Расширенная выборка данных в .NET приложениях

Method Chaining

- Метод возвращает экземпляр класса
- У метода должен быть тип void

Связи

Связь «ОДИН-К-ОДНОМУ»

Добавим поле с типом модели в User.cs

```
public UserProfile UserProfile { get; set; }
```

Создавая первичный ключ просто укажем в декораторе связанную модель:

```
// UserProfile.cs
[Key]
[ForeignKey("User")]
public int Id { get; set; }
```

И добавим ссылку на модель User:

```
public User User { get; set; }
```

Связь «МНОГИЕ-К-ОДНОМУ»

В Product.cs укажем, что хотим связать его с категорией:

```
public int? CategoryId { get; set; }
public Category Category { get; set; }
```

И добавим саму коллекцию в Category.cs:

```
public Icollection<Product> Products { get; set; }
```

Создавая конструктор, создадим экземпляр `IList<Product>`

Связь «МНОГИЕ-КО-МНОГИМ»

Реализуется наличием коллекций объектов в каждой связанной модели.

```
public ICollection<Player> Players { get; set; }

public ICollection<Team> Teams { get; set; }
```

При **Code-First** подходе будет создана таблица **TeamPlayers**

System.Data.Entity

Where()

Аналог ключевого слова WHERE. Производит выборку данных.

```
_context.Where(
        x => x.UserName == “John”
        && x.Profile.Age == 19
).ToList();
```

Include()

Соединяет выбираемую модель используя внешний ключ

```
_context.Include(
    p => p.Players
).ToList();
```

OrderBy()

Сортирует данные модели в порядке возрастания по определенному выражению или предикату.

```
_context.OrderBy(
    p => p.FirstName
).ToList();
```

OrderByDescending()

Сортирует данные модели в порядке убывания по определенному выражению или предикату.

```
_context.OrderByDescending(
    p => p.FirstName
).ToList();
```

Take()

Аналог ключевого слова LIMIT в SQL.

```
_context.OrderBy(
    p => p.FirstName
).Take(3).ToList();
```

Как быть с составными запросами?

Составные запросы можно создавать используя тип

`IQueryable<T>`

IQueryable<T>

Предоставляет функциональные возможности для оценки запросов по определенным источникам данных в случае, если тип данных известен.

```
public interface IQueryable<out T> :
                IEnumerable<T>,
                IEnumerable,
                IQueryable
```

IQueryable<T> пример

```
// объявим переменную с типом IQueryable<T>
// присвоим коллекцию DbSet<Patients> как значение
IQueryable<Patient> patients = db.Patients;

if(...)
   patients = patients.Where(x => …);
if(...)
   patients = patients.OrderBy(s => s.Name);
```

Проецируемая выборка

Или как мы память экономим

Что такое проекция

При использовании проекции мы сами решаем, что нужно выбрать из БД

Для этого используем следующий подход:

- Используем метод `Select()`
- Создаем анонимный объект как аргумент метода
- Проецируем все нужные поля на него

Пример с анонимным классом

```
var patients = _ctx.Select(p => new {
  Name = p.FullName,
  Company = p.Insurance.Title,
  Gender = p.Gender,
  Workplace = p.Workplace.Title
}).ToList();
```

Таким образом мы спроецировали новый тип

Пример с обычным классом

Предполагается, что у вас имеется готовый класс, который может принять эти данные.

Данный подход ничем не отличается от анонимного:

```
var patients = _ctx.Select(p => new PatientInfoCard { … }
```

Для чего это все?

- Четкая типизация данных для определенных ситуаций
- Экономия памяти за счет выборки только определенных данных