ASP.NET Core Entity Framework Core

FSMVÜ Computer Eng. Dr. Öğr Üyesi Ali NİZAM

Object-Relational Mapper (ORM)

- Object-Relational Mapper (ORM) tools such as Entity Framework:
  - Database first : We start by defining our database objects and their relations, then write our classes to match them, and we bind them together
  - Code first : We start by designing our classes as Plain Old CLR Objects (POCOs) to model the concepts that we wish to represent, without caring (too much!) how they will be persisted in the database

ORM

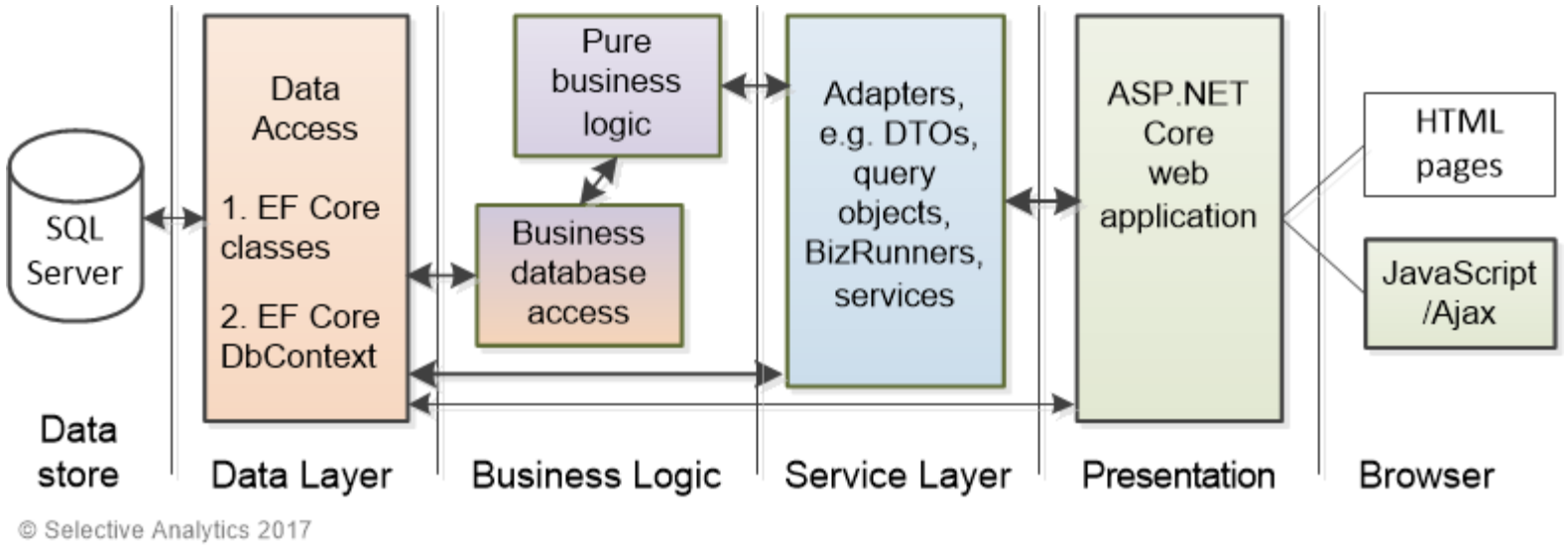
- Database servers and object-oriented software use different principles:
  - databases use primary keys to define that a row is unique,
  - whereas .NET class instances are, by default, considered unique by their reference.

https://livebook.manning.com/#!/book/entity-framework-core-in-action/chapter-1/v-11/47

EF Core

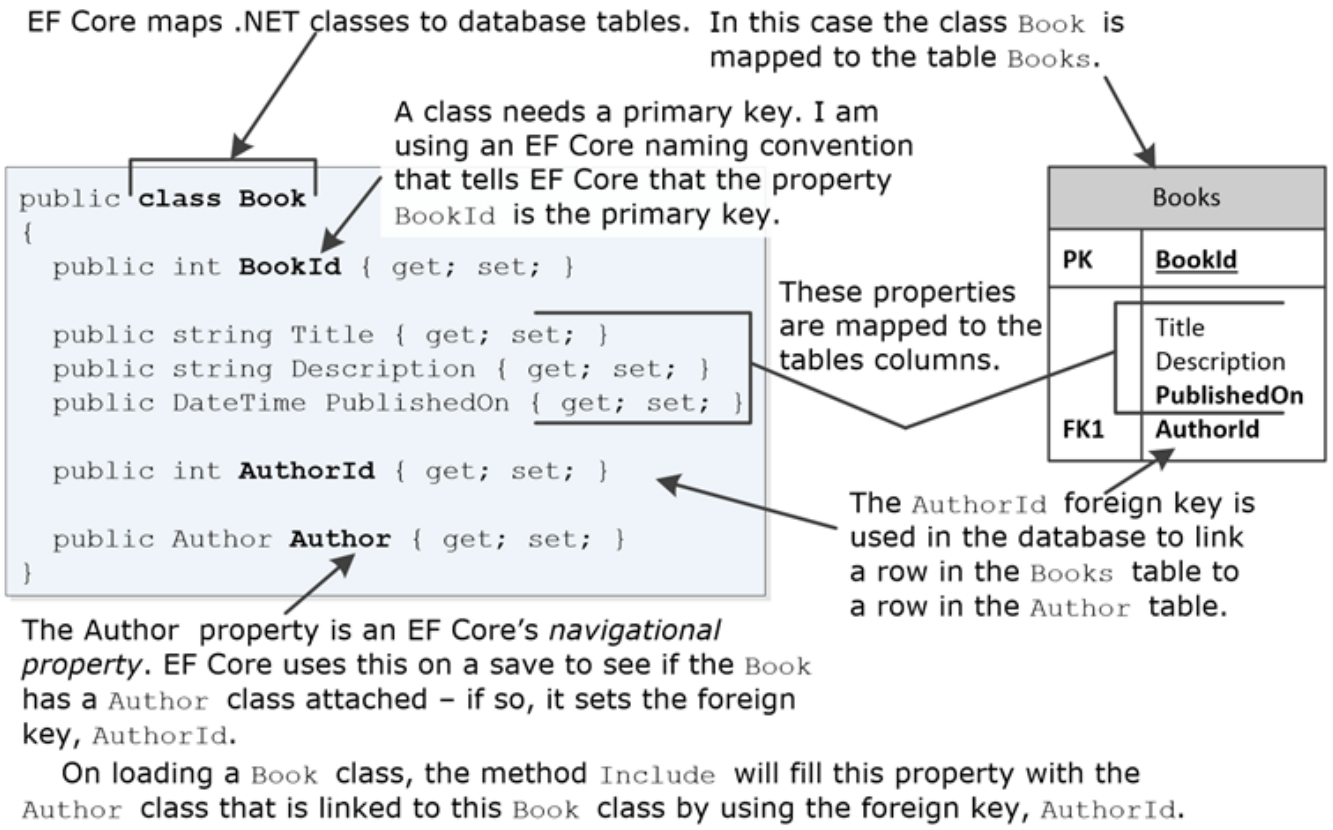
- In EF6, you could use an EDMX/database designer to visually design your database, an option known as design-first.
- EF Core doesn’t support the design-first approach, and there are no plans to add it.

Structure



https://www.thereformedprogrammer.net/six-ways-to-build-better-entity-framework-core-and-ef6-applications/

EF Core Class Structure



https://livebook.manning.com/#!/book/entity-framework-core-in-action/chapter-1/v-11/120

To create a local database from model

- NuGet Package Manager > Package Manager Console.
- PMC menüsü
- In the PMC, enter the following commands:
- Add-Migration Initial
- Update-Database

Migration

- Migrations are used to keep the database schema in sync with the model.

To create a model from local database

- Reverse engineer your model

- Tools –> NuGet Package Manager –> Package Manager Console
- Run the following command to create a model from the existing database:
- PowerShell

- Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models –

- You can specify which tables you want to generate entities for by adding the -Tables argument to the command above. For example, -Tables Blog,Post.

https://docs.microsoft.com/tr-tr/ef/core/get-started/aspnetcore/existing-db

Querying

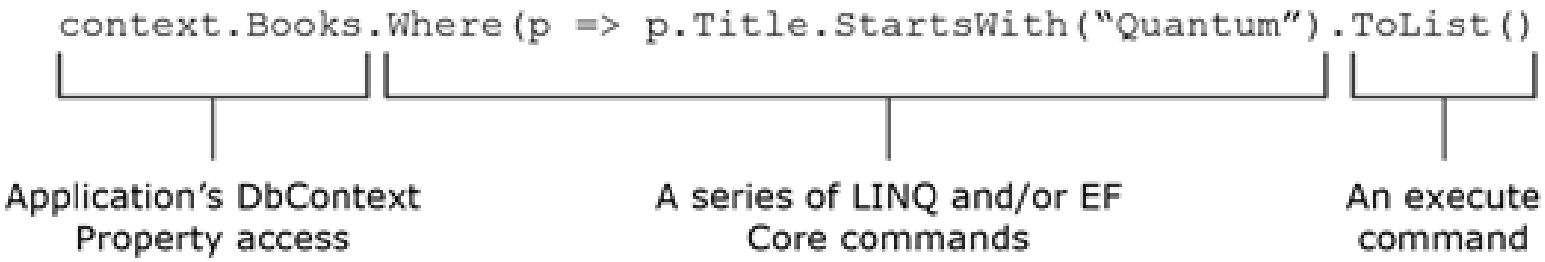

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Read

- The code to read all the books and output them to the console

```
public static void ListAll()
{
    using (var db = new AppDbContext())             //
    {
        foreach (var book in        db.Books.AsNoTracking()             //

            .Include(a => a.Author))                //
        {
            var webUrl = book.Author.WebUrl == null
                ? "- no web URL given -"
                : book.Author.WebUrl;
            Console.WriteLine(
                $"{book.Title} by {book.Author.Name}");
            Console.WriteLine("     " +
                "Published on " +
                $"{book.PublishedOn:dd-MMM-yyyy}" +
                $". {webUrl}");
        }
    }
```

https://livebook.manning.com/#!/book/entity-framework-core-in-action/chapter-1/v-11/120


context.Books.Where(p => p.Title.StartsWith("Quantum").ToList()
Application's DbContext Property access
A series of LINQ and/or EF Core commands
An execute command

Connection Configuration

- Read the connection string using the ConfigurationManager

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {

optionsBuilder.UseSqlServer(ConfigurationManager.ConnectionStrings["
BloggingDatabase"].ConnectionString);
    }
}
```

ASP.NET Core Connection Settings

- connection string could be stored in appsettings.json
- {
- "ConnectionStrings": {
- "BloggingDatabase": "Server=(localdb)\\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;"
- },
- }
- The context is typically configured in Startup.cs with the connection string being read from configuration
- public void ConfigureServices(IServiceCollection services)
- {
- services.AddDbContext<BloggingContext>(options =>
- options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
- }

Persistencey Policy and Transections


```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var context = new BloggingContext())
        {
            using (var transaction = context.Database.BeginTransaction())
            {
                context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/dotnet"});
                context.SaveChanges();

                context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/visualstudio"});
                context.SaveChanges();

                transaction.Commit();
            }
        }
    });
}
```

Create a Model


```
class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Blog>()
                .Property(b => b.Url)
                .IsRequired();
        }
    }
```

Use Data Annotation


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

Conventions

- By convention, public properties with a getter and a setter will be included in the model.

- Data Annotations
- You can use Data Annotations to exclude a property from the model.

- C#

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

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
```

Data Annotations


```
class Car
{
    [Key]
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Or


```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasKey(c => c.LicensePlate);
    }
}
```

Length


```
public class Blog
{
    public int BlogId { get; set; }
    [MaxLength(500)]
    public string Url { get; set; }
}
```

Or

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasMaxLength(500);
    }
```

Fluent API


```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure default schema
        modelBuilder.HasDefaultSchema("Admin");

        //Map entity to table
        modelBuilder.Entity<Student>().ToTable("StudentInfo");
        modelBuilder.Entity<Standard>().ToTable("StandardInfo","dbo");
    }
```

http://www.entityframeworktutorial.net/code-first/configure-entity-mappings-using-fluent-api.aspx

Configure Column Name, Type and Order

The default Code-First convention creates a column for a property with the same name, order, and datatype. You can override this convention, as shown below.

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }
    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure Column
        modelBuilder.Entity<Student>()
                    .Property(p => p.DateOfBirth)
                    .HasColumnName("DoB")
                    .HasColumnOrder(3)
                    .HasColumnType("datetime2");
    }
}
```

IsOptional


```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure Null Column
            modelBuilder.Entity<Student>()
            .Property(p => p.Heigth)
            .IsOptional();

        //Configure NotNull Column
        modelBuilder.Entity<Student>()
            .Property(p => p.Weight)
            .IsRequired();
    }
```

Model Used in Example


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

public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

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

Foreign Key


```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
            .HasOne<Grade>(s => s.Grade)
            .WithMany(g => g.Students)
            .HasForeignKey(s => s.CurrentGradeId);
    }
```

Configure Cascade Delete using Fluent API

- Cascade delete means automatically deleting child rows when the related parent row is deleted.
- For example, if Grade is deleted then all the students in that Grade should also be deleted automatically. The following code configures the cascade delete using the WillCascadeOnDelete method.

```
modelBuilder.Entity<Grade>()
    .HasMany<Student>(g => g.Students)
    .WithRequired(s => s.CurrentGrade)
    .WillCascadeOnDelete();
```

Create a one to one map


```
modelBuilder.Entity<BlogDetail>()
     .HasKey(b => b.BlogId);

modelBuilder.Entity<BlogDetail>()
    .HasOne(b => b.Blog)
    .WithOne(b => b.Detail)
    .IsRequired();
```

Creating one-to-many maps

- an entity can be associated with one or more entities of another type, and each of these entities is associated with at most one entity of the first type, we call that one-to-many

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // configures one-to-many relationship
        modelBuilder.Entity<Student>()
            .HasRequired<Grade>(s => s.CurrentGrade)
            .WithMany(g => g.Students)
            .HasForeignKey<int>(s => s.CurrentGradeId);          }
    }
```

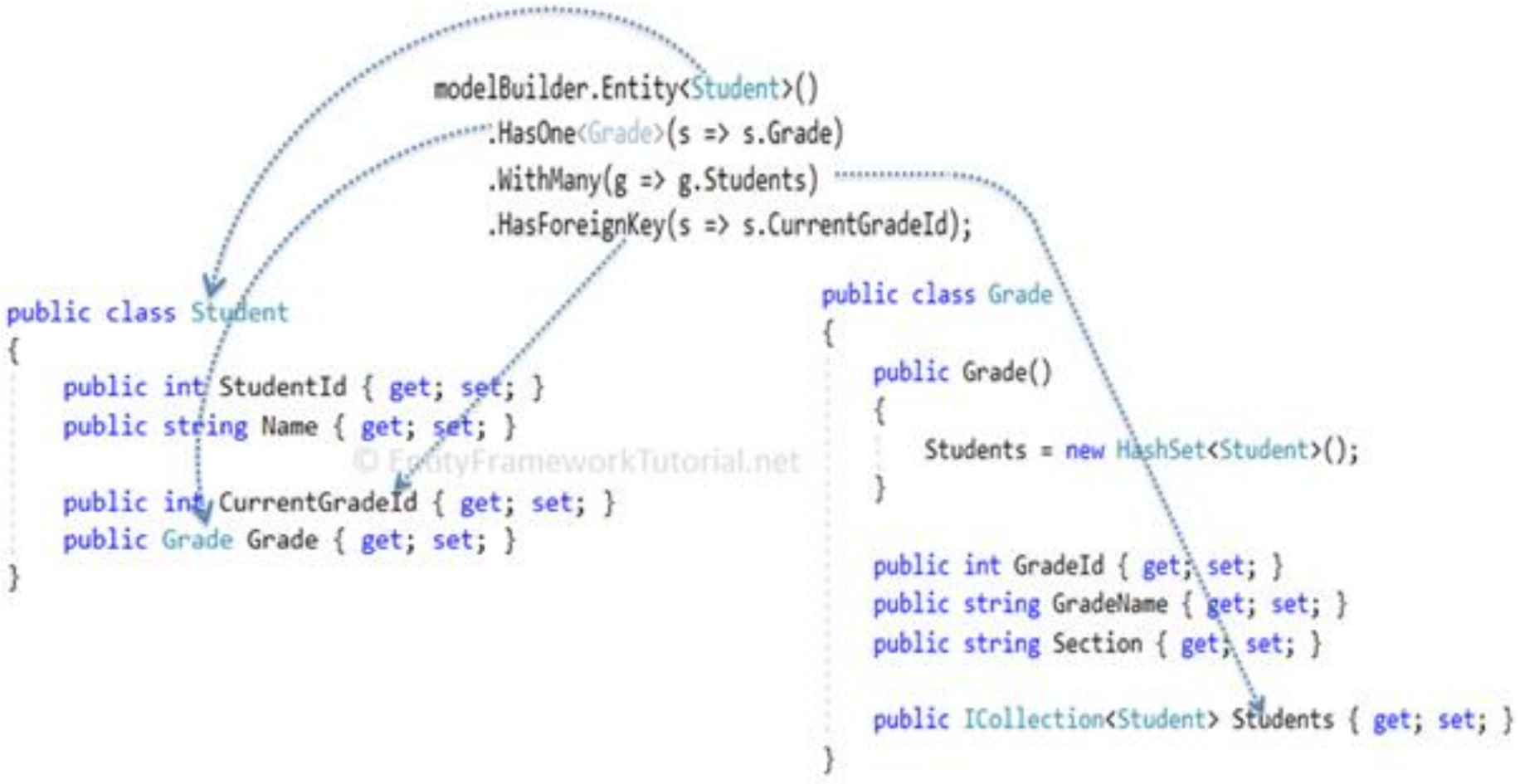
Many To Many

- the default conventions for many-to-many relationships creates a joining table with the default naming conventions.
- Use Fluent API to customize a joining table name and column names

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Student>()
            .HasMany<Course>(s => s.Courses)
            .WithMany(c => c.Students)
            .Map(cs =>
                  {
                     cs.MapLeftKey("StudentRefId");
                     cs.MapRightKey("CourseRefId");
                     cs.ToTable("StudentCourse");
                  });

}
```

http://www.entityframeworktutorial.net/code-first/configure-many-to-many-relationship-in-code-first.aspx

Implementing Foreign Key



Conventions

- By convention, a property named Id or <type name>Id will be configured as the key of an entity.

```
class Car
{
    public string Id { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Excluding an entity from mapping

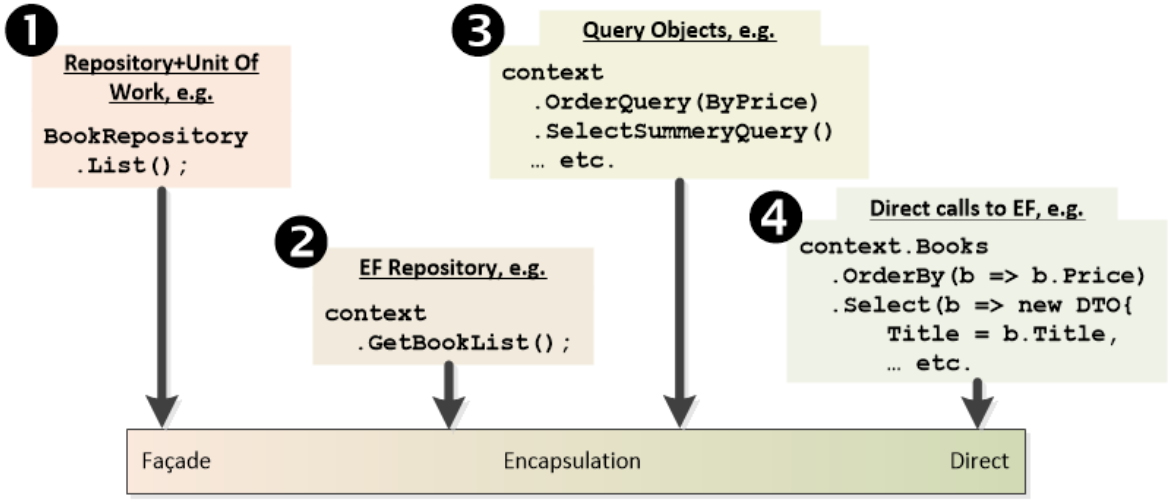

```
public class SampleContext : DbContext
{
    public DbSet<Contact> Contacts { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<AuditLog>();
    }
}
```

Excluding a property from mapping


```
public class SampleContext : DbContext
{
    public DbSet<Contact> Contacts { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Contact>().Ignore(c => c.FullName);
    }
}
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName => $"{FirstName} {LastName}";
    public string Email { get; set; }
}
```

Database access patterns

- There are a number of different ways we can form your EF database access inside an application, with different levels of hiding the EF access code from the rest of the application. Four different data access patterns.



Database access patterns

- **Repository + Unit of Work (Repo+UOW)**. This hides all the EF Core behind code that provides a different interface to EF. The idea being you could replace EF with another database access framework with no change to the methods that call the Repo+UOW.
- **EF repository**. This is a repository patterns that doesn't try and hide the EF code like the Repo+UOW pattern does. EF repositories assume that you as developer know the rules of EF, such as using tracked entities and calling SaveChanges for updates, and you will abide by them.
- **Query Object**. Query objects encapsulate the code for a database query, that is a database read. They hold the whole code for a query or for complex queries it might hold part of a query. Query objects are normally built as extension methods with IQueryable<T> inputs and outputs so that they can be chained together to build more complex queries.
- **Direct calls to EF**. This represents the case where you simply place the EF code you need in the method that needs it. For instance, all the EF code to build a list of books would be in the ASP.NET action method that shows that list.

Defining a DbContext derived class

- The recommended way to work with context is to define a class that derives from DbContext and exposes DbSet properties that represent collections of the specified entities in the context. If you are working with the EF Designer, the context will be generated for you. If you are working with Code First, you will typically write the context yourself.

- C#
- public class ProductContext : DbContext
- {
- public DbSet<Category> Categories { get; set; }
- public DbSet<Product> Products { get; set; }
- }

DB Context

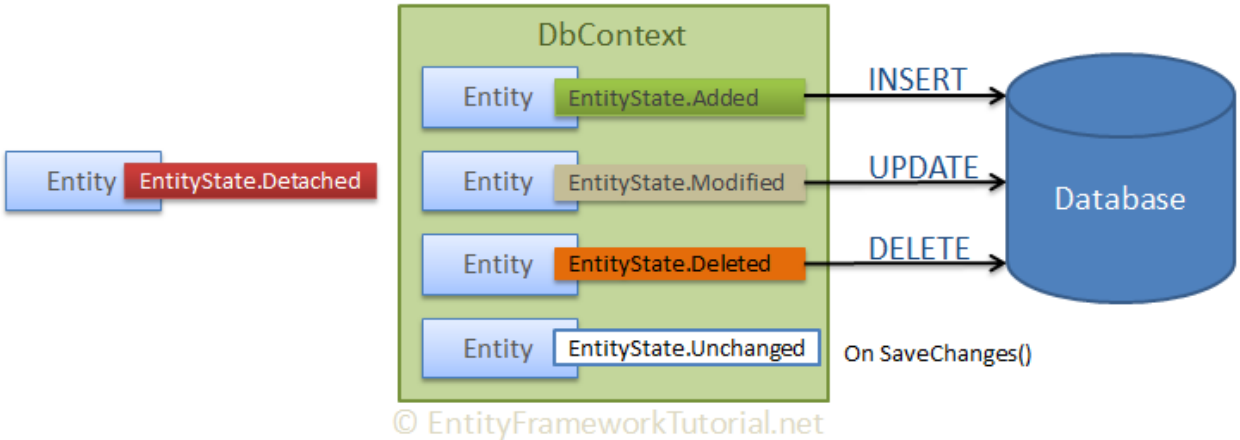
- When working with Web applications, use a context instance per request.
- By default, the context manages connections to the database. The context opens and closes connections as needed. For example, the context opens a connection to execute a query, and then closes the connection when all the result sets have been processed.

Service Layer

- Service Layer acts as an adapter
- Service Layer becomes a crucial layer, as it can be the layer that understands both sides and can transform the data between the two worlds

EntityState in Entity Framework

- EF API maintains the state of each entity during an its lifetime. Each entity has a state based on the operation performed on it via the context class.
  - Added
  - Modified
  - Deleted
  - Unchanged
  - Detached



https://www.entityframeworktutorial.net/basics/entity-states.aspx

Entity states and SaveChanges

- An entity can be in one of five states as defined by the EntityState enumeration. These states are:
  - Added: the entity is being tracked by the context but does not yet exist in the database
  - Unchanged: the entity is being tracked by the context and exists in the database, and its property values have not changed from the values in the database
  - Modified: the entity is being tracked by the context and exists in the database, and some or all of its property values have been modified
  - Deleted: the entity is being tracked by the context and exists in the database, but has been marked for deletion from the database the next time SaveChanges is called
  - Detached: the entity is not being tracked by the context

Adding a new entity to the context

- A new entity can be added to the context by calling the Add method on DbSet. This puts the entity into the Added state, meaning that it will be inserted into the database the next time that SaveChanges is called. For example:

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

Not Changed


```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);

    // Do some more work...

    context.SaveChanges();
}
```

Not Changed (Another Way)


```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

Insert or update pattern

- A common pattern for some applications is to either Add an entity as new (resulting in a database insert) or Attach an entity as existing and mark it as modified (resulting in a database update) depending on the value of the primary key. For example, when using database generated integer primary keys it is common to treat an entity with a zero key as new and an entity with a non-zero key as existing. This pattern can be achieved by setting the entity state based on a check of the primary key value. For example:

```
public void InsertOrUpdate(Blog blog)
{
    using (var context = new BloggingContext())
    {
        context.Entry(blog).State = blog.BlogId == 0 ?
                                    EntityState.Added :
                                    EntityState.Modified;

        context.SaveChanges();
    }
}
```

Updating

- Attaching an existing but modified entity to the context
- If you have an entity that you know already exists in the database but to which changes may have been made then you can tell the context to attach the entity and set its state to Modified. For example:

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Modified;

    // Do some more work...

    context.SaveChanges();
}
```

Language Integrated Query (LINQ)

- Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language.
- Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support.
- Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

LINQ : Simple queries

- One of the things that make Entity Framework Core has first-class support for LINQ. This makes simple queries remarkably easy to execute. The DbSet in the context implements an interface called IQueryable, which allows you to chain function calls together to filter, project, sort, or any number of other things. Let's try a few quick examples:

- Get all the ants named Bob:

```
context.Ants.Where(x => x.Name == "Bob")
```

- You can also do compound queries where you provide multiple constraints. Here we want all the ants named Bob who are older than 30 days.

```
context.Ants.Where(x => x.Name == "Bob" && x.AgeInDays > 30)
```

from clause

- from clause comes first in order to introduce the data source

  var queryAllCustomers = **from** cust in customers

  select cust;

Filtering

- Where: filter causes the query to return only those elements for which the expression is true.

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Ordering

- Often it is convenient to sort the returned data. The orderby clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example, the following query can be extended to sort the results based on the Name property. Because Name is a string, the default comparer performs an alphabetical sort from A to Z.

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Grouping

- The group clause enables you to group your results based on a key that you specify. For example you could specify that the results should be grouped by the City so that all customers from London or Paris are in individual groups. In this case, cust.City is the key.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
 var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

 // customerGroup is an IGrouping<string, Customer>
 foreach (var customerGroup in queryCustomersByCity)
 {
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
       Console.WriteLine("    {0}", customer.Name);
    }
 }
```

Joining

- Join operations create associations between sequences that are not explicitly modeled in the data sources.
  - var innerJoinQuery =
  - from cust in customers
  - join dist in distributors on cust.City equals dist.City
  - select new { CustomerName = cust.Name, DistributorName = dist.Name };

Insert Update Delete

- There is no direct Linq Expression for Insert, Update or Delete

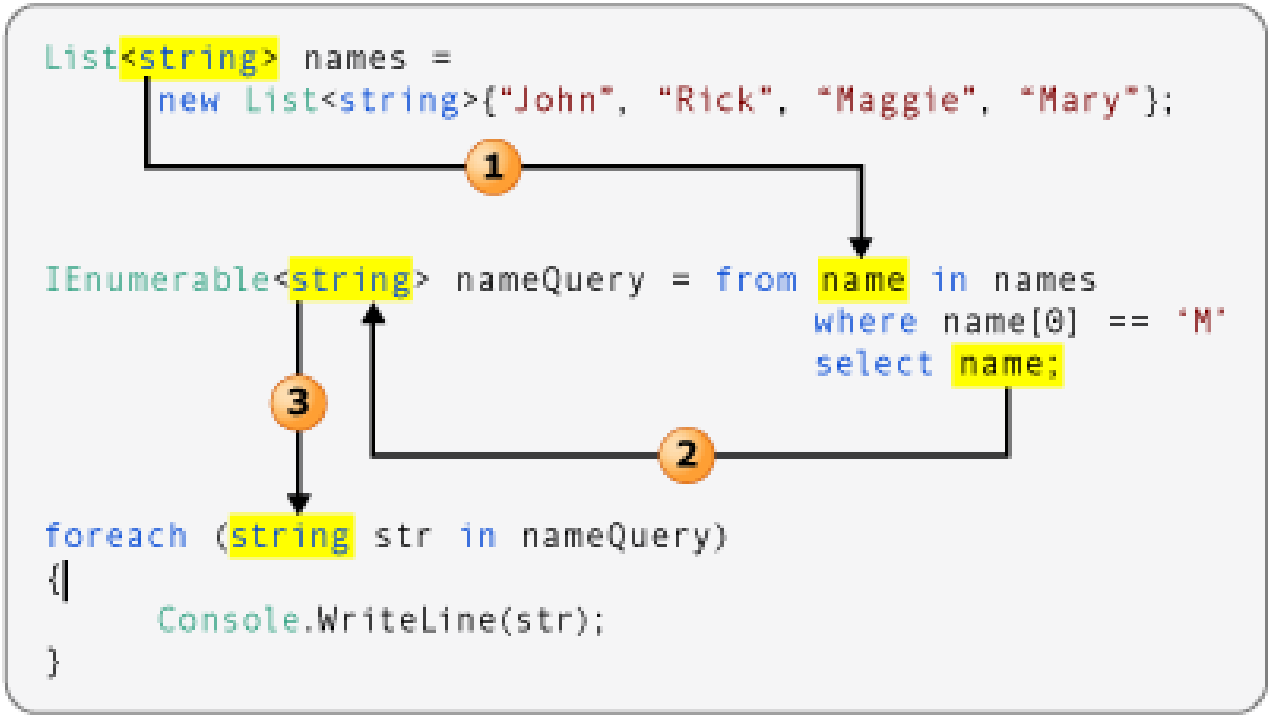
Inserting


```
Order ord = new Order
{
    OrderID = 12000,
    ShipCity = "Seattle",
   // …
};

// Add the new object to the Orders collection.
db.Orders.InsertOnSubmit(ord);

// Submit the change to the database.
try
{
    db.SubmitChanges();
}
catch (Exception e)
{
    Console.WriteLine(e);
    // Make some adjustments  and Try again.
    db.SubmitChanges();
}
```

Linq and list


```
List<string> words = new List<string>() { "an",
"apple", "a", "day" };
  var query = from word in words
              select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);
```

Linq Type conversation



Standard Query Operator Extension Methods

The following example shows a simple query expression and the semantically equivalent query written as a method-based query.

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};


        //Query syntax:
        IEnumerable<int> numQuery1 = from num in numbers      where num % 2 == 0  orderby
num    select num;
        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);


        foreach (int i in numQuery1)         {             Console.Write(i + " ");         }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)         {             Console.Write(i + " ");         }
     }
}
```

Query Building


```
private async static Task<IEnumerable<Ant>> Search(Context context, int? age, string name,
string game)
{
    var query = context.Ants as IQueryable<Ant>;

    if (age.HasValue)
        query = query.Where(x => x.AgeInDays == age);

    if (!String.IsNullOrEmpty(name))
        query = query.Where(x => x.Name == name);

    if (!String.IsNullOrEmpty(game))
        query = query.Where(x => x.FavoriteAntGame == game);

    return await query.ToListAsync();
}
```

https://stackify.com/entity-framework-core-tutorial/

Updating data

- Changing data retrieved from the context is really easy, thanks to the fact that all the entities used are tracked. If we wanted to load an Ant and then change the name, it is as simple as:

```
var ant = await context.Ants.FirstOrDefaultAsync(x => x.Id == id);
ant.Name = "Bob";
await context.SaveChangesAsync();
```

Performance tip: Profiling

- You can easily build queries in Entity Framework Core that seem reasonable, but end up being very costly when transformed to SQL. In order to watch the queries you're making Prefix like tools.
- You can use Prefix to spot common issues like n+1 problems or slow queries. Your users will be grateful that you've taken the time to install and run some profiling.

Seeing SQL Statemens

- public static readonly Microsoft.Extensions.Logging.LoggerFactory _myLoggerFactory = new Microsoft.Extensions.Logging.LoggerFactory(new[] { new Microsoft.Extensions.Logging.Debug.DebugLoggerProvider() });

- protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
- {
- optionsBuilder.UseLoggerFactory(_myLoggerFactory);
- }

Complex queries

- LINQ is a really nice domain-specific language for manipulating and querying objects, however, sometimes you have to relax the abstraction and get back to the relational model. If you find yourself building crazy queries that bend your mind with the complexity of the LINQ, then take a step back: you can drop to SQL to perform your queries.

- This is done using the FromSql for queries:

- context.Ants.FromSql<Ant>("select * from ants");
- or using the ExecuteSqlCommandAsync:

- context.Database.ExecuteSqlCommandAsync("delete from ants where name='Bob’");

- Unfortunately, you must use a real entity for your SQL queries and you cannot use a projection.
- This was functionality that was available in EF.

Using GUIDs

- protected override void OnModelCreating(ModelBuilder modelBuilder) { modelBuilder .Entity<Blog>() .Property(b => b.BlogId) .ForSqlServerUseSequenceHiLo(); base.OnModelCreating(modelBuilder); }

- Guis as a key
- public Guid BlogId { get; set; }

Implementing Inheritence

- modelBuilder .Entity<Post Content >()
  .HasBaseType<BlogContent>();

Validation

- FluentValidation can be integrated with ASP.NET Core. Once enabled, MVC will use FluentValidation to validate objects that are passed in to controller actions by the model binding infrastructure.

- To enable MVC integration, you'll need to add a reference to the FluentValidation.AspNetCore assembly by installing the appropriate NuGet package:

```
public void ConfigureServices(IServiceCollection services) {
    services.AddMvc(setup => {
      //...mvc setup...
    }).AddFluentValidation();

    services.AddTransient<IValidator<Person>, PersonValidator>();
    //etc
}
```

Buid in Validators


```
public class EmployeeInputModel
  {
     public int Id { get; set; }

     [Required]
     public string EmployeeNo { get; set; }

     [StringLength(10)]
     [MinLength(3)]
     public string Surname { get; set; }

     [EmailAddress]
     public string Email { get; set; }

     [Url]
     public string BlogUrl { get; set; }

     [DataType(DataType.Date)]
     [Display(Name = "Date of Birth")]
     public DateTime BirthDate { get; set; }

    [Range(0, 10000.00)]
     public decimal Salary { get; set; }
}
```

Complex Validation: Add validation Code to Enties


```
public class Person {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public int Age { get; set; }
}

public class PersonValidator : AbstractValidator<Person> {
        public PersonValidator() {
                RuleFor(x => x.Id).NotNull();
                RuleFor(x => x.Name).Length(0, 10);
                RuleFor(x => x.Email).EmailAddress();
                RuleFor(x => x.Age).InclusiveBetween(18, 60);
        }
}
```

Error Handling and Validation


```
private static IStatusGeneric ExecuteValidation(this DbContext context)
{
    var status = new StatusGenericHandler();
    foreach (var entry in
        context.ChangeTracker.Entries()
            .Where(e =>
                (e.State == EntityState.Added) ||
                (e.State == EntityState.Modified)))
    {
        var entity = entry.Entity;
        var valProvider = new ValidationDbContextServiceProvider(context);
        var valContext = new ValidationContext(entity, valProvider, null);
        var entityErrors = new List<ValidationResult>();
        if (!Validator.TryValidateObject(
            entity, valContext, entityErrors, true))
        {
            status.AddValidationResults(entityErrors);
        }
    }

    return status;
}
```

SaveChangesWithValidation


```
public static IStatusGeneric SaveChangesWithValidation (this DbContext context, IGenericServicesConfig config)
{
    var status = context.ExecuteValidation();
    if (!status.IsValid) return status;

    context.ChangeTracker.AutoDetectChangesEnabled = false;
    try
    {
        context.SaveChanges();
    }
    catch (Exception e)
    {
        var exStatus = config?.SaveChangesExceptionHandler(e, context);
        if (exStatus == null) throw;       //error wasn't handled, so rethrow
        status.CombineStatuses(exStatus);
    }
    finally
    {
        context.ChangeTracker.AutoDetectChangesEnabled = true;
    }
     return status;
}
```