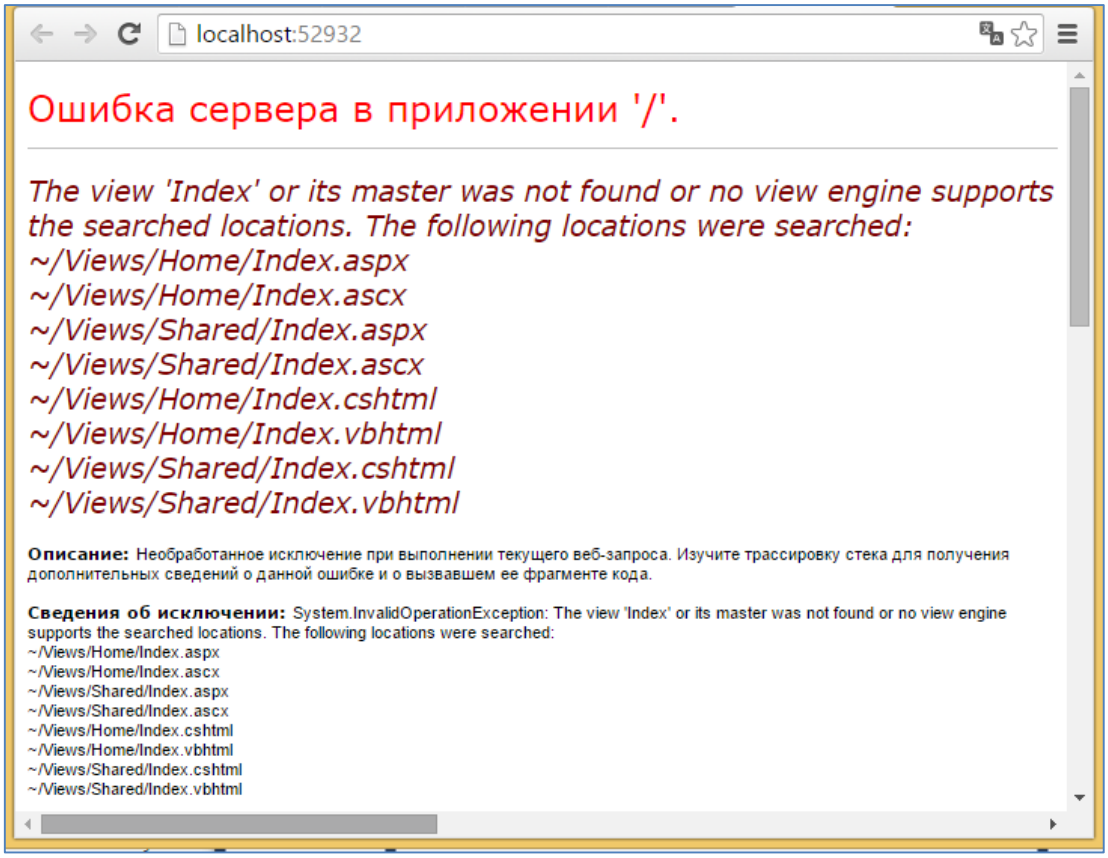


Рисунок 4.1 - MVC Framework пытается найти представление по умолчанию

Это сообщение об ошибке весьма полезно. Оно объясняет не только то, что MVC не смог найти представление для нашего метода, но оно также показывает, где он искал. Это еще один хороший пример MVC соглашения: представления связаны с методами при помощи имен. Наш метод действия называется Index, а наш контроллер называется Home, и вы можете увидеть на рисунке 4.1, что MVC пытается найти различные файлы в папке Views с таким именем.

Примечание. Расширение файла .cshtml обозначает C# представление, которое будет обрабатываться Razor. Предыдущие версии MVC опирались на движок представлений ASPX, для которого файлы представления имели расширение .aspx.

Настраиваем роут по умолчанию

В платформе MVC запрос по умолчанию, поступающие в корень сайта (http://mysite/), необходимо отображать в метод действия. Например, метод действия `Index` класса HomeController. Для этого в методе RegisterRoutes в файле App_Start/RouteConfig.cs указываем настройки как показано в листинге 2.

Листинг 4.2. Роут по умолчанию. RouteConfig.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace WebApplication2
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home",
                                action = "Index",
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

Совет. Обратите внимание, что в листинге 2 установлено контроллеру значение Home, а не HomeController, что является именем класса. Это часть схемы именования ASP.NET MVC, в которой имена классов контроллеров всегда заканчиваются на Controller, и при обращении к классу эта часть имени опускается.

4.2 Основные особенности языка

Листинг 4.3. Класс Product.cs.

```
public class Product
{
private int productID;
private string name;
private string description;

public int ProductID
{   get { return productID; }
    set { productID = value; }
}
public string Name
{   get { return name; }
    set { name = value; }
}
public string Description
{   get { return description; }
    set { description = value; }
}
}
```

Использование автоматически реализуемых свойств

Листинг 4.4. Класс Product.cs.

```
public class Product
{
    private int productID { get; set; }
    private string name { get; set; }
    private string description { get; set; }
}
```

Есть несколько пунктов, на которые стоит обратить внимание при использовании автоматических свойств. Первое, мы не определяем тело для выражений получения и установки. Второе, мы не определяем поле, поддерживающее свойство. Все это сделает для нас C# компилятор, когда мы построим наш класс. Использование автоматических свойств ничем не отличается от использования обычных свойств.

С помощью автоматических свойств мы экономим время на наборе текста, создаем код, который легче читать, и при этом сохраняем ту гибкость, которую предоставляют свойства. Если наступит момент, когда нам нужно будет изменить способ реализации свойства, мы сможем вернуться к обычному формату свойства. Давайте представим, что мы должны изменить способ реализации свойства Name, как показано в листинге 4.5.

Листинг 4.5. Класс Product.cs.

```
public class Product
{
    private string name;
    public int ProductID { get; set; }

    public string Name
    {   get  {  return ProductID + name;   }
        set  {  name = value;    }
    }
    public string Description { get; set; }
```

```
    public decimal Price { get; set; }
    public string Category { set; get; }
}
```

Использование инициализаторов объектов и коллекций

Другой утомительной задачей программирования является создание нового объекта, а затем присвоение значений свойствам, как показано в листинге 4.6.

Листинг 4.6.

```
// создание нового объекта Product
            Product myProduct = new Product();
            // установка значений свойств
            myProduct.ProductID = 100;
            myProduct.Name = "Kayak";
            myProduct.Description = "A boat for one person";
            myProduct.Price = 275M;
            myProduct.Category = "Watersports";
```

Мы можем использовать **функцию инициализации объекта**, которая позволяет создавать и заполнять экземпляр объекта Product за один шаг, как показано в листинге 4.7.

```
// создание и заполнение нового объекта Product
            Product myProduct = new Product
            {
                ProductID = 100,
                Name = "Kayak",
                Description = "A boat for one person",
                Price = 275M,
                Category = "Watersports"
            };
```

Фигурные скобки (**{}**) после вызова названия Product образуют инициализатор, который мы используем для передачи значений параметров как часть процесса создания.

Эта же функция (т.е. функцию инициализации) позволяет нам инициализировать содержимое коллекций и массивов как часть процесса создания, что показано в листинге 4.8.

Листинг 4.8.

```
string[] stringArray = { "apple", "orange", "plum" };
List<int> intList = new List<int> { 10, 20, 30, 40 };
Dictionary<string, int> myDict = new Dictionary<string, int>
{    { "apple", 10 }, { "orange", 20 }, { "plum", 30 }
};
```

5 ЛЕКЦИЯ. ЛЯМБДА-ВЫРАЖЕНИЯ В C#)

Лямбда-выражение — это анонимная функция, с помощью которой можно создавать типы делегатов или деревьев выражений. С помощью лямбда-выражений можно написать локальные функции, которые затем можно передавать в другие функции в качестве аргументов или возвращать из них в качестве значения. Лямбда-выражения особенно полезны при написании выражений запросов LINQ.

Для создания лямбда-выражения можно определить входные параметры (если таковые имеются) слева лямбда-оператора => и поместить выражение или блок операторов на другую сторону. Например, лямбда-выражение x => x * x принимает параметр с именем x и возвращает значение x, возведённое в квадрат. Можно назначить это выражение типу делегата, как показано в следующем примере:

C#

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Создание типа дерева выражений:

C#

```
using System.Linq.Expressions;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Expression<del> myET = x => x * x;
        }
    }
}
```

Оператор **=>** имеет такой же приоритет, как и присваивание (**=**), и является правоассоциативным (см. раздел "Ассоциативность" статьи об операторах).

Лямбда-операторы используются в запросах LINQ с синтаксисом на основе методов в качестве аргументов методов стандартных операторов запроса, таких как Where.

При использовании синтаксиса на основе методов для вызова метода Where класса Enumerable (как это происходит в LINQ to Objects и LINQ to XML) параметром является тип делегата System.Func<T, TResult>. Лямбда-выражение — это наиболее удобный способ создания такого делегата. При вызове того же метода, к примеру, в классе System.Linq.Queryable (как это делается в LINQ to SQL) типом параметра будет System.Linq.Expressions.Expression<Func>, где Func — это любые делегаты Func с числом входных параметров не более шестнадцати.Опять же, лямбда-выражения представляют собой самый быстрый способ построения такого дерева выражений. Лямбда-выражения позволяют вызовам **Where** выглядеть одинаково, хотя, на самом деле, объекты, созданные из лямбда-выражений, будут иметь разные типы.

Обратите внимание: в приведённом выше примере сигнатура делегата имеет один неявный входной параметр типа **int** и возвращает значение типа **int**. Лямбда-выражение можно преобразовать в делегат соответствующего типа, поскольку он также имеет один входной параметр (x) и возвращает значение, которое компилятор может неявно преобразовать в тип **int**. (Вывод типов более подробно рассматривается в следующих разделах.) Делегат, вызываемый посредством входного параметра 5, возвращает результат 25.

Лямбда-выражения не разрешены в левой части оператора is или as.

Все ограничения, применяемые к анонимным методам, применяются также к лямбда-выражениям.Дополнительные сведения см. в разделеАнонимные методы (Руководство по программированию в C#).

Выражения-лямбды

Лямбда-выражение с выражением с правой стороны оператора => называется *выражением-лямбдой*. Выражения-лямбды широко используются при создании таких конструкций как Деревья выражений (C# и Visual Basic). Выражения-лямбды возвращают результат выражения и имеют следующую основную форму:

```
(input parameters) => expression
```

Если лямбда имеет только один входной параметр, скобки можно не ставить, во всех остальных случаях они обязательны. Если входных параметров два и более, то они разделяются запятыми и заключаются в скобки:

C#
```
(x, y) => x == y
```

Иногда компилятору бывает трудно или даже невозможно вывести типы входных параметров. В этом случае типы можно указать в явном виде, как показано в следующем примере:

C#
```
(int x, string s) => s.Length > x
```

Отсутствие входных параметров задаётся пустыми скобками.

C#
```
() => SomeMethod()
```

Обратите внимание на предыдущий пример: тело выражения-лямбды может состоять из вызова метода. Однако при создании деревьев выражений, которые вычисляются вне .NET Framework, например в SQL Server, не следует использовать вызовы методов в лямбда-выражениях.Эти методы не имеют смысла вне контекста среды CLR .NET.

Лямбды операторов

Лямбда операторов (или операторная лямбда) напоминает выражение-лямбду, за исключением того, что оператор (или операторы) заключается в фигурные скобки:

```
(input parameters) => {statement;}
```

Тело лямбды операторов может состоять из любого количества операторов; однако на практике обычно используется не больше двух-трёх.

C#
```
delegate void TestDelegate(string s);
...
TestDelegate myDel = n => { string s = n + " " + "World";
Console.WriteLine(s); };
myDel("Hello");
```

Лямбды операторов, как и анонимные методы, не могут использоваться для создания деревьев выражений.

Асинхронные лямбда-выражения

Можно легко создавать лямбда-выражения и операторы, которые включают асинхронную обработку с помощью ключевых слов async и await направит. Например, следующий фрагмент приложения Windows Forms содержит обработчик событий, который вызывает и ожидает асинхронный метод — ExampleMethodAsync:

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

Можно переписать этот обработчик событий, используя асинхронное лямбда-выражение. Чтобы добавить этот обработчик, добавьте модификатор **async** перед списком параметров лямбда-выражения как показано в следующем примере:

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event
handler.\r\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

Дополнительные сведения о создании и использовании асинхронных методов см. в разделе Асинхронное программирование с использованием ключевых слов Async и Await (C# и Visual Basic).

Лямбды со стандартными операторами запросов

Многие стандартные операторы запросов имеют входной параметр, тип которого принадлежит семейству универсальных методов-делегатов Func<T, TResult>. Эти делегаты используют параметры типа для определения количества и типов входных параметров, а также тип возвращаемого значения делегата.Делегаты **Func** очень полезны для инкапсуляции пользовательских выражений, которые применяются к каждому элементу в наборе исходных данных. В качестве примера рассмотрим следующий тип делегата:

C#
```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

Можно создать экземпляр этого делегата как Func<int,bool> myFunc, где **int** — тип входного параметра, а **bool** — тип возвращаемого значения. Возвращаемое значение всегда указывается в последнем параметре типа. **Func<int, string, bool>** определяет делегат с двумя входными параметрами, типы которых — **int** и **string**, и типом возвращаемого значения **bool**. Следующий делегат **Func** при вызове возвращает значение true или false которое показывает равен ли входной параметр 5:

C#
```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4); // returns false of course
```

Также лямбда-выражения можно использовать, когда аргумент имеет тип **Expression<Func>**, например в стандартных операторах запроса, которые определены в System.Linq.Queryable. При задании аргумента типа **Expression<Func>** лямбда-выражение компилируется в дерево выражения. Пример использования стандартного оператора запроса, метода Count, показан ниже:

C#
```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
```

Компилятор может вывести тип входного параметра; а также его можно определить явно. Данное конкретное лямбда-выражение подсчитывает количество целых чисел (n), которые при делении на два дают остаток 1.
Следующая строка кода создает последовательность, которая содержит все элементы массива numbers, расположенные слева от 9, поскольку это первое число последовательности, не удовлетворяющее условию:

C#
```
var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);
```

В этом примере показано, как задать несколько входных параметров путём заключения их в скобки. Этот метод возвращает все элементы в массиве чисел до того числа, величина которого меньше номера его позиции.Не следует путать лямбда-оператор (**=>**) с оператором "больше или равно" (**>=**).

C#
```
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
```

Вывод типа в лямбда-выражениях

При написании лямбда-выражений обычно не требуется указывать тип входных параметров, поскольку компилятор может выводить этот тип на основе тела лямбда-выражения, типа делегата параметра и других факторов, как описано в спецификации языка C#. Для большинства стандартных операторов запроса первый входной параметр имеет тот же тип, что и элементы в исходной

последовательности. Поэтому, если производится запрос к **IEnumerable<Customer>**, то переменная входного параметра расценивается как объект типа Customer, а это означает, что у вас есть доступ к его методам и свойствам.

C#
```
customers.Where(c => c.City == "London");
```

Используются следующие основные правила для лямбда-выражений:

- лямбда-выражение должно содержать то же число параметров, что и тип делегата;
- Каждый входной параметр в лямбда-выражении должен быть неявно преобразуемым в соответствующий параметр делегата.
- возвращаемое значение лямбда-выражения (если таковое имеется) должно быть неявно преобразуемым в возвращаемый тип делегата.

Обратите внимание: лямбда-выражения сами по себе не имеют типа, поскольку система общих типов не имеет встроеной концепции "лямбда-выражения". Однако иногда бывает удобно оперировать неформальным понятием "типа" лямбда-выражения. При этом под типом понимается тип делегата или тип Expression, в который преобразуется лямбда-выражение.

Область действия переменной в лямбда-выражениях

Лямбда-выражения могут ссылаться на *внешние переменные* (см. Анонимные методы (Руководство по программированию в C#)), находящиеся в области метода, в котором определена лямбда-функция, или в области типа, который содержит лямбда-выражение.Переменные, полученные таким способом, сохраняются для использования в лямбда-выражениях, даже если бы в ином случае они оказались за границами области действия и уничтожились сборщиком мусора. Внешней переменной должно быть присвоено определённое значение, прежде чем она сможет использоваться в лямбда-выражениях. Следующий пример демонстрирует применение этих правил:

C#
```
delegate bool D();
delegate bool D2(int i);

class Test
{
    D del;
    D2 del2;
    public void TestMethod(int input)
    {
        int j = 0;
        // Initialize the delegates with lambda expressions.
        // Note access to 2 outer variables.
        // del will be invoked within this method.
        del = () => { j = 10;  return j > input; };

        // del2 will be invoked after TestMethod goes out of scope.
        del2 = (x) => {return x == j; };

        // Demonstrate value of j:
        // Output: j = 0
        // The delegate has not been invoked yet.
        Console.WriteLine("j = {0}", j);        // Invoke the delegate.
        bool boolResult = del();

        // Output: j = 10 b = True
        Console.WriteLine("j = {0}. b = {1}", j, boolResult);
    }
```

```
    static void Main()
    {
        Test test = new Test();
        test.TestMethod(5);

        // Prove that del2 still has a copy of
        // local variable j from TestMethod.
        bool result = test.del2(10);

        // Output: True
        Console.WriteLine(result);

        Console.ReadKey();
    }
}
```

Следующие правила применимы к области действия переменной в лямбда-выражениях.

- Захваченная переменная не будет уничтожена сборщиком мусора до тех пор, пока делегат, который на нее ссылается, не перейдет в статус подлежащего уничтожению при сборке мусора.
- Переменная, объявленная в лямбда-выражении, невидима во внешнем методе.
- Лямбда-выражение не может непосредственно захватывать параметры **ref** или **out** из метода, в котором они находятся.
- Оператор Return в лямбда-выражении не приводит к возврату (завершению) метода, в котором объявлено/вызвано лямбда-выражение.
- Лямбда-выражение не может содержать оператора **goto**, оператора **break** или оператора **continue** внутри лямбда-функции, если целевой объект перехода находится вне блока.Если целевой объект находится внутри блока, то наличие оператора перехода за пределами лямбда-функции также будет ошибкой.

Источник
https://msdn.microsoft.com/ru-ru/library/bb397687.aspx

6 ЛЕКЦИЯ
ТЕРМИНОЛОГИЯ ПЛАТФОРМЫ ENTITY FRAMEWORK

В этом разделе определены термины, которые часто встречаются в документации по Entity Framework.Даны ссылки на соответствующие разделы, содержащие дополнительные сведения.

Термин	Определение
ассоциация	Определение отношения между двумя типами сущностей. Дополнительные сведения см. в разделах Association Element (CSDL) и тип ассоциации.
набор ассоциаций	Логический контейнер для экземпляров ассоциаций одного типа. Дополнительные сведения см. в разделах AssociationSet Element (CSDL) и набор ассоциаций.
Code First	Начиная с версии 4.1 платформы Entity Framework, модель можно создавать программно с помощью шаблона разработки Code First.Шаблон разработки Code First имеет два различных сценария.В обоих случаях разработчик определяет модель, задавая в коде определения классов .NET Framework, а затем выборочно определяет дополнительные сопоставления или конфигурации с помощью заметок к данным или fluent API. Следует иметь в виду, что шаблон разработки Code First является частью платформы Entity Framework 5.0.Платформа Entity Framework 5.0 не является частью платформы .NET Framework, но построена на .NET Framework 4.5.Платформа Entity Framework 5.0 доступна в качестве пакета NuGet«Entity Framework».Дополнительные сведения см. в разделе Выпуски и управление версиями платформы Entity Framework.
дерево команд	Типовое программное представление всех запросов Entity Framework, состоящих из одного или нескольких выражений. Для получения дополнительной информации см. Общие сведения о платформе Entity Framework.
сложный тип	Класс .NET Framework, представляющий сложное свойство согласно определению в концептуальной модели.Сложные типы позволяют организовать скалярные свойства внутри сущностей.Сложные объекты являются экземплярами сложных типов.Дополнительные сведения см. в разделах ComplexType Element (CSDL) и сложный тип.
ComplexType	Спецификация типа данных, которая представляет нескалярное свойство типа сущности, не имеющего ключевого свойства. Дополнительные сведения см. в разделах ComplexType Element (CSDL) и сложный тип.

концептуальная модель	Абстрактная спецификация для типов сущностей, сложных типов, сопоставлений, контейнеров сущностей, наборов сущностей и наборов сопоставлений в домене приложения Entity Framework.Концептуальная модель определяется на языке CSDL в CSDL-файле. Для получения дополнительной информации см. Моделирование и сопоставление.
CSDL-файл	XML-файл, содержащий концептуальную модель на языке CSDL.
CSDL (язык определения концептуальной схемы)	Основанный на XML язык, используемый для определения типов сущностей, ассоциаций, контейнеров сущностей, наборов сущностей и наборов ассоциаций концептуальной модели. Для получения дополнительной информации см. Спецификация языка CSDL.
контейнер	Логическое группирование наборов сущностей и ассоциаций. Дополнительные сведения см. в разделах EntityContainer Element (CSDL) и контейнер сущностей.
параллельность	Позволяет нескольким пользователям одновременно обращаться и изменять совместно используемые данные.По умолчанию в платформе Entity Framework реализуется модель оптимистичного параллелизма.
направление	Указывает асимметричную природу некоторых сопоставлений.Направление указывается с помощью атрибутов **FromRole**и **ToRole** элемента **NavigationProperty** или **ReferentialConstraint** в схеме. Дополнительные сведения см. в разделах NavigationProperty Element (CSDL) и свойство навигации.
безотлагательная загрузка	Процесс загрузки конкретного набора связанных между собой объектов вместе с объектами, которые были запрошены явным образом.
EDMX-файл	XML-файл, содержащий концептуальную модель (на языке CSDL), модель хранения (на языке SSDL) и сопоставления между ними (на языке MSL).Edmx-файл создается средствами EDM (модель данных с использованием сущностей). Дополнительные сведения см. в разделе .edmx File Overview.
end	Сущность, участвующая в ассоциации. Дополнительные сведения см. в разделах End Element (CSDL) и конечная точка ассоциации.
сущность	Концепция в области приложения, по которой определен тип данных. Дополнительные сведения см. в разделах EntityType Element (CSDL) и тип сущности.

EntityClient	Независимый от хранилища поставщик данных ADO.NET, содержащий такие классы, как **EntityConnection**,**EntityCommand** и **EntityDataReader**.Работает с Entity SQL и подключается к зависящим от хранилища поставщикам данных ADO.NET, таким как **SqlClient**. Для получения дополнительной информации см. Поставщик EntityClient для платформы Entity Framework.
контейнер сущностей	Задает наборы сущностей и наборы ассоциаций, которые будут реализованы в заданном пространстве имен. Дополнительные сведения см. в разделах EntityContainer Element (CSDL) и контейнер сущностей.
Модель EDM	Набор понятий, описывающих структуру данных (например, сущности и отношения) независимо от формы ее хранения. Для получения дополнительной информации см. Модель EDM.
Entity Framework	Набор технологий, который поддерживает разработку приложений, связанных с обработкой данных, позволяя программистам работать с концептуальными моделями, сопоставленными логическим схемам в источниках данных. Для получения дополнительной информации см. Общие сведения о платформе Entity Framework.
набор сущностей	Логический контейнер для сущностей данного типа и его подтипов.Наборы сущностей сопоставляются таблицам в базе данных. Дополнительные сведения см. в разделах EntitySet Element (CSDL) и набор сущностей.
Entity SQL	Независимый от хранилища диалект SQL, который работает непосредственно с концептуальными схемами сущностей и поддерживает такие понятия концептуальной модели, как наследование и отношения. Для получения дополнительной информации см. Язык Entity SQL.
тип сущности	Класс .NET Framework, представляющий сущность согласно определению в концептуальной модели.Типы сущностей могут иметь скалярные и сложные свойства, а также свойства навигации.Объекты являются экземплярами типов сущностей.Для получения дополнительной информации см. Работа с объектами.
EntityType	Спецификация для типа данных, которая содержит ключ и именованный набор свойств, и представляет элемент верхнего уровня в концептуальной модели или модели хранения. Дополнительные сведения см. в разделах EntityType Element (CSDL) и тип сущности.

явная загрузка	Когда запрос возвращает объекты, связанные объекты не загружаются.По умолчанию они не загружаются до тех пор, пока не будут явным образом запрошены вызовом метода **Load** для свойства навигации.
сопоставление на основе внешнего ключа	Сопоставление между сущностями, управляемая через свойства внешнего ключа.
идентифицирующее отношение	Отношение, в котором первичный ключ основной сущности является частью первичного ключа зависимой сущности.В таком отношении зависимая сущность не может существовать без основной.
независимое сопоставление	Сопоставленение между сущностями, представляемое и отслеживаемое независимым объектом.
клавиша	Атрибут типа сущности, который указывает, какое свойство или набор свойств используется для определения уникальных экземпляров типа сущности.Представлен на уровне объектов классом EntityKey. Дополнительные сведения см. в разделах Key Element (CSDL) и ключ сущности.
отложенная загрузка	Когда запрос возвращает объекты, связанные объекты не загружаются.Вместо этого они загружаются автоматически, когда производится доступ к свойству навигации.
LINQ to Entities	Синтаксис запроса, который определяет набор операторов запроса, обеспечивающих операции просмотра, фильтрации и проекции, выражаемые прямым, декларативным способом в Visual C# и Visual Basic. Для получения дополнительной информации см. LINQ to Entities.
сопоставление	Спецификация соответствий между элементами в концептуальной модели и элементами в модели хранения. Для получения дополнительной информации см. Спецификация языка MSL.
MSL-файл	XML-файл, содержащий сопоставление концептуальной модели и модели хранения, описанное на языке MSL.
MSL (язык определения соответствий)	Основанный на XML язык, используемый для сопоставления элементов, определенных в концептуальной модели, элементам в модели хранилища. Для получения дополнительной информации см. Спецификация языка MSL.

функции изменения	Хранимые процедуры, которые используются для вставки, обновления и удаления данных, находящихся в источнике данных.Эти функции используются вместо сформированных команд Entity Framework.Функции изменения определены элементом **Function** в модели хранения.Элемент ModificationFunctionMapping сопоставляет эти функции изменения операциям вставки, обновления и удаления для сущностей, определенных в концептуальной модели.
кратность	Количество сущностей, которые могут существовать на каждой стороне связи, как определено ассоциацией.Также называется мощностью или количеством элементов. Дополнительные сведения см. в разделах End Element (CSDL) и конечная точка ассоциации.
несколько наборов сущностей на тип	Возможность определить тип сущности в более чем одном наборе сущностей. Дополнительные сведения см. в разделах EntitySet Element (CSDL) и How to: Define a Model with Multiple Entity Sets per Type.
свойство навигации	Свойство типа сущности, которое представляет связь с другим типом сущности, как определено ассоциацией.Свойства навигации используются, чтобы возвратить связанные объекты как EntityCollection<TEntity> или EntityReference<TEntity>, в зависимости от кратности другого элемента сопоставления. Дополнительные сведения см. в разделах NavigationProperty Element (CSDL) и свойство навигации.
путь запроса	Строковое представление пути, которое показывает, какие связанные объекты будут возвращены при выполнении запроса объектов.Путь запроса определяется путем вызова метода Include объекта ObjectQuery<T>. Для получения дополнительной информации см. Loading Related Objects.
контекст объекта	Представляет контейнер сущностей, определенный в концептуальной модели.Содержит соединение с базовым источником данных и предоставляет такие службы, как отслеживание изменений и разрешение идентификаторов.Контекст объекта представлен экземпляром класса ObjectContext или **DbContext**. **DbContext** является частью платформы Entity Framework 5.0.Платформа Entity Framework 5.0 не является частью платформы .NET Framework, но построена на .NET Framework 4.5.Платформа Entity Framework 5.0 доступна в качестве пакета NuGet«Entity Framework».Дополнительные сведения см. в разделе Выпуски и управление версиями платформы Entity Framework.
уровень объектов	Типы сущностей и определения контекста объектов, используемых платформой Entity Framework.

запросы объектов	Запрос, выполняемый в контексте объекта на концептуальной модели, возвращающий данные как объекты. Для получения дополнительной информации см. Object Queries.
объектно-реляционное сопоставление	Метод преобразования данных из реляционной базы данных в типы данных, которые могут быть использованы в объектно-ориентированных приложениях. Платформа Entity Framework обеспечивает объектно-реляционное сопоставление, которое сопоставляет реляционные данные, определенные в модели хранения, с типами данных, определенными в концептуальной модели. Для получения дополнительной информации см. Моделирование и сопоставление.
службы объектов	Службы, предоставленные платформой Entity Framework, которые позволяют коду приложения работать с такими сущностями, как объекты .NET Framework.
объекты, игнорирующие сохраняемость	Объект, который не содержит никакой логики, относящейся к хранилищу данных.Также называется сущностью POCO.
POCO	Традиционный объект среды CLR.Объект, который не является производным от другого класса и не реализует интерфейсы.
сущность POCO	Сущность Entity Framework, которая не является производной от классов EntityObject и ComplexObject и не реализует интерфейсы Entity Framework.Сущности POCO часто представляют существующие объекты домена, используемые в приложении Entity Framework.Эти сущности поддерживают пропуск сохраняемости.Для получения дополнительной информации см. Working with POCO Entities.
прокси-объект	Объект, который является производным от класса POCO и формируется платформой Entity Framework для поддержки отложенной загрузки и отслеживания изменений.Для получения дополнительной информации см. Requirements for Creating POCO Proxies.
справочное ограничение	Ограничение, которое определено в концептуальной модели и указывает, что сущность имеет зависимое отношение с другой сущностью.Это ограничение означает, что экземпляр зависимой сущности не может существовать без соответствующего экземпляра главной сущности. Дополнительные сведения см. в разделах ReferentialConstraint Element (CSDL) и ограничение ссылочной целостности.
отношение	Логическое соединение между сущностями.

роль	Имя, данное каждому **End** сопоставления, чтобы сделать более ясной семантику отношения. Дополнительные сведения см. в разделах End Element (CSDL) и конечная точка ассоциации.
скалярное свойство	Свойство сущности, которое сопоставляется одному полю в модели хранения.
сущность с самостоятельным отслеживанием	Сущность, построенная средствами преобразования текстовых шаблонов (T4), которая может записывать изменения скалярных свойств, сложных свойств и свойств навигации.
простой тип	Тип-примитив, который используется для определения свойств в концептуальной модели. Дополнительные сведения см. в разделах Conceptual Model Types (CSDL) и Модель EDM. Примитивные типы данных.
разделенная сущность	Тип сущности, сопоставляемый с двумя отдельными типами в модели хранения. Для получения дополнительной информации см. How to: Define a Model with a Single Entity Mapped to Two Tables.
модель хранения	Определение для логической модели данных в поддерживаемом источнике данных, таком как реляционная база данных.Модель хранения определяется на языке SSDL в SSDL-файле модели хранения. Дополнительные сведения см. в разделах Моделирование и сопоставление и Спецификация языка SSDL.
SSDL-файл	XML-файл, содержащий модель хранения, описанную на языке SSDL.
SSDL (язык определения структуры схемы)	Язык на основе XML, который используется для определения типов сущностей, ассоциаций, контейнеров сущностей, наборов сущностей и наборов ассоциаций модели хранения, которая часто соответствует схеме базы данных. Для получения дополнительной информации см. Спецификация языка SSDL.
одна таблица на иерархию	Метод моделирования иерархии типов в базе данных, согласно которому атрибуты всех типов в иерархии помещаются в одну таблицу.
одна таблица на тип	Метод моделирования иерархии типов в базе данных, согласно которому для моделирования разных типов используются разные таблицы, связанные отношением «один к одному».

7 ЛЕКЦИЯ. ЗАГРУЗКА СВЯЗАННЫХ ОБЪЕКТОВ (ПЛАТФОРМА ENTITY FRAMEWORK)

В этом разделе описаны *шаблоны*, которые можно использовать для загрузки связанных сущностей. В типах сущности могут определяться *свойства навигации*, которые представляют ассоциации в модели данных. Эти свойства можно использовать для загрузки сущностей, которые связаны с возвращенной сущностью посредством определенной ассоциации. Когда сущности формируются на основе модели данных, *свойства навигации* создаются для сущностей в обоих элементах ассоциации. Данные свойства навигации возвращают либо ссылку на элемент «один» в связи типа «один ко многим» или «многие к одному», либо коллекцию элементов «многие» в связи типа «один ко многим» или «многие ко многим». Дополнительные сведения см. в разделах Свойства навигации и Определение отношений и управление отношениями (платформа Entity Framework).

Загрузка шаблона	**Описание**
Указывается в запросе	Можно сформировать запрос Entity SQL или LINQ to Entities , который будет явно переходить по связям при помощи свойств навигации. При выполнении такого запроса происходит возврат связанных сущностей, которые включены в качестве свойств навигации в самую внешнюю проекцию запроса. Дополнительные сведения см. в разделе Как переходить по связям с помощью свойств навигации (платформа Entity Framework).
Явная загрузка	При явной загрузке сущностей в ObjectContext требуется несколько циклов обмена данными с базой данных; кроме того, могут потребоваться несколько активных результирующих наборов, однако объем возвращаемых данных ограничен только загружаемыми сущностями. Для явной выборки связанных сущностей из источника данных используйте метод**Load** для объекта EntityCollection или EntityReference или метод LoadProperty для объекта **ObjectContext**. При каждом вызове метода **Load** открывается соединение с базой данных для получения связанной информации. Это гарантирует, что запрос никогда не выполняется без явного обращения к связанной сущности. Явная загрузка является поведением по умолчанию для среды Entity Framework .
Отложенная загрузка	Во время загрузки этого типа связанные сущности загружаются из источника данных автоматически при доступе к свойству навигации. Используя загрузку этого типа, необходимо учитывать, что доступ к каждому свойству навигации приводит к выполнению отдельного запроса к источнику данных, если сущность еще не находится в **ObjectContext**.
Безотложная загрузка	Если известен точный вид графа связанных сущностей, необходимых для конкретного приложения, то можно применить метод Include к

или Определение путей запроса с включением	объекту ObjectQuery для определения пути запроса, управляющего тем, какие связанные сущности должны быть возвращены в составе начального запроса. При определении пути запроса для возврата всех сущностей, которые задаются путем в одном результирующем наборе, требуется только один запрос к базе данных. Для каждого объекта, возвращаемого запросом, загружаются все связанные сущности того типа, который задан в пути.

7.1 Переход по связям с помощью свойств навигации (платформа Entity Framework)

В этом разделе показано, как переходить по связям с помощью свойств навигации. В этом примере возвращаются все заказы контактов с фамилией «Zhou». Свойство навигации **Contact.SalesOrderHeader** используется для получения коллекции объектов **SalesOrderHeader** для каждого контактного лица. Один и тот же пример приводится с использованием всех следующих *технологий запросов* платформы Entity Framework .

- LINQ to Entities
- Entity SQL с ObjectQuery<T>
- Методы построителя запросов ObjectQuery<T>

Пример LINQ to Entities.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new { LastName = contact.LastName, Orders =
contact.SalesOrderHeaders };

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Name: {0}", order.LastName);
        foreach (SalesOrderHeader orderInfo in order.Orders)
        {
            Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due:
{2}",
                orderInfo.SalesOrderID, orderInfo.OrderDate,
orderInfo.TotalDue);
        }
        Console.WriteLine("");
    }
}
```

Пример Entity SQL (не рекомендуется!!).

```
using (AdventureWorksEntities context =
    new AdventureWorksEntities())
{
    string esqlQuery = @"SELECT c.FirstName, c.SalesOrderHeaders
        FROM AdventureWorksEntities.Contacts AS c where c.LastName = @ln";
```

```
    ObjectQuery<DbDataRecord> query = new
ObjectQuery<DbDataRecord>(esqlQuery, context);
    query.Parameters.Add(new ObjectParameter("ln", "Zhou"));

    foreach (DbDataRecord rec in query)
    {

        // Display contact's first name.
        Console.WriteLine("First Name {0}: ", rec[0]);
        List<SalesOrderHeader> list = rec[1] as List<SalesOrderHeader>;
        // Display SalesOrderHeader information
        // associated with the contact.
        foreach (SalesOrderHeader soh in list)
        {
            Console.WriteLine("   Order ID: {0}, Order date: {1}, Total Due:
{2}",
                soh.SalesOrderID, soh.OrderDate, soh.TotalDue);
        }
    }
}
```

Пример метода построителя запросов.

```
string lastName = "Zhou";
using (AdventureWorksEntities context =
    new AdventureWorksEntities())
{
    // Define a query that returns a nested
    // DbDataRecord for the projection.
    ObjectQuery<DbDataRecord> query =
        context.Contacts.Select("it.FirstName, "
            + "it.LastName, it.SalesOrderHeaders")
        .Where("it.LastName = @ln", new ObjectParameter("ln", lastName));

    foreach (DbDataRecord rec in
        query.Execute(MergeOption.AppendOnly))
    {

        // Display contact's first name.
        Console.WriteLine("First Name {0}: ", rec[0]);
        List<SalesOrderHeader> list = rec[2]
            as List<SalesOrderHeader>;
        // Display SalesOrderHeader information
        // associated with the contact.
        foreach (SalesOrderHeader soh in list)
        {
            Console.WriteLine("   Order ID: {0}, " +
                "Order date: {1}, Total Due: {2}",
                soh.SalesOrderID, soh.OrderDate, soh.TotalDue);
        }
    }
}
```

Источник
https://msdn.microsoft.com/ru-ru/library/bb896321.aspx#_QueryBuilder

ЛЕКЦИЯ 8 (продолжение лекции 7).
ПЕРЕХОД ПО СВЯЗЯМ С ПОМОЩЬЮ СВОЙСТВ НАВИГАЦИИ

7.1 LINQ TO ENTITIES

Большинство приложений на данный момент создаются на основе реляционных баз данных. Этим приложениям необходимо взаимодействовать с данными, представленными в реляционном виде. Схемы баз данных не всегда идеально подходят для создания приложений, а концептуальные модели приложений не всегда совпадают с логическими моделями баз данных.

EDM (модель данных с использованием сущностей) представляет собой **концептуальную модель данных**, которую можно использовать для моделирования данных конкретного домена (ПО), что позволяет приложениям взаимодействовать с данными как с объектами.

В модели EDM (модель данных с использованием сущностей) реляционные данные представлены в виде **объектов** в среде .NET. Благодаря этому поддержка LINQ эффективно реализуется на уровне объектов, что позволяет составлять запросы баз данных на языке, используемом для сборки бизнес-логики. Эта функция называется **LINQ to Entities**. .

LINQ to Entities обеспечивает поддержку LINQ при запросах к сущностям. Компонент позволяет разработчикам писать запросы к концептуальной модели Entity Framework на языке Visual Basic или Visual C#. Запросы к платформе Entity Framework представляются в виде дерева команд запроса, выполняемого на контексте объектов. Технология LINQ to Entities преобразует запросы Language-Integrated Queries (LINQ) в запросы в виде дерева команд, выполняет эти запросы на платформе Entity Framework и возвращает объекты, которые могут использоваться как платформой Entity Framework, так и технологией LINQ.

Далее описывается процесс создания и исполнения запроса в технологии LINQ to Entities.

7.1.1 Запросы в LINQ to Entities

Запрос представляет собой выражение, извлекающее данные из источника данных.Запросы обычно выражаются на специализированном языке запросов, например SQL для реляционных баз данных и XQuery для XML.Поэтому разработчикам приходится учить новый язык запросов для каждого типа источника данных и формата данных, для которых выполняется запрос.

Интегрированный в язык запрос (LINQ) предлагает упрощенную согласованную модель работы с данными для различных типов источников данных и различных форматов данных.Запросы LINQ всегда работают с программируемыми объектами.

Операция запроса LINQ состоит из трех действий:

- получение одного или нескольких источников данных,
- создание запроса
- выполнение.

К источникам данных, реализующим общий интерфейс IEnumerable<T> или IQueryable<T>, можно выполнять запрос с помощью LINQ. Экземпляры универсального класса ObjectQuery<T>, реализующего общий интерфейс IQueryable<T>, служат источником данных для запросов LINQ to Entities. Универсальный класс ObjectQuery<T> представляет запрос, который возвращает коллекцию, включающую ноль или более типизированных объектов. Можно также разрешить компилятору вывести тип сущности при помощи ключевого слова C# **var.**

В запросе указываются данные, которые необходимо получить из источника данных. В запросе можно также указать, как следует сортировать, группировать и формировать возвращаемую информацию. В LINQ запрос хранится в переменной. Если запрос возвращает последовательность значений, то переменная запроса должна иметь тип данных, который может быть запрошен. Эта переменная запроса не выполняет никаких действий и не возвращает данные. Она только хранит информацию о запросе. После создания запроса его необходимо выполнить, чтобы получить данные.

7.1.2 Синтаксис запроса

Запросы LINQ to Entities могут быть составлены одним из двух способов: с использованием

- **синтаксиса выражения запроса** (см. пункт 7.2)
- **синтаксиса запросов на основе методов** (см. пункт 7.3)

Синтаксис выражения запроса появился в языках C# 3.0 и Visual Basic 9.0 и состоит из набора предложений, написанных в декларативном стиле. Тем не менее среда CLR платформы .NET Framework не может прочитать выражение запроса сама. Таким образом, во время компиляции выражения запроса преобразуются в то, что понятно CLR - вызовы методов. Эти методы называются *стандартными операторами запроса*. Разработчик может вызывать их напрямую, используя **синтаксис методов** вместо синтаксиса запроса.

7.2 СИНТАКСИС ВЫРАЖЕНИЙ ЗАПРОСОВ (LINQ TO ENTITIES)

Выражения запроса используют декларативный синтаксис запроса. Этот синтаксис позволяет разработчикам писать запросы на высокоуровневом языке. С помощью синтаксиса выражения запроса можно выполнять даже сложную фильтрацию, упорядочение и группирование операций в источнике данных с помощью минимального программного кода.Дополнительные сведения см. в разделе Basic Query Operations (Visual Basic).Примеры, показывающие применение синтаксиса выражения запросов, см. в следующих разделах.

Примеры синтаксиса выражений запросов: проекция.

Примеры, приведенные в этом разделе, показывают, как использовать метод **Select** и ключевые слова **From … From …** для выполнения запросов к модели AdventureWorks Sales с применением синтаксиса выражений запросов. **From … From …** - это основанный на запросе эквивалент метода **SelectMany**.Модель AdventureWorks Sales, которая используется в этих примерах, состоит из таблиц Contact, Address, Product, SalesOrderHeader и SalesOrderDetail образца базы данных AdventureWorks.

В примерах, приведенных в этом разделе, используются следующие инструкции **using/Imports**:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

Пример. В следующем примере метод **Select** используется для возврата всех строк из таблицы *Product* и отображения названий продуктов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery = from product in context.Products
                                        select product;

    Console.WriteLine("Product Names:");
    foreach (var prod in productsQuery)
    {
        Console.WriteLine(prod.Name);
    }
```

```
}
```

Пример. В следующем примере используется метод Select для возврата последовательности, состоящей только из названий продуктов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> productNames =
        from p in context.Products
        select p.Name;

    Console.WriteLine("Product Names:");
    foreach (String productName in productNames)
    {
        Console.WriteLine(productName);
    }
}
```

В следующем примере используется метод Select<TSource, TResult> для проецирования свойств **Product.Name** и **Product.ProductID** в последовательность анонимных типов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from product in context.Products
        select new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        };

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```

В следующем примере используется инструкция **From … From …** (эквивалент метода SelectMany) для выделения всех заказов, в которых значение *TotalDue* меньше 500,00.

```
decimal totalDue = 500.00M;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from contact in contacts
        from order in orders
        where contact.ContactID == order.Contact.ContactID
            && order.TotalDue < totalDue
```

```
        select new
        {
            ContactID = contact.ContactID,
            LastName = contact.LastName,
            FirstName = contact.FirstName,
            OrderID = order.SalesOrderID,
            Total = order.TotalDue
        };

    foreach (var smallOrder in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3}
Total Due: ${4} ",
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
            smallOrder.OrderID, smallOrder.Total);
    }
}
```

Примеры синтаксиса выражений запросов: фильтрация

В следующем примере возвращаются заказы, объем которых больше 2 и меньше 6.

```
int orderQtyMin = 2;
int orderQtyMax = 6;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from order in context.SalesOrderDetails
        where order.OrderQty > orderQtyMin && order.OrderQty < orderQtyMax
        select new
        {
            SalesOrderID = order.SalesOrderID,
            OrderQty = order.OrderQty
        };

    foreach (var order in query)
    {
        Console.WriteLine("Order ID: {0} Order quantity: {1}",
            order.SalesOrderID, order.OrderQty);
    }
}
```

Примеры синтаксиса выражений запросов: упорядочение. Примеры в этом разделе демонстрируют, как использовать методы OrderBy и OrderByDescending для выполнения запросов к модели БД с использованием синтаксиса выражений запроса.

В следующем примере выражение OrderBy используется для возвращения списка контактов, упорядоченного по фамилии.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedNames =
        from n in context.Contacts
        orderby n.LastName
        select n;
```

```
    Console.WriteLine("The sorted list of last names:");
    foreach (Contact n in sortedNames)
    {
        Console.WriteLine(n.LastName);
    }
}
```

Примеры синтаксиса выражений запросов: Операторы соединения.

Соединение - важная операция в запросах, которые обращаются к источникам данных без доступных для навигации взаимосвязей, например к таблицам реляционной базы данных. Соединение двух источников данных представляет собой взаимосвязь объектов одного источника данных с объектами, использующими общий атрибут в другом источнике данных.

Примеры в этом разделе демонстрируют, как использовать методы GroupJoin и Join для выполнения запросов к модели БД с использованием синтаксиса выражений запроса. Модель AdventureWorks Sales, которая используется в этих примерах, состоит из таблиц Contact, Address, Product, SalesOrderHeader и SalesOrderDetail образца базы данных AdventureWorks.

GroupJoin. Пример. В следующем примере выполняется соединение GroupJoin таблиц SalesOrderHeader и SalesOrderDetail, чтобы найти количество заказов для каждого клиента.Групповое соединение эквивалентно левому внешнему соединению, которое возвращает каждый элемент первого (левого) источника данных, даже если в другом источнике данных не имеется соответствующих элементов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query =
        from order in orders
        join detail in details
        on order.SalesOrderID
        equals detail.SalesOrderID into orderGroup
        select new
        {
            CustomerID = order.SalesOrderID,
            OrderCount = orderGroup.Count()
        };

    foreach (var order in query)
    {
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}",
            order.CustomerID,
            order.OrderCount);
    }
```

```
}
```

Join.

Пример. В следующем примере выполняется соединение таблиц SalesOrderHeader и SalesOrderDetail для определения количества заказов через Интернет за август.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query =
        from order in orders
        join detail in details
        on order.SalesOrderID equals detail.SalesOrderID
        where order.OnlineOrderFlag == true
        && order.OrderDate.Month == 8
        select new
        {
            SalesOrderID = order.SalesOrderID,
            SalesOrderDetailID = detail.SalesOrderDetailID,
            OrderDate = order.OrderDate,
            ProductID = detail.ProductID
        };

    foreach (var order in query)
    {
        Console.WriteLine("{0}\t{1}\t{2:d}\t{3}",
            order.SalesOrderID,
            order.SalesOrderDetailID,
            order.OrderDate,
            order.ProductID);
    }
}
```

7.3 СИНТАКСИС ЗАПРОСОВ, ОСНОВАННЫХ НА МЕТОДЕ (LINQ TO ENTITIES)

Другим способом создания запросов LINQ to Entities является синтаксис запроса, основанного на методе. Он представляет собой последовательность непосредственных вызовов методов операторов LINQ, передающих лямбда-выражения в качестве параметров.

Дополнительные сведения см. в разделе Лямбда-выражения (Руководство по программированию в C#).

Примеры, показывающие применение синтаксиса на основе методов, приведены ниже.

В примерах, приведенных в этом разделе, используются следующие инструкции **using**

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

Примеры синтаксиса запросов на основе методов: проекция

Примеры в этом разделе демонстрируют, как использовать методы Select и SelectMany для выполнения запросов к модели AdventureWorks Sales с использованием синтаксиса запросов на основе методов.Модель AdventureWorks Sales, которая используется в этих примерах, состоит из таблиц Contact, Address, Product, SalesOrderHeader и SalesOrderDetail образца базы данных AdventureWorks.

Пример. В следующем примере используется метод Select<TSource, TResult> для проецирования свойств **Product.Name** и **Product.ProductID** в последовательность анонимных типов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Products
        .Select(product => new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        });

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
```

```
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```

В следующем примере используется метод Select для возвращения последовательности только названий продуктов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> productNames = context.Products
        .Select(p => p.Name);

    Console.WriteLine("Product Names:");
    foreach (String productName in productNames)
    {
        Console.WriteLine(productName);
    }
}
```

SelectMany

Пример. В следующем примере используется метод **SelectMany** для выборки всех заказов, в которых *TotalDue* меньше 500,0

```
decimal totalDue = 500.00M;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
    contacts.SelectMany(
        contact => orders.Where(order =>
            (contact.ContactID == order.Contact.ContactID)
                && order.TotalDue < totalDue)
            .Select(order => new
            {
                ContactID = contact.ContactID,
                LastName = contact.LastName,
                FirstName = contact.FirstName,
                OrderID = order.SalesOrderID,
                Total = order.TotalDue
            }));

    foreach (var smallOrder in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3}
Total Due: ${4} ",
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
            smallOrder.OrderID, smallOrder.Total);
    }
}
```

Примеры синтаксиса запросов на основе методов: фильтрация

Примеры, приведенные в этом разделе, показывают, как использовать методы **Where** и **Where…Contains** для выполнения запросов к модели AdventureWorks Sales с применением синтаксиса запросов, основанного на методе.Учтите, что конструкцию Where…**Contains** нельзя использовать в составе скомпилированного запроса.

Модель AdventureWorks Sales, которая используется в этих примерах, состоит из таблиц Contact, Address, Product, SalesOrderHeader и SalesOrderDetail образца базы данных AdventureWorks.

Where

Пример. В следующем примере возвращаются все заказы, совершенные через Интернет.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var onlineOrders = context.SalesOrderHeaders
        .Where(order => order.OnlineOrderFlag == true)
        .Select(s => new { s.SalesOrderID, s.OrderDate, s.SalesOrderNumber
});

    foreach (var onlineOrder in onlineOrders)
    {
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number:
{2}",
            onlineOrder.SalesOrderID,
            onlineOrder.OrderDate,
            onlineOrder.SalesOrderNumber);
    }
}
```

Примеры синтаксиса запросов на основе методов: упорядочение

Примеры в этом разделе демонстрируют, как использовать метод ThenBy, чтобы выполнить запрос к модели AdventureWorks Sales с помощью синтаксиса запросов на основе методов.Модель AdventureWorks Sales, которая используется в этих примерах, состоит из таблиц Contact, Address, Product, SalesOrderHeader и SalesOrderDetail образца базы данных AdventureWorks.

ThenBy

Пример. В следующем примере, в синтаксисе запросов на основе методов, используются методы **OrderBy** и **ThenBy**, чтобы вернуть список контактов, отсортированный сначала по фамилии, а затем по имени.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
```

```
    IQueryable<Contact> sortedContacts = context.Contacts
        .OrderBy(c => c.LastName)
        .ThenBy(c => c.FirstName);

    Console.WriteLine("The list of contacts sorted by last name then by
first name:");
    foreach (Contact sortedContact in sortedContacts)
    {
        Console.WriteLine(sortedContact.LastName + ", " +
sortedContact.FirstName);
    }
}
```

ThenByDescending

Пример. В следующем примере используются методы **OrderBy** и **ThenByDescending**, чтобы вначале выполнить сортировку по стоимости, а затем сортировку по убыванию по названию продуктов.

```
sing (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IOrderedQueryable<Product> query = context.Products
        .OrderBy(product => product.ListPrice)
        .ThenByDescending(product => product.Name);

    foreach (Product product in query)
    {
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price
{2}",
            product.ProductID,
            product.Name,
            product.ListPrice);
    }
}
```

GroupJoin

Пример. В следующем примере выполняется соединение **GroupJoin** таблиц SalesOrderHeader и SalesOrderDetail, чтобы найти количество заказов для каждого клиента.Групповое соединение эквивалентно левому внешнему соединению, которое возвращает каждый элемент первого (левого) источника данных, даже если в другом источнике данных не имеется соответствующих элементов.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;
```

```
    var query = orders.GroupJoin(details,
        order => order.SalesOrderID,
        detail => detail.SalesOrderID,
        (order, orderGroup) => new
        {
            CustomerID = order.SalesOrderID,
            OrderCount = orderGroup.Count()
        });

    foreach (var order in query)
    {
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}",
            order.CustomerID,
            order.OrderCount);
    }

}
```

Join

Пример. В следующем примере выполняется соединение с таблицами Contact и SalesOrderHeader.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        contacts.Join(
            orders,
            order => order.ContactID,
            contact => contact.Contact.ContactID,
            (contact, order) => new
            {
                ContactID = contact.ContactID,
                SalesOrderID = order.SalesOrderID,
                FirstName = contact.FirstName,
                Lastname = contact.LastName,
                TotalDue = order.TotalDue
            });

    foreach (var contact_order in query)
    {
        Console.WriteLine("ContactID: {0} "
                        + "SalesOrderID: {1} "
                        + "FirstName: {2} "
                        + "Lastname: {3} "
                        + "TotalDue: {4}",
            contact_order.ContactID,
            contact_order.SalesOrderID,
            contact_order.FirstName,
            contact_order.Lastname,
            contact_order.TotalDue);
```

```
    }
}
```

Примеры синтаксиса запросов на основе методов: навигация по связям.

Свойства навигации в модели Entity Framework - это свойства быстрого доступа, используемые для нахождения сущностей в элементах ассоциации.Свойства навигации позволяют пользователю переходить от одной сущности к другой или от сущности к связанным сущностям в наборе ассоциаций.В этом разделе содержатся примеры синтаксиса запросов на основе методов для навигации по связям с помощью свойств навигации в запросах LINQ to Entities.

Модель AdventureWorks Sales, которая используется в этих примерах, состоит из таблиц Contact, Address, Product, SalesOrderHeader и SalesOrderDetail образца базы данных AdventureWorks.

Пример. В следующем примере синтаксиса запроса на основе методов используется метод SelectMany, чтобы получить все заказы контактных лиц с фамилией «Zhou».

Свойство навигации **Contact.SalesOrderHeader** используется для получения коллекции объектов **SalesOrderHeader** для каждого контактного лица.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> ordersQuery = context.Contacts
        .Where(c => c.LastName == lastName)
        .SelectMany(c => c.SalesOrderHeaders);

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due:
{2}",
            order.SalesOrderID, order.OrderDate, order.TotalDue);
    }
}
```

Источники.

https://msdn.microsoft.com/ru-ru/library/bb896310(v=vs.110).aspx
https://msdn.microsoft.com/ru-ru/library/bb399367(v=vs.110).aspx

ДОПОЛНИТЕЛЬНЫЙ МАТЕРИАЛ

Методы построителя запросов (ПЛАТФОРМА ENTITY FRAMEWORK)

Класс ObjectQuery поддерживает запросы LINQ to Entities и Entity SQL к концептуальной модели. Класс **ObjectQuery** также реализует набор методов построителя запросов, которые можно использовать для последовательного построения команд запросов, эквивалентных Entity SQL. Ниже приведены **методы построителя запросов ObjectQuery** наряду с эквивалентными инструкциями Entity SQL

Метод ObjectQuery	Инструкция Entity SQL
Distinct	DISTINCT
Except	EXCEPT
GroupBy	GROUP BY
Intersect	INTERSECT
OfType	OFTYPE
OrderBy	ORDER BY
Select	SELECT
SelectValue	SELECT VALUE
Skip	SKIP
Top	TOP и LIMIT
Union	UNION
UnionAll	UNION ALL
Where	where

Каждый метод построителя запросов возвращает новый экземпляр **ObjectQuery**. Это позволяет построить запрос, результирующий набор которого основан на операциях последовательности предыдущих экземпляров **ObjectQuery**. В следующем примере показано применение

метода **Where** для фильтрации возвращенных объектов **Product** по значению **ProductID**.

C#
```
// Return Product objects with the specified ID.
ObjectQuery<Product> query =
   context.Products
   .Where("it.ProductID = @product",
   new ObjectParameter("product", productId));
```

Поскольку экземпляр **ObjectQuery** реализует методы IQueryable и IEnumerable, можно объединить методы построителя запросов, реализованные в **ObjectQuery**, со стандартными методами оператора запроса LINQ, такими как First и Count. В отличие от методов построителя запросов операторы LINQ не возвращают экземпляр **ObjectQuery**. Дополнительные сведения см. в разделе Общие сведения о стандартных операторах запросов в документации по Visual Studio 2008.

Выбор данных

По умолчанию **ObjectQuery** возвращает нуль или больше объектов сущности конкретного типа. Вызов последующих методов запросов, таких как **Where** и **OrderBy**, влияет на коллекцию объектов, возвращаемых первоначальным методом **ObjectQuery**. Некоторые методы, такие как **Select** и **GroupBy**, возвращают вместо типа сущности проекцию данных, такую как DbDataRecord. Дополнительные сведения см. в разделе Запросы объектов (платформа Entity Framework). В следующем примере возвращается коллекция объектов **DbDataRecord**, содержащая вложенные типы сущностей **SalesOrderHeader**.

C#
```
// Define a query that returns a nested
// DbDataRecord for the projection.
ObjectQuery<DbDataRecord> query =
    context.Contacts.Select("it.FirstName, "
        + "it.LastName, it.SalesOrderHeaders")
    .Where("it.LastName = @ln", new ObjectParameter("ln", lastName));
```

Хотя методы построителя запросов применяются последовательно, можно построить такой же тип вложенных запросов, который поддерживается языком Entity SQL . Для этого необходимо включить подзапрос в метод как код Entity SQL . В следующем примере подзапрос Entity SQL SELECT используется в методе **Select**, чтобы включить записи **LastName**, вложенные в результирующий набор и отсортированные в алфавитном порядке по первой букве фамилии:

C#
VB
```
// Define the query with a GROUP BY clause that returns
// a set of nested LastName records grouped by first letter.
ObjectQuery<DbDataRecord> query =
    context.Contacts
    .GroupBy("SUBSTRING(it.LastName, 1, 1) AS ln", "ln")
    .Select("it.ln AS ln, (SELECT c1.LastName " +
    "FROM AdventureWorksEntities.Contacts AS c1 " +
    "WHERE SubString(c1.LastName, 1, 1) = it.ln) AS CONTACT")
    .OrderBy("it.ln");
```

Примечание
Используйте метод ToTraceString, чтобы увидеть команду источника данных, которая будет сформирована методом **ObjectQuery**.Дополнительные сведения см. в разделе Запросы объектов (платформа Entity Framework).

Псевдонимы

Методы построителя запросов применяются последовательно, чтобы построить совокупную команду запроса. Это означает, что текущая команда **ObjectQuery** рассматривается как подзапрос, к которому применяется текущий метод.

Примечание
Свойство CommandText возвращает команду для экземпляра **ObjectQuery**.

В методе построителя запросов для ссылки на текущую команду **ObjectQuery** используется псевдоним. По умолчанию строка «it» является псевдонимом, которая представляет текущую команду, как в следующем примере:

C#
VB

```
int cost = 10;
// Return Product objects with a standard cost
// above 10 dollars.
ObjectQuery<Product> productQuery =
    context.Products
    .Where("it.StandardCost > @cost", new ObjectParameter("cost", cost));
```

Если свойству Name присваивается значение **ObjectQuery**, это значение становится псевдонимом в последующих методах. Предыдущий пример расширен в следующем примере путем задания для имени **ObjectQuery** значения «product» и последующим использованием этого псевдонима в последующем методе **OrderBy**.

C#
VB

```
int cost = 10;
// Return Product objects with a standard cost
// above 10 dollars.
ObjectQuery<Product> productQuery =
    context.Products
    .Where("it.StandardCost > @cost", new ObjectParameter("cost", cost));

// Set the Name property for the query and then
// use that name as the alias in the subsequent
```

```
// OrderBy method.
productQuery.Name = "product";
ObjectQuery<Product> filteredProduct = productQuery
    .OrderBy("product.ProductID");
```

Параметры

Все методы построителя запросов, которые принимают входную строку Entity SQL , также поддерживают параметризованные запросы.Имена параметров в Entity SQL определены в выражениях запросов с символом (@) в качестве префикса. Дополнительные сведения см. в разделе Параметры (язык Entity SQL). Параметры передаются в методы построителя запросов как массив экземпляров ObjectParameter. В следующем примере два параметра передаются в метод **Where**:

C#

VB

```
// Get the contacts with the specified name.
ObjectQuery<Contact> contactQuery = context.Contacts
    .Where("it.LastName = @ln AND it.FirstName = @fn",
    new ObjectParameter("ln", lastName),
    new ObjectParameter("fn", firstName));
```

Замечания по использованию параметров

При использовании параметров с методами построителя запросов следует учитывать следующие замечания.

- Параметры, переданные в методы построителя запросов, собираются последующими экземплярами **ObjectQuery** в последовательности. К ним можно обратиться с помощью свойства Parameters. После добавления параметров их можно удалить из коллекции и очистить коллекцию до момента компиляции или выполнения запроса. Имена параметров изменить нельзя, но значения можно изменить в любое время.

- Параметры в коллекции ObjectParameterCollection должны быть уникальными. Коллекция не может содержать два параметра с одинаковыми именами.

- При использовании композиционных методов, таких как **Union**, **UnionAll**, **Intersect** и **Except**, коллекции параметров объединяются.Исключение возникает в случае несовместимых, неполных наборов параметров или если в коллекциях параметров обоих запросов существуют одинаковые имена.

Источник
https://msdn.microsoft.com/ru-ru/library/bb896238(v=vs.100).aspx
https://msdn.microsoft.com/ru-ru/library/bb386964(v=vs.110).aspx
https://msdn.microsoft.com/ru-ru/library/bb397947(v=vs.120).aspx
https://msdn.microsoft.com/ru-ru/library/bb738551(v=vs.110).aspx

9 ЛЕКЦИЯ. СТАНДАРТНЫЕ ОПЕРАТОРЫ ЗАПРОСОВ В ЗАПРОСАХ LINQ TO ENTITIES

В запросе указывается, какие данные надо получить из источника. В запросе можно также указать, как следует сортировать, группировать и формировать возвращаемую информацию. Технология LINQ предоставляет набор стандартных методов запросов, которые можно использовать в запросах. Большинство этих методов работают с последовательностями.

В данном контексте последовательность - это объект, тип которого реализует интерфейс IEnumerable<T> или IQueryable<T>.

Функциональные возможности стандартных операторов запросов включают

- фильтрацию,
- проекцию,
- статистическую обработку,
- сортировку,
- группирование,
- разбиение на страницы и т. д.

У некоторых наиболее часто используемых стандартных операторов запросов существует выделенный синтаксис ключевого слова, поэтому оператор можно вызвать с помощью синтаксиса выражения запроса. Выражение запроса - это другой, более легко читаемый способ создания запросов, отличающийся от способа, основанного на использовании методов. Предложения выражений запросов преобразуются в вызовы методов запросов во время компиляции. Список стандартных операторов запросов, имеющих эквивалентные предложения выражений запросов, см. в разделе Standard Query Operators Overview.

8.1 Методы проекции и фильтрации

Проекция - это преобразование элементов результирующего набора в нужную форму. Например, можно проецировать необходимое подмножество свойств из каждого объекта в результирующем наборе, или свойство и произвести с ним математическое вычисление, или целый объект из результирующего набора. Методами проекции являются методы **Select** и **SelectMany**.

Фильтрация - это операция по ограничению значений в результирующем наборе только элементами, соответствующими указанному условию.Методом фильтрации является метод **Where.**

Большинство перегрузок методов проекции и фильтрации поддерживаются в LINQ to Entities, за исключением тех, которые принимают позиционные аргументы.

8.2 Методы соединения

Соединение является важной операцией в запросах, обращающихся к целевым источникам данных, не имеющим связей друг с другом, по которым можно перемещаться. Соединение двух источников данных - это ассоциация объектов в одном источнике данных с объектами в другом источнике, имеющем общий атрибут или свойство. Методами соединения являются методы **Join** и **GroupJoin**.

Поддерживается большинство перегрузок методов соединения, за исключением тех, в которых используется интерфейс IEqualityComparer<T>. Это происходит потому, что сравнивающий класс нельзя преобразовать в источник данных.

8.3 Методы наборов

Операции с наборами в LINQ — это операции запросов, результирующие наборы которых основываются на наличии или отсутствии эквивалентных элементов в одной или другой коллекции (или наборе).

Методами наборов являются методы: **All**, **Any**, **Concat**, **Contains**, **DefaultIfEmpty**, **Distinct**, **EqualAll**, **Except**, **Intersect** и **Union**.

Большинство перегрузок методов наборов поддерживаются в LINQ to Entities, хотя и существуют некоторые отличия в поведении по сравнению с технологией LINQ to Objects.Однако методы наборов, в которых используется интерфейс IEqualityComparer<T>, не поддерживаются, поскольку сравнивающий класс нельзя преобразовать в источник данных.

8.3 Методы упорядочивания

Упорядочение (или сортировка) — это упорядочение элементов в результирующем наборе на основе одного или нескольких атрибутов.Указав более одного критерия сортировки, можно разорвать связи внутри группы.

Поддерживается большинство перегрузок методов упорядочивания, за исключением тех, в которых используется интерфейс IComparer<T>.Это происходит потому, что сравнивающий класс нельзя преобразовать в источник данных.Методами упорядочивания являются методы **OrderBy**, **OrderByDescending**, **ThenBy**, **ThenByDescending** и **Reverse**.

Поскольку запрос выполняется к источнику данных, поведение упорядочивания может отличаться от поведения запросов, выполненных в среде CLR.Это происходит, потому что в источнике данных могут быть заданы параметры упорядочения, такие как упорядочение с учетом регистра, упорядочение с учетом символов кандзи и упорядочение значений null.В зависимости от источника данных эти параметры упорядочения могут приводить к результатам, отличающимся от результатов в среде CLR.

Если один и тот же селектор ключа будет указан в более чем одной операции сортировки, будет выполнена повторяющаяся сортировка.Это недопустимо, поэтому будет выдано исключение.

8.4 Методы группирования

Группирование — это размещение данных в группах таким образом, чтобы у элементов в каждой группе был общий атрибут.Методом группирования является метод **GroupBy**.

Поддерживается большинство перегрузок методов группирования, за исключением тех, в которых используется интерфейс IEqualityComparer<T>.Это происходит потому, что сравнивающий класс нельзя преобразовать в источник данных.

Методы группирования сопоставляются с источником данных с помощью отдельного вложенного запроса для селектора ключа.Вложенный запрос сравнения селектора ключа выполняется с использованием семантики источника данных, включая и вопросы, связанные со сравнением значений **null**.

8.5 Методы статистической обработки

Статистическая операция вычисляет одно значение по коллекции значений.Например, статистической операцией является вычисление средней дневной температуры с использованием значений дневной температуры за месяц.Статистическими методами являются методы **Aggregate**, **Average**, **Count**, **LongCount**, **Max**, **Min** и **Sum**.

Поддерживается перегрузка большинства статистических методов.Поведение методов статистической обработки, связанное со значениями null, определяется семантикой источника данных.Поведение методов статистической обработки может отличаться в том случае, если присутствуют значения null. Это зависит от используемого конечного источника данных.Поведение методов статистической обработки с использованием семантики источника данных также может отличаться от поведения, ожидаемого от методов среды CLR.Например, по умолчанию метод **Sum** на SQL Server пропускает значения null, а не вызывает исключение.

Любые исключения, являющиеся результатом статистической обработки, например переполнение, вызванное функцией **Sum**, вызываются как исключения источников данных или как исключения платформы Entity Framework во время материализации результатов запросов.

В методах, в которых выполняется вычисление над последовательностью, таких как **Sum** или**Average**, реальное вычисление выполняется на сервере.В результате на сервере могут произойти преобразования типов и потеря точности, а результаты могут отличаться от результатов, ожидаемых при использовании семантики среды CLR.

Поведение по умолчанию методов статистической обработки, по отношению к значениям NULL и значениям, отличным от NULL, отображено в следующей таблице:

Метод	Нет данных	Все значения NULL	Некоторые значения NULL	Нет значений NULL
Average	Возвращает значение NULL.	Возвращает значение NULL.	Возвращает среднее значение от значений последовательности, отличных от NULL.	Вычисляет среднее значение для последовательности числовых значений.
COUNT	Возвращает значение 0	Возвращает число значений null в последовательности.	Возвращает число значений null и значений, отличных от null, в последовательности.	Возвращает число элементов в последовательности.
max	Возвращает значение NULL.	Возвращает значение NULL.	Возвращает максимальное значение, отличное от NULL, в последовательности.	Возвращает максимальное значение в последовательности.
min	Возвращает значение NULL.	Возвращает значение NULL.	Возвращает минимальное значение, отличное от NULL, в последовательности.	Возвращает минимальное значение в последовательности.
Sum	Возвращает значение NULL.	Возвращает значение NULL.	Возвращает сумму значений, отличных от NULL, в последовательности.	Вычисляет сумму последовательности числовых значений.

10 ЛЕКЦИЯ. HTML-хелперы

ASP.NET MVC- **HTML HELPERS**

В ASP.NET MVC HTML хелперы очень похожи на традиционные элементы управления веб-формы. Но HTML хелперы являются более легкими по сравнению с элементами управления ASP.NET. В отличие от элементов управления Web Form, HTML- хелперы не имеет модели событий и состояния представлений.

В ASP.NET MVC вы можете создавать свои собственные HTML хелперы, или использовать встроенные. В большинстве случаев, HTML хелпер это просто метод, который возвращает строку.

9.1 Стандартные HTML хелперы

HTML Links

Простым способом визуализации HTML ссылки является использование - Html.ActionLink (). В ASP.NET MVC Html.ActionLink () не связывает с представлением, создает ссылку на действие контроллера.

Razor Syntax:

```
@Html.ActionLink("About this Website", "About")
```

Первый параметр это текст ссылки, а второй параметр определяет действие контроллера.

Html.ActionLink() хелпер выводит следующий HTML:

```
<a href="/Home/About">About this Website</a>
```

9.2 Как мы увидели из прошлых примеров, представления используют разметку html для визуализации содержимого. Однако фреймворк ASP.NET MVC обладает также таким мощным инструментом как HTML-хелперы, позволяющие генерировать html-код.

Строчные хелперы

Строчные хелперы похожи на обычные определения методов на языке C#, только начинаются с тега @helper. Например, создадим в представлении хелпер для вывода названий книг в виде списка:

```
@helper BookList(IEnumerable<BookStore.Models.Book> books)
{
   <ul>
      @foreach (BookStore.Models.Book b in books)
```

```
    {
        <li>@b.Name</li>
    }
</ul>
}
```

Данный хелпер мы можем определить в любом месте представления. И также в любом месте представления мы можем его использовать, передавая в него объект IEnumerable<BookStore.Models.Book>:

```
<h3>Список книг</h3>
@BookList(ViewBag.Books)
<!-- или если используется строго типизированное представление -->
@BookList(Model)
```

Строчные html-хелперы удобно использовать, если необходимо создать один метод, который предполагается использовать в представлении многократно. Например:

```
@helper CreateList(string[] all)
{
    <ul>
        @foreach (string s in all)
        {
            <li>@s</li>
        }
    </ul>
}
@{
    string[] cities = new string[] { "Лондон", "Париж", "Москва" };
}
@{
    string[] countries = new string[] { "Великобритания", "Франция", "Россия"
};
}
<h3>Города</h3>
@CreateList(cities)
<br />
<h3>Страны</h3>
@CreateList(countries)
```

При отсутсвии подобного хелпера, то нам бы пришлось по сути дублировать один и тот же html-код для создания списка. Однако этот хелпер еще довольно простой, а если нам приходится создавать по сто раз более сложную, но однотипную разметку html, тогда хелперы окажутся еще более полезными.

Но данный подход имеет один недостаток - если хелпер очень объемный, то он может очень сильно захламлять разметку представления. И в этом случае его лучше вынести в отдельный файл кода. Так, перепишем предыдущий пример. Для этого нам надо создать новый класс с методом расширения - то есть таким методом, который расширяет функциональность уже существующих классов. А эти классы указываются в качестве первого параметра метода. Итак, создадим в проекте новую папку Helpers и добавим в нее новый класс ListHelper:

```
using System;
using System.Web;
using System.Web.Mvc;
using System.Linq;

namespace BookStore.Helpers
{
   public static class ListHelper
   {
      public static MvcHtmlString CreateList(this HtmlHelper html, string[]
items)
      {
         TagBuilder ul = new TagBuilder("ul");
         foreach (string item in items)
         {
            TagBuilder li = new TagBuilder("li");
            li.SetInnerText(item);
            ul.InnerHtml += li.ToString();
         }
         return new MvcHtmlString(ul.ToString());
      }
   }
}
```

В новом классе хелпера определен один статический метод CreateList, принимающий в качестве первого параметра объект, для которого создается метод. Так как данный метод расширяет функциональность html-хелперов,

которые представляет класс HtmlHelper, то именно объект этого типа и передается в данном случае в качестве первого параметра. Второй параметр метода CreateList - массив строк-значений, которые потом будут выводиться в списке.

В самом методе с помощью объекта TagBuilder конструируется стандартный элемент html - элемент ul. При обходе массива все строковые значения обертываются в тег li и добавляются в список. И на выходе возвращается полноценные элемент ul.

Класс TagBuilder имеет ряд членов, которые можно использовать при таком подходе:

- Свойство **InnerHtml** позволяет установить или получить содержимое тега в виде строки
- Метод **MergeAttribute (string, string, bool)** позволяет добавить к элементу один атрибут. Для получения всех атрибутов можно использовать коллекцию **Attributes**
- Метод **SetInnerText(string)** устанавливает текстовое содержимое внутри элемента
- Метод **AddCssClass(sting)** добавляет класс css к элементу

После создания нового хелпера мы его можем использовать в представлении. Перепишем предыдущий пример следующим образом:

```
@{
   string[] cities = new string[] { "Лондон", "Париж", "Москва" };
}
@{
   string[] countries = new string[] { "Великобритания", "Франция", "Россия"
};
}
@using BookStore.Helpers
<h3>Города</h3>
@Html.CreateList(cities)
<br />
<h3>Страны</h3>
<!-- или можно вызвать так -->
@ListHelper.CreateList(Html, countries)
```

Добавление атрибутов

Стандартные реализации хелперов позволяют добавить к создаваемым элементам атрибуты, например, атрибут class, id и другие. В кастомные хелперы мы также можем добавлять атрибуты. Например, изменим хелпер CreateList следующим образом:

```
using System.Collections.Generic;
using System.Linq;
```

```
using System.Reflection;
using System.Web.Mvc;

public static class ListHelper
{
   public static MvcHtmlString CreateList(this HtmlHelper html, string[] items,
object obj)
   {
      var type = obj.GetType();
      var props = type.GetProperties();

      Dictionary<string, string> dic= props.ToDictionary(x => x.Name, x =>
x.GetValue(obj, null).ToString());

      TagBuilder ul = new TagBuilder("ul");
      foreach (string item in items)
      {
         TagBuilder li = new TagBuilder("li");
         li.SetInnerText(item);
         ul.InnerHtml += li.ToString();
      }
      foreach (var attr in dic)
      {
         ul.MergeAttribute(attr.Key.ToString(), attr.Value.ToString());
      }
      return new MvcHtmlString(ul.ToString());
   }
}
```

Теперь в качестве второго параметра передается объект obj, свойства которого должны представлять собой передаваемые в хелпер атрибуты. С помощью механизма рефлексии этот объект конвертируется в словарь Dictionary.

И в этом случае в коде представления мы можем передать в хелпер значения для атрибутов:

```
@Html.CreateList(new string[] { "Лондон", "Париж", "Берлин" }, new {
@class = "btn", id = "citiesList" })
```

11 ЛЕКЦИЯ. РАБОТА С ФОРМАМИ

11.1 Элемены управления формы HTML

Следующие HTML хелперы используются для вывода HTML элементов управления:

- BeginForm()
- EndForm()
- TextArea()
- TextBox()
- CheckBox()
- RadioButton()
- ListBox()
- DropDownList()
- Hidden()
- Password()

Пример. ASP.NET Syntax C#:

```
<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>
<% using (Html.BeginForm()){%>
<p>
<label for="FirstName">First Name:</label>
<%= Html.TextBox("FirstName") %>
<%= Html.ValidationMessage("FirstName", "*") %>
</p>
<p>
<label for="LastName">Last Name:</label>
<%= Html.TextBox("LastName") %>
<%= Html.ValidationMessage("LastName", "*") %>
</p>
<p>
<label for="Password">Password:</label>
<%= Html.Password("Password") %>
<%= Html.ValidationMessage("Password", "*") %>
</p>
<p>
<label for="Password">Confirm Password:</label>
<%= Html.Password("ConfirmPassword") %>
<%= Html.ValidationMessage("ConfirmPassword", "*") %>
</p>
<p>
```

```
<label for="Profile">Profile:</label>
<%= Html.TextArea("Profile", new {cols=60, rows=10})%>
</p>
<p>
<%= Html.CheckBox("ReceiveNewsletter") %>
<label for="ReceiveNewsletter" style="display:inline">Receive
Newsletter?</label>
</p>
<p>
<input type="submit" value="Register" />
</p>
<%}%>
```

10.2 Хотя мы можем сами написать любой требуемый хелпер, но фреймворк MVC уже предоставляет большой набор встроенных html-хелперов, которые позволяют генерировать ту или иную разметку, главным образом, для работы с формами. Поэтому в большинстве случаев не придется создавать свои хелперы, и можно будет воспользоваться встроенными.

Хелпер Html.BeginForm

Для создания форм мы вполне можем использовать стандартные элементы html, например:

```
<form method="post" action="/Home/Buy">

    <input type="hidden" value="@ViewBag.BookId" name="BookId" />

    <table>

        <tr><td><p>Введите свое имя </p></td>

            <td><input type="text" name="Person" /> </td></tr>

        <tr><td><p>Введите адрес :</p></td>

            <td><input type="text" name="Address" /> </td></tr>

        <tr><td><input type="submit" value="Отправить" /> </td>

                <td></td></tr>

    </table>

</form>
```

Это обычная html-форма, которая по нажатию на кнопку отправляет все введенные данные запросом POST на адрес /Home/Buy. Встроенный хелпер **BeginForm/EndForm** позволяет создать ту же самую форму:

```
@using(Html.BeginForm("Buy", "Home", FormMethod.Post))
{
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <table>
        <tr><td><p>Введите свое имя </p></td>
            <td><input type="text" name="Person" /> </td></tr>
        <tr><td><p>Введите адрес :</p></td>
            <td><input type="text" name="Address" /> </td></tr>
        <tr><td><input type="submit" value="Отправить" /> </td>
                <td></td></tr>
    </table>
}
```

Метод **BeginForm** принимает в качестве параметров имя метода действия и имя контроллера, а также тип запроса. Данный хелпер создает как открывающий тег <form>, так и закрывающий тег </form>. Поэтому при рендеринге представления в выходной поток у нас получится тот же самый html-код, что и с применением тега form. Поэтому оба способа идентичны.

Здесь есть один момент. Если у нас в контроллере определены две версии одного метода - для методов POST и GET, например:

```
[HttpGet]
public ActionResult Buy()
{
    return View();
}

[HttpPost]
public string Buy(Purchase purchase)
{
    ..............
    return "Спасибо за покупку книги";;
}
```

То есть фактически вызов страницы с формой и отправка формы осуществляется одним и тем же действием Buy. В этом случае можно не указывать в хелпере `Html.BeginForm` параметры:

```
@using(Html.BeginForm())
{
    .............
}
```

Ввод информации

В предыдущем примере вместе с хелпером Html.BeginForm использовались стандартные элементы html. Однако набор html-хелперов содержит также хелперы для ввода информации пользователем. В MVC определен широкий набор хелперов ввода практически для каждого html-элемента. Что выбрать - хелпер или стандартный элементы ввода html, уже решает сам разработчик.

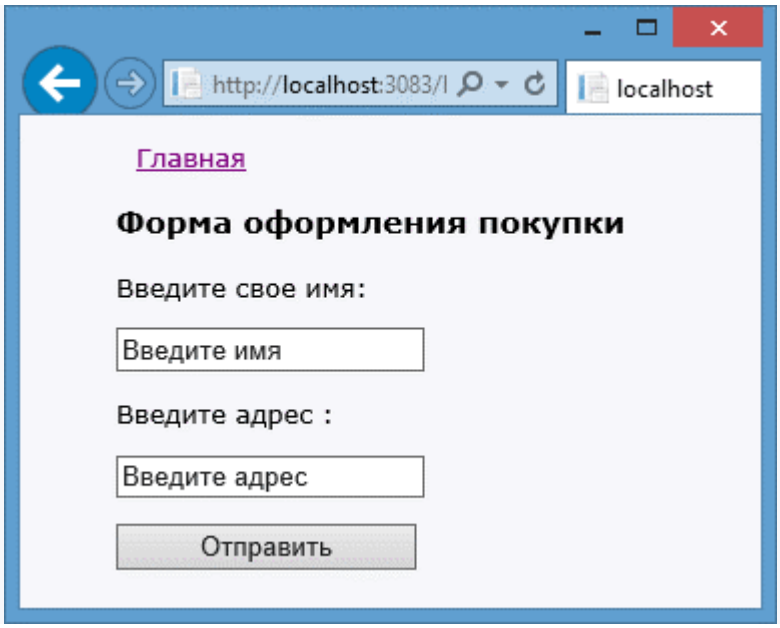
Вне зависимости от типа все базовые html-хелперы используют как минимум два параметра: первый параметр применяется для установки значений для атрибутов `id` и `name`, а второй параметр - для установки значения атрибута `value`

Html.TextBox

Хелпер `Html.TextBox` генерирует тег `input` со значением атрибута `type` равным `text`. Хелпер `TextBox` используют для получения ввода пользователем информации. Так, перепишем предыдущую форму с заменой полей ввода на хелпер Html.TextBox:

```
@using(Html.BeginForm("Buy", "Home", FormMethod.Post))
{
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <p>Введите свое имя: </p>
    @Html.TextBox("Person", "Введите имя")
    <p>Введите адрес :</p>
    @Html.TextBox("Address", "Введите адрес")
    <p><input type="submit" value="Отправить" /></p>
}
```

Мы получим тот же результат:



Html.TextArea

Хелпер `TextArea` используется для создания элемента `<textarea>`, который представляет многострочное текстовое поле. Результатом выражения `@Html.TextArea("text", "привет <br/> мир")`

будет следующая html-разметка:

```
<textarea cols="20" id="text" name="text" rows="2">привет <br/> мир

</textarea>
```

Обратите внимание, что хелпер декодирует помещаемое в него значение, в том числе и html-теги, (все хелперы декодируют значения моделей и значения атрибутов). Другие версии хелпера `TextArea` позволяют указать число строк и столбцов, определяющих размер текстового поля.

```
@Html.TextArea("text", "привет <br /> мир", 5, 50, null)
```

Этот хелпер сгенерирует следующую разметку:

```
1    <textarea cols="50" id="text" name="text" rows="5">привет <br /> мир
2    </textarea>
```

Html.Hidden

В примере с формой мы использовали скрытое поле `input type="hidden"`, вместо которого могли бы вполне использовать хелпер`Html.Hidden`. Так, следующий вызов хелпера:

```
@Html.Hidden("BookId", "2")
```

сгенерирует разметку:

```
1        <input id="BookId" name="BookId" type="hidden" value="2" />
```

А при передаче переменной из ViewBag нам надо привести ее к типу string: `@Html.Hidden("BookId", @ViewBag.BookId as string)`

Html.Password

`Html.Password` создает поле для ввода пароля. Он похож на хелпер `TextBox`, но вместо введенных символов отображает маску пароля. Следующий код:

```
@Html.Password("UserPassword", "val")
```

генерирует разметку:

```
<input id="UserPassword" name="UserPassword" type="password" value="val"
/>
```

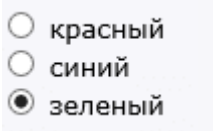
Html.RadioButton

Для создания переключателей применяется хелпер `Html.RadioButton`. Он генерирует элемент `input` со значением `type="radio"`. Для создания группы переключателей, надо присвоить всем им одно и то же имя (свойство `name`):

```
@Html.RadioButton("color", "red")
<span>красный</span> <br />
@Html.RadioButton("color", "blue")
<span>синий</span> <br />
@Html.RadioButton("color", "green", true)
<span>зеленый</span>
```

Этот код создает следующую разметку:

```
<input id="color" name="color" type="radio" value="red" />
        <span>красный</span> <br />
        <input id="color" name="color" type="radio" value="blue" />
<span>синий</span> <br />
<input checked="checked" id="color" name="color" type="radio" value="green" />
<span>зеленый</span>
```



Html.CheckBox

`Html.CheckBox` может применяться для создания сразу двух элементов. Возьмем, к примеру, следующий код:

```
@Html.CheckBox("Enable", false)
```

Это выражение будет генерировать следующий HTML:

```
1<input id="Enable" name="Enable" type="checkbox" value="true" />
2<input name="Enable" type="hidden" value="false" />
```

То есть кроме собственно поля флажка, еще и генерируется скрытое поле. Зачем оно нужно? Дело в том, что браузер посылает значение флажка только тогда, когда флажок выбран или отмечен. А скрытое поле гарантирует, что для элемента `Enable` будет установлено значение даже, если пользователь не отметил флажок.

Html.Label

Хелпер `Html.Label` создает элемент `<label/>`, а передаваемый в хелпер параметр определяет значение атрибута `for` и одновременно текст на элементе. Перегруженная версия хелпера позволяет определить значение атрибута `for` и текст на метке независимо друг от друга. Например, объявление хелпера `Html.Label("Name")` создает следующую разметку:

```
1<label for="Name">Name</label>
```

Элемент `label` представляет простую метку, предназначенную для прикрепления информации к элементам ввода, например, к текстовым полям. Атрибут `for` элемента `label` должен содержать ID ассоциированного элемента ввода. Если пользователь нажимает на метку, то браузер автоматически передает фокус связанному с этой меткой элементу ввода.

Html.DropDownList

Хелпер `Html.DropDownList` создает выпадающий список, то есть элемент `<select />`. Для генерации такого списка нужна коллекция объектов `SelectListItem`, которые представляют элементы списка. Объект `SelectListItem` имеет свойства `Text` (отображаемый текст),`Value` (само значение, которое может не совпадать с текстом) и `Selected`. Можно создать коллекцию объектов `SelectListItem` или использовать хелпер `SelectList`. Этот хелпер просматривает объекты `IEnumerable` и преобразуют их в последовательность объектов`SelectListItem`. Так, код -

```
@Html.DropDownList("countires", new SelectList(new string[] {"Russia","USA", "Canada","France"}),"Countries")
```

генерирует следующую разметку:

```
1    <select id="countires" name="countires"><option value="">Countries</optio
2    <option>Russia</option>
3    <option>USA</option>
4    <option>Canada</option>
5    <option>France</option>
6    </select>
```

Теперь более сложный пример. Выведем в список коллекцию элементов Book. В контроллере передадим этот список через ViewBag:

```
BookContext db = new BookContext();

public ActionResult Index()
{
   SelectList books = new SelectList(db.Books, "Author", "Name");
   ViewBag.Books = books;
   return View();
}
```

Здесь мы создаем объект SelectList, передавая в его **конструктор** набор значений для списка (db.Books), название свойства модели Book, которое будет использоваться в качестве значения (Author), и название свойства модели Book, которое будет использоваться для отображения в списке. В данном случае необязательно устанавливать два разных свойства, можно было и одно установить и для значения и отображения.

Тогда в представлении мы можем так использовать этот SelectList:

```
@Html.DropDownList("Author", ViewBag.Books as SelectList)
```

И при рендеринге представления все элементы SelectList добавятся в выпадающий список

Html.ListBox

Хелпер `Html.ListBox`, также как и `DropDownList`, создает элемент `<select />`, но при этом делает возможным множественное выделение элементов (то есть для атрибута `multiple` устанавливается значение `multiple`). Для создания списка, поддерживающего множественное выделение, вместо `SelectList` можно использовать класс `MultiSelectList`:

```
@Html.ListBox("countires", new MultiSelectList(new string[] {"Россия","США", "Китай
```

Этот код генерирует следующую разметку:

```
<select Length="9" id="countries" multiple="multiple" name="countires">
<option>Россия</option>
```

```
<option>США</option>
<option>Китай</option>
<option>Индия</option>
</select>
```

С передачей одиночных значений на сервер все понятно, но как передать множественные значения? Допустим, у нас есть следующая форма:

```
@using (Html.BeginForm())
{
   @Html.ListBox("countries",
         new MultiSelectList(new string[] { "Россия", "США", "Китай", "Индия"
}))
   <p><input type="submit" value="Отправить" /></p>
}
```

Тогда метод контроллера мог бы получать эти значения следующим образом:

```
[HttpPost]
public string Index(string[] countries)
{
   string result = "";
   foreach (string c in countries)
   {
      result += c;
      result += ";";
   }
   return "Вы выбрали: " + result;
}
```

Форма с несколькими кнопками

Как правило, на форме есть только одна кнопка для отправки. Однако в определенных ситуациях может возникнуть потребность, использовать более одной кнопки. Например, есть поле для ввода значения, а две кнопки указывают, надо это значение удалить или, наоборот, добавить:

```
@using (Html.BeginForm("MyAction", "Home", FormMethod.Post))
{
   <input type="text" name="product" /><br/>
   <button name="action" value="add">Добавить</button>
   <button name="action" value="delete">Удалить</button>
}
```

Самое простое решение состоит в том, что для каждой кнопки устанавливается одниковое значение атрибута `name`, но разное для атрибута `value`. А метод, принимающий форму, может выглядеть следующим образом:

```
[HttpPost]
public ActionResult MyAction(string product, string action)
{
    if(action=="add")
    {

    }
    else if(action=="delete")
    {

    }
    // остальной код метода
}
```

И с помощью условной конструкции в зависимости от значения параметра action, который хранит значение атрибута value нажатой кнопки, производятся определенные действия.

Источник.
http://www.w3schools.com/aspnet/mvc_htmlhelpers.asp
http://metanit.com/sharp/mvc5/4.6.php

12 ЛЕКЦИЯ
(ПРОДОЛЖЕНИЕ ЛЕКЦИИ 11)

Доменная модель представляет реальный мир в вашем приложении, она содержит представления ваших объектов, процессов и правил. Доменная модель является сердцем MVC приложения, а все остальное, в том числе представления и контроллеры – это всего лишь средства для взаимодействия с доменной моделью.

ASP.NET MVC не диктует того, какая технология должна использоваться для доменной модели.

Вы вольны выбрать любую технологию, которая будет взаимодействовать с .NET Framework, и тут есть много вариантов. Тем не менее, ASP.NET MVC предлагает инфраструктуру и соглашения, чтобы помочь подключить **классы доменной модели к контроллерам и представлениям**, а также к самому MVC Framework. Есть три ключевые функциональные возможности:

- ***Связывание данных модели*** является функцией, которая автоматически заполняет объекты модели, используя входные данные, как правило, отправленные из HTML формы.
- *Метаданные модели* позволяют описать фреймворку смысл классов модели (см. предыдущие лекции). Например, вы можете предоставить читабельное описание их свойств или дать подсказки о том, как они должны отображаться. MVC Framework может автоматически представить изображение или редактор UI для классов модели в представлениях.
- *Валидация*, которая выполняется во время связывания данных и применяет правила, которые могут быть определены как метаданные.

10.1 Связывание данных модели.

Модель данных.

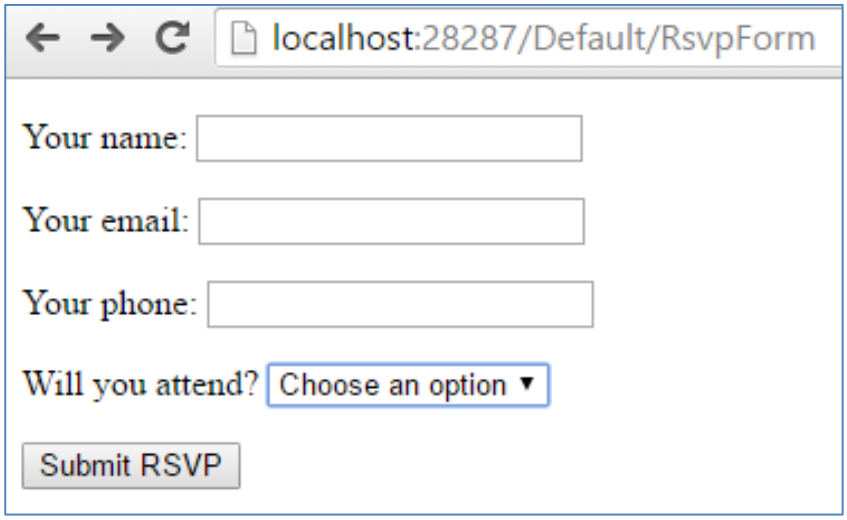
```
namespace PartyInvites.Models
{
public class Class1
{
public string Name { get; set; }
public string Email { get; set; }
public string Phone { get; set; }
public bool? WillAttend { get; set; }
}}
```

Метод действия RsvpForm контроллера будет вызываться в ответ на HTTP POST запрос, GuestResponse является типом класса C #.

```
[HttpPost]
```

```
public ViewResult RsvpForm(Class1 guestResponse)
{
   if (ModelState.IsValid)
   {
      // TODO: Email response to the party organizer
      return View("Thanks", guestResponse);
   }
   else
   {
      // there is a validation error
      return View();
   }
}
```

Запрос на обработку методу действия RsvpForm контроллера поступают от следующего *представления*.




```
@model WebApplication10.Models.Class1
@{    Layout = null;}
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>RsvpForm</title>
</head>
<body>
   <div>
    @using (Html.BeginForm())
    {
     <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
     <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
```

```
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
new SelectListItem() {Text = "Yes, I'll be there", Value =
bool.TrueString},
new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
}, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
        }

    </div>
</body>
</html>
```

Ответом является ***связывание данных модели*** – чрезвычайно полезная функциональная особенность MVC, когда входные данные разбиваются (парсятся) и пары ключ/значение в HTTP запросе используются **для заполнения свойств типа доменной модели**. Этот процесс является противоположностью использования вспомогательных методов HTML; это когда при создании данных формы для отправки клиенту, мы генерировали HTML элементы input, где значения атрибутов id и name были получены из названий свойств классов моделей.

13 ЛЕКЦИЯ. СТРОГО ТИПИЗИРОВАННЫЕ ХЕЛПЕРЫ

Кроме базовых хелперов в ASP.NET MVC имеются их двойники - **строго типизированные хелперы**. Этот вид хелперов принимает в качестве параметра *лямбда-выражение*, в котором указывается то *свойство модели*, к которому должен быть привязан данный хелпер. Важно учитывать, что строго типизированные хелперы могут использоваться только в строго типизированных представлениях, а тип модели, которая передается в хелпер, должен быть тем же самым, что указан для всего представления с помощью директивы @model.

Посмотрим на примере. Модель `Purchase` используется для оформления покупки книги:

```
public class Purchase
{
    public int PurchaseId { get; set; }
    public string Person { get; set; }
    public string Address { get; set; }
    public int BookId { get; set; }
    public DateTime Date { get; set; }
}
```

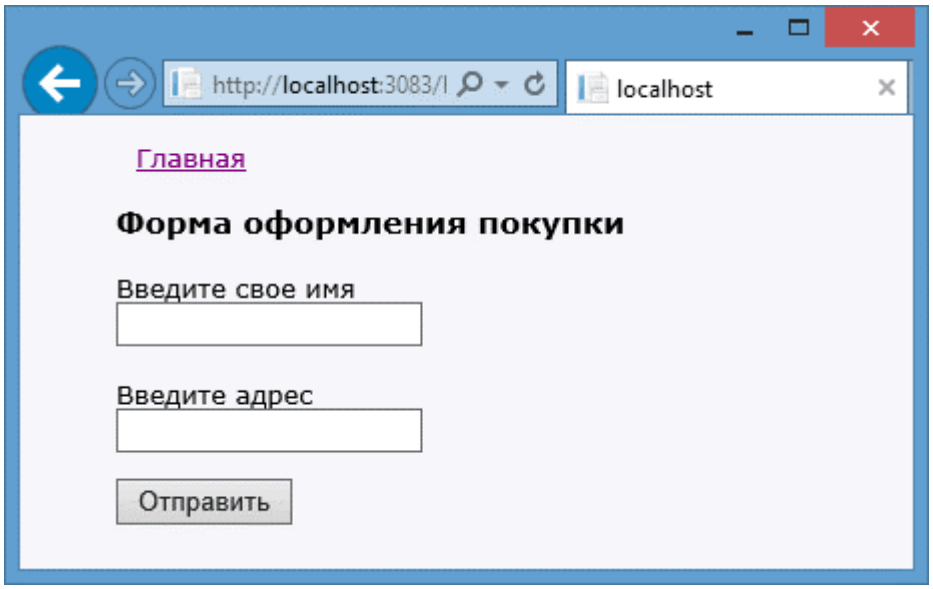
И для ее использования применялась следующая форма:

```
<form method="post" action="">
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
        <table>
            <tr><td><p>Введите свое имя </p></td>
                    <td><input type="text" name="Person" /> </td></tr>
            <tr><td><p>Введите адрес :</p></td><td>
                   <input type="text" name="Address" /> </td></tr>
            <tr><td><input type="submit" value="Отправить" /> </td>
                    <td></td></tr>
    </table>
</form>
```

Перепишем этот пример с использованием хелперов:

```
@model BookStore.Models.Purchase
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

    <div>
        <h3>Форма оформления покупки</h3>
        @using(Html.BeginForm("Buy", "Home", FormMethod.Post))
        {
            @Html.HiddenFor(m=>m.BookId)
            @Html.LabelFor(m => m.Person,"Введите свое имя")
            <br />
```

```
            @Html.TextBoxFor(m=>m.Person)
            <br /><br />
            @Html.LabelFor(m => m.Address, "Введите адрес")
            <br />
            @Html.TextBoxFor(m=>m.Address)
            <p><input type="submit" value="Отправить" /></p>
        }
    </div>
```



Строго типизированный хелпер похож на обычный, только в конце прибавляется суффикс **For**: LabelFor. Так как строго типизированные хелперы могут использоваться только в строго типизированных представлениях, то вначале представления указываем модель, которая будет использоваться: @model BookStore.Models.Purchase. То есть, в вызове @Html.TextBoxFor(m=>m.Person)параметр m представляет переменную модели Purchase. А лямбда-выражение m=>m.Person указывает, что данный хелпер будет генерировать текстовое поле для свойства Person. Таким образом, хелпер @Html.TextBoxFor(m=>m.Person) сгенерирует текстовое поле<input id="Person" name="Person" type="text" value="" />.

Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер:

- **Html.CheckBoxFor**

Выражение @Html.CheckBoxFor(m=>m.Enable, false) создает разметку:

```
<input id="Enable" name="Enable" type="checkbox" value="true" />
<input name="Enable" type="hidden" value="false" />
```

- **Html.HiddenFor**

Выражение @Html.HiddenFor(m=> m.Name) создает разметку:

```
<input id="Name" name="Name" type="hidden"
value="[значение_m.Name]" />
```

- **Html.LabelFor**

Хелпер @Html.LabelFor(m => m.Name,"Имя") генерирует разметку:

```
<label for="Name">Имя</label>
```

- **Html.PasswordFor**

Хелпер @Html.PasswordFor(m => m.Password) оздает разметку:

```
<input id="Password" name="Password" type="password" />
```

- **Html.RadioButtonFor**

@Html.RadioButtonFor(m => m.Option, "val") генерирует разметку:

```
<input id="Option" name="Option" type="radio" value="val" />
```

- **Html.TextBoxFor**

Выражение
@Html.TextBoxFor(m => m.Name) создает разметку:

```
<input id="Name" name="Name" type="text" />
```

- **Html.TextAreaFor**

Хелпер @Html.TextAreaFor(m => m.Name,10,9,null) генерирует код:

```
<textarea cols="9" id="Name" name="Name" rows="10" ></textarea>
```

14 ЛЕКЦИЯ.
СОЗДАНИЕ, ДОБАВЛЕНИЕ, ИЗМЕНЕНИЕ И УДАЛЕНИЕ ОБЪЕКТОВ (платформа Entity Framework)

Объекты в контексте объекта — это экземпляры типов сущностей, представляющие данные, которые находятся в источнике данных. Объекты в контексте объекта можно изменять, создавать и удалять, а Entity Framework отслеживает производимые над ними изменения. При вызове метода SaveChanges Entity Framework создает и выполняет команды, которые производят эквивалентные операции вставки, изменения и удаления в хранилище данных.

Создание и добавление объектов

Чтобы вставить данные на источнике данных, создайте экземпляр типа сущности и добавьте объект к контексту объекта.Чтобы сохранить новый объект в источнике данных, сначала необходимо задать все свойства, не поддерживающие значения **null**. При работе с классами, созданными службами Entity Framework , рассмотрите возможность использования статического метода типа сущности **Create** *ObjectName* для создания нового экземпляра типа сущности. Средства модель EDM (сущностная модель данных) при создании типов сущности включают этот метод в каждый класс. Он служит для создания экземпляров объекта и задания всех свойств класса, которые не могут иметь значение **null**.

Добавление объектов к контексту объекта производится одним из следующих методов.

- Метод AddObject в объекте ObjectSet.
- Метод AddObject в объекте ObjectContext.
- Метод Add в объекте EntityCollection.

При добавлении новых объектов следует принимать во внимание следующие соображения.

- До вызова метода **SaveChanges** Entity Framework создает временное значение ключа для каждого нового объекта.После вызова **SaveChanges** оно заменяется значением идентификатора, назначенного источником данных при вставке новой строки.
- Если источник данных не создал значение ключа для сущности, то следует назначить уникальное значение вручную. Если пользователь назначил двум объектам одно и то же значение ключа, при вызове метода**SaveChanges** будет создано исключение InvalidOperationException. Если это случится, нужно присвоить уникальные значения и попытаться повторить операцию.

Удаление объектов

Вызов метода DeleteObject для объекта **ObjectSet** или метода DeleteObject для объекта **ObjectContext** помечает указанный объект на удаление. Строка не удаляется из источника данных до тех пор, пока не будет вызван метод**SaveChanges**. Entity Framework выполняет удаление объектов по-разному, в зависимости от типа отношения, к которому принадлежит объект.

При идентифицирующем отношении, когда первичный ключ главной сущности является частью первичного ключа зависимой сущности, удаление объекта может привести к удалению связанных с ним объектов. Зависимые объекты не могут существовать без заданного отношения с родительским объектом. Удаление родительского объекта приводит к удалению всех дочерних объектов. Это действие выполняется так же, как и при включении атрибута <OnDelete Action="Cascade" /> сопоставления этого отношения.

Для неидентифицирующего отношения, которое представлено сопоставлением на основе внешнего ключа, при удалении главного объекта Entity Framework для зависимых объектов задает для свойств внешнего ключа, допускающих значения null, значение **null**.

Изменение объектов

Entity Framework отслеживает изменения объектов, присоединенных к контексту **ObjectContext**. Средства модель EDM (сущностная модель данных) создают пару разделяемых методов с именами **On***Property***Changing** и **On***Property***Changed**.Эти методы вызываются в методах задания свойств. Чтобы реализовать пользовательские процедуры бизнес-логики, которые будут выполняться при изменении свойства, расширьте их в разделяемых классах данных.

Пример

Ниже описывается изменение объектов в контексте объекта и сохранение данных в базе данных.

Пример основан на модели Adventure Works Sales. Чтобы запустить код, используемый в данном разделе, нужно сначала добавить к проекту модель Adventure Works Sales и настроить его для использования платформы Entity Framework.

В этом примере запрос объектов возвращает один объект SalesOrderHeader на основе указанного идентификатора SalesOrderID. Состояние этого заказа изменяется с 5 (доставлен) на 1 (обрабатывается), к заказу добавляется новый элемент, а первый существующий элемент удаляется. Для записи изменений в базу данных вызывается метод SaveChanges.

```
// Specify the order to update.
int orderId = 43680;

using (AdventureWorksEntities context =
   new AdventureWorksEntities())
{
   try
   {
      var order = (from o in context.SalesOrderHeaders
                   where o.SalesOrderID == orderId
                   select o).First();

      // Change the status and ship date of an existing order.
      order.Status = 1;
      order.ShipDate = DateTime.Today;

      // You do not have to call the Load method to load the details for the order,
      // because  lazy loading is set to true
      // by the constructor of the AdventureWorksEntities object.
      // With  lazy loading set to true the related objects are loaded when
      // you access the navigation property. In this case SalesOrderDetails.

      // Delete the first item in the order.
      context.DeleteObject(order.SalesOrderDetails.First());

      // Create a new SalesOrderDetail object.
      // You can use the static CreateObjectName method (the Entity Framework
      // adds this method to the generated entity types) instead of the new
operator:
      // SalesOrderDetail.CreateSalesOrderDetail(1, 0, 2, 750, 1,
(decimal)2171.2942, 0, 0,
      //   Guid.NewGuid(), DateTime.Today));
      SalesOrderDetail detail = new SalesOrderDetail
      {
         SalesOrderID = 1,
         SalesOrderDetailID = 0,
         OrderQty = 2,
         ProductID = 750,
         SpecialOfferID = 1,
         UnitPrice = (decimal)2171.2942,
         UnitPriceDiscount = 0,
         LineTotal = 0,
         rowguid = Guid.NewGuid(),
         ModifiedDate = DateTime.Now
```

```
        };

        order.SalesOrderDetails.Add(detail);

        // Save changes in the object context to the database.
        int changes = context.SaveChanges();

        Console.WriteLine(changes.ToString() + " changes saved!");
        Console.WriteLine("Updated item for order: "
            + order.SalesOrderID.ToString());

        foreach (SalesOrderDetail item in order.SalesOrderDetails)
        {
            Console.WriteLine("Item ID: "
                + item.SalesOrderDetailID.ToString() + "  Product: "
                + item.ProductID.ToString() + "  Quantity: "
                + item.OrderQty.ToString());
        }
    }
    catch (UpdateException ex)
    {
        Console.WriteLine(ex.ToString());
    }
}
```

Добавление нового обекта в контекст (Adding a new entity to the context)

Новый объект может быть добавлен к контексту путем вызова метода Add класса DbSet. Объект переходит в состояние Added (добавлено), что означает, что этот объект будет добавлен в таблицу БД при вызове метода SaveChanges. Например:

```
using (var context = new BloggingContext())
{
   var blog = new Blog { Name = "ADO.NET Blog" };
   context.Blogs.Add(blog);
   context.SaveChanges();
}
```

Another way to add a new entity to the context is to change its state to Added. For example:

Другой способ, добавления нового объекта к контексту, чтобы изменить состояние Added (Добавлено). Например:

```
using (var context = new BloggingContext())
{
   var blog = new Blog { Name = "ADO.NET Blog" };
   context.Entry(blog).State = EntityState.Added;
   context.SaveChanges();
}
```

Наконец, вы можете добавить новый объект к контексту, используя связи с другим объектом, который уже отслеживается.. Например:

Finally, you can add a new entity to the context by hooking it up to another entity that is already being tracked. This could be by adding the new entity to the collection navigation property of another entity or by setting a reference navigation property of another entity to point to the new entity. For example:

```
using (var context = new BloggingContext())
{
   // Add a new User by setting a reference from a tracked Blog
   var blog = context.Blogs.Find(1);
   blog.Owner = new User { UserName = "johndoe1987" };

   // Add a new Post by adding to the collection of a tracked Blog
   var blog = context.Blogs.Find(2);
```

```
    blog.Posts.Add(new Post { Name = "How to Add Entities" });

    context.SaveChanges();
}
```

Редактирование модели

В предыдущей теме мы посмотрели, как с помощью шаблонных хелперов отображать значения модели. Теперь же займемся логикой редактирования модели. Вначале добавим в контроллер действие, которые будет получать по Id модель и выводить в представление ее свойства для редактирования:

```
[HttpGet]
public ActionResult EditBook(int? id)
{
    if (id == null)
    {
        return HttpNotFound();
    }
    Book book = db.Books.Find(id);
    if (book != null)
    {
        return View(book);
    }
    return HttpNotFound();
}
```

На случай, если пользователи не укажут id, мы устанавливаем в качестве параметра не int, а int?. И если такой параметр не передан, то возвращаем результат метода `HttpNotFound`.

А представление у нас будет содержать набор хелперов EditorFor для некоторых полей модели:

```
@{
    ViewBag.Title = "Редактировать книгу";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model BookStore.Models.Book
<h2>Книга № @Model.Id</h2>
@using (Html.BeginForm("EditBook", "Home", FormMethod.Post))
{
    <fieldset>
```

```
        @Html.HiddenFor(m => m.Id)
        <p>
            @Html.LabelFor(m => m.Name, "Название книги")
            <br />
            @Html.EditorFor(m => m.Name)
        </p>
        <p>
            @Html.LabelFor(m => m.Author, "Автор")
            <br />
            @Html.EditorFor(m => m.Author)

        </p>
        <p>
            @Html.LabelFor(m => m.Price, "Цена")
            <br />
            @Html.EditorFor(m => m.Price)
        </p>
        <p><input type="submit" value="Отправить" /></p>
    </fieldset>
}
```

Так как уникальный идентификатор id книги нам не надо редактировать, то поле для его отображения сделаем скрытым, то есть воспользуемся хелпером `Html.HiddenFor`.

Теперь нам нужен сам код сохранения. Определим в контроллере действие EditBook, которое будет обрабатывать POST-запросы:

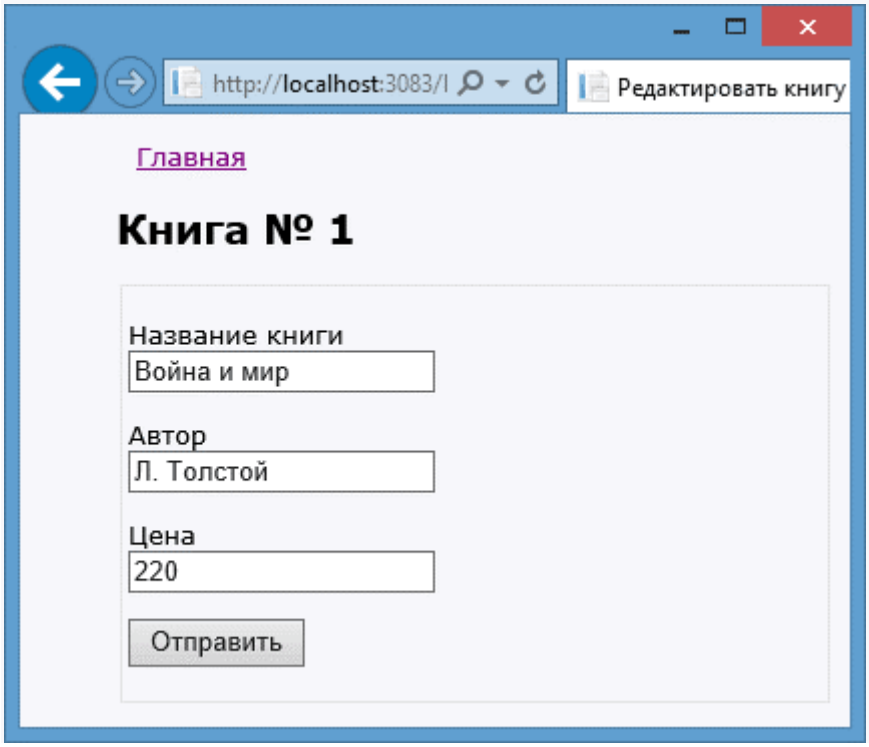
```
[HttpPost]
public ActionResult EditBook(Book book)
{
    db.Entry(book).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

С помощью строки `db.Entry(book).State = EntityState.Modified;` мы указываем, что объект существует `book` уже в базе данных, и для него надо внести в базу измененное значение, а не создавать новую запись. После чего перенаправляемся на главную страницу.

Стоит отметить, что хотя Entity Framework позволяет нам абстрагироваться от запросов sql и структуры бд, но на низком уровне, когда мы устанавливаем значение `db.Entry(book).State = EntityState.Modified;`, то мы тем самым

указываем методу`db.SaveChanges()`, что надо сгенерировать и выполнить комаду UPDATE для обновления модели в БД.

Обратимся к методу EditBook, например, `Home/EditBook/1`:



Хелпер `Html.EditorFor` сгенерировал нам поля для редактирования. Мы можем изменить модель, и отправить ее на сервер, где произойдет ее сохранение.

Добавление модели

В предыдущей теме мы посмотрели, как редактировать модель. Продолжим работу с моделью Book и теперь посмотрим, как мы можем ее добавлять и удалить из БД. Для добавления модели вначале определим пару действий:

```
[HttpGet]
public ActionResult Create()
{
    return View();
}
[HttpPost]
public ActionResult Create(Book book)
{
    db.Books.Add(book);
    db.SaveChanges();

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

Первый метод возвращает пользователю представление с формой для добавления, а второй - принимает данные этой формы. Теперь создадим представление.

А представление будет выглядеть следующим образом:

```
@model BookStore.Models.Book

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Новая книга</h2>

@using (Html.BeginForm())
{
    @Html.LabelFor(model => model.Name, "Название книги")
    <br />
    @Html.EditorFor(model => model.Name)
    <br /><br />
    @Html.LabelFor(model => model.Author, "Автор")
    <br />
    @Html.EditorFor(model => model.Author)
    <br /><br />
    @Html.LabelFor(model => model.Price, "Цена")
    <br />
    @Html.EditorFor(model => model.Price)
    <br /><br />
    <input type="submit" value="Добавить" />
}
```

При получении модели book в действии Create метод `db.Books.Add(book)` будет устанавливать значение `Added` в качестве состояния модели. Поэтому метод `db.SaveChanges()` сгенерирует выражение INSERT для вставки модели в таблицу. То есть метод Create мы могли бы переписать следующим образом:

```
[HttpPost]
public ActionResult Create(Book book)
{
    db.Entry(book).State = EntityState.Added;
    db.SaveChanges();

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

Удаление модели

Теперь самая важная часть - удаление модели. Даже не в плане реализации, сколько в плане безопасности. Добавим простое действие, которое удаляет модель из базы данных:

```
public ActionResult Delete(int id)
{
    Book b = db.Books.Find(id);
    if (b != null)
    {
        db.Books.Remove(b);
        db.SaveChanges();
    }
    return RedirectToAction("Index");
}
```

Вначале мы проверяем, а есть ли такой объект в бд, и если есть, то вызываем метод `db.Books.Remove(b)`. Он установит статус модели в `Deleted`, благодаря чему EntityFramework при вызове метода `db.SaveChanges` сгенерирует sql-выражение DELETE. Но мы можем сами указать статус явным образом:

```
public ActionResult Delete(int id)
{
    Book b = new Book { Id = id };
    db.Entry(b).State = EntityState.Deleted;
    db.SaveChanges();

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

Подобный подход имеет один плюс - мы избегаем первого запроса к бд, который у нас был в выражении `Book b = db.Books.Find(id);`. То есть вместо двух запросов к БД

теперь у нас только один. Но в целом подобный метод на удаление имеет один минус в плане безопасности.

Допустим, нам пришло электронное письмо, в которое была внедрена картинка посредством тега:

```
<img src="http://адрес_нашего_сайта/Home/Delete/1" />
```

В итоге при открытии письма 1-я запись в таблице может быть удалена. Уязвимость касается не только писем, но может проявляться и в других местах, но смысл один - GET-запрос к методу Delete несет потенциальную уязвимость. Поэтому переделаем метод следующим образом:

```
[HttpGet]
public ActionResult Delete(int id)
{
    Book b = db.Books.Find(id);
    if (b == null)
    {
        return HttpNotFound();
    }
    return View(b);
}
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Book b = db.Books.Find(id);
    if (b == null)
    {
        return HttpNotFound();
    }
    db.Books.Remove(b);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Теперь вместо одного метода Delete целых два. Атрибут `ActionName("Delete")` указывает, что метод DeleteConfirmed будет восприниматься как действие Delete. Первый метод передает удаляемую модель в представление. На представлении с помощью нажатия кнопки мы сможем подтвержить удаление. И удаляемый id уйдет второму методу по запросу POST. Таким образом, мы уйдем от уязвимости GET-запроса. Ну и само представление:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model BookStore.Models.Book
<h2>Удаление книги</h2>
<dl>
```

```
    <dt>Название</dt>
    <dd>
        @Html.DisplayFor(model => model.Name)
    </dd>

    <dt>Автор</dt>
    <dd>
        @Html.DisplayFor(model => model.Author)
    </dd>

    <dt>Цена</dt>
    <dd>
        @Html.DisplayFor(model => model.Price)
    </dd>
</dl>

@using (Html.BeginForm())
{
    <input type="submit" value="Удалить" />
}
```

15 ЛЕКЦИЯ. ПЕРЕОПРЕДЕЛЕНИЕ ШАБЛОНОВ ОТОБРАЖЕНИЯ И РЕДАКТИРОВАНИЯ

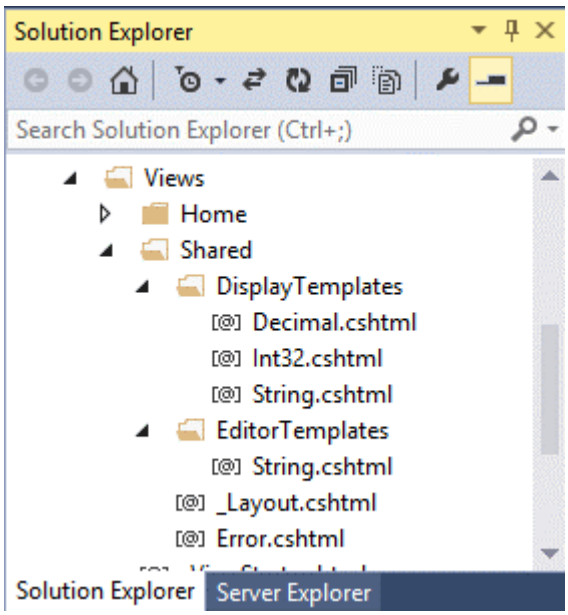
При использовании методов **DisplayFor()/EditorFor()** фреймворк MVC сам определяет, каким образом и какую разметку html создавать для отображения и редактирования полей модели. Но мы можем переопределить подобное поведение, указав, как именно MVC должен работать с отдельными типами данных.

Допустим, у нас есть следующая модель:

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Модель содержит свойства трех типов данных: int, string, decimal. Переопределим создание разметки для этих свойств.

Для этого добавим в проект в каталог *Views/Shared* две новых папки: **DisplayTemplates** (для шаблонов отображения) и**EditorTemplates** (для шаблонов редактирования):



Так как класс использует свойства трех разных типов данных, в шаблонах отображения в папке DisplayTemplates я создал по файлу для каждого типа. Названия файлов носят названия классов, которые представляют данный тип. Например, файл *Int32.cshtml*, который используется для визуализации значений int, имеет следующую разметку:

```
@model Int32
```

```
<b>@Model</b>
```

Директива `@model Int32` указывает, что в качестве модели будет использоваться тип int или Int32. А само содержимое модели делается жирным шрифтом с помощью тега `<b>`

Файл *String.cshtml* имеет следующее содержимое:

```
@model String

<b>"@Model"</b>
```

Тут тоже самое, только название указывается в кавычках. В общем-то мы можем определить разные теги или структуру тегов, можно, например, задать классы (`<span class="string">"@Model"</span>`) и т.д.

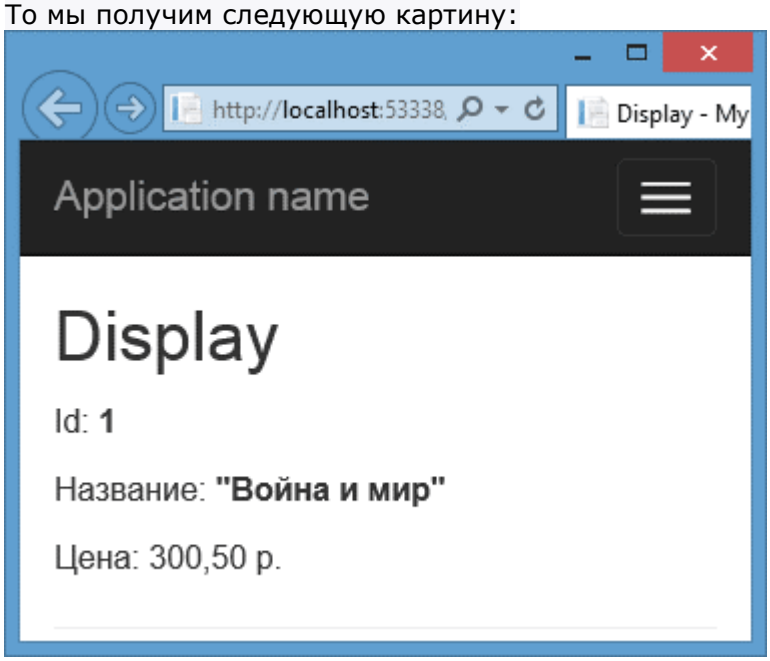
Файл *Decimal.cshtml*:

```
@model decimal

@{
    IFormatProvider formatProvider =
        new System.Globalization.CultureInfo("ru-RU");
    <span class="currency">@Model.ToString("C", formatProvider)</span>
}
```

Здесь уже определяется логика форматирования значения в качестве денежной единицы.
Теперь если мы используем хелпер DisplayFor() для вывода свойств модели:

```
@model CustomDisplayEditorTemplates.Models.Book

@{
    ViewBag.Title = "Display";
}

<h2>Display</h2>

<div>
    <div>
        <p>
            Id: @Html.DisplayFor(model => model.Id)
        </p>
        <p>
            Название: @Html.DisplayFor(model => model.Name)
        </p>
        <p>
            Цена: @Html.DisplayFor(model => model.Price)
        </p>

    </div>
</div>
```

То мы получим следующую картину:



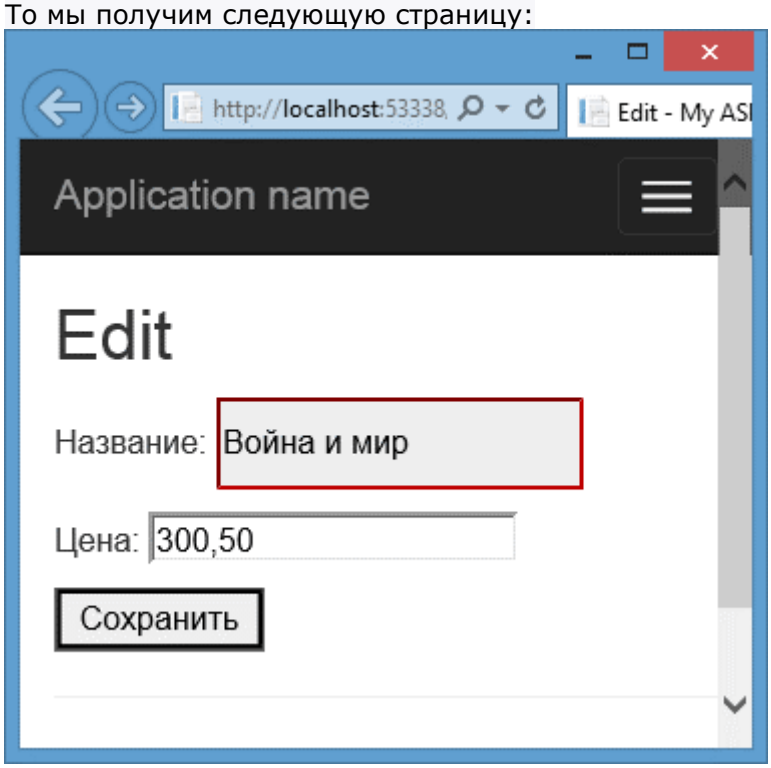
В папку *EditorTemplates* я добавил только один шаблон форматирования - *String.cshtml*:

```
@model string

<input type="text" name="name" style="border-color:red; height:40px;
background-color:#eee;" value="@Model" />
```

И допустим, у нас есть следующее представление для редактирования данных:

```
@model CustomDisplayEditorTemplates.Models.Book

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.HiddenFor(model => model.Id)
    <p>Название: @Html.EditorFor(model => model.Name)</p>
    <p>Цена: @Html.EditorFor(model => model.Price)</p>
    <p><input type="submit" value="Сохранить" /></p>
}
```

То мы получим следующую страницу:



При создании подобных шаблонов на основе элементов ввода следует учитывать их ограниченность: в данном случае элемент имеет атрибут `name="name"`, поэтому его значение будет сопоставляться со свойством Name. Если у нас она модель в которой есть только одно строковое свойство, которое называется Name, то подобный подход еще сработает. Но у нас может быть несколько свойств с типом string, которые могут называться по-разному. В этом случае мы можем определить шаблон вывода или редактирования модели целиком.

Например, добавим в папку DisplayTemplates следующий файл *Book.cshtml*:

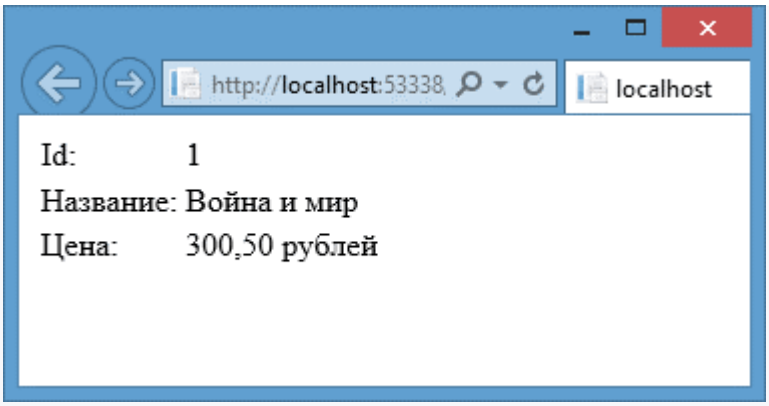
```
@model CustomDisplayEditorTemplates.Models.Book

<table>
    <tr><td>Id:</td><td>@Model.Id</td></tr>
    <tr><td>Название:</td><td>@Model.Name</td></tr>
    <tr><td>Цена:</td><td>@Model.Price рублей</td></tr>
</table>
```

CustomDisplayEditorTemplates - в данном случае это название моего проекта, в котором определен класс Book.

Значения всех свойств модели выводятся через таблицу. Тогда представление для отображения модели будет выглядеть так:

```
@model CustomDisplayEditorTemplates.Models.Book
@{
    Layout = null;
```

```
}
@Html.DisplayForModel()
```

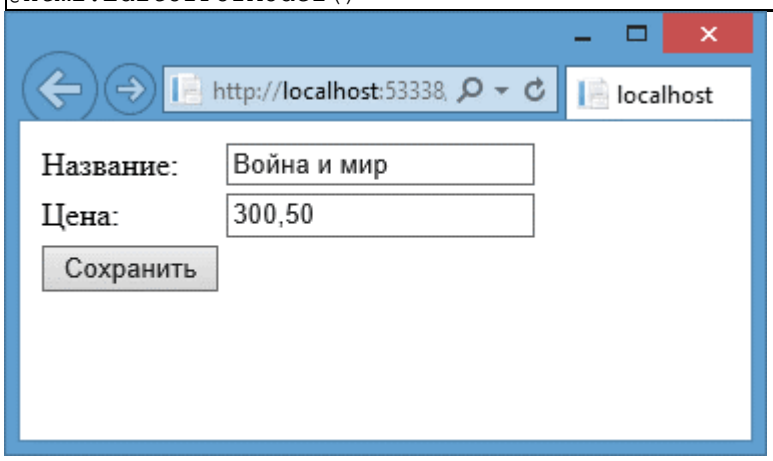


Также добавим в каталог EditorTemplates шаблон для редактирования *Book.cshtml*:

```
@model CustomDisplayEditorTemplates.Models.Book

@using (Html.BeginForm())
{
    @Html.HiddenFor(model => model.Id)
    <table>
        <tr><td>Название: </td><td><input type="text" name="name" value="@Model.Name" /
        <tr><td>Цена: </td><td><input type="text" name="price" value="@Model.Price" /><
        <tr><td><input type="submit" value="Сохранить" /></td><td></td></tr>
    </table>
}
```

Используем этот шаблон для создания формы редактирования:

```
@model CustomDisplayEditorTemplates.Models.Book

@{
    Layout = null;
}
@Html.EditorForModel()
```



16 ЛЕКЦИЯ. ШАБЛОННЫЕ ХЕЛПЕРЫ

Кроме базовых html-хелперов, рассмотренных в прошлой главе и генерирующих определенные элементы разметки html, фреймворк ASP.NET MVC также имеет **шаблонные (или шаблонизированные) хелперы**. В отличие от рассмотренных в прошлой главе html-хелперов они не генерируют определенный элемент html. Шаблонные хелперы смотрят на свойство модели и генерируют тот элемент html, который наиболее подходит данному свойству, исходя из его типа и метаданных.
В ASP.NET MVC имеются следующие шаблонные хелперы:

- **Display**
  Создает элемент разметки для отображения значения указанного свойства модели: `Html.Display("Name")`
- **DisplayFor**
  Строго типизированный аналог хелпера Display: `Html.DisplayFor(m => m.Name)`
- **Editor**
  Создает элемент разметки для редактирования указанного свойства модели: `Html.Editor("Name")`
- **EditorFor**
  Строго типизированный аналог хелпера Editor: `Html.EditorFor(m => m.Name)`
- **DisplayText**
  Создает выражение для указанного свойства модели в виде простой строки: `Html.DisplayText("Name")`
- **DisplayTextFor**
  Строго типизированный аналог хелпера DisplayText: `Html.DisplayTextFor(m => m.Name)`

Это были одиночные хелперы, которые генерируют разметку только для одного свойства модели. Но кроме них во фреймворке также есть еще несколько шаблонов, которые позволяют создать разом все поля для всех свойств модели:

- **DisplayForModel**
  Создает поля для чтения для всех свойств модели: `Html.DisplayForModel()`
- **EditorForModel**
  Создает поля для редактирования для всех свойств модели: `Html.EditorForModel()`

Например, определим в контроллере некоторое действие BookView, которое по id будет выводить информацию об определенной книге:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@model BookStore.Models.Book

<h2>Книга № @Model.Id</h2>
@Html.DisplayForModel()
```

И обратимся к этому ресурсу, набрав в адресной строке браузера *Home/BookView/1*: