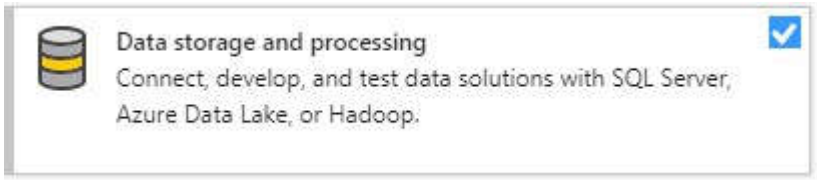
Database

1.mora install

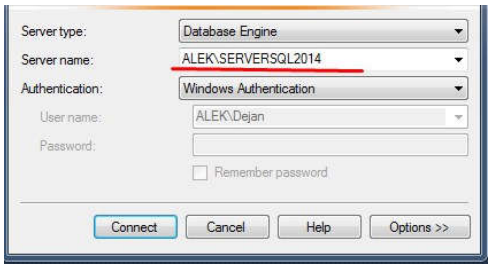


2.
Microsoft.EntityFrameworkCore.SqlServer

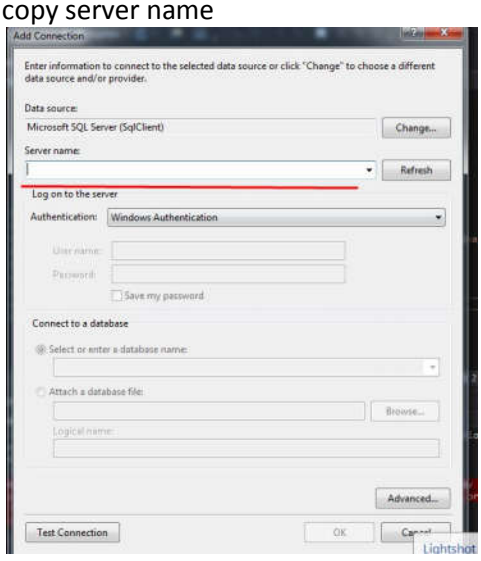
Microsoft.EntityFrameworkCore.Design

Microsoft.EntityFrameworkCore.Tools potrebno za migracii

3.
Tools
Connect to Database
Enter Server Name and database vo Advance ima conection string
can not find server



copy server name



enter server name and choose database -> ok
ili
Server Explorer DataConections ima connection string

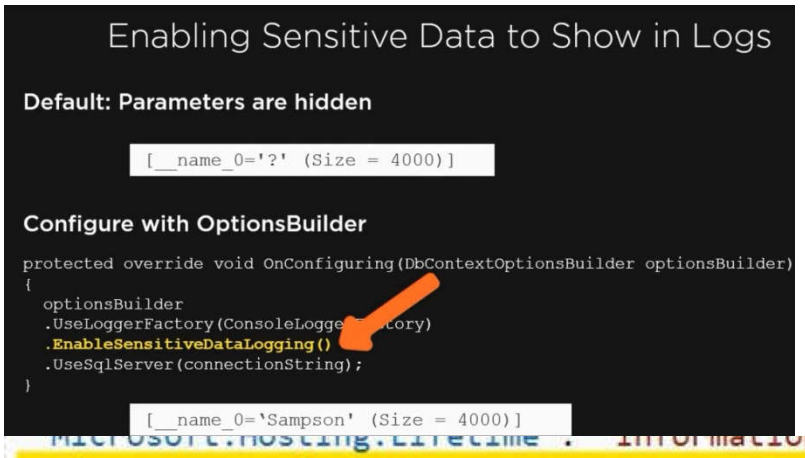
Adding the `DbContext` to dependency injection:

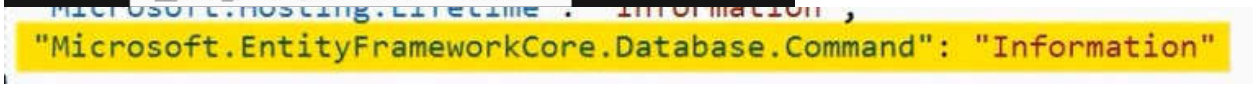
```
public void ConfigureServices(IServiceCollection services)
        {

//in appsettings.json file

"ConnectionStrings": {
    "conn": "Data Source=ALEK;Initial Catalog=EFCore;Integrated Security=True"
  }
```

What is the difference between the following in a database connection string
Trusted_Connection=True;
Integrated Security=SSPI;
Integrated Security=true;
All the above 3 settings specify the same thing, use Integrated Windows Authentication to connect to SQL Server instead of using SQL Server authentication.

We can use either AddDbContext() or AddDbContextPool() method to register our application specific DbContext class with the ASP.NET Core dependency injection system.
The difference between AddDbContext() and AddDbContextPool() methods is, AddDbContextPool() method provides DbContext pooling. With DbContext pooling, an instance from the DbContext pool is provided if available, rather than creating a new instance.
From a performance standpoint AddDbContextPool() method is better over AddDbContext() method.
AddDbContextPool() method is introduced in ASP.NET Core 2.0.
UseSqlServer() extension method is used to configure our application specific DbContext class to use Microsoft SQL Server as the database.

```
services.AddDbContext<AppDbContext>(options =>
        options.UseSqlServer(configuration.GetConnectionString("MyConnectionString")) //adding connection string

        .EnableSensitiveDataLogging()   //default parameters are hidden this will enable them to be showed
```






```
      );



        }

//or you can add connection string in the DatabaseContext class
  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(connectionString); // add connection string

        }


  public AppDbContext(DbContextOptions<AppDbContext> options):base(options)
        {

        }

      public  DbSet<Samurai> Samurais { get; set; } //the table will be named Samurais or
you can specify the name by adding attribute [Table("Samurais")]
```

Migrations

in `Package Manager Console you can exceute power schell commands`

get-help about_entityframeworkcore.

Add-Migration SomeName Adds a new migration.it creates new file in Migration Folder with the name of the migration that files contain two methods the Up() method gets executed if migration is appllied with Update-Database the Down() method executes when the migration is removed.

```
protected override void Up(MigrationBuilder migrationBuilder)
        {}
    protected override void Down(MigrationBuilder migrationBuilder)
        {}
```

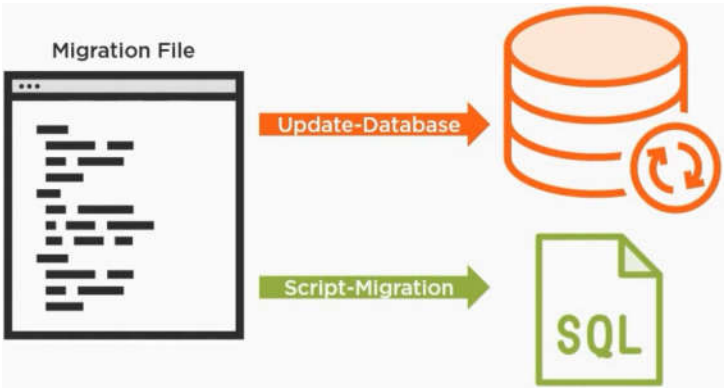
Update-Database SomeName(optional) - Updates the database to a specified migration by default the latest migration applies if name is not specefied.this command can also remove applied migrations with Update-Database SomeName it all migrations after SomeName will be removed will get back to SomeName

`Remove-Migration remove latest migration that is not applied(with Update-Database) yet` update-database 0 This will wipe the database and allow you to remove the Migration Snapshot on your Solution

- Use migrations to keep domain models and database schema in sync
- To add a new migration use Add-Migration command
- To update the database with the latest migration use Update-Database command
- To remove the latest migration that is not yet applied to the database use Remove-Migration
- __EFMigrationsHistory table is used to keep track of the migrations that are applied to the database
- ModelSnapshot.cs file contains the snapshot of the current model and is used to determine what has changed when adding the next migration

Add-Migration	Adds a new migration.
Drop-Database	Drops the database.
Get-DbContext	Gets information about a DbContext type.
Remove-Migration	Removes the last migration.
Scaffold-DbContext	Scaffolds a DbContext and entity types for a database.
Script-DbContext	Generates a SQL script from the current DbContext.
Script-Migration	Generates a SQL script from migrations.
Update-Database	Updates the database to a specified migration.

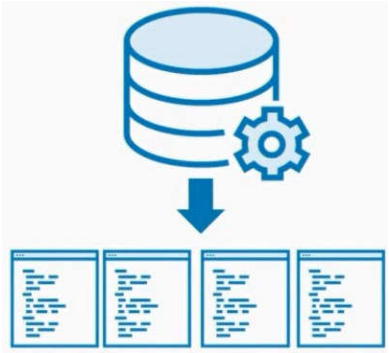
if you add migration and then write Script-Migration a script will be generated and promped to you.



from a migration file you can update databse so ef core can create the database or generate script

update-database -verbose will let you see everything the update-database command is doing

Create DbContext and classes from database



Parameters:

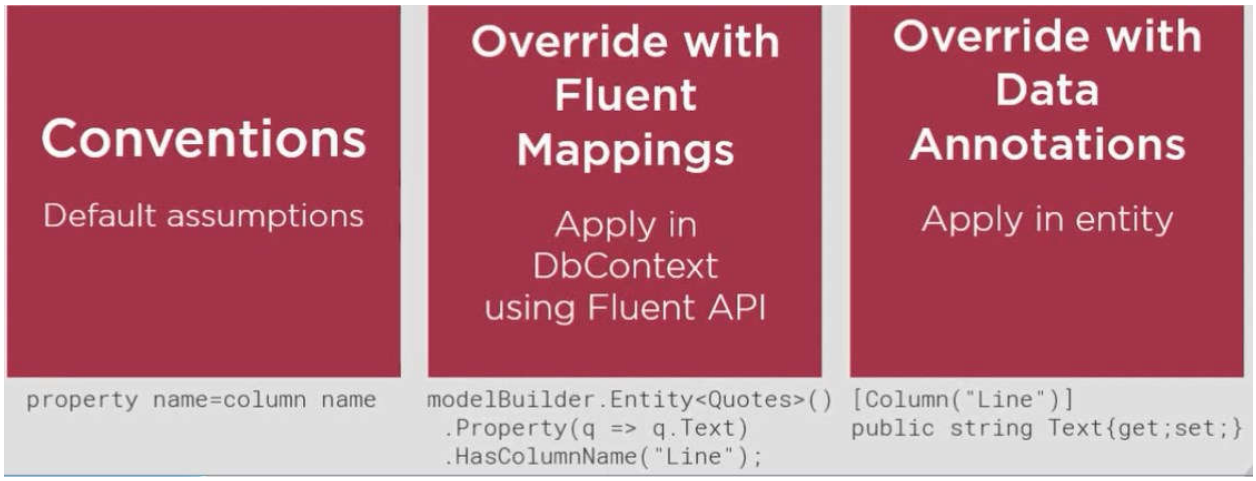
SCAFFOLD-DBCONTEXT

Parameter	Description

Parameter	Description
-Connection <String>	The connection string to the database. For ASP.NET Core 2.x projects, the value can be *name=<name of connection string>*. In that case the name comes from the configuration sources that are set up for the project. This is a positional parameter and is required.
-Provider <String>	The provider to use. Typically this is the name of the NuGet package, for example: `Microsoft.EntityFrameworkCore.SqlServer`. This is a positional parameter and is required.
-OutputDir <String>	The directory to put files in. Paths are relative to the project directory.
-ContextDir <String>	The directory to put the `DbContext` file in. Paths are relative to the project directory.
-Namespace <String>	The namespace to use for all generated classes. Defaults to generated from the root namespace and the output directory. (Available from EFCore 5.0.0 onwards.)
-ContextNamespace <String>	The namespace to use for the generated `DbContext` class. Note: overrides `-Namespace`. (Available from EFCore 5.0.0 onwards.)
-Context <String>	The name of the `DbContext` class to generate.
-Schemas <String[]>	The schemas of tables to generate entity types for. If this parameter is omitted, all schemas are included.
-Tables <String[]>	The tables to generate entity types for. If this parameter is omitted, all tables are included.
-DataAnnotations	Use attributes to configure the model (where possible). If this parameter is omitted, only the fluent API is used.
-UseDatabaseNames	Use table and column names exactly as they appear in the database. If this parameter is omitted, database names are changed to more closely conform to C# name style conventions.
-Force	Overwrite existing files.
-NoOnConfiguring	Suppresses generation of the `OnConfiguring` method in the generated `DbContext` class. (Available from EFCore 5.0.0 onwards.)

provider and connection string are required

```
PM> scaffold-dbcontext -provider Microsoft.EntityFrameworkCore.SqlServer -connection “Data Source
= (localdb)\MSSQLLocalDB; Initial Catalog = SamuraiAppData"
```



Ef Core reads Dbcontext(DbSets) and classes to determen database design this is mapping by convention

to overide these conventions use Fluent Mappings in onModelCreating

or another way to overide conventions you can use Database Annotations

OnModelCreating

The DbContext class has a method called **OnModelCreating** that takes an instance of ModelBuilder as a parameter. This method is called by the framework when your context is first created and when new migration is added (se koristi za mapiranje na modelot migraciite gi koristat rabotite definirani vo OnModelCreating) to build the model and its mappings in memory.. You can override this method to add your own configurations:

```
public class SampleContext : DbContext
{
   // Specify DbSet properties etc

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
      // add your own configuration here
   }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
     Seeding
   modelBuilder.Entity[Employee]().HasData(
      new Employee
      {
         Id = 1,
```

```
        Name = "Mark",
        Department = Dept.IT,
        Email = "mark@pragimtech.com"
    }
```

Specify Table Name for Entety

```
modelBuilder.Entity<Job>().ToTable("TableName");
```

mapping many to many

```
  modelBuilder.Entity<CompanyJob>()
          .HasKey(t => new { t.JobId, t.CompanyId });

            modelBuilder.Entity<CompanyJob>()
                .HasOne(pt => pt.Company)
                .WithMany(p => p.CompanyJobs)
                .HasForeignKey(pt => pt.CompanyId);


            modelBuilder.Entity<CompanyJob>()
                .HasOne(pt => pt.Job)
                .WithMany(t => t.CompanyJobs)
                .HasForeignKey(pt => pt.JobId);
```

Define Shadow prop

```
modelBuilder.Entity<Samurai>()
            .Property<DateTime>("LastModified");
```

add shadow prop in every entity

```
foreach (var entityType in modelBuilder.Model.GetEntityTypes())
{
    modelBuilder.Entity(entityType.Name).Property<DateTime>("Created");
    modelBuilder.Entity(entityType.Name).Property<DateTime>("LastModified");
}
```

```
  );
```

Fluent API in Entity Framework Core

The term *Fluent API* refers to a pattern of programming where method calls are chained together with the end result being certainly less verbose and arguably more readable than a series of statements:

```
1.  // series of statements
2.  modelBuilder.Entity<Order>().Property(t => t.OrderDate).IsRequired();
3.  modelBuilder.Entity<Order>().Property(t => t.OrderDate).HasColumnType("Date");
4.  modelBuilder.Entity<Order>().Property(t => t.OrderDate).HasDefaultValueSql("GetDate()");
5.
6.  // fluent api chained calls
7.  modelBuilder.Entity<Order>()
8.      .Property(t => t.OrderDate)
9.          .IsRequired()
10.         .HasColumnType("Date")
11.         .HasDefaultValueSql("GetDate()");
```

Entity Framework Fluent API is used to configure domain classes to override conventions. EF Fluent API is based on a Fluent API design pattern (a.k.a Fluent Interface) where the result is formulated by method chaining.
In Entity Framework Core, the ModelBuilder class acts as a Fluent API. By using it, we can configure many different things, as it provides more configuration options than data annotation attributes.
Entity Framework Core Fluent API configures the following aspects of a model:

1. Model Configuration: Configures an EF model to database mappings. Configures the default Schema, DB functions, additional data annotation attributes and entities to be excluded from mapping.
2. Entity Configuration: Configures entity to table and relationships mapping e.g. PrimaryKey, AlternateKey, Index, table name, one-to-one, one-to-many, many-to-many relationships etc.
3. Property Configuration: Configures property to column mapping e.g. column name, default value, nullability, Foreignkey, data type, concurrency column etc.

Fluent API Configurations

Override the `OnModelCreating` method and use a parameter `modelBuilder` of type `ModelBuilder` to configure domain classes

The following table lists important methods for each type of configuration.

Configurations	Fluent API Methods	Usage
Model Configurations	HasDbFunction()	Configures a database function when targeting a relational database.
	HasDefaultSchema()	Specifies the database schema.

	HasAnnotation()	Adds or updates data annotation attributes on the entity.
	HasSequence()	Configures a database sequence when targeting a relational database.
Entity Configuration	HasAlternateKey()	Configures an alternate key in the EF model for the entity.
	HasIndex()	Configures an index of the specified properties.
	HasKey()	Configures the property or list of properties as Primary Key.
	HasMany()	Configures the Many part of the relationship, where an entity contains the reference collection property of other type for one-to-Many or many-to-many relationships.
	HasOne()	Configures the One part of the relationship, where an entity contains the reference property of other type for one-to-one or one-to-many relationships.
	Ignore()	Configures that the class or property should not be mapped to a table or column.
	OwnsOne()	Configures a relationship where the target entity is owned by this entity. The target entity key value is propagated from the entity it belongs to.
	ToTable()	Configures the database table that the entity maps to.
Property Configuration	HasColumnName()	Configures the corresponding column name in the database for the property.
	HasColumnType()	Configures the data type of the corresponding column in the database for the property.
	HasComputedColumnSql()	Configures the property to map to computed column in the database when targeting a relational database.
	HasDefaultValue()	Configures the default value for the column that the property maps to when targeting a relational database.
	HasDefaultValueSql()	Configures the default value expression for the column that the property maps to when targeting relational database.

	HasField()	Specifies the backing field to be used with a property.
	HasMaxLength()	Configures the maximum length of data that can be stored in a property.
	IsConcurrencyToken()	Configures the property to be used as an optimistic concurrency token.
	IsRequired()	Configures whether the valid value of the property is required or whether null is a valid value.
	IsRowVersion()	Configures the property to be used in optimistic concurrency detection.
	IsUnicode()	Configures the string property which can contain unicode characters or not.
	ValueGeneratedNever()	Configures a property which cannot have a generated value when an entity is saved.
	ValueGeneratedOnAdd()	Configures that the property has a generated value when saving a new entity.
	ValueGeneratedOnAddOrUpdate()	Configures that the property has a generated value when saving new or existing entity.
	ValueGeneratedOnUpdate()	Configures that a property has a generated value when saving an existing entity.

Reference Loop Handling

```
Install-Package Microsoft.AspNetCore.Mvc.NewtonsoftJson
 services.AddControllersWithViews().AddNewtonsoftJson(opt =>
        {
           opt.SerializerSettings.ReferenceLoopHandling = ReferenceLoopHandling.Ignore;
  });

ReferenceLoopHandling.Ignore;          ako e ignore ne pecati loop
ReferenceLoopHandling.Error;          error frla isklucok
ReferenceLoopHandling.Serialize;      pecati  loop
```

Mappings

name conventions + fluent Api + Data Annotations

One to One

Ef 6

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public virtual StudentAddress Address { get; set; }
}

public class StudentAddress
{
    [ForeignKey("Student")]
    public int StudentAddressId { get; set; }

    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public virtual Student Student { get; set; }
}
```

Efcore

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public StudentAddress Address { get; set; }
}

public class StudentAddress
{
    public int StudentAddressId { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public int StudentId { get; set; }
    public Student Student { get; set; }
}
```

EF Core creates a unique index on the NotNull foreign key column `StudentId` in the `StudentAddresses` table, as shown above. This ensures that the value of the foreign key column `StudentId` must be unique in the `StudentAddress` table, which is necessary of a one-to-one relationship.

Many to Many

```
public class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    public Course()
    {
        this.Students = new HashSet<Student>();
    }

    public int CourseId { get; set; }
    public string CourseName { get; set; }

    public virtual ICollection<Student> Students { get; set; }
}
```

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }

    public IList<StudentCourse> StudentCourses { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public string Description { get; set; }

    public IList<StudentCourse> StudentCourses { get; set; }
}
public class StudentCourse
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

```
modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.StudentId, sc.CourseId });
```

foreign key

if foreign key are not specified ef core will create shadow properties but can mix up the principle and dependent entity ef core will guess for example making the dependand entity principle which you dont want

one to one

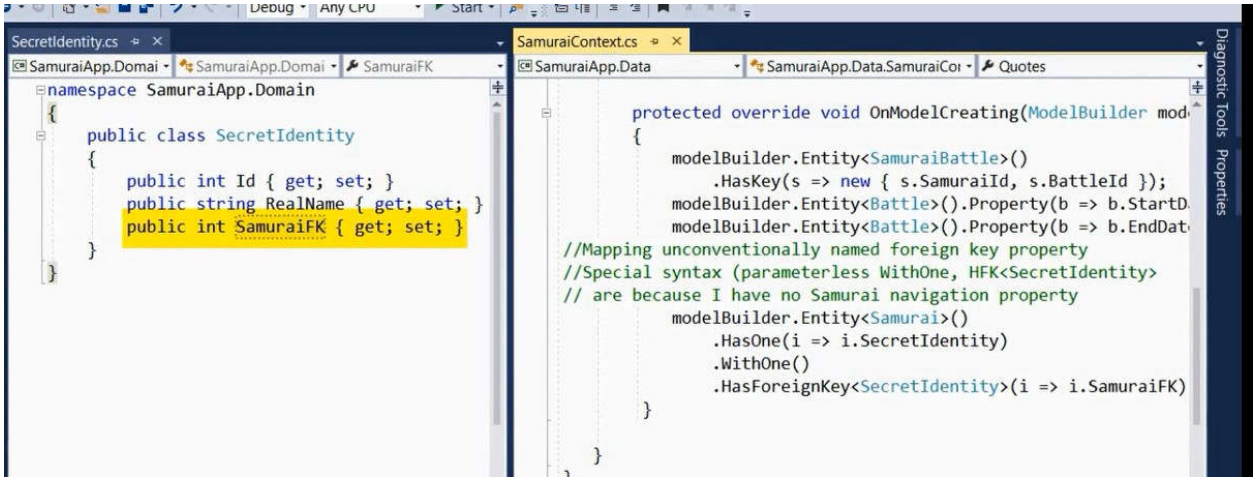
```
modelBuilder.Entity<Samurai>()
    .HasOne(s => s.SecretIdentity)
    .WithOne(i => i.Samurai).IsRequired();
```

with fluent api you can specefied dependent and principle in this case SecretIdenitty is dependent

one to one Samurai Secret Identity
the foreign key of SecretIdentity will the samurai primary key

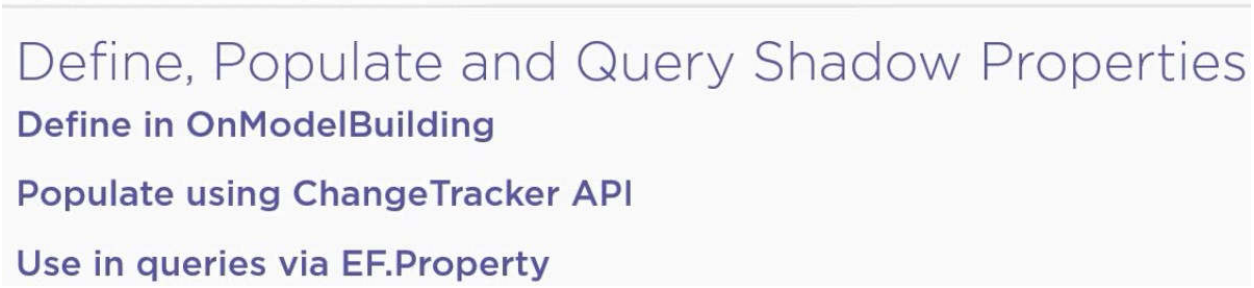
```
private static void AddSecretIdentityToExistingSamurai()
{
    Samurai samurai;
    using (var separateOperation = new SamuraiContext())
    {
        samurai = _context.Samurais.Find(2);
    }
    samurai.SecretIdentity = new SecretIdentity { RealName = "Julia" };
    _context.Samurais.Attach(samurai);
    _context.SaveChanges();
}
```

fk with fluent api without name convention or annotation for one to one



Shadow Properties

```
modelBuilder.Entity<Samurai>()
            .Property<DateTime>("LastModified");

_context.Entry(samurai)
        .Property("LastModified").CurrentValue = DateTime.Now;

_context.Samurais
        .OrderBy(s => EF.Property<DateTime>(s, "LastModified"));
```



One to One

```
 public class Samurai
   {

       public int Id { get; set; }
       public string Name { get; set; }
       public Horse horse { get; set; } //required


   }
```

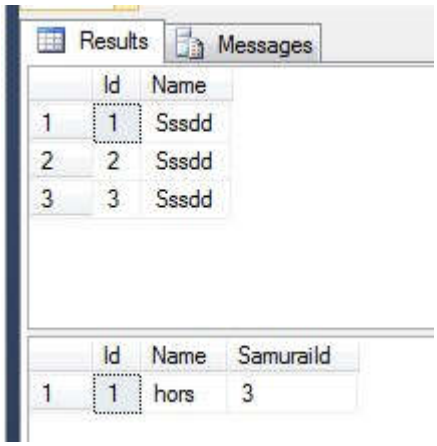
Samurai can be without Horse

Horse must have Samurai

except if `public int SamuraiId is null`

```
public class Horse
    {

        public int Id { get; set; }
        public string Name { get; set; }

        public int SamuraiId { get; set; } //required

        [ForeignKey("SamuraiFk")] //if naming conventional are not followed for SamuraiId
```

```
        public Samurai samurai { get; set; }

    }
select * from Samurai

select * from Horse
```

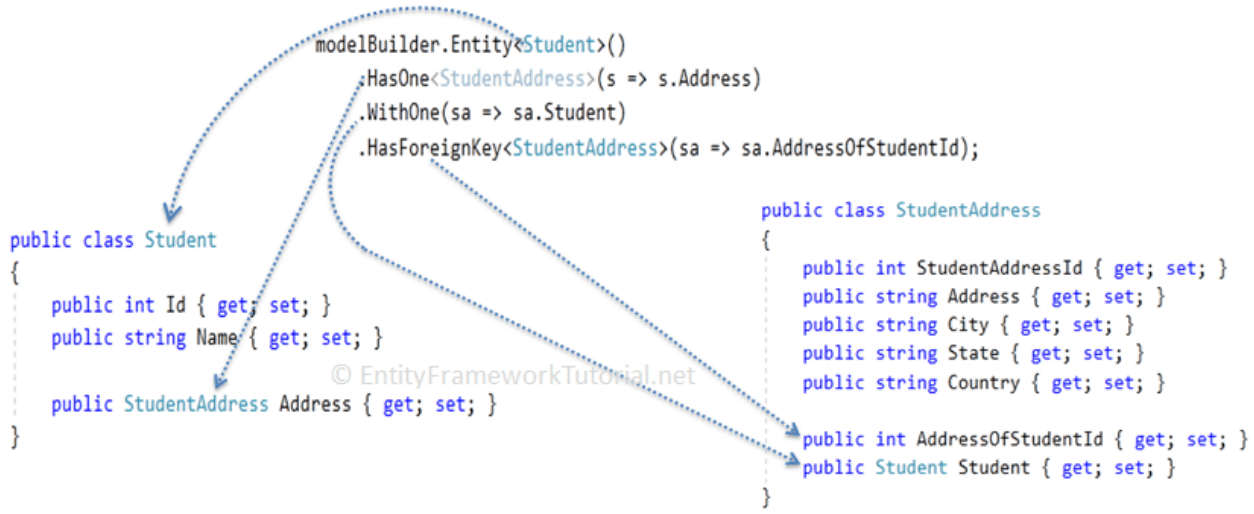


	Id	Name
1	1	Sssdd
2	2	Sssdd
3	3	Sssdd

	Id	Name	Samuraiid
1	1	hors	3

Fluent Api one to one

```
modelBuilder.Entity<Student>()
    .HasOne<StudentAddress>(s => s.Address)
    .WithOne(ad => ad.Student)
    .HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId);
```



One to Many

The following code shows a one-to-many relationship between `Blog` and `Post`

```
public class Blog

{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }

  [ForeignKey("BlogFk")] //if naming conventional are not followed for BlogId(class name
+ Id)
    public Blog Blog { get; set; }
}
```

- `Post` is the dependent entity
- `Blog` is the principal entity
- `Blog.BlogId` is the principal key (in this case it is a primary key rather than an alternate key)
- `Post.BlogId` is the foreign key
- `Post.Blog` is a reference navigation property
- `Blog.Posts` is a collection navigation property
- `Post.Blog` is the inverse navigation property of `Blog.Posts` (and vice versa)

While it is recommended to have a foreign key property defined in the dependent entity class, it is not required. If no foreign key property is found, a shadow foreign key property will be introduced

Fluent Api one to many

```
modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
 .HasForeignKey(p => p.BlogForeignKey);
```

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogForeignKey { get; set; }
    public Blog Blog { get; set; }
}
```

with no foreign key will create shadow prop

```
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
```

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

you can add custom name for shodow prop

```
       // Add the shadow property to the model
        modelBuilder.Entity<Post>()
.Property<int>("BlogForeignKey");

        // Use the shadow property as a foreign key
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
.HasForeignKey("BlogForeignKey");
```

Many-to-many

relationships without an entity class to represent the join table are not yet supported. However, you can represent a many-to-many relationship by including an entity class for the join table and mapping two separate one-to-many relationships.

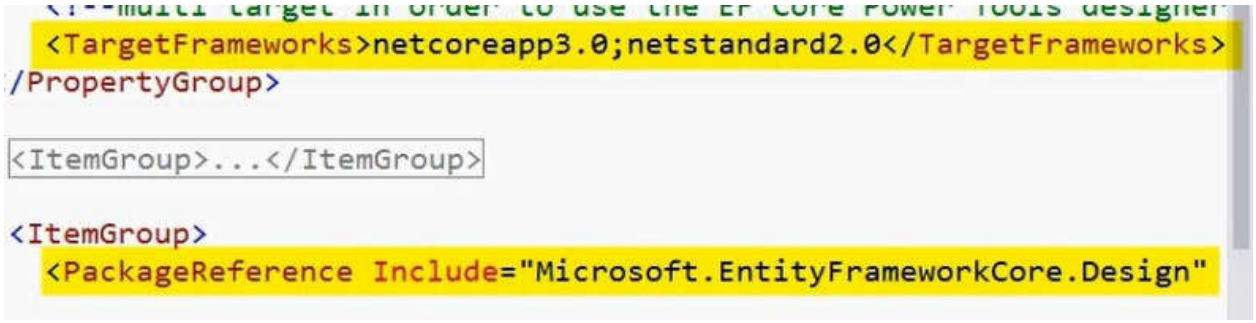
you can add many to many object directly in dbcontext.add() method

```
class MyContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Tag> Tags { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<PostTag>()
            .HasKey(t => new { t.PostId, t.TagId });

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Post)
            .WithMany(p => p.PostTags)
            .HasForeignKey(pt => pt.PostId);

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Tag)
            .WithMany(t => t.PostTags)
            .HasForeignKey(pt => pt.TagId);
    }
```

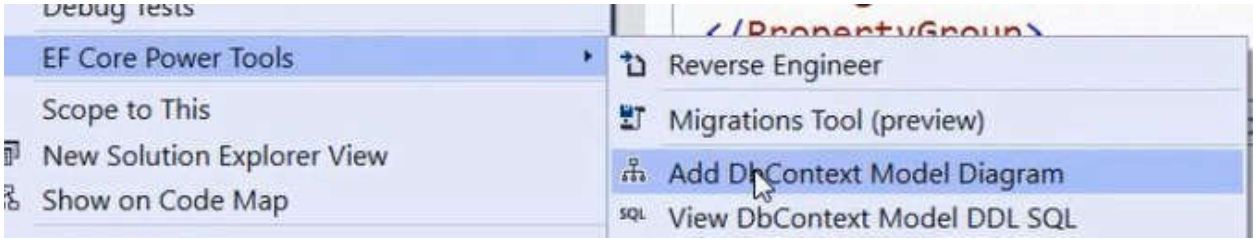
```
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class Tag
{
    public string TagId { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class PostTag
{
    public int PostId { get; set; }  //required

    public Post Post { get; set; }  //optional

    public string TagId { get; set; } //required
    public Tag Tag { get; set; }   //optional

}
```

Visualizing how Ef Core see my Model

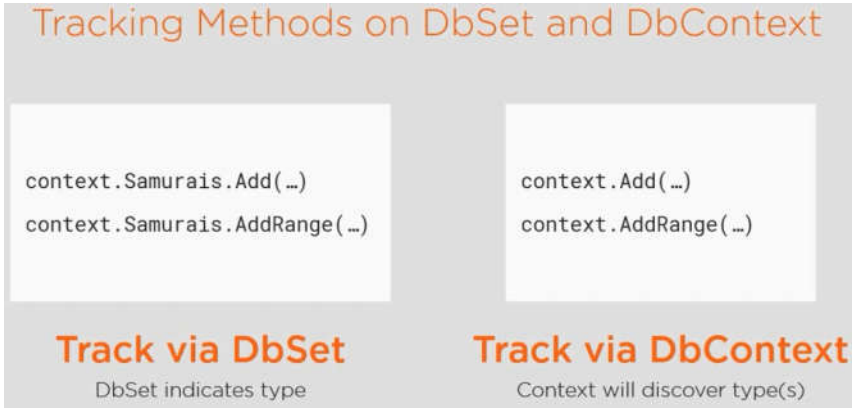
Individual components ->DGML editor must be installed



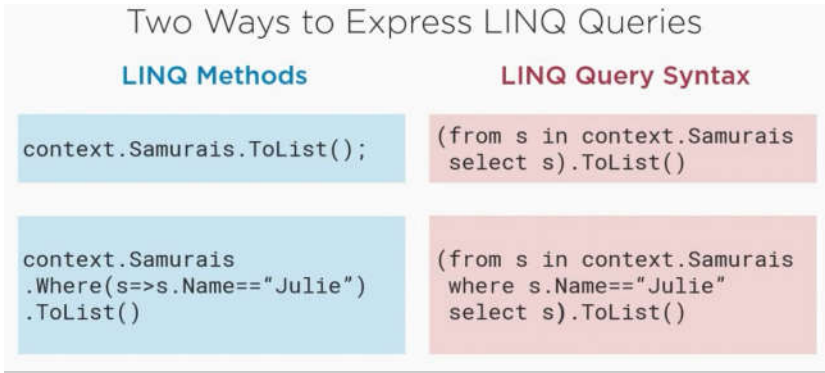
righ click the project where dbcontext live and choose

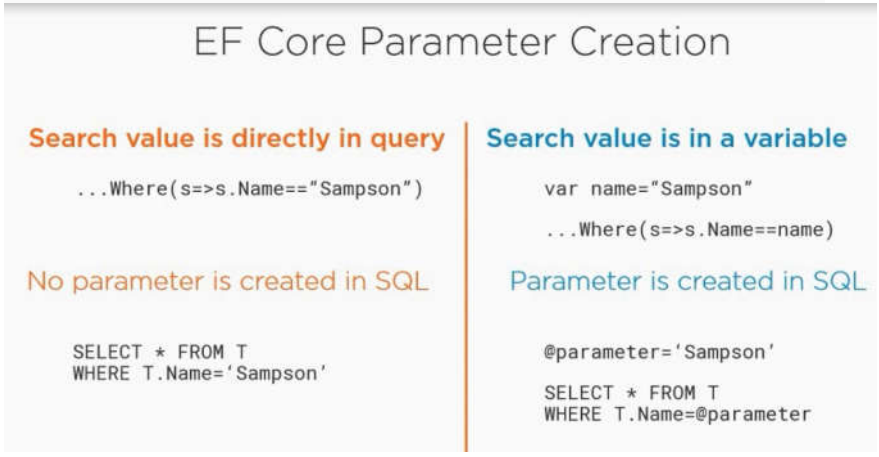

Debug Tests
EF Core Power Tools
Scope to This
New Solution Explorer View
Show on Code Map
Reverse Engineer
Migrations Tool (preview)
Add DbContext Model Diagram
View DbContext Model DDL SQL

Interacting with Data



with 4 objects and more it will bulk insert





Use **FromSqlRaw** to execute a SQL query or stored procedure that returns entities.
Use **ExecuteSqlRaw** to execute a SQL query or stored procedure that performs database operations but does not return entities example insert update delete but return number of rows affected

```
_context.Samurais.FromSQLRaw("some sql string").ToList();
_context.Samurais.FromSQLRawAsync("some sql string").ToList();
_context.Samurais.FromSQLInterpolated($"some sql string {var}").ToList();
_context.Samurais.FromSQLInterpolatedAsync($"some sql {var}").ToList();
```

DbSet Methods to Run Raw SQL

Synchronous and asynchronous options

Special method for interpolated strings

Creates an IQueryable, so you still need an execution method

Use parameters to avoid SQL injection!!

```
_context.Database.ExecuteSQLRaw("some SQL string");
_context.Database.ExecuteSQLRawAsync("some SQL string");
_context.Database.ExecuteSQLInterpolated($"some SQL string {variable}");
_context.Database.ExecuteSQLInterpolatedAsync($"some SQL string {var}");
```

Run Raw SQL for Non-Query Commands

Only result is number of rows affected

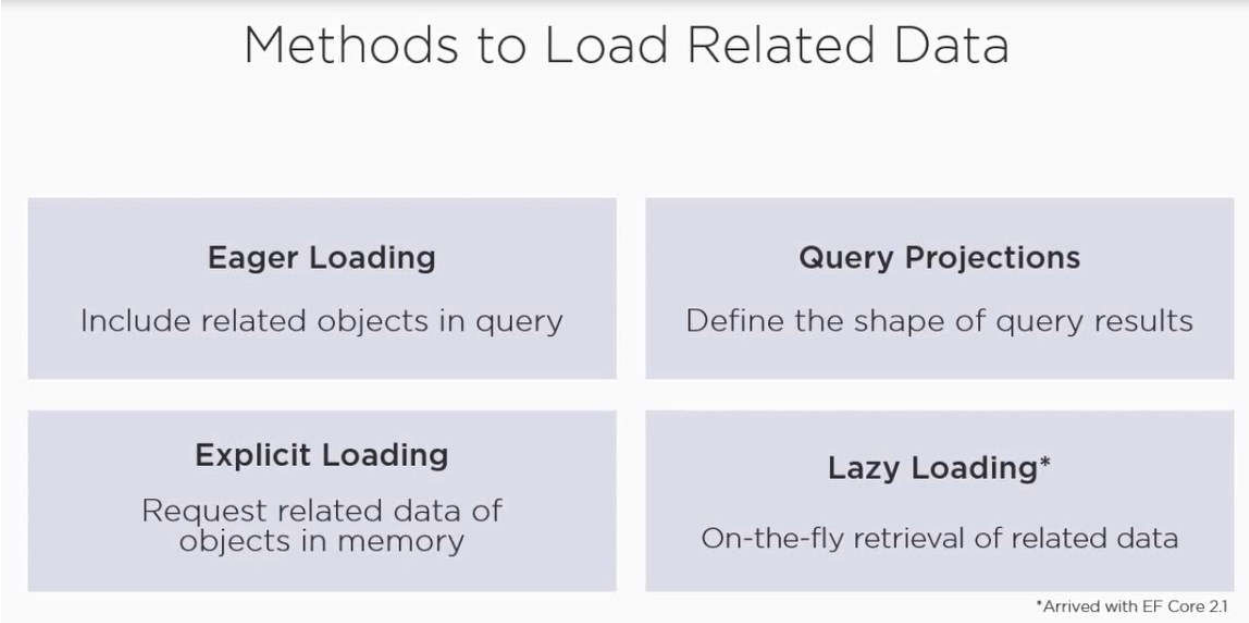
On-the-fly SQL or Stored Procedures

Using Related Data to filter

```
private static void FilteringWithRelatedData()
{
  var samurais = _context.Samurais
                         .Where(s => s.Quotes.Any(q => q.Text.Contains("happy")))
                         .ToList();
}
```

with this you dont load related data just use the related data to filter samurais

Loading Related Data



Eager Loading

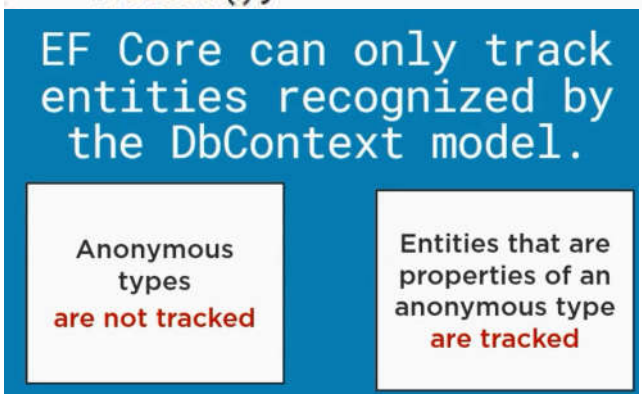
```
_context.Samurais
  .Include(s=>s.Quotes)
  .ThenInclude(q=>q.Translations)
```

◀ Include children & grandchildren

Include always loads the entire set of related objects you can not filter related data

Query Projections

```
var samuraisWithHappyQuotes = _context.Samurais
    .Select(s => new {
      Samurai=s,
      HappyQuotes = s.Quotes.Where(q => q.Text.Contains("happy"))
    })
    .ToList();
```




```
var samuraisWithHappyQuotes = _context.Samurais
    .Select(s => new {
      Samurai=s,
      HappyQuotes = s.Quotes.Where(q => q.Text.Contains("happy"))
    })
    .ToList();
var firstsamurai = samuraisWithHappyQuotes[0].Samurai.Name += " The Happiest";
}
```

enteties are beign tracked by this query projection and will mark first item as modified

Explcit Loading

```
With samurai object already in memory

_context.Entry(samurai).Collection(s => s.Quotes).Load();

_context.Entry(samurai).Reference(s => s.Horse).Load();
```

```
var samurai = _context.Samurais.FirstOrDefault(s => s.Name.Contains("Julie"));
_context.Entry(samurai).Collection(s => s.Quotes).Load();
_context.Entry(samurai).Reference(s => s.Horse).Load();
```

samurai will be in loaded in memory
collections for collections properties
Reference for single propertie

Filter loaded data using the Query method

```
var happyQuotes = context.Entry(samurai)
      .Collection(b => b.Quotes)
      .Query()
      .Where(q => q.Quote.Contains("Happy")
      .ToList();
```

Lazy Loading

lazy loading is off by default

what is lazy loading? see image

```
private static void LazyLoadQuotes()
{
  var samurai = _context.Samurais.FirstOrDefault(s => s.Name.Contains("Julie"));

  var quoteCount = samurai.Quotes.Count();
}
```

Enable with these requirements:

Every navigation property must be virtual

Microsoft.EntityFramework.Proxies package

ModelBuilder.UseLazyLoadingProxies()

Views and Procedures

you can use migrations to add views or procedures

in Up method migrationBuilder.Sql(@" ")

in down method delete procedure or views

for views

```
public DbSet<SamuraiBattleStat> SamuraiBattleStats { get; set; }
```

```
modelBuilder.Entity<SamuraiBattleStat>().HasNoKey().ToView("SamuraiBattleStats");
```

Ef core will not track enteties marked with HasNoKey()

for procedures

```
var text = "Happy";
var samurais = _context.Samurais.FromSqlRaw(
  "EXEC dbo.SamuraisWhoSaidAWord {0}", text).ToList();
```

```
var text = "Happy";
var samurais = _context.Samurais.FromSqlInterpolated(
  $"EXEC dbo.SamuraisWhoSaidAWord {text}").ToList();
```

```
_context.Database.ExecuteSQLRaw("some SQL string");
_context.Database.ExecuteSQLRawAsync("some SQL string");
_context.Database.ExecuteSQLInterpolated($"some SQL string {variable}");
_context.Database.ExecuteSQLInterpolatedAsync($"some SQL string {var}");
```

Run Raw SQL for Non-Query Commands

Only result is number of rows affected

On-the-fly SQL or Stored Procedures

```
var samuraiId = 22;
//var x =_context.Database
//          .ExecuteSqlRaw("EXEC DeleteQuotesForSamurai {0}", samuraiId );
samuraiId = 31;
 _context.Database
       .ExecuteSqlInterpolated($"EXEC DeleteQuotesForSamurai {samuraiId}");
```

Owned type

Ef core assumes that every class is an entity

if we want to create class that is not entity we must mapp it explicitly

```
modelBuilder.Entity<Samurai>().OwnsOne(s => s.BetterName);
```

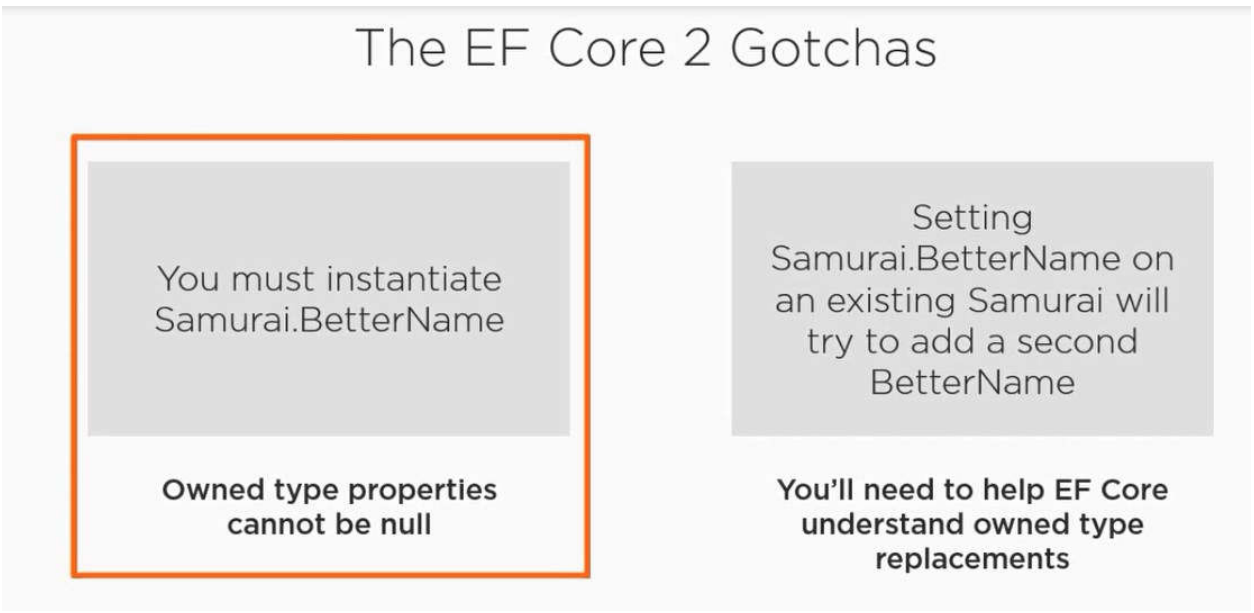
so the property (class) in the class Samurai BetterName can be resolved in with the property's (class's) properties, now the Samurai entity will have the BetterName class properties define in itself

to have the BetterName class properties defined in another table

```
modelBuilder.Entity<Samurai>().OwnsOne(s => s.BetterName).ToTable("BetterNames");
```

to change the column names

```
}
modelBuilder.Entity<Samurai>().OwnsOne(s => s.BetterName).Property(b => b.GivenName).HasColumnName("GivenName")
modelBuilder.Entity<Samurai>().OwnsOne(s => s.BetterName).Property(b => b.SurName).HasColumnName("SurName");
```



get id from entity

```
   var std = new Student(){ StudentName = "Steve" };
     context.Add(std);
     context.SaveChanges();


     Console.Write(std.StudentID); // 1
```

It will be negative until you save your changes. Just call `Save` on the context.
_dbContext.Locations.Add(location);
_dbContext.Save();
After the save, you will have the ID which is in the database.

Logging

```
public static readonly ILoggerFactory MyLoggerFactory
    = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter((category, level) =>
                    category == DbLoggerCategory.Database.Command.Name
                    && level == LogLevel.Information)
                .AddConsole();
        });
```

In this example, the log is filtered to only return messages:

- in the 'Microsoft.EntityFrameworkCore.Database.Command' category
- at the 'Information' level

apply thr logger

```
optionsBuilder.UseLoggerFactory(
  DbCommandConsoleLoggerFactory).EnableSensitiveDataLogging();
or
    services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(configuration.GetConnectionString("MyConnectionString")));
            //.EnableSensitiveDataLogging()
            //.UseLoggerFactory(DbCommandConsoleLoggerFactory)
            //loging



public static readonly LoggerFactory ChangeTrackingAndSqlConsoleLoggerFactory
```

```
 = new LoggerFactory(new[] {
     new ConsoleLoggerProvider (
       (category, level) =>
       (category == DbLoggerCategory.ChangeTracking.Name |
       category==DbLoggerCategory.Database.Command.Name)
       && level==LogLevel.Debug ,true)
 });
```

Apart from the Log Levels, the logger API defines several DBLogger categories. We can use them to filter out the log.

DBLogger Category	Description
DbLoggerCategory.ChangeTracking.Name	Logger category for messages from change detection and tracking.
DbLoggerCategory.Database.Name	Logger categories for messages related to database interactions.
DbLoggerCategory.Database.Connection.Name	Logger category for messages related to connection operations.
DbLoggerCategory.Database.Transaction.Name	Logger category for messages related to transaction operations.
DbLoggerCategory.Database.Command.Name	Logger category for command execution, including SQL sent to the database.
DbLoggerCategory.Infrastructure.Name	Logger category for miscellaneous messages from the Entity Framework infrastructure.
DbLoggerCategory.Migrations.Name	Logger category messages from Migrations.
DbLoggerCategory.Query.Name	Logger category for messages related to queries, excluding the generated SQL, which is in the DbLoggerCategory.Database.Command category.
DbLoggerCategory.Scaffolding.Name	Logger category for messages from scaffolding/reverse engineering.
DbLoggerCategory.Update.Name	Logger category for messages related to SaveChanges(), excluding messages specifically relating to database interactions which are covered by the DbLoggerCategory.Database categories.
DbLoggerCategory.Model.Name	Logger categories for messages related to model building and metadata.

DBLogger Category	Description
DbLoggerCategory.Model.Validation.Name	Logger category for messages from model validation.