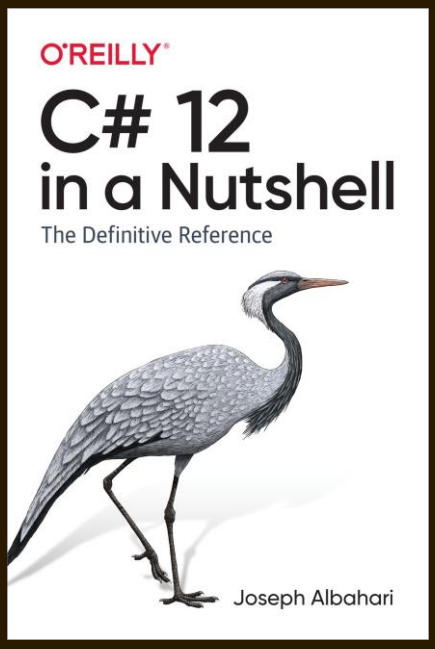
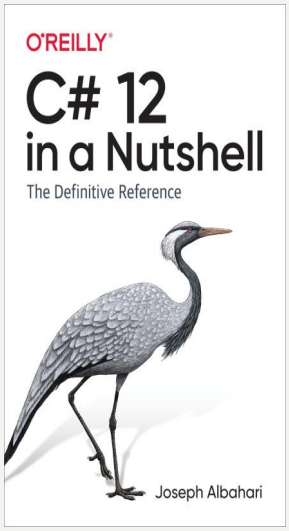
Chapter 8
EF CORE



- EF Core Entity Classes
- DbContext
- Object Tracking
- Change Tracking
- Navigation Properties

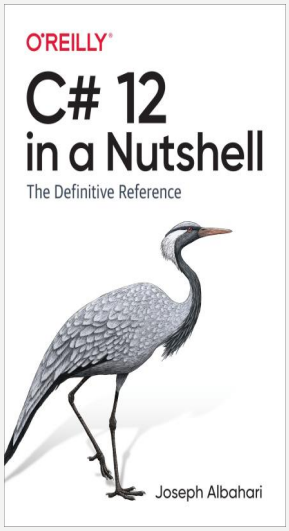


EF Core
Entity Classes

EF Core به ما اجازه میده تا از کلاس‌ها برای نگهداری دیتا استفاده کنیم به شکلی که هر پراپرتی public کلاس یک ستون جدول محسوب میشه که میتونیم روش کوئری بزنیم.

برای مثال ما میتونیم به ازای جدول Customer در دیتابیس یک entity class به شکل زیر تعریف کنیم:

```
0 references | 0 changes | 0 authors, 0 changes
public class Customer
{
    0 references | 0 changes | 0 authors, 0 changes
    public int Id { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string? Name { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string? Address { get; set; }
}
```



DbContext

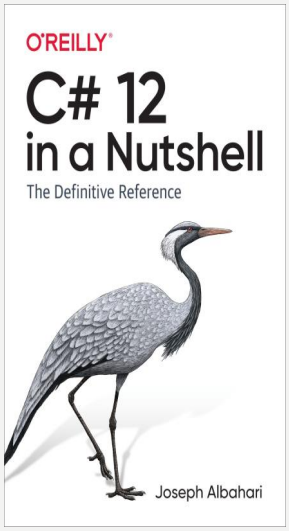
بعد از تعریف entity class ها، حالا وقتشه که یه کلاس بسازیم که از DbContext ارث ببره. این کلاس بهمون کمک میکنه تا بتونیم با session های دیتابیس کار کنیم.

معمولا این کلاس شامل یک پراپرتی DbSet<T> به ازای هر entity class هست:

```
0 references | 0 changes | 0 authors, 0 changes
public class NutshellDbContext : DbContext
{
    0 references | 0 changes | 0 authors, 0 changes
    public DbSet<Customer> Customers { get; set; }
}
```

یک کلاس DbContext سه تا کار انجام میده:

- به عنوان یه factory برای تولید آبجکتهای DbSet<> تا بتونیم روش کوئری بزنیم.
- هرگونه تغییر روی entity ها رو نگه میداره. (change tracking)
- متدهای virtual ای فراهم میکنه که باهاش بتونید تنظیمات مربوط به اتصال به دیتابیس رو مدیریت کنید.



DbContext

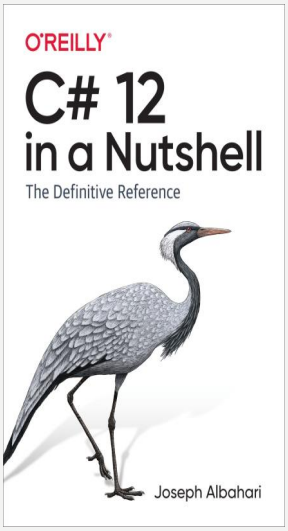
با override کردن متد OnConfiguring شما میتونید نوع دیتابیسی که قراره ازش استفاده کنید و همینطور connection strting خودتونو مشخص کنید.

```
0 references | 0 changes | 0 authors, 0 changes
public class NutshellDbContext : DbContext
{
    0 references | 0 changes | 0 authors, 0 changes
    public DbSet<Customer> Customers { get; set; }

    0 references | 0 changes | 0 authors, 0 changes
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(local);Database=Nutshell;Trusted_Connection=True");
    }
}
```

توی این مثال کانکشن استرینگ مستقیم به صورت متنی تعریف شده. توی یک برنامه عملیاتی معمولا میتونید اونو از یک فای کانفیگ مثل appSettings.json بخونید.

UseSqlServer یک اکستنشن متد هستش که در اسمبلی Microsoft.EntityFramework.SqlServer تعریف شده. پکیج‌های دیگه‌ای هم برای کار با سایر دیتابیس‌ها مثل Oracle, MySQL, PostgreSQL و SQLite وجود داره.



DbContext

اگه از ASP.Net استفاده میکنید میتونید از طریق Dependency Injection با ساخت یه constructor برای کلاس DbContext تنظیمات پیش‌فرض رو بهش بدید.

تو این حالت اگه بخواهید متد OnConfiguring رو هم override کنید میتونید از پراپرتی configured برای چک کردن اینکه قبلا کانفیگ اتفاق افتاده یا نه استفاده کنید:

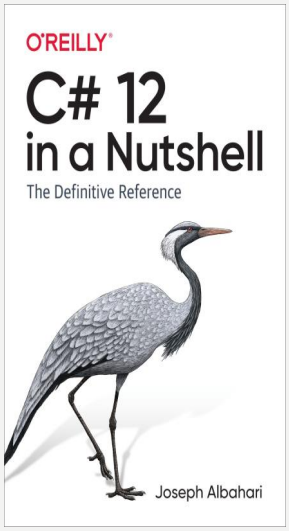
```
2 references | 0 changes | 0 authors, 0 changes
public class NutshellDbContext : DbContext
{
    0 references | 0 changes | 0 authors, 0 changes
    public NutshellDbContext(DbContextOptions<NutshellDbContext> options) : base(options)
    {
    }

    0 references | 0 changes | 0 authors, 0 changes
    public DbSet<Customer> Customers { get; set; }

    0 references | 0 changes | 0 authors, 0 changes
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (optionsBuilder.IsConfigured)
            return;

        optionsBuilder.UseSqlServer(@"Server=(local);Database=Nutshell;Trusted_Connection=True");
    }
}
```

در متد OnConfiguring شما میتونید ویژگی‌های دیگه‌ای روهم فعال کنید. (مثل lazy loading)



DbContext

به صورت دیفالت EF Core مبتنی بر یک قرارداده، به این معنی که schema دیتابیس رو براساس کلاس‌ها و اسم پراپرتی‌هاشون برامون شبیه‌سازی میکنه.

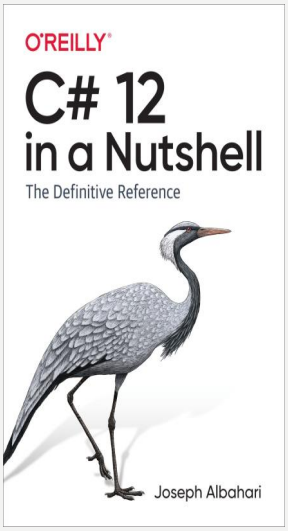
البته میشه با استفاده از fluent api با override کردن متد OnModelCreating دیفالت‌ها رو تغییر داد. مثلا ما میتونیم عینا اسم جدول رو مشخص کنیم:

```
1 reference | 0 changes | 0 authors, 0 changes
public class NutshellDbContext(DbContextOptions<NutshellDbContext> options) : DbContext(options)
{
    0 references | 0 changes | 0 authors, 0 changes
    public DbSet<Customer> Customers { get; set; }

    0 references | 0 changes | 0 authors, 0 changes
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder
              .Entity<Customer>()
              .ToTable("Customer");

    }
```

بدون این کد EF Core جدول کلاس customer رو به جدول customers وصل میکنه چرا که ما پراپرتی DbSet<Customer> رو به اسم Customers تعریف کردیم.



DbContext

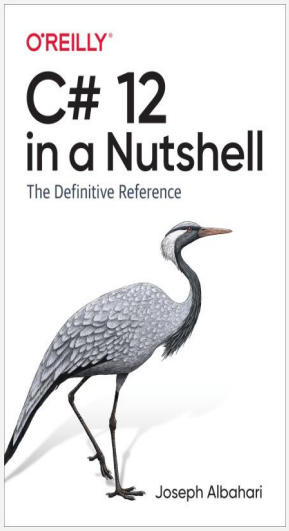
Fluent api ها میتونند روی پراپرتی‌های کلاس‌ها هم تاثیر بذارند و اونا رو مدیریت کنند، این مثال رو ببینید:

```
modelBuilder.Entity<Customer>(entity =>
{
    entity.ToTable("Customer");
    entity
        .Property(propertyExpression => propertyExpression.Name)
        .IsRequired()                  // Column is not nullable
        .HasColumnName("CustomerName"); // Column name is 'CustomerName'
});
```

اینجا پراپرتی Name در کلاس Customer به ستون CustomerName وصل شده و همینطور تاکید کردیم که اجباری باشه و nullable نباشه.

در جدولی که در اسلاید بعدی میاد میتونید لیست fluent api های مهم رو ببینید.

- به جای این روش میتونید از attribute ها هم در خود کلاس برای کانفیگ کردن تنظیمات استفاده کنید. ولی خب این روش باعث میشه انعطاف‌پذیری کانفیگ‌هاتون کم بشه و همینطور بعضی ویژگی ها فقط توسط fluent api پشتیبانی میشه.



DbContext

Method	Purpose	Example

`ToTable`	Specify the database table name for a given entity	`builder`
		`  .Entity<Customer>()`
		`  .ToTable("Customer");`

`HasColumnName`	Specify the column name for a given property	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .HasColumnName("Full Name");`

`HasKey`	Specify a key (usually that deviates from convention)	`builder.Entity<Customer>()` `  .HasKey(c => c.CustomerNr);`

`IsRequired`	Specify that the property requires a value (is not nullable)	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .IsRequired();`

`HasMaxLength`	Specify the maximum length of a variable-length type (usually a string) whose width can vary	`builder.Entity<Customer>()`
		`  .Property(c => c.Name)`
		`  .HasMaxLength(60);`

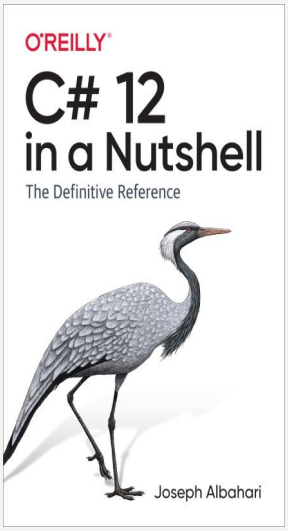
`HasColumnType`	Specify the database data type for a column	`builder.Entity<Purchase>()`
		`  .Property(p => p.Description)`
		
		`.HasColumnType("varchar(80)");`

`Ignore`	Ignore a type	`builder.Ignore<Products>();`
`Ignore`	Ignore a property of a type	`builder.Entity<Customer>()` `  .Ignore(c => c.ChatName);`

`HasIndex`	Specify a property (or combination of properties) should serve in the database as an index	`// Compound index:`
		`builder.Entity<Purchase>()`
		`  .HasIndex(p =>`
		`  new { p.Date, p.Price });`
		
		`// Unique index on one property`
		`builder`
		`  .Entity<MedicalArticle>()`
		`  .HasIndex(a => a.Topic)`
		`  .IsUnique();`

`HasOne`	See "Navigation Properties"	`builder.Entity<Purchase>()`
		`  .HasOne(p => p.Customer)`
		`  .WithMany(c => c.Purchases);`

`HasMany`	See "Navigation Properties"	`builder.Entity<Customer>()`
		`  .HasMany(c => c.Purchases)`
		`  .WithOne(p => p.Customer);`

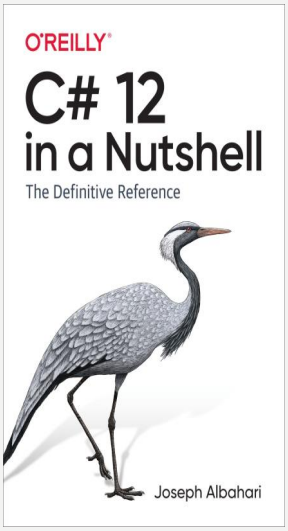


DbContext

EF Core مدل code-first رو پشتیبانی میکنه، به این معنی که شما میتونید اول entity class ها رو تعریف کنید و بعد از EF Core بخواهید که دیتابیس رو براتون ایجاد کنه.
ساده‌ترین راه برای ایجاد دیتابیس فراخوانی متدی در dbContext است:

```
var dbCreated = nutshellDbContext.Database.EnsureCreated();
Console.WriteLine(dbCreated);
```

روش بهتر استفاده از امکان migration در EF Core هست. این روش نه تنها برای ایجاد دیتابیس بهمون کمک میکنه، بلکه هر تغییری در ساختار کلاسها رو هم میتونه به صورت خودکار تشخیص بده و روی دیتابیس اعمال کنه.



DbContext

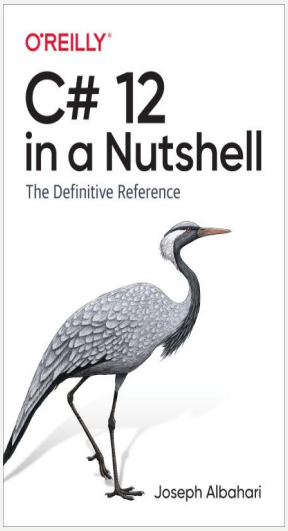
برای فعال کردن این قابلیت میتونید دستورات زیر رو در package manager console وارد کنید:

```
Install-Package Microsoft.EntityFrameworkCore.Tools
Add-Migration InitialCreate
Update-Database
```

دستور اول پکیج لازم برای مدیریت EF Core رو نصب میکنه.

دستور دوم یک کلاس C# که به عنوان code migration شناخته میشه رو ایجاد میکنه که شامل دستورالعمل‌های لازم برای ساخت دیتابیس است.

دستور آخر دستورالعمل‌ها رو روی سرور اجرا میکنه و دیتابیس رو ایجاد میکنه.



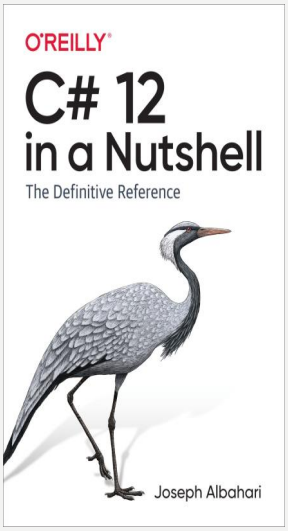
DbContext

حالا بعد از تعریف entity class ها و همینطور ساختن dbContext وقتشه که ازش استفاده کنیم. کد زیر یک کوئری برای خوندن تعداد مشتریا روی دیتابیس اجرا میکنه:

```
using var dbContext = new NutshellDbContext();
Console.WriteLine(dbContext.Customers.Count());
// Executes "SELECT COUNT(*) FROM [Customer] AS [c]"
```

همینطور میتونید یک دیتای جدید در جدول Cusomer/ ثبت کنید:

```
Customer cust = new Customer()
{
    Name = "Sara Wells"
};
dbContext.Customers.Add(cust);
dbContext.SaveChanges();    // Writes changes back to database
```



DbContext

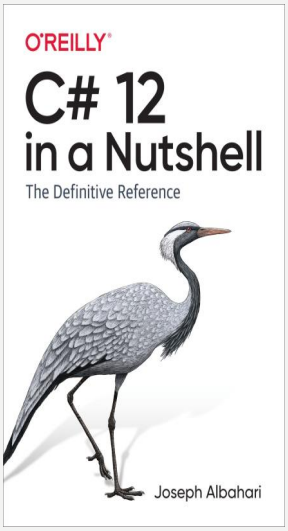
برای خوندن رکورد مشتری که تازه ثبتش کردیم میتونیم به این شکل عمل کنیم:

```
Customer customer = dbContext.Customers
  .Single(c => c.Name == "Sara Wells");
```

و اینجوری هم آپدیتش کنید:

```
customer.Name = "Dr. Sara Wells";
dbContext.SaveChanges();
```

عملگر single برای برگردوندن یک آیتم به وسیله primary key مناسب است و برخلاف دستور First اگر بیش از یک آیتم پیدا کنه خطا میده.



Object Tracking

DbContext میتونه آبجکتها رو track کنه و هر تغییری که اتفاق میفته رو نگه داره. به عبارت دیگه هیچ وقت در lifetime یک dbContext نمیتونیم دو entity جداگانه که به یک سطر از جدول اشاره کنند رو داشته باشیم. به این قابلیت میگن object tracking.

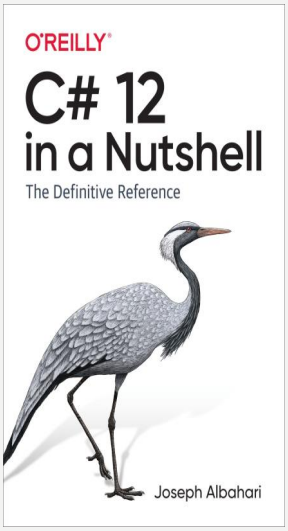
مثال زیر رو ببینید:

```
Customer a = dbContext.Customers.OrderBy(c => c.Name).First();
Customer b = dbContext.Customers.Single(c => c.Id == 2);

a.Name = "it's changed";

Console.WriteLine(a.Name + " " + b.Name);  // it's changed it's changed
```

اینجا a و b هر دو به یک نحوی یک سطر واحد از جدول customer رو در خود نگه میداره. پس اگه فیلدی از a تغییر کنه روی همون فیلد از b هم تاثیر میذاره.



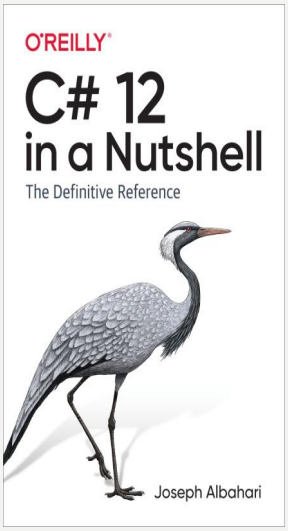
Object Tracking

حالا بیاییم یه بررسی کنیم ببینیم وقتی کوئری دوم در اسلاید قبل اجرا میشه چه اتفاقی میفته. اولین اتفاق اینه که میره توی دیتابیس سرچ میکنه و آیتم مدنظر رو پیدا میکنه. بعد primary key رو میخونه و در cache مربوط به entity ذخیره‌ش میکنه. در ادامه میبینه که قبلا با این primary key یه آبجکت دیگه وجود داره، پس همون آبجکت رو بدون هیچ تغییری برمیگردونه. مثلا اگه همون لحظه کاربر دیگه‌ای بیاد و اسم مشتری رو عوض کنه این تغییر نادیده گرفته میشه.

اگر بخواهید میتونید با فراخوانی متد AsNoTracking این امکان رو غیرفعال کنید.

```
Customer a = dbContext.Customers.OrderBy(c => c.Name).First();
Customer b = dbContext.Customers.Single(c => c.Id == 2);
Customer c = dbContext.Customers.AsNoTracking().Single(c => c.Id == 2);
List<Customer> customers = dbContext.Customers.ToList();
List<Customer> customersNotracking = dbContext.Customers.AsNoTracking().ToList();
a.Name = "it's changed";

Console.WriteLine(
                  $"\n{a.Name} " +                      // it's changed
                  $"\n{b.Name} " +                      // it's changed
                  $"\n{c.Name}" +                       // David
                  $"\n{customers[1].Name}" +            // it's changed
                  $"\n{customersNotracking[1].Name}" +  // David
                  $"");
```



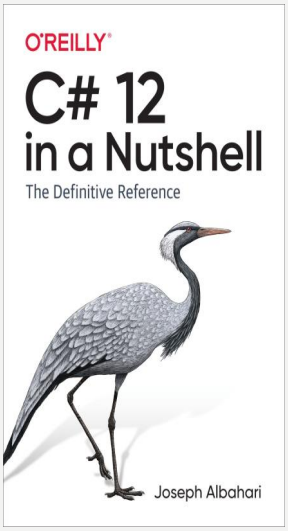
Change Tracking

وقتی مقدار یک پراپرتی در entity که با dbContext لود شده تغییر میکنه، EF Core این تغییرات رو شناسایی میکنه و با فراخوانی متد SaveChanges دیتابیس رو آپدیت میکنه.
برای اینکار موقع لود یک entity در dbcontext یک snapshot از وضعیتش نگه میداره و موقع صدازدن متد SaveChanges وضعیت فعلی رو با وضعیت اصلی مقایسه میکنه.

همچنین به راحتی میتونیم روی تغییرات پیمایش کنیم:

```
List<Customer> customers = dbContext.Customers.ToList();
customers[0].Address = "London";
customers[1].Name = "Joe";
dbContext.Customers.Add(new Customer { Name = "Peter", Address = "Paris" });

foreach (var e in dbContext.ChangeTracker.Entries())
{
    Console.WriteLine($"{e.Entity.GetType().FullName} is {e.State}");
    foreach (var m in e.Members)
        Console.WriteLine(
          $"  {m.Metadata.Name}: '{m.CurrentValue}' modified: {m.IsModified}");
}
```



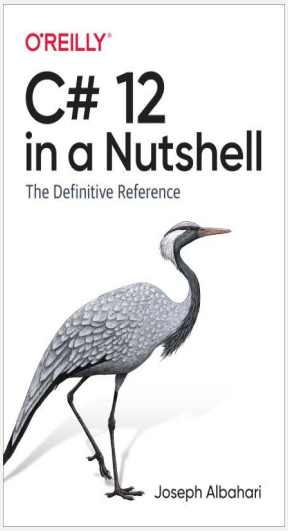
Change Tracking

اینم نتیجه کد اسلاید قبل:

```
CSharp12Nutshell.Chapter08.EFCore.Entities.Customer is Added
  Id: '-2147482647' modified: False
  Address: 'Paris' modified: False
  Name: 'Peter' modified: False
CSharp12Nutshell.Chapter08.EFCore.Entities.Customer is Modified
  Id: '1' modified: False
  Address: 'London' modified: True
  Name: 'Dr. Sara Wells' modified: False
CSharp12Nutshell.Chapter08.EFCore.Entities.Customer is Modified
  Id: '2' modified: False
  Address: '' modified: False
  Name: 'Joe' modified: True
CSharp12Nutshell.Chapter08.EFCore.Entities.Customer is Unchanged
  Id: '3' modified: False
  Address: '' modified: False
  Name: 'Jack' modified: False
```

نکته اینکه اگه از AsNoTracking استفاده کنیم اون آبجکت track نمیشه. وقتی SaveChanges رو استفاده میکنیم EF Core اطلاعات موجود در changeTracker رو به دستورات SQL تبدیل میکنه و دیتابیس رو براساس اون آپدیت میکنه.

همچنین اگر یکی از دستورات به خطا بخوره مابقی دستورات هم اجرا نمیشن و هیچ تغییری در دیتابیس اتفاق نمیفته.



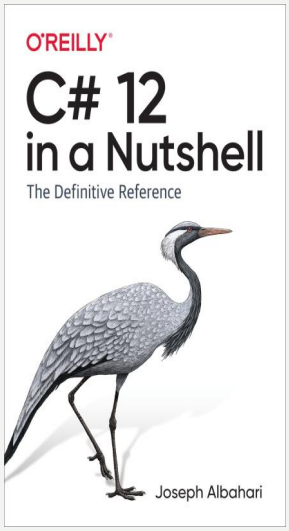
Change Tracking

برای اینکه track کردن تغییرات رو بهینه کنیم میتونیم اینترفیس‌های INotifyPropertyChanged و INotifyPropertyChanging رو روی entity هامون پیاده‌سازی کنیم.

INotifyPropertyChanged بهمون اجازه میده از سربار اضافی موقع مقایسه آبجکت تغییر کرده با آبجکت اصلی جلوگیری کنیم.

INotifyPropertyChanging بهمون اجازه میده تا در کل بتونیم جلوی ذخیره entity رو بگیریم.

(اینجا یه بدهی برای خودم میذارم در مورد این موضوع بیشتر بخونم و یه پست مجزا بذارم براش)



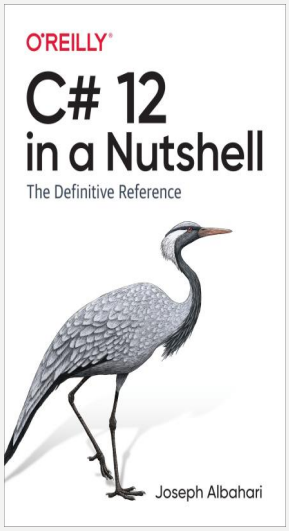
Navigation Properties

Navigation Properties بهمون اجازه میده کارای زیر رو انجام بدیم:

- بدون استفاده از join روی جداولی که باهم relation دارند کوئری بزنیم.
- ایجاد، حذف و ویرایش سطرهای مرتبط به هم بدون به روزرسانی صریح foreign keys.

برای مثال فرض کنید هر مشتری تعدادی خرید داره. میتونیم یک ارتباط یک به چند بین دو entity ایجاد کنیم.

```
2 references | 0 changes | 0 authors, 0 changes
public class Purchase
{
    0 references | 0 changes | 0 authors, 0 changes
    public int ID { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public DateTime Date { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Description { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public decimal Price { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int? CustomerID { get; set; }     // Foreign key field
    0 references | 0 changes | 0 authors, 0 changes
    public Customer Customer { get; set; }   // Parent navigation property
}
```



Navigation Properties

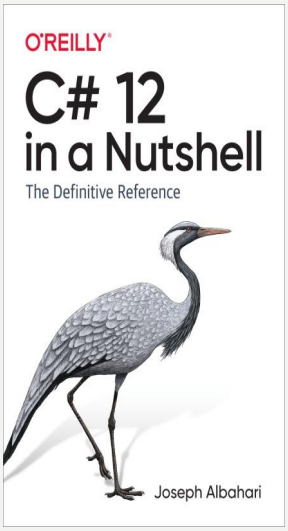
طبق قوانین naming convention در EF Core فیلد Id در جدول Customer با فیلد CustomerId در جدول Purchase ارتباط پیدا میکنه.

اگه به هردلیلی EF Core نتونه ارتباط رو شناسایی کنه میتونیم به راحتی به صورت دستی اون رو تعریف کنیم.

```
modelBuilder.Entity<Purchase>()
        .HasOne(e => e.Customer)
        .WithMany(e => e.Purchases)
        .HasForeignKey(e => e.CustomerID);
```

حالا میتونیم یه همچین کوئری‌ای بنویسیم:

```
using (var dbContext = new NutshellDbContext())
{
    List<Customer> customers = dbContext.Customers.ToList();
    var customersWithPurchases = customers.Where(c => c.Purchases.Any());
}
```



Navigation Properties

وقتی یک entity جدید به کالکشن navigation property اضافه میکنیم، EF Core موقعی که saveChanges رو صدا میزنیم foreign key رو مشخص میکنه و نیاز نیست ما اونو بهش بگیم:

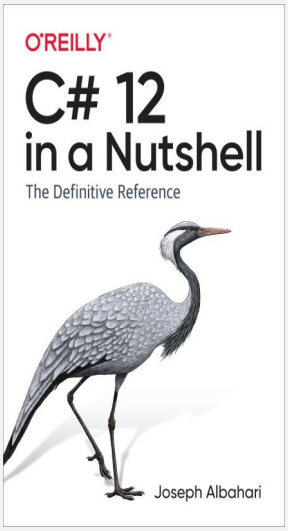
```
Customer cust = dbContext.Customers.Single(c => c.Id == 1);

Purchase p1 = new Purchase { Description = "Bike", Price = 500 };
Purchase p2 = new Purchase { Description = "Tools", Price = 100 };

cust.Purchases.Add(p1);
cust.Purchases.Add(p2);

dbContext.SaveChanges();
```

توی این مثال EF Core به صورت خودکار مقدار CustomerId رو در دو آبجکت purchase برابر ۱ قرار میده.



Navigation Properties

وقتی یک entity رو با استفاده از EF Core میگیریم به صورت پیش فرض navigation property ها برنمیگرده:

```
var customer = dbContext.Customers.First();
Console.WriteLine(customer.Purchases.Count);    // Always 0
```

یک راه حل برای این موضوع استفاده از متد Include هست:

```
var customerIncludePurchases = dbContext.Customers
          .Include(c => c.Purchases)
          .Where(c => c.Id == 2).First();
```

راه دیگه استفاده از projection است، این روش میتونه برای مواقعی که فقط یه سری از پراپرتی‌ها نیاز دارید:

```
var custInfo = dbContext.Customers
          .Where(c => c.Id == 2)
          .Select(c => new
          {
              Name = c.Name,
              Purchases = c.Purchases.Select(p => new { p.Description, p.Price })
          })
          .First();
```