Итак, Вы закончили с SQL Server (на самом деле только начали, т.к. все самое интересное будет в Вашей профессиональной практике).

Переходим к C#. Опять же мы здесь не будем расписывать все справочные данные по C#. Только то, что нам нужно, чтобы начать разрабатывать веб-приложения.

В этой главе мы будем создавать классы для доступа к данным и управления ними.

Очень сильно жизнь разработчика упрощает Entity Framework (или другая ORM, мы используем именно EF).

Задание 1. Найдите что такое ORM, какие бывают ORM.

Если бы не было EF - то нам пришлось бы для каждой таблицы в базе создавать свой класс со всеми полями и прописывать код ADO.NET для управления этими таблицами.

ADO.NET - это такая встроенная библиотека, которая позволяет выполнять SQL запросы и получать данные из базы.Так мы раньше и делали. Но после начала использования ORM все гораздо упростилось.

По сути Вам надо выполнить следующие шаги:
- создать модель базы EDMX через специальный мастер (все необходимые классы для выбранных таблиц базы)
- создать класс-репозиторий, который будет выполнять основные операции по работе с данными: Create, Read, Update, Delete (CRUD)
- настроить кеширование для некоторых сущностей для повышения производительности приложения

Видео по EF - http://youtu.be/8dQ2DuktQAQ
В этом же видео рассматривается, как сделать Repository класс - об этом позже.

Задание 2. Сделайте для своего тестового проекта подобный EDMX.

Примечание. Я не упомянул, как создавать проект в целом для ASP.NET. Пару опорных пунктов:
- качаете VS Express 2013 for Web или Visual Studio 2015 Community
- создаете проект типа Web Project ASP.NET MVC
- запускаете проект (F5). Остановить - Shift + F5

http://msdn.microsoft.com/ru-ru/library/ee410104(v=vs.100).aspx

Пару слов о классах

В C# все строится в виде классов - набор полей и методов работы над ними. Класс - это по сути определенная сущность, обладающая набором некоторых данных и имеющая некоторое поведение.
Вы можете создавать конкретные экземпляры классов (да, это объекты).
По сути класс - это трафарет. А экземпляр - это конкретная реализация этого класса.
Например, класс Human, а экземпляр Human {id=12, FIO="Иванов Иван Петрович"}

Статический метод класса - это такой метод класса, которые может быть вызван без создания класса.

Примечание. Аккуратнее со статическими методами. Раньше в нашей CMS класс провайдер был сделан как статический. В каждом методе вызывался свой объект доступа к БД (Object Context). Получалось, что при каждом запросе создавалась куча новых объектов контекстов данных (хотя в идеале можно было обойтись одним) + они некорректно освобождали память. В итоге мы переделали класс провайдера со статического класса на обычный + использовали по возможности только один объект контекста данных для всех запросов к базе данных в рамках одного HTTP запроса к серверу от клиента.

У класса есть поля, методы и свойства (по сути те же методы). У каждого из них может быть уровень доступа:

- private - доступен только внутри класса
- protected - доступен внутри класса и его потомках (т.е. наследующих его)
- public - доступен извне.

Члены класса могут быть статическими и не статическими. Если метод статический, то его можно вызывать без создания экземпляра класса. Обычно это какая-то общая функциональность, свойственная в целом для класса, а не для конкретного экземпляра. По возможности не задействуйте внешние ресурсы (например базу) в статических членах класса.

Интерфейс - это такой класс, у которого не прописывается реализация конкретных методов. Он нужен для того, чтобы декларативно определить что должна делать та или иная сущность. Т.е. часто вы работаете именно с интерфейсом и его методами. Вас не интересует какой именно класс будет отдуваться за реализацию этого интерфейса - вы просто используете его методы и не задумываетесь о реализации. По сути, интерфейс - это контракт, который должен выполнить класс, наследующий этот интерфейс. Лучшая пословица характеризующая это - “назвался груздем - полезай в кузов”).

Задание 3. Найдите книгу Эндрю Троелсена и прочитайте главу про классы. Вы должны понимать, что такое конструктор, наследование, полиморфизм, скрытый член класса, интерфейс.

Создание класса репозитория и менеджера

В общем случае лучше разделять на 2 уровня. Класс доступа к данным (DAL) и класс бизнес логики(BLL). Если проект относительно простой - то можно использовать только менеджер.

Что содержит типичный Репозиторий:

- Ссылка на объект контекста доступа к БД (обычно мы называем эту переменную db)
- Gets, Save, Delete для каждой сущности - т.е. простейшие операции работы с сущностью. Причем в них нет никакой дополнительной логики - только сами операции работы с сущностью. Метод GetItems возвращает Iqueryble<entity> - т.е. не сам фактический список, а просто начальный запрос на доступ к данным

Что содержит типичный класс Менеджера:

- кеширование данных
- атомарные операции, например метод ЯПроверилЗадачу (а не ИзменитьСтатусЗадачи(statusID)). Это именно бизнес-логика. Здесь нужно писать методы именно в терминах работы приложения, а не терминах базы данных
- логирование действий
- работа уведомлений о различных ситуациях
- проверка возможности выполнения операции данным пользователем
- обработка исключений и факта получения некорректных данных

Желательно придерживаться именно такого подхода, т.е.:

- не нужно размазывать бизнес-логику по всему приложения (писать ее в контроллерах, в каких то частных случаях обрабатывать исключения и т.д.).
- каждую логику прописывать в Менеджере один раз (например Метод проверки прав у пользователя на Проект - т.е. везде должен использоваться именно он, а не какие-то частные свои реализации).
- писать методы в терминах бизнес-логики. Т.е. здесь методов может быть гораздо больше, чем на уровне DAL. Например, товар можно сохранять в DAL при помощи SaveProduct. А в менеджере вызывать методы: ChangeProductPrice (залогировать кто изменил и какое было старое значение), ChangeProductOrder (изменение порядка вывода + возможно указать комментарий по изменению сущности - т.е. зачем он это сделал).

Небольшой момент по контексту базы в Репозитории. На мой взгляд, его лучше сделать public(т.е. есть возможность обратиться к нему извне). Это будет иметь смысл в некоторых ситуациях:

- для второстепенной сущности нет методов репозитория, но требуется получить некоторые данные по этой сущности
- в целях производительности в некоторых местах будет использоваться прямой SQL прямо в Менеджере. Использоваться будет как раз этот объект контекста.

Классы для менеджера и репозитория лучше используйте нестатические, дабы не иметь проблем с освобождением памяти.

В тех случаях, когда у вас класс менеджера и репозитория совпадают - вы просто делаете методы DAL приватными (private), а методы BLL - public (сам класс в итоге называется менеджером - например CatalogManager).

Не нужно создавать на каждую сущность в БД отдельный менеджер. Создавайте менеджер для всей подсистемы, например Каталога. Так будет проще перенести свое решение в другой проект.

По переносу своего решения в другой проект - делайте свой метод как можно более независимым от внешнего мира за счет использования интерфейсов. Т.е. вы используете не конкретную реализацию класса, который вы используете в менеджере, а его интерфейс. Допустим, в одном из методов менеджера вам надо отправить сообщение (не важно куда - на email, скайп и т.д.). Для этого вы используете не конкретный класс MailSender, а интерфейс ISender (он просто содержит метод Send без реализации). Таким образом при переносе в другой проект вы не будете зависеть от конкретного класса, а только от интерфейса, и в новом проекте он может быть реализован по-своему. Когда вы создаете объект менеджера - вы просто указываете объект этого интерфейса (на пример, объект класса MailSender)

```
// класс менеджера
public class ItemManager{
        private ISender sender;
        public ItemManager(ISender sender){
                this.sender = sender
        }
        public DoSomething(){
                ….
                sender.Send("text")
        }
```

```
}

// использование в проекте
void MyMethod(){
        var mng = new ItemManager(new MailSender());
        mng.DoSomething();
}
```

Теперь давайте посмотрим типовой класс репозитория.

```
namespace project1.DAL
{
        public class ItemRepository
        {
        #region System
        public ProjectDBContext db;
        private bool _disposed;

        public LogsMaItemRepositorynager()
        {
                db = new ProjectDBContext ();
                _disposed = false;
                //SERIALIZE WILL FAIL WITH PROXIED ENTITIES
                //dbContext.Configuration.ProxyCreationEnabled = false;
                //ENABLING COULD CAUSE ENDLESS LOOPS AND PERFORMANCE PROBLEMS
                //dbContext.Configuration.LazyLoadingEnabled = false;
        }

        public void Dispose()
        {
                Dispose(true);
                GC.SuppressFinalize(this);
        }
        protected virtual void Dispose(bool disposing)
        {
                if (!_disposed)
                {
                if (disposing)
                {
                        if (db != null)
                        db.Dispose();
                }
                db = null;
                _disposed = true;
                }
        }
        #endregion

        public IQuesryble<item> GetItems()
```

```
{
        var res = db.items;
                return res;
}

public int SaveItem(item element)
{
                if (element.id == 0)
        {
                db.items.Add(element);
                db.SaveChanges();
        }
        else
        {
                try
                {
                db.Entry(element).State = EntityState.Modified;
                db.SaveChanges();
                }
                catch (OptimisticConcurrencyException ex)
                {
                RDL.Debug.LogError(ex);
                }
        }

                return element.id;
}
public bool DeleteItem(int id)
{
        bool res = false;
                var item = db.items.SingleOrDefault(x => x.id == id);
        if (item != null)
        {
                db.Entry(item).State = System.Data.Entity.EntityState.Deleted;
                db.SaveChanges();
        }
        res = true;
                return res;
}
```

Примечание

1. Не нужно выделяться особой фантазией и придумывать каждый раз новые имена для менеджера и объекта контекста. Мы, к примеру, используем всегда 2 имени - db для контекста БД и для mng для экземпляра класса менеджера. Так гораздо и проще понимать друг друга. Если вы хотите где-то использовать метод менеджера - то вы просто набираете mng (он объявлен где-то выше и вы в курсе этого) и просто используете. А не начинаете судорожно искать “как же я назвал этот объект менеджера в этот раз”. Еще раз повторю, соблюдение, понимание и знание правил наименований - значительно ускоряет работу над проектом.

2. Возьмите за привычку использовать регионы в коде. На одну сущность один регион. Основной смысл - это структуризация кода. Вы можете сворачивать отдельные регионы. При этом разумно используйте регионы. Не нужно делать очевидные объединения, которые только усложняют поиск по коду (например, регионы Поля, Методы, Неактуальные методы).

Кеширование

Есть память сервера и память базы. У сервера память небольшая, но быстрая. У базы памяти много, но она медленнее.
Часто используемые данные имеет смысл держать в памяти сервера, чтобы доступ к ним осуществлялся быстрее.
Те, кто только начал использовать кеширование, допускают ошибку - помещают в базу весь каталог сразу. Тем самым Ваше приложение начинает потреблять очень много памяти.

Другая дурацкая ошибка - это попытка кешировать Iqueryble объект - помните что такой объект по сути не содержит данных, он содержит только sql запрос.

Кешируйте только часто используемые данные. Память на сервере не безгранична. Если вы слишком много храните в кеше, это может сказаться на общей производительности сервера и в частности на вашем приложении.

Видео про кеширование - http://youtu.be/IOEOrIBiInA

Обычно стратегия работы с кешем такова: смотрим, есть ли в кеше то, что нам нужно (по ключу). Если есть - то берем. Если нет - то берем из базы и записываем эти данные в кеш, чтобы в следующий раз взять их уже из кеша.
На первых порах не используйте кеш. Это Вам сейчас скорее потребуется в тех случаях, когда будете разбираться с чужим кодом.

Проверка прав на выполнение операции

Запомните, никогда не доверяйте тому, что пришло от браузера. Неважно это простой запрос или ajax запрос. Эти данные можно подделать. Для конфиденциальных данных или критически важных операций обязательно надо проверять - может ли данный пользователь сделать это действие.

Для этого в менеджера создавайте отдельные методы проверки доступа данного пользователя к данному объекта.

Обычно пользователь определяется своим именем User.Identity.Username. У пользователя могут быть роли или права на операции. Вы можете проверять есть ли у пользователя соответствующая роль (Roles.UserIsInRole) или право (кастомный метод, т.е. нет в стандартном функционале платформы).

Большой плюс такого подхода (создание отдельных функций проверки) в том, что вы запросто можете сменить потом способ проверки (например, добавилась еще одна роль в системе и ее нужно обрабатывать также в проверке - если вы не сделали такого метода - вам придется бегать по всему коду и менять проверку).

Еще один прием - это работа с Guid, полученным от пользователя. Если вы какому-то неавторизованному пользователю дали ссылку с уникальным Guid - то с высокой долей вероятности при получении Guid в запросе вы можете принять, что это именно тот пользователь (конечно, этот метод нельзя использовать для строгой авторизации на доступ к каким либо операциям, но его стоит иметь в виду при работе с неавторизованными пользователями). Пример такого использования - корзина в интернет-магазине для нового незарегистрированного посетителя.

Обработка исключений

Обязательно используйте стандартные средства обработки исключений. Возьмите за привычку использовать примерно такой паттерн:

```
private string GetString(string param1,int param2){
	var res = "";
	try{
		...
		res = ...
	}
	catch(Excecption ex){
		RDL.Log.Error(ex, new{param1, param2});
		res = "";
	}
	return res;
}
```

Важные моменты:

- обозначьте сразу что будет результатом. Мы всегда, к примеру, используем для результата название res.
- поставьте всю функцию в try catch
- используйте стандартный метод обработки исключения (у нас это RDL.Debug.LogError)

Что делает RDL.Debug.LogError - фиксирует ошибку (запись в БД, отправка на почту уведомления). Важный момент - поставьте try catch внутри LogError, но не используйте LogError в блоке catch иначе может возникнуть бесконечный цикл.

LINQ

LINQ - это технология доступа к данным. В нашем случае мы рассматриваем LINQ To SQL. Т.е. конструкции на LINQ в итоге преобразуются в SQL запросы.

Есть 2 синтаксиса LINQ - SQL подобный и с лямбда-выражениями (это по сути функции в одну строку со специальным синтаксисом).

Предлагаю вам изучать только один синтаксис - с лямбда выражениями, он более функциональный и более легко читаемый.

Как извлечь все записи с ID больше 10:

```
var res = items.Where(x=>x.id>10).ToList()
```

Сам запрос по факту выполнится только при выполнении ToList() - помните об этом. Старайтесь как можно больше операций фильтрации сделать до ToList. Однако не все операции могут быть преобразованы в SQL автоматически, например indexOf выдаст ошибку, что невозможно преобразовать. В этом случае сначала нужно получить фактические данные, а потом уже применить indexOf'

```
var res = items.Where(x=>x.id>10).ToList().Where(x=>x.indexOf(name)>=0);
```

Помните, что если у вас коллекция IQueryble<>, то будет включаться механизм преобразования в SQL запрос. Если же у вас List<>, то обработка будет уже вестись в C#.

Основные операции LINQ

- SELECT - определить формат выходных данных
- WHERE - фильтрации данных
- FirstOrDefault().First() - получить первый элемент коллекции (их отличие в том, что First вызывает исключение если нет элементов, а FirstOrDefault возвращает значение по умолчанию (т.е. null для объектов)).
- Skip, Take - пропускает элементы и берет элементы. Используется для создание пейджинга.

- Any, All - возвращают True, если хотя бы один элемент удовлетворяет условию (а второй - если все элементы удовлетворяют условию).
- Max, Min, Count, Sum, Average и т.д. - агрегирующие функции

Пару моментов по LINQ

- Не используйте неявных циклов внутри LINQ запросов, например
  `items.Where(x=> DoHardSomething(x))`
- Не используйте отложенную загрузку
  `items.Select(x=>x.cats.name)` - вызовет кучу дублирующих запросов если items - это уже List<>. Решение для этого случая - использование Include("связанная таблица-справочник, а не вложенные сущности") либо извлечь весь справочник заранее и потом в LINQ извлекать их через отдельную функцию.
- Взять проекцию объекта можно следующим образом (это часто требуется в запросах Ajax когда не нужны все данные, а только некоторые поля):
  items.Select(x=> new {x.id, x.name})
- Когда у вас идет проверка по нескольким условиям, правильно структурируйте через && или || (например, так обычно делается фильтр по данным некоторой таблицы)

```
items.Where(x=>
        (....) &&
        (....) &&
        (....) &&
        (....)
)
```

В некоторых случаях в целях производительности лучше использовать ADO.NET, а не LINQ. Далеко не всегда LINQ генерирует оптимальный запрос. Также если извлекается очень много данных - то LINQ делает маппинг данных в объекты, что тоже бывает затратно по ресурсам. Эти 2 момента можно решить с использованием библиотеки Dapper. Т.е. он позволяет выполнить SQL запрос и загрузить результат в коллекцию нужных вам объектов очень быстро. Если же вы хотите максимальную скорость маппинга в объекты - используйте DataReader. Это самый быстрый способ, т.к. данные читаются только в одном направлении.

Задание 4. Найдите, как посмотреть SQL, который генерирует LINQ.

Задание 5. Создайте для своего проекта Класс Репозиторий и Класс Менеджера по всем правилам, которые мы указали здесь.

Задание 6. Найдите и опишите самые популярные классы .NET и их часто используемые методы. Небольшая подсказка: String, Object, List, ArrayList, Membership, FilePath,File, Directory, Stream, DataReader, Roles, HttpContext, Request, Response

Задание 7 (для продвинутых).
Создайте класс корзины со всеми соответствующими методами. Подсказка: будет еще один класс - элемент корзины (товар в корзине).

Задание 8 (для самых продвинутых).
Найдите библиотеку Dapper и попробуйте ее использовать для извлечения записей из большой таблицы (в предыдущих главах мы ее делали). Сделайте LINQ аналог и сравните производительность по данным трассировки и MiniProfiler.

Итак, сейчас Вы уже имеете небольшое приложение с кодом C#, которое получает данные из базы. Правда у него пока нет интерфейса. Именно этим мы и займемся в следующем уроке. Мы расскажем как вывести данные, которые есть у нас в базе и рассмотрим механизм работы страниц в ASP.NET.