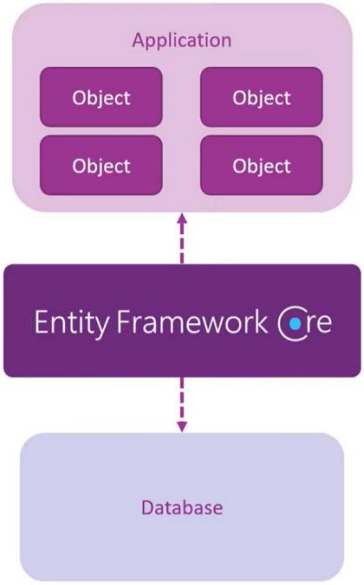
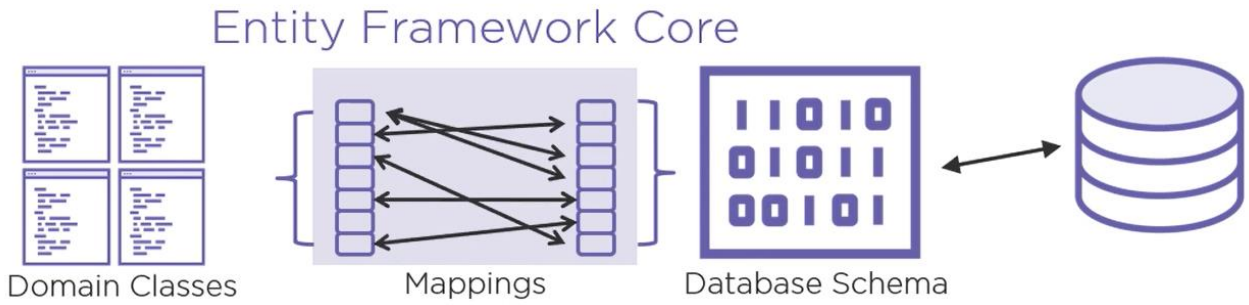
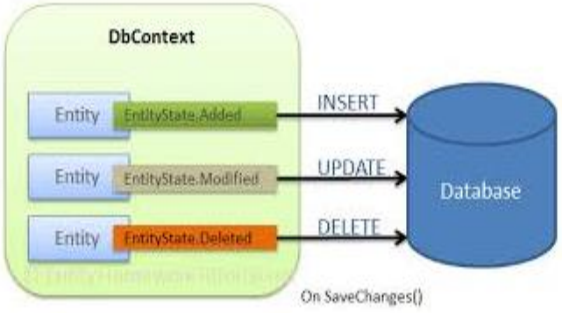
Entity Framework Core

Entity Framework Core (EF Core) is an Object-Relational Mapper (ORM) for .NET that allows developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write. EF Core is a lightweight, extensible, open-source, and cross-platform version of the Entity Framework, which is the ORM introduced with the .NET Framework.







1. Basic Concepts of EF Core

- **ORM (Object-Relational Mapping)**: EF Core serves as an ORM that allows .NET developers to interact with a database using C# objects and LINQ queries rather than SQL queries.
- **DbContext**: This is the primary class that interacts with the database. It represents a session with the database and provides APIs to perform CRUD (Create, Read, Update, Delete) operations.
- **Entities**: Entities are C# classes that represent tables in the database. Each instance of the entity class represents a row in the corresponding table.
- **DbSet**: Each DbSet property on the DbContext represents a table in the database. DbSet provides the methods to query and work with data from the database.

---

2. Setup and Configuration

- **Installing EF Core**: To use EF Core, you need to install the necessary packages:
  - Microsoft.EntityFrameworkCore
  - Microsoft.EntityFrameworkCore. tools
  - Microsoft.EntityFrameworkCore.SqlServer
  -
- **DbContext Class**: You create a custom **DbContext** class that inherits from **Microsoft.EntityFrameworkCore.DbContext.** In this class, you define the **DbSet** properties for each entity you want to work with.
- **Connection String**: EF Core needs a connection string to connect to the database.

---

3. Creating Models and Migrations

- **Model Creation**: Models are plain C# classes where each property represents a column in a table. Data annotations or Fluent API can be used to configure entity relationships, constraints, and other metadata.

  Data annotations are part of the System.ComponentModel.DataAnnotations,they help define constraints, relationships, and other metadata directly on properties and classes in your model.

  Here's a list of commonly used data annotations in EF Core:

  1. [Key] Specifies that the property is the primary key of the entity.

```
public class Product
{
   [Key]
   public int ProductId { get; set; }
}
```

2. [Required] Marks a property as required, meaning it cannot be null.

```
public class Product
{
   [Required]
   public string Name { get; set; }
}
```

3. [MaxLength] and [MinLength] Specifies the maximum or minimum length of a string. **[MaxLength]** can also work on array properties like byte[].

```
public class Product
{
   [MaxLength(100)]
   public string Name { get; set; }
}
```

4. [StringLength] Specifies both the maximum and optional minimum length of a string.

```
public class Product
{
   [StringLength(100, MinimumLength = 5)]
   public string Name { get; set; }
}
```

5. [Column] Configures the column name and data type of a property in the database.

```
public class Product
{
   [Column("ProductName", TypeName = "nvarchar(200)")]
   public string Name { get; set; }
}
```

6. [Table] Configures the table name and optional schema of an entity.

```
[Table("Products", Schema = "store")]
public class Product {
   public int ProductId { get; set; }
   public string Name { get; set; }
}
```

7. [ForeignKey] Specifies the foreign key property for a relationship, Use it with navigation properties to define the linking foreign key property.

```
public class Order
{
    public int OrderId { get; set; }

    [ForeignKey("Customer")]
    public int CustomerId { get; set; }

    public Customer Customer { get; set; }
}
```

8. [InverseProperty] Specifies the inverse (opposite) side of a relationship, Useful when you have multiple relationships between two entities.

```
public class Customer
{
    public int CustomerId { get; set; }

    [InverseProperty("Customer")]
    public List<Order> Orders { get; set; }
}

public class Order
{
    public int OrderId { get; set; }
    public int CustomerId { get; set; }

    [ForeignKey("CustomerId")]
    [InverseProperty("Orders")]
    public Customer Customer { get; set; }
}
```

9. [NotMapped] Specifies that a property should not be mapped to a database column. Useful for calculated properties or properties that should only exist in memory.

```
public class Product
{
    public int ProductId { get; set; }

    [NotMapped]
    public string TemporaryValue { get; set; }
}
```

10. [Index] Specifies an index on a column or set of columns.

```
[Index(nameof(Name), IsUnique = true)]
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
}
```

11. [EnumDataType] Specifies the data type of an enumeration.

```
public enum ProductStatus
{
    Available,
    OutOfStock
}

public class Product
{
    public int ProductId { get; set; }

    [EnumDataType(typeof(ProductStatus))]
    public ProductStatus Status { get; set; }
}
```

12. [Range] Specifies the minimum and maximum value for numeric properties. Primarily useful for validation, but it can also be used to guide model constraints.

```
public class Product
{
    [Range(1, 100)]
    public int Stock { get; set; }
}
```

13. [RegularExpression] Defines a regular expression that a string property must match.

```
public class Product
{
    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    public string Name { get; set; }
}
```

14. Identity, Default and unique Constraints **[DatabaseGenerated]**

Here's an example showing a model with an identity column, a default value, and a unique constraint:

```
public class Product
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int ProductId { get; set; } // Identity column
    public string Name { get; set; }

    [Index(IsUnique = true)]
    public string SKU { get; set; } // Unique constraint

    public decimal Price { get; set; }

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public DateTime CreatedAt { get; set; } = DateTime.Now; // Default value
}
```

- **Migrations**: EF Core supports migrations, which are a way to create and update the database schema over time. Migrations track changes to your model and apply those changes to the database.

  The commands used this aim are the following:

  - add-migration [MigrationName]
  - update-database
  - remove-migration

---

4. Handling Relationships

1. One-to-Many Relationship

In a one-to-many relationship, a single record in one table is associated with multiple records in another table. For example, a Category can have multiple Product items.

Setting Up a One-to-Many Relationship

```
public class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }

    public ICollection<Product> Products { get; set; } // Navigation property
}

public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    [ForeignKey("Category")]
    public int CategoryId { get; set; } // Foreign key property
    public Category Category { get; set; } // Navigation property
}
```

In this example:

- The Category entity has a navigation property Products, which is a collection of Product entities.
- The Product entity has a foreign key property CategoryId and a navigation property Category.

Configuring the One-to-Many Relationship with Fluent API (Alternative way)

The relationship can also be configured using the Fluent API in OnModelCreating:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>()
        .HasOne(p => p.Category)
        .WithMany(c => c.Products)
        .HasForeignKey(p => p.CategoryId);
}
```

---

2. One-to-One Relationship

A one-to-one relationship is when one record in a table is associated with one and only one record in another table. For example, a User entity might be associated with only one UserProfile.

Setting Up a One-to-One Relationship

```
public class User
{
    public int UserId { get; set; }
    public string Username { get; set; }
    public UserProfile Profile { get; set; } // Navigation property
}
public class UserProfile
{
    public int UserProfileId { get; set; }
    public string Bio { get; set; }
    [ForeignKey("User")]
    public int UserId { get; set; } // Foreign key property
    public User User { get; set; } // Navigation property
}
```

In this example:

- User has a navigation property Profile for the UserProfile.
- UserProfile has a navigation property User and a foreign key UserId

3. Many-to-Many Relationship

A many-to-many relationship occurs when multiple records in one table are associated with multiple records in another table. For example, **Student** and **Course** have a many-to-many relationship because a student can enroll in multiple courses, and each course can have multiple students.

In EF you can configure a many-to-many relationship without defining an explicit join entity. EF Core automatically creates an implicit join table.

Setting Up a Many-to-Many Relationship

```
public class Student
{
   public int StudentId { get; set; }
   public string Name { get; set; }
   public ICollection<Course> Courses { get; set; } // Navigation property
}

public class Course
{
   public int CourseId { get; set; }
   public string Title { get; set; }
   public ICollection<Student> Students { get; set; } // Navigation property
}
```

Alternative Way

```
public class Employee
 {

    [Key]
    public int Ssn { get; set; }
    [Required]
    public string Fname { get; set; }
    public virtual ICollection<WorksOn> WorksOnProjects { get; set; }
 }
```

```
public class Project
{
    [Key]
    public int Pnumber { get; set; }
    [Required]
    public string Pname { get; set; }
    public string Plocation { get; set; }
    public virtual ICollection<WorksOn> WorksOnEmployees { get; set; }
}

[PrimaryKey(nameof(Essn), nameof(Pno))]
public class WorksOn
{
    [ForeignKey("Employee")]
    public int Essn { get; set; }
    public virtual Employee Employee { get; set; }
    [ForeignKey("Project")]
    public int Pno { get; set; }
    public virtual Project Project { get; set; }
    public decimal Hours { get; set; }

}
```

---

4. Inverse Property

The **[InverseProperty]** attribute in Entity Framework Core is particularly useful when you have multiple relationships between the same two entities. It helps EF Core understand which navigation property corresponds to which foreign key when there's ambiguity.

Scenario

1. Each **Department** can have multiple **Employees** working in it.
2. Each **Department** has one **Manager**, who is also an **Employee**.

```
public class Department
{
    public int DepartmentId { get; set; }
    public string DepartmentName { get; set; }

    // Collection of employees who work in this department
    [InverseProperty("WorkingDepartment")]
    public ICollection<Employee> Employees { get; set; } = new List<Employee>();

    // Manager of the department
    [ForeignKey("ManagerId")]
    public int? ManagerId { get; set; } // Nullable to allow departments without a manager
    [InverseProperty("ManagedDepartment")]
    public Employee Manager { get; set; }
}

public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }

    // Foreign key for WorkingDepartment
    [ForeignKey("DepartmentId")]
    public int DepartmentId { get; set; }
    [InverseProperty("Employees")]
    public Department WorkingDepartment { get; set; }

    // Department that this employee manages
    [InverseProperty("Manager")]
    public Department ManagedDepartment { get; set; }
}
```

5. Self-relationship

```
public class Employee
{

    [Key]
    public int Ssn { get; set; }

    [Required]
    public string Fname { get; set; }
    public string Minit { get; set; }
    [Required]
    public string Lname { get; set; }
    public DateTime Bdate { get; set; }
    public string Address { get; set; }
    public string Sex { get; set; }
    public decimal Salary { get; set; }

    [ForeignKey("Supervisor")]
    public int? SuperSsn { get; set; }
    public virtual Employee Supervisor { get; set; }

    [InverseProperty("Supervisor")]
    public virtual ICollection<Employee> Supervisees { get; set; }
}
```