EF

Objective

- Entity Framework
  - Entity Framework Core Features
  - Entity Framework Versions
- Approaches
  - Code First Approach
  - Data First Approach
- Database Providers
- EF Core Installation
- Install EF Core Tools
- Configuring EF – SQL Server
- Scaffold - DbContext Command
- DbContext
- CRUD Operation
- EF Core Conventions
- Schema
- Table
- Column
- Relationship
- One-To-Many
- One-to-One

TIME FOR CASE STUDY

Case Study (continued)

- As a Shopon customer, I want to know all bank offers

Thought

- Each bank will have different offers.
- As we have to fetch the offer details based on the bank, we can use EF to achieve this.

Knowledge Byte

Solution

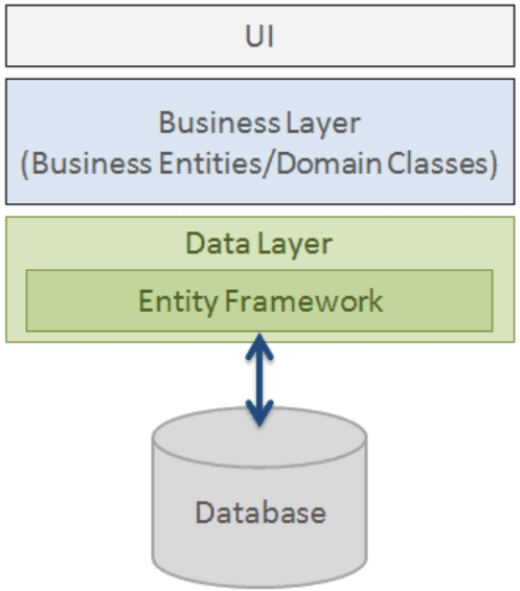
Knowledge Byte

- EF Core
- Installation
- EF Core Features
- CRUD Operation
- EF Version
- EF Core Conventions
- Approaches
- Relationship
- Providers

Entity Framework

- Entity Framework is an **object-relational mapper** (O/RM) that enables .NET 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.

With the Entity Framework, developers can work at a higher level of abstraction when they deal with data, and can create and maintain data-oriented applications with less code compared with traditional applications.



Entity Framework Core Features

- Cross-platform
- Modeling
- Querying
- Change Tracking
- Saving
- Concurrency
- Transactions
- Caching
- Built-in Conventions
- Configurations
- Migrations

Entity Framework Versions

EF 6	EF Core
✓ First released in 2008 with .NET Framework 3.5 SP1	✓ First released in June 2016 with .NET Core 1.0
✓ Stable and feature rich	✓ New and evolving
✓ Windows only	✓ Windows, Linux, OSX
✓ Works on .NET Framework 3.5+	✓ Works on .NET Framework 4.5+ and .NET Core
✓ Open-source	✓ Open-source

EF Core on GitHub: https://github.com/aspnet/EntityFrameworkCore
EF Core Roadmap: docs.microsoft.com/en-us/ef/core/what-is-new/roadmap
Track EF Core's issues
at https://github.com/aspnet/EntityFrameworkCore/issues
EF Core Official Documentation: https://docs.microsoft.com/ef/core

Approaches

- EF Core supports

Data First

Code First

Code First Approach

- In this approach, we first create our domain class
- We also create Context class which is derived from EF **DBContext** class
- Based on the Domain class and Context class, EF Core creates Database and relevant tables
- EF uses its conventions to create Database and database tables

Data First Approach

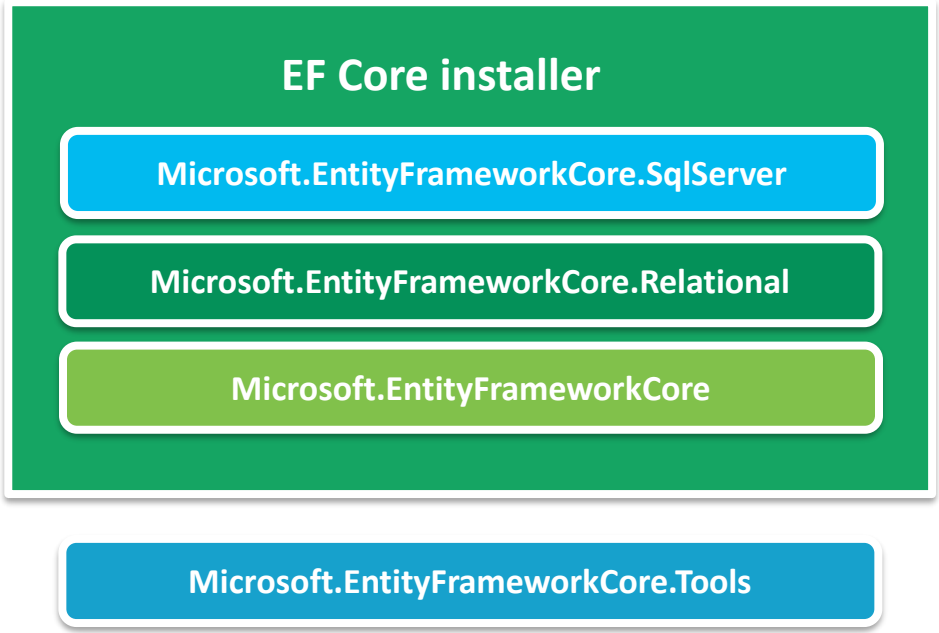
- EF Core API creates the domain and context classes based on your existing database using EF Core commands. This has limited support in EF Core as it does not support visual designer or wizard.

Database Providers

- EF Core supports many relational and non-relational databases by using plug-in libraries called Database Providers.
- These database providers are available as NuGet packages.
- Database providers sits between EF Core and Database. It will have functionality specific to the database it supports.
- Functionalities that is common across database is present in EFCore Component.
- Refer following link to know more on providers: https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

EF Core Installation

- We have to install



Microsoft.EntityFrameworkCore.SqlServer

- The **Microsoft.EntityFrameworkCore.SqlServer** contains functionality that is specific to Microsoft's SqlServer.
- This package is dependent upon **Microsoft.EntityFrameworkCore.Relational**
- As other packages are dependent upon this package, if we install this package other two packages will be installed automatically.

Install
Microsoft.EntityFrameworkCore.SqlServer

Configuring EF – SQL Server

Microsoft.EntityFrameworkCore.Relational

- The **Microsoft.EntityFrameworkCore.Relational** contains functionality related to all relational database like SqlServer, Oracle, MySql, etc.
- This has dependency on **Microsoft.EntityFrameworkCore**

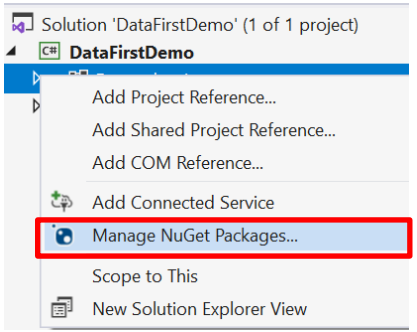
Microsoft.EntityFrameworkCore

- The **Microsoft.EntityFrameworkCore** contains core functionality related to EntiryFrameworkCore, that is common to all databases.

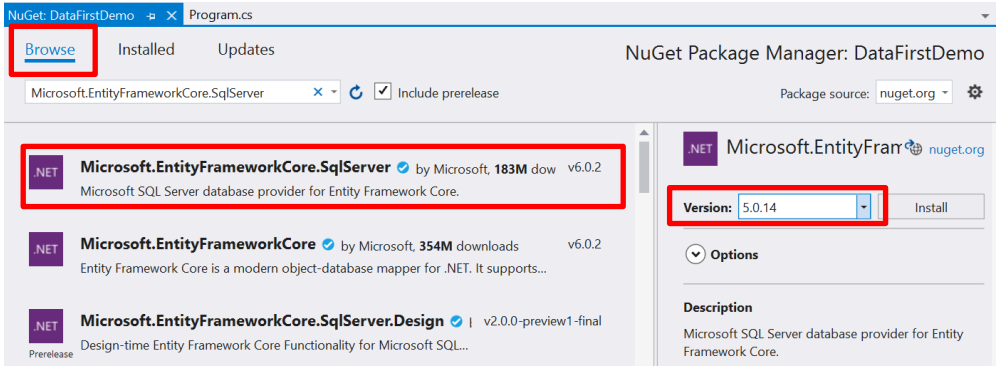
Install - Microsoft.EntityFrameworkCore.SqlServer

- To **Install Microsoft.EntityFrameworkCore.SqlServer** navigate to NuGet package manager and add new package

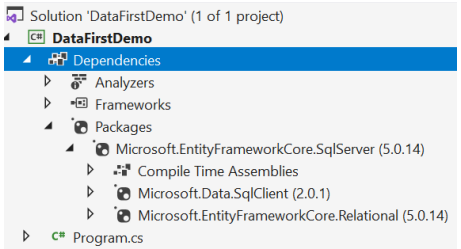
NuGet Packages



NuGet Package Manager

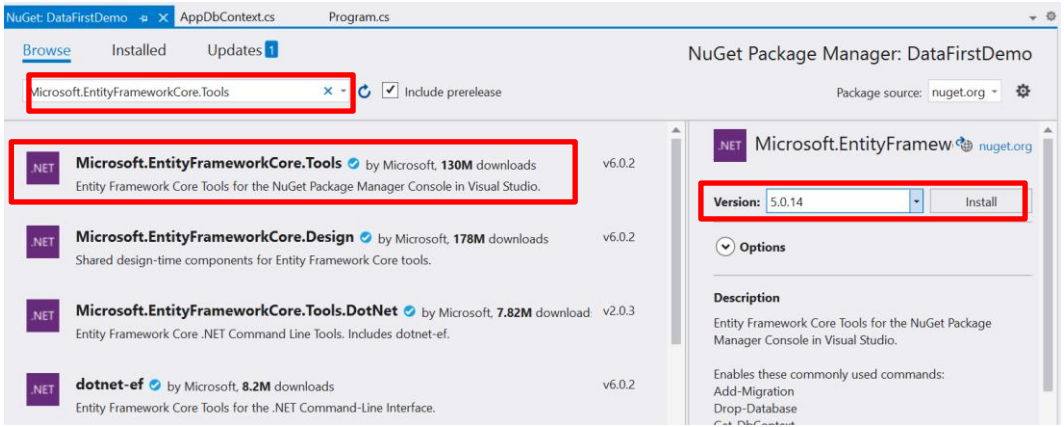


After installation



Install EF Core Tools

- EF tools is used to execute EF Core commands.
- These make it easier to perform several EF Core-related tasks in your project at design time, such as migrations, scaffolding, etc.
- EF Tools are available as NuGet packages.



Configuring EF – SQL Server

- To work with different databases(relational and non-relational), EF Core support wide range of providers.
- Following link list all the providers which is supported by EF Core https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

Database - First approach

- Creating entity & context classes for an existing database is called Database-First approach.
- Using the Scaffold-DbContext command we can create entity and context classes(by deriving DbContext) based on the schema of the existing database.

Scaffold - DbContext Command

- Use Scaffold-DbContext to create a model based on your existing database.
- The following parameters can be specified with Scaffold-DbContext in Package Manager Console:

```
Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>] [-Tables>] [-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]
```

- In Visual Studio, select menu Tools -> NuGet Package Manger -> Package Manger Console and run the following command:

```
Scaffold-DbContext "Data Source=.;Initial Catalog=db_Shopon;Integrated Security=True" Microsoft.EntityFrameworkCore.SqlServer -table Product, Category, Company -OutputDir Models
```

DbContext

- The **DbContext** class is an integral part of Entity Framework.
- An instance of DbContext represents a session with the database which can be used to query and save instances of our entities to a database.
- DbContext is a combination of the Unit Of Work and Repository patterns.

DbContext

- DbContext in EF Core allows us to perform following tasks:
  - Manage database connection
  - Configure model & relationship
  - Querying database
  - Saving data to the database
  - Configure change tracking
  - Caching
  - Transaction management

DbContext - Methods

Method	Usage
Add	Adds a new entity to `DbContext` with Added state and starts tracking it. This new entity data will be inserted into the database when SaveChanges() is called.
AddAsync	Asynchronous method for adding a new entity to `DbContext` with Added state and starts tracking it. This new entity data will be inserted into the database when SaveChangesAsync() is called.
AddRange	Adds a collection of new entities to `DbContext` with Added state and starts tracking it. This new entity data will be inserted into the database when SaveChanges() is called.
AddRangeAsync	Asynchronous method for adding a collection of new entities which will be saved on SaveChangesAsync().
Attach	Attaches a new or existing entity to `DbContext` with Unchanged state and starts tracking it.
AttachRange	Attaches a collection of new or existing entities to `DbContext` with Unchanged state and starts tracking it.

DbContext - Methods

Entry	Gets an `EntityEntry` for the given entity. The entry provides access to change tracking information and operations for the entity.
Find	Finds an entity with the given primary key values.
FindAsync	Asynchronous method for finding an entity with the given primary key values.
Remove	Sets Deleted state to the specified entity which will delete the data when SaveChanges() is called.
RemoveRange	Sets Deleted state to a collection of entities which will delete the data in a single DB round trip when SaveChanges() is called.
SaveChanges	Execute INSERT, UPDATE or DELETE command to the database for the entities with Added, Modified or Deleted state.
SaveChangesAsync	Asynchronous method of SaveChanges()
Set	Creates a `DbSet<TEntity>` that can be used to query and save instances of TEntity.

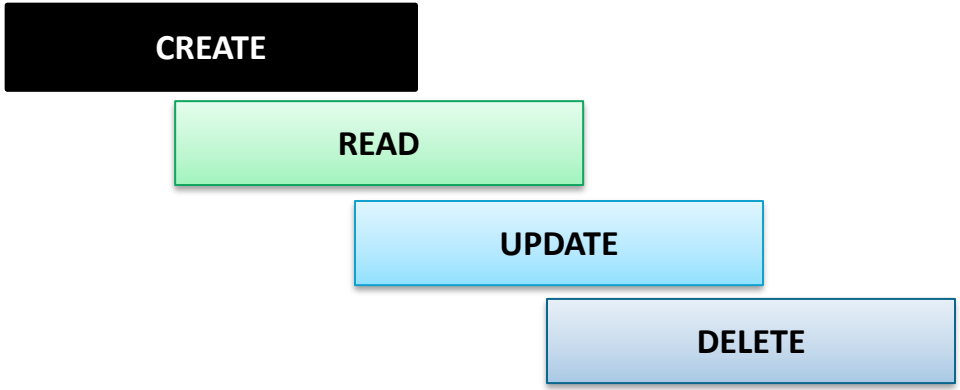
DbContext - Methods

Update	Attaches disconnected entity with Modified state and start tracking it. The data will be saved when SaveChagnes() is called.
UpdateRange	Attaches a collection of disconnected entities with Modified state and start tracking it. The data will be saved when SaveChagnes() is called.
OnConfiguring	Override this method to configure the database (and other options) to be used for this context. This method is called for each instance of the context that is created.
OnModelCreating	Override this method to further configure the model that was discovered by convention from the entity types exposed in `DbSet<TEntity>` properties on your derived context.

DbContext - Properties

Method	Usage
ChangeTracker	Provides access to information and operations for entity instances this context is tracking.
Database	Provides access to database related information and operations for this context.
Model	Returns the metadata about the shape of entities, the relationships between them, and how they map to the database.

CRUD Operation



CREATE

ProductRepo.cs

```
public class ProductRepo
{
    private db_ShoponContext context = new db_ShoponContext();

    /// <summary>
    /// Method to add product
    /// </summary>
    /// <param name="product"></param>
    1 reference
    public void AddProduct(Product product)
    {
        try
        {
            this.context.Products.Add(product);
            this.context.SaveChanges();
        }
        catch (Exception e)
        {
            Console.WriteLine($"Error while inserting product {e.Message}");
        }
    }
}
```

Program.cs

```
private static void AddProduct()
{
    Product product = new Product()
    {
        Availablestatus = "Y",
        Categoryid = 2001,
        ImageUrl = "images/oneplus/nord2_5g.jpg",
        Companyid = 1010,
        Price = 49999,
        Productname = "Nord 2 5G(Gray)"
    };
    productRepo.AddProduct(product);
}
```

The DbSet.Add and DbContext.Add methods add a new entity to a context (instance of DbContext) which will insert a new record in the database when we call the SaveChanges() method.

READ

ProductRepo.cs

```
class Program
{
    private static ProductRepo productRepo = new ProductRepo();
    0 references
    static void Main(string[] args)
    {
        ReadProducts();
    }

    1 reference
    private static void ReadProducts()
    {
        Console.WriteLine("PId \t Name \t\t Price");
        Console.WriteLine("-------------------------------------------------");
        foreach (var product in productRepo.GetProducts())
        {
            Console.WriteLine($"{product.Pid} \t {product.Productname} \t\t {product.Price}");
        }
    }
}
```

Program.cs

```
public class ProductRepo
{
    private db_ShoponContext context = new db_ShoponContext();

    /// <summary>
    /// Method to get all products
    /// </summary>
    /// <returns></returns>
    1 reference
    public IEnumerable<Product> GetProducts() => this.context.Products;
}
```

UPDATE

ProductRepo.cs

```
public void UpdateProduct(Product product)
{
    var existingProduct = this.context.Products.
        FirstOrDefault(x => x.Pid == product.Pid);
    if(existingProduct != null)
    {
        existingProduct.IsDeleted = product.IsDeleted;
        existingProduct.ImageUrl = product.ImageUrl;
        existingProduct.Availablestatus = product.Availablestatus;
        existingProduct.Categoryid = product.Categoryid;
        existingProduct.Companyid = product.Companyid;
        existingProduct.Price = product.Price;
        existingProduct.Productname = product.Productname;

        this.context.SaveChanges();
    }
}
```

As soon as we modify the FirstName, the context sets its EntityState to Modified because of the modification performed in the scope of the DbContext instance (context). So, when we call the SaveChanges() method, it builds and executes the Update statement in the database.

Program.c

```
private static void UpdateProduct()
{
    Product productToUpdate = new Product()
    {
        Pid = 108,
        Availablestatus = "N",
        Categoryid = 2001,
        ImageUrl = "images/oneplus/nord2_5g.jpg",
        Companyid = 1010,
        Price = 48999,
        Productname = "Nord 2 5G(Gray)",
        IsDeleted = false
    };
    productRepo.UpdateProduct(productToUpdate);
}
```

Delete

ProductRepo.cs

```
public void DeleteProdut(int pid)
{
    var productToDelete = this.context.Products
        .FirstOrDefault(x => x.Pid == pid);
    if(productToDelete != null)
    {
        this.context.Products.Remove(productToDelete);
        this.context.SaveChanges();
    }
}
```

Program.cs

```
private static void DeleteProduct()
{
    productRepo.DeleteProdut(108);
}
```

Context.Students.Remove(std) or context.Remove<Students>(std) marks the std entity object as Deleted. Therefore, EF Core will build and execute the DELETE statement in the database.

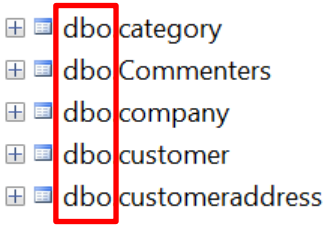
EF Core Conventions

- Conventions are default rules using which Entity Framework builds a model based on your domain (entity) classes.
- EF Core API creates a database schema based on domain and context classes, without any additional configurations because domain classes were following the conventions.

Schema | Table | Column

Schema

- EF Core will create all the database objects in the **dbo** schema by default.



Table

- EF Core will create database tables for all DbSet<TEntity> properties in a context class with the same name as the property.
- It will also create tables for entities which are not included as DbSet properties but are reachable through reference properties in other DbSet entities.

Table (Continued)

- dbo.product
  - Columns
    - pid (PK, int, not null)
    - productname (varchar(20), null)
    - price (float, null)
    - companyid (FK, int, null)
    - categoryid (FK, int, null)
    - availablestatus (char(1), null)
    - imageUrl (varchar(50), null)
    - isDeleted (bit, null)
- dbo.category
  - Columns
    - categoryid (PK, int, not null)
    - category (varchar(20), null)
  - Keys
  - Constraints
  - Triggers
  - Indexes
  - Statistics
- dbo.Commenters
- dbo.company
  - Columns
    - companyid (PK, int, not null)
    - companyname (varchar(20), null)
    - companystatus (char(1), null)
    - isdeleted (bit, null)

```
public partial class Product
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    6 references
    public int? Companyid { get; set; }
    6 references
    public int? Categoryid { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }

    1 reference
    public virtual Category Category { get; set; }

    1 reference
    public virtual Company Company { get; set; }
}
```

Column

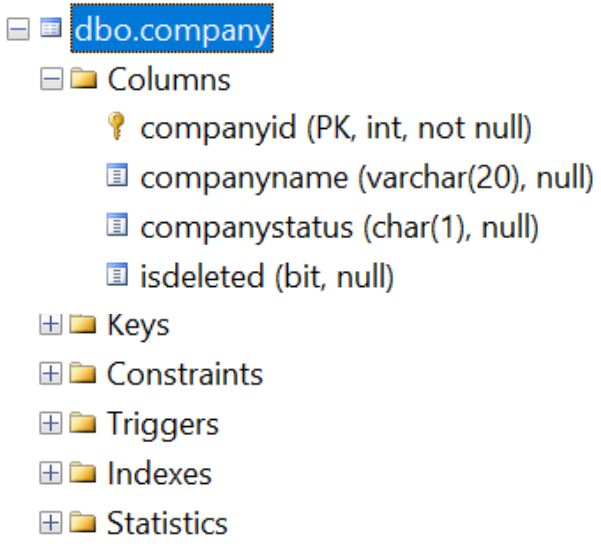
- EF Core will create columns for all the scalar properties of an entity class with the same name as the property, by default.
- It uses the reference and collection properties in building relationships among corresponding tables in the database.

Column (continued)

```
public partial class Company
{
    0 references
    public Company()
    {
        Products = new HashSet<Product>();
    }

    1 reference
    public int Companyid { get; set; }
    1 reference
    public string Companyname { get; set; }
    1 reference
    public string Companystatus { get; set; }
    1 reference
    public bool? Isdeleted { get; set; }

    2 references
    public virtual ICollection<Product> Products { get; set; }
}
```



Column Data Type

Column Data Type

- The data type for columns in the database table is depending on how the provider for the database has mapped C# data type to the data type of a selected database.
- The following table lists mapping between C# data type to SQL Server column data type.

C# Data Type	Mapping to SQL Server Data Type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime
bool	bit
byte	tinyint
short	smallint
long	bigint
double	float
char	No mapping
sbyte	No mapping (throws exception)
object	No mapping

Column Data Type (continued)

- **Nullable Column** - EF Core creates null columns for all reference data type and nullable primitive type properties e.g. string, Nullable<int>, decimal?.

- **NotNull Column** - EF Core creates NotNull columns in the database for all primary key properties, and primitive type properties e.g. int, float, decimal, DateTime etc..

- **Primary Key** - EF Core will create the primary key column for the property named Id or <Entity Class Name>Id (case insensitive). For example, EF Core will create a column as PrimaryKey in the Product table if the Product class includes a property named id, ID, iD, Id, Productid, ProductId, PRODUCTID, or ProDUCTID.

Column Data Type (continued)

- Foreign Key - As per the foreign key convention, EF Core API will create a foreign key column for each reference navigation property in an entity with one of the following naming patterns.
  - `<Reference Navigation Property Name>Id`
  - `<Reference Navigation Property Name><Principal Primary Key Property Name>`

Column Data Type (continued)

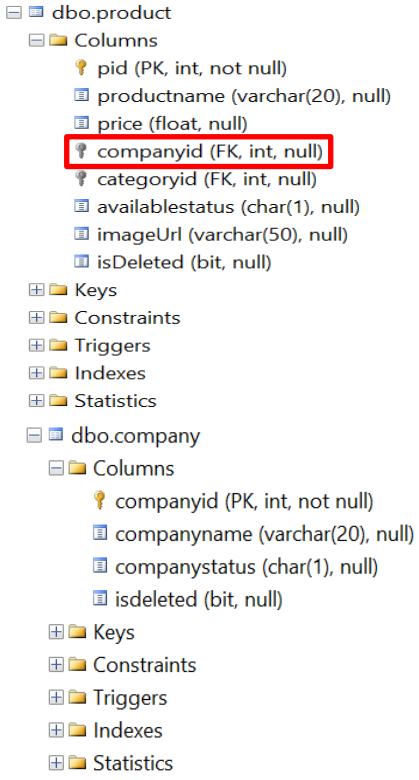
```
public partial class Product  Dependent Entity
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    6 references
    public int? Companyid { get; set; }  Foreign Key Property
    6 references
    public int? Categoryid { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }

    1 reference
    public virtual Category Category { get; set; }
    1 reference
    public virtual Company Company { get; set; } Reference Property
}
```

```
public partial class Company  Principal Entity
{
    0 references
    public Company()
    {
        Products = new HashSet<Product>();
    }

    1 reference
    public int Companyid { get; set; }
    1 reference
    public string Companyname { get; set; }
    1 reference
    public string Companystatus { get; set; }
    1 reference
    public bool? Isdeleted { get; set; }

    2 references
    public virtual ICollection<Product> Products { get; set; }
}
```



Relationship

- EF Core supports the relationship conventions between two entity classes that result in one-to-many or one-to-one relationships between corresponding tables in the database.

One-To-Many | One-To-One

One-To-Many

- Entity Framework Core follows the same convention as **Entity Framework 6.x conventions for one-to-many relationship**. The only difference is that EF Core creates a foreign key column with the same name as navigation property name and not as <NavigationPropertyName>_<PrimaryKeyPropertyName>

Convention 1 | Convention 2 | Convention 3 | Convention 4

Convention-1

- When we want to establish a one-to-many relationship where many products are associated with one company, this can be achieved by including a reference navigation property in the dependent entity as shown below. (here, the Product entity is the dependent entity, and the Company entity is the principal entity).

```
public partial class Company
{
    1 reference
    public int Companyid { get; set; }
    1 reference
    public string Companyname { get; set; }
    1 reference
    public string Companystatus { get; set; }
    1 reference
    public bool? Isdeleted { get; set; }
}
```

```
public partial class Product
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    5 references
    public int Companyid { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }

    1 reference
    public Company Company { get; set; }
}
```

Convention 2

- Another convention is to include a collection navigation property in the principal entity as shown below.

```
public partial class Company
{
    1 reference
    public int Companyid { get; set; }
    1 reference
    public string Companyname { get; set; }
    1 reference
    public string Companystatus { get; set; }
    1 reference
    public bool? Isdeleted { get; set; }

    1 reference
    public virtual ICollection<Product> Products { get; set; }
}
```

```
public partial class Product
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }
}
```

Convention 3

- Another EF convention for the one-to-many relationship is to include navigation property at both ends, which will also result in a one-to-many relationship (convention 1 + convention 2).

```
public partial class Company
{
    1 reference
    public int Companyid { get; set; }
    1 reference
    public string Companyname { get; set; }
    1 reference
    public string Companystatus { get; set; }
    1 reference
    public bool? Isdeleted { get; set; }

    1 reference
    public virtual ICollection<Product> Products { get; set; }
}
```

```
public partial class Product
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    6 references
    public int Companyid { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }

    1 reference
    public Company Company { get; set; }
}
```

Convention 4

- Defining the relationship fully at both ends with the foreign key property in the dependent entity creates a one-to-many relationship.

```
public partial class Company
{
    1 reference
    public int Companyid { get; set; }
    1 reference
    public string Companyname { get; set; }
    1 reference
    public string Companystatus { get; set; }
    1 reference
    public bool? Isdeleted { get; set; }

    1 reference
    public virtual ICollection<Product> Products { get; set; }
}
```

```
public partial class Product
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    6 references
    public int? Companyid { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }

    1 reference
    public virtual Company Company { get; set; }
}
```

One-to-One

- Entity Framework Core introduced default conventions which automatically configure a One-to-One relationship between two entities (EF 6.x or prior does not support conventions for One-to-One relationship).
- In EF Core, a one-to-one relationship requires a reference navigation property at both sides. The following **Customer** and **CustomerAddress** entities follow the convention for the one-to-one relationship.

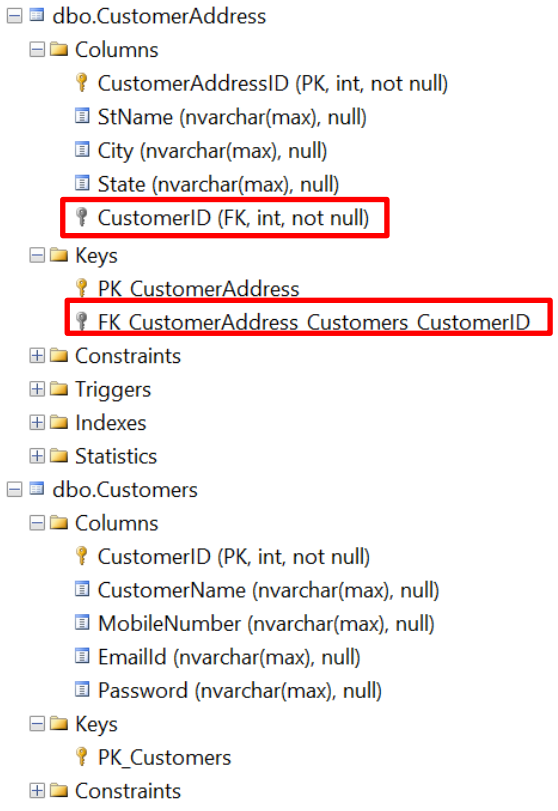
One-to-One (continued)

```
public class CustomerAddress
{
    0 references
    public int CustomerAddressID { get; set; }
    0 references
    public string StName { get; set; }
    0 references
    public string City { get; set; }
    0 references
    public string State { get; set; }

    0 references
    public int CustomerID { get; set; }
    0 references
    public Customer Customer { get; set; }
}

public class Customer
{
    0 references
    public int CustomerID { get; set; }
    0 references
    public string CustomerName { get; set; }
    0 references
    public string MobileNumber { get; set; }
    0 references
    public string EmailId { get; set; }
    0 references
    public string Password { get; set; }

    0 references
    public CustomerAddress CustomerAddress { get; set; }
}
```



Configurations in Entity Framework Core

- EF Core supports configuring with Conventions and/or Convention over Configuration.
- EF Core allows us to configure domain classes in order to customize the EF model to database mappings. This is called Convention over Configuration.
- Ways to configure domain classes in EF Core

By using Data Annotation Attributes

By using Fluent API

Data Annotation Attributes

- Data Annotations is a simple attribute-based configuration method where different .NET attributes can be applied to domain classes and properties to configure the model.

- Data annotation attributes are not dedicated to Entity Framework, as they are also used in ASP.NET MVC. This is why these attributes are included in separate namespace

***System.ComponentModel.DataAnnotations*.**

Example

Data Annotation Attributes - Example

Bank.cs

```
public class Bank
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    0 references
    public int BankId { get; set; }  public string BankName { get; set; }
    0 references
    public string City { get; set; } public string IFSC { get; set; }
    3 references
    public List<Offer> Offers { get; set; }
    0 references
    public Bank()
    {
        this.Offers = new List<Offer>();
    }
    0 references
    public void AddOffer(Offer offer)
    {
        this.Offers.Add(offer);
    }
    0 references
    public List<Offer> GetOffers()
    {
        return this.Offers;
    }
}
```

Offer.cs

```
[Table("BankOffers")]
5 references
public class Offer
{
    [Column("OfferId")]
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    0 references
    public int ID { get; set; }

    [Column("ValidateUpto")]
    0 references
    public DateTime OfferTime { get; set; }
    0 references
    public int Discount { get; set; }
    0 references
    public string Remark { get; set; }
    0 references
    public int BankId { get; set; }

    [ForeignKey("BankId")]
    0 references
    public Bank Bank { get; set; }
}
```

Data Annotation Attributes - Example

db_ShoponContext.cs

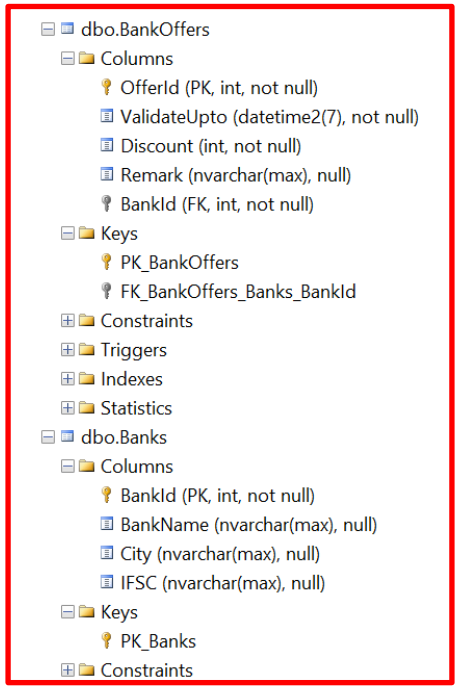
```
public partial class db_ShoponContext : DbContext
{
    1 reference
    public db_ShoponContext()...

    0 references
    public db_ShoponContext(DbContextOptions<db_ShoponContext> options)...

    0 references
    public virtual DbSet<Category> Categories { get; set; }
    0 references
    public virtual DbSet<Company> Companies { get; set; }
    5 references
    public virtual DbSet<Product> Products { get; set; }
    0 references
    public virtual DbSet<Customer> Customers { get; set; }
    0 references
    public virtual DbSet<CustomerAddress> CustomerAddress { get; set; }

    0 references
    public virtual DbSet<Bank> Banks { get; set; }
    0 references
    public virtual Offer Offers { get; set; }
```

Create new **DbSet** in Context class



After creating class, we must add migration and update database

```
PM> Add-Migration initBank
PM> Update-Database
```

Fluent API

- Entity Framework Fluent API is used to configure domain classes to override conventions. EF Fluent API is based on a Fluent API design pattern 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.

Fluent API (continued)

- Entity Framework Core Fluent API configures the following aspects of a model:
  - **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.
  - **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.
  - **Property Configuration**: Configures property to column mapping e.g. column name, default value, nullability, Foreignkey, data type, concurrency column etc.

Fluent API (continued)

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.

Fluent API (continued)

Configurations	Fluent API Methods	Usage
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.

Fluent API (continued)

Configurations	Fluent API Methods	Usage
Property Configuration	HasColumnName()	Configures the corresponding column name in the database for the property.
	HasDefaultValue()	Configures the default value for the column that the property maps to when targeting a relational database.
	HasMaxLength()	Configures the maximum length of data that can be stored in a property.
	IsRequired()	Configures whether the valid value of the property is required or whether null is a valid value.
	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.

Version 1.0

Fluent API (continued)

- Relationship with Fluent API – Using Fluent API we can set different relationships between tables like

One-to-Many Relationships | One-to-One Relationships | Many -to-Many Relationships

One-to-Many Relationships

- Generally, we don't need to configure one-to-many relationships because EF Core includes enough conventions which will automatically configure them.

- Entity Framework Core made it easy to configure relationships using Fluent API.

Example

One-to-Many Relationships - Example

Commenter.cs

```
public class Commenter
{
    0 references
    public int CommenterId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public int Age { get; set; }
    0 references
    public string Gender { get; set; }

    public virtual ICollection<CustomerComment>
        1 reference
        CustomerComments { get; set; }
}
```

CustomerComment.cs

```
public class CustomerComment
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string Comment { get; set; }
    1 reference
    public string Location { get; set; }
    1 reference
    public int CommenterId { get; set; }

    1 reference
    public virtual Commenter Commenter { get; set; }
}
```

db_ShoponContext.cs

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasAnnotation("Relational:Collation", "SQL_Latin1_General_CP1_CI_AS");

    modelBuilder.Entity<CustomerComment>(entity =>
    {
        entity.HasKey(d => d.Id)
            .HasName("PK__CustomerComment__DD37D91A44BAD04F");

        entity.ToTable("CustomerComments");

        entity.Property(d => d.Location)
            .HasColumnName("City");

        entity.Property(d => d.Comment)
            .HasColumnName("CustomerComment")
            .HasMaxLength(250);

        entity.HasOne(d => d.Commenter)
            .WithMany(c => c.CustomerComments)
            .HasForeignKey(d => d.CommenterId)
            .HasForeignKey("CommenterId");
    });
```

After creating class, we must add migration and update database

```
PM> Add-Migration initCustomerComments
```

```
PM> Update-Database -Verbose
```

One-to-One Relationships

- Generally, you don't need to configure one-to-one relationships manually because EF Core includes Conventions for One-to-One Relationships. However, if the key or foreign key properties do not follow the convention, then you can use data annotation attributes or Fluent API to configure a one-to-one relationship between the two entities.

Example

One-to-One Relationships - Example

Student.cs

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

    public StudentAddress Address { get; set; }
}
```

StudentAddress.cs

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

db_ShoponContext.cs

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>()
        .HasOne<StudentAddress>(s => s.Address)
        .WithOne(ad => ad.Student)
        .HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId);
}
```

Many -to-Many Relationships

- The steps for configuring many-to-many relationships would the following:
  - Define a new joining entity class which includes the foreign key property and the reference navigation property for each entity.
  - Define a one-to-many relationship between other two entities and the joining entity, by including a collection navigation property in entities at both sides (Product and Vendor, in this case).
  - Configure both the foreign keys in the joining entity as a composite key using Fluent API.

Example

Many -to-Many Relationships - Example

```
public partial class Product
{
    6 references
    public int Pid { get; set; }
    5 references
    public string Productname { get; set; }
    5 references
    public double? Price { get; set; }
    6 references
    public int? Companyid { get; set; }
    6 references
    public int? Categoryid { get; set; }
    5 references
    public string Availablestatus { get; set; }
    5 references
    public string ImageUrl { get; set; }
    4 references
    public bool? IsDeleted { get; set; }

    1 reference
    public virtual Category Category { get; set; }
    1 reference
    public virtual Company Company { get; set; }

    public ICollection<ProductVendor>
        0 references
        ProductVendors { get; set; }
}
```

```
public class Vendor
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string EmailID { get; set; }
    0 references
    public string MobileNo { get; set; }

    public ICollection<ProductVendor>
        0 references
        ProductVendors { get; set; }
}
```

```
public class ProductVendor
{
    1 reference
    public int ProductId { get; set; }
    0 references
    public Product Product { get; set; }

    1 reference
    public int VendorId { get; set; }
    0 references
    public Vendor Vendor { get; set; }
}
```

Many -to-Many Relationships - Example

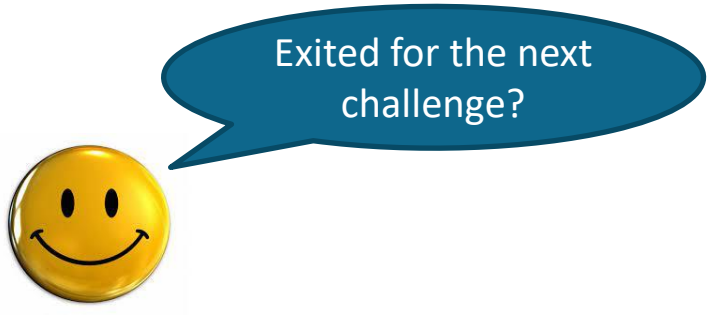
- Now, the foreign keys must be the composite primary key in the joining table. This can only be configured using Fluent API, as below.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasAnnotation("Relational:Collation", "SQL_Latin1_General_CP1_CI_AS");
    modelBuilder.Entity<ProductVendor>().HasKey(d => new {d.VendorId, d.ProductId });
    modelBuilder.Entity<CustomerComment>(entity =>...);
    modelBuilder.Entity<Category>(entity =>...);
    modelBuilder.Entity<Company>(entity =>...);
```

After creating class, we must add migration and update database

```
PM> Add-Migration initProductVendor
PM> Update-Database -Verbose
```

Next Step



Recap

Useful links

Thank you

Recap

Useful Links

- Pls paste links from all the slides

US – CORPORATE HEADQUARTERS

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK

89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

INDIA

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000

Pune
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666

www.xoriant.com