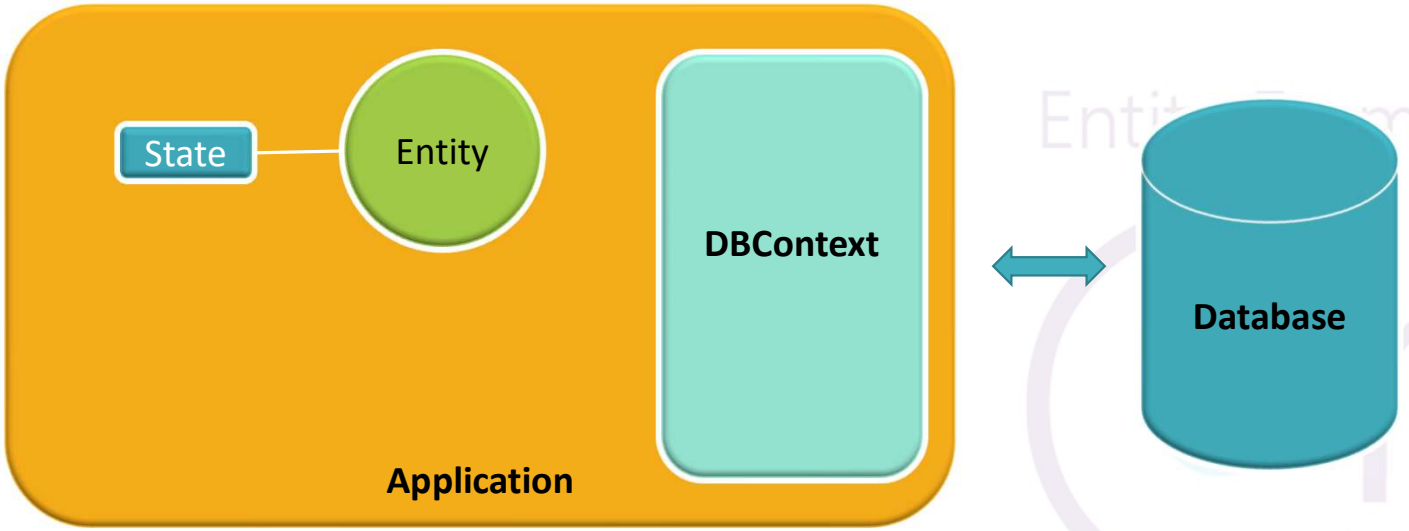
Entity's Changes Tracking

Entity Change Tracking

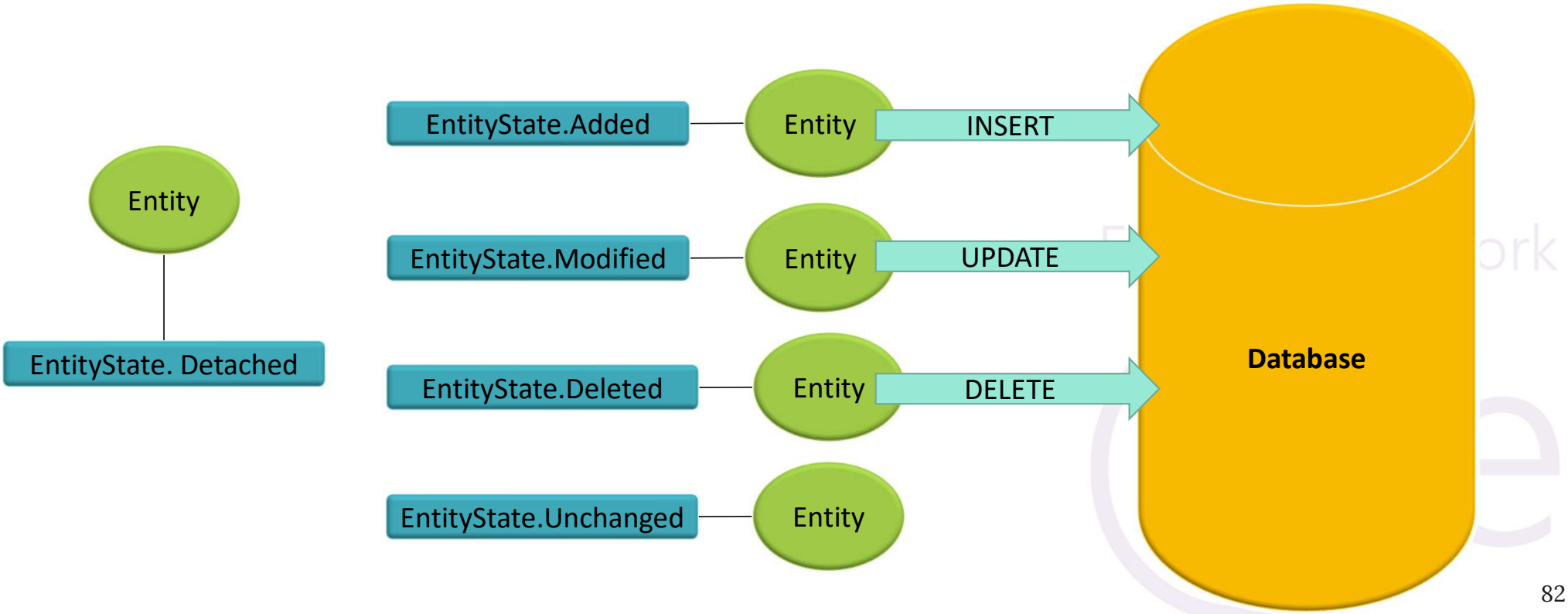
- `DbContext` tracks changes in entities (*Change Tracking*)



Entity Change Tracking

- Entity State
  - Enum `System.Data.Entity.EntityState`
    - `Added`
    - `Deleted`
    - `Modified`
    - `Unchanged`
    - `Detached` (`AsNoTracking()`, `state`, add with `new`)
- Unchanged to Modified is the only state change that is Automatically Done by Context
- Other changes must be explicitly done
  - using context methods (`Add`, `Remove`)
  - change Entity's state by code

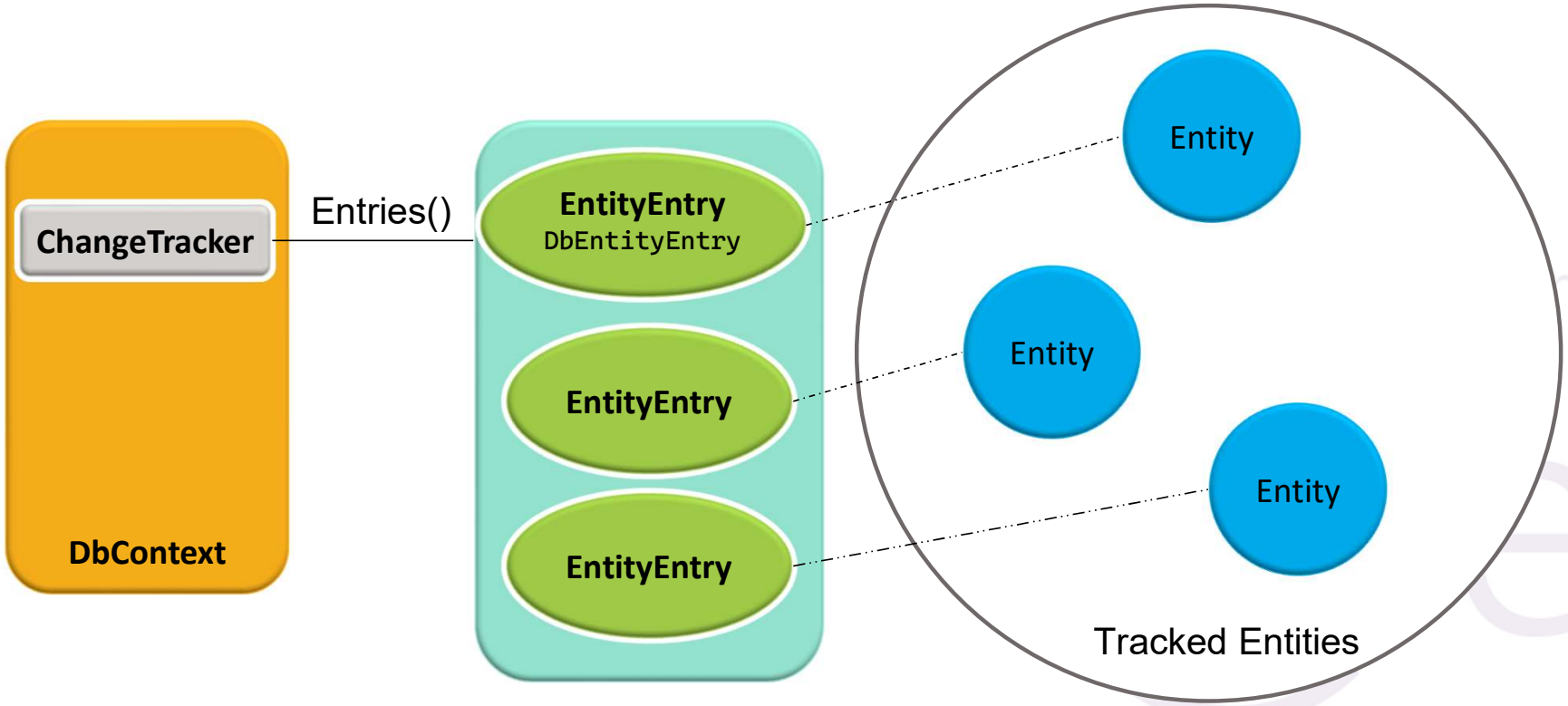
Entity State and `DbContext.SaveChanges()`



Change Tracking in Entity Framework Core

- Provides access to change tracking information and operations for entity instances that the context is tracking
- `DbContext.ChangeTracker` property
  - Type (`ChangeTracker` class)
  - `Entries()` method
    - Return collection of `DbEntityEntry`. contains object for each entity being tracked
- Every entity must have primary key to be tracked

Change Tracking in Entity Framework Core



Change Tracking in Entity Framework Core

```
using (var context = new CompanyDbContext())
{
    Console.WriteLine("Get First Employee");
    var emp = context.Employees.FirstOrDefault();
    emp.Name = "Modified Name";

    Console.WriteLine($"Context Tracking Changes of{ context?.ChangeTracker.Entries().Count()}
    Entities");
    Console.WriteLine("Get First Department");

    var Dept = context.Departments.FirstOrDefault();

    Console.WriteLine($"Context Tracking Changes of{ context?.ChangeTracker.Entries().Count()}
    Entities");
    Display(context.ChangeTracker);
}
```

Change Tracking in Entity Framework Core

```
public static void Display(ChangeTracker  tracker)
{
    var Entries = tracker.Entries();
    System.Console.WriteLine("**********************************");
    foreach (var Entry in Entries)
    {
        System.Console.WriteLine($"Entry Name={Entry.Entity.GetType().Name}");
        System.Console.WriteLine($"State={Entry.State}");
    }
}
```

EntityEntry class

- Provides access to change tracking information and operations for a given entity.
- Used for get information about an entity (state, current value, original value)

```
using (var ctx = new CompanyDBContext())
{
  var emp = ctx.Employees.Include(e=>e.Department).FirstOrDefault();
  emp.EmployeeName = "Modified Name";
  Console.WriteLine(ctx.Entry(emp).State);
  Console.WriteLine("============================");
  foreach (var property in ctx.Entry(emp).Properties)
  {
    Console.WriteLine("Property Name = " + property.Metadata.Name);
    Console.WriteLine("Property Original value = " + property.OriginalValue);
    Console.WriteLine("Property Original value = " + property.CurrentValue);
  }
}
```

EntityEntry class

- State on property level

```
var em = ctx.Employees.Find(1);
                em.EmployeeName = "New Name";
                var entry = ctx.Entry(em);

                if (entry.Property(x => x.EmployeeName).IsModified)
                {
                    Console.WriteLine("Name is Modified");
                }
                if (entry.Property("EmployeeName").IsModified)
                {
                    Console.WriteLine("Name is Modified");
                }
```

EntityEntry class

- Methods

Method Name	Return Type	Description

Collection	CollectionEntry	Gets an object that represents the collection **navigation property** from this entity to a collection of related entities.
		Example:
		var DeptEntityEntry =dbContext.Entry(DepartmentEntity);
		var collectionProperty =DeptEntityEntry.Collection(d =>d.Employees);

Reference	ReferenceEntry<TEntity, TProperty>	Gets an object that represents the reference (i.e. noncollection) Navigation property from this entity to another entity. Example: Var EmployeeDBEntityEntry=dbContext.Entry(EmployeEntity); Var referenceProperty=EmployeeDBEntityEntry.Reference(s=>s.Department);

EntityEntry class

- Methods

Method Name	Return Type	Description
Property	PropertyEntry	Gets an object that represents a scalar or complex property of this entity. Example: var studentDBEntityEntry =dbContext.Entry(studentEntity); string propertyName =studentDBEntityEntry.Property("StudentName").Name;
ComplexProperty	ComplexPropertyEntry	Gets an object that represents a **complex property** of this entity. Example: var studentDBEntityEntry =dbContext.Entry(studentEntity); var complexProperty =studentDBEntityEntry.ComplexProperty(stud.StudentStandard);

EntityEntry class

Method Name	Return Type	Description

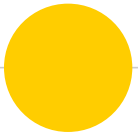
GetDatabaseValues	PropertyValues	Queries the database for copies of the values of the tracked entity as they currently exist in the database. Changing the values in the returned dictionary will not update the values in the database. If the entity is not found in the database then null is returned.
		Example:
		var studentEntityEntry =dbContext.Entry(studentEntity);
		var dbPropValues =studentDBEntityEntry.GetDatabaseValues();

Reload	void	Reloads the entity from the database overwriting any property values with values from the database. The entity will be in the Unchanged state after calling this method.
		Example:
		var studentDBEntityEntry =dbContext.Entry(studentEntity);
		studentDBEntityEntry.Reload();

Type of Entities

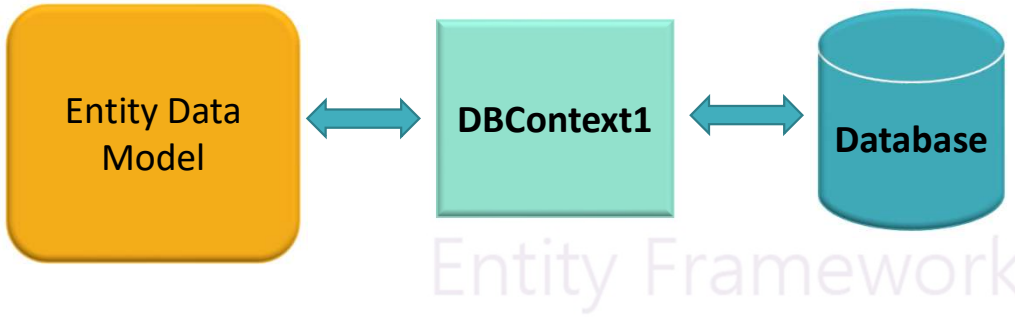
- Tracked Entities
- Detached Entities
- Disconnected Entities

Persistence in Entity Framework

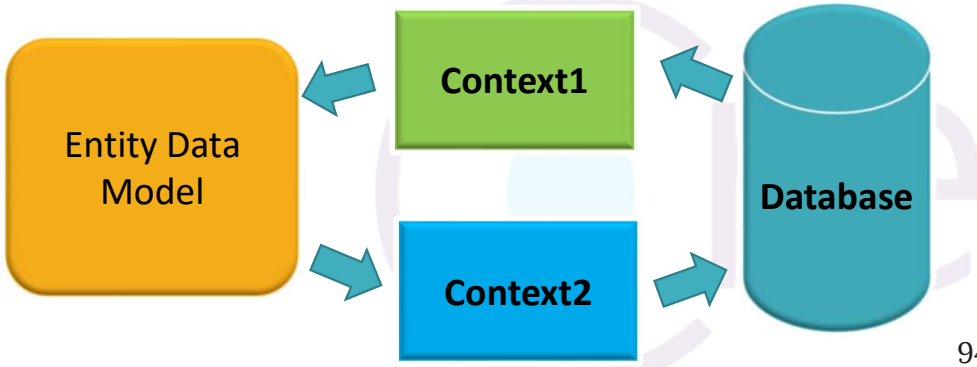


Persistence in Entity Framework

- Connected
  - Entity retrieved from DB and same Context used to persist (save) to DB



- Disconnected
  - Entity retrieved from DB using Context and another Context used to persist it to DB



Persistence in Entity Framework

- Disconnected
  - Web application (short lived DbContext)
    - connect to the database only for the duration of a single web page request. They don't maintain persistent database connections

Connected Entities

- CURD Operation (Add-Update-Read-Delete)

```
using (var context = new CompanyDBContext())
{
  var EmployeeList = context.Employees.ToList<Employee>();
  //Perform create operation
  context.Employees.Add(new Employee() { EmployeeName = "New Employee" });
  //Perform Update operation
   Employee employeeToUpdate =
   EmployeeList.Where(e => e.EmployeeName == "Emp1").FirstOrDefault<Employee>();
   employeeToUpdate.EmployeeName = "Edited Employee1";
   //Perform delete operation
   context.Employees.Remove(EmployeeList.ElementAt<Employee>(0));
   //Execute Insert, Update & Delete queries in the database
   context.SaveChanges();
 }
```

Connected Entities

- Add Related

```
using (var context = new CompanyDBContext())
{
  var Dept=context.Departments.FirstOrDefault();
  var Employee=new Employee{Name="Admed_Added",Department=Dept};
  context.Employees.Add(Employee);
  context.SaveChanges();
 }
```

Connected Entities

- Add Related

```
using (var context = new CompanyDBContext())
{
  var empArr=new Employee[]{
                new Employee{Name="New1",},
                new Employee{Name="New2",},
            };
 var Department =new Department{Name="new_dept",Employees=empArr};
 context.Add(Department);
 context.SaveChanges();
}
```

Connected Entities

- Delete Related

```
using (var context = new CompanyDBContext())
{
  var Department=context.Departments
                .Where(d=>d.Name=="new_dept")
                .FirstOrDefault();
                if(Department!=null)
                {
                    context.Departments.Remove(Department);
                }
                context.SaveChanges();
}
```

Connected Entities

- Udpate Related
- For DbContext to detects modification
  - `context.ChangeTracker.AutoDetectChangesEnabled = true`
  - Set it to false would stop tracking updated entities but continue track inserted and deleted Entities
  - Calling Context.ChangeTracker.DetectChanges() berfore saveChanges() would allow to track updated entities
  - Adding or deleting entities must be through context.DbSet (ctx.employees) otherwise the changes wont be detected

```
var EmployeeList = ctx.Employees.ToList();
EmployeeList.Add(new Employee() {EmployeeName = "mmmmmmm" }); // wont be detected
ctx. Employees.Add(new Student() {EmployeeName = "mmmmmmm" });
```

Disconnected Entities

- Attach entities with the new context instance and make context aware about these entities.
- Set appropriate EntityStates to these entities manually
  - Add new Entity using DBContext
  - Update Existing Entity using DBContext
  - Delete Entity using DBContext
  - Add Entity Graph using DBContext

Disconnected Entities – INSERT

- DbContext Methods

DbContext Methods	DbSet Methods	Description
DbContext.Attach	DbSet.Attach	Attach an entity to DbContext. Set Unchanged state for an entity whose Key property has a value and Added state for an entity whose Key property is empty or the default value of data type.
DbContext.Add	DbSet.Add	Attach an entity to DbContext with Added state.
DbContext.AddRange	DbSet.AddRange	Attach a collection of entities to DbContext with Added state.
DbContext.Entry	-	Gets an EntityEntry for the specified entity which provides access to change tracking information and operations.

Disconnected Entities - INSERT

- DbContext Methods

DbContext Methods	DbSet Methods	Description
DbContext.AddAsync	DbSet.AddAsync	Asynchronous method for attaching an entity to DbContext with Added state and start tracking it if not. Data will be inserted into the database when SaveChangesAsync() is called.
DbContext.AddRange Async	DbSet.AddRangeAsync	Asynchronous method for attaching multiple entities to DbContext with Added state in one go and start tracking them if not. Data will be inserted into the database when SaveChangesAsync() is called.

Disconnected Entities – INSERT

```
var newEmployee = new Employee();
newEmployee.EmployeeName = "New Name";
using(var ctx=new CompanyDBContext())
{
    ctx.Employees.Add(newEmployee);
    ctx.SaveChanges();
}
```

```
var newEmployee = new Employee();
newEmployee.EmployeeName = "New Name";
using (var ctx=new CompanyDBContext())
{
    dbCtx.Entry(newEmployee).State = EntityState.Added;
    dbCtx.SaveChanges();
}
```

Disconnected Entities - INSERT

```
Employee attached_employee1,attached_employee2;
EntityState state1,state2;
using (var context1 = new CompanyDbContext())
{
  attached_employee1=context1.Employees.FirstOrDefault();
  attached_employee2=new Employee{Name="Attached_new"};
  state1=context1.Entry(attached_employee1).State;  //Unchanged
  state2=context1.Entry(attached_employee2).State;   //Detached
}
  //attached_employee1.Name="ModName";
using (var context2 = new CompanyDbContext())
{
  context2.Attach(attached_employee1);
  context2.Attach(attached_employee2);
  state1=context2.Entry(attached_employee1).State; //Unchanged
  state2=context2.Entry(attached_employee2).State; // Added
}
```

Disconnected Entities – UPDATE

```
Employee emp;
using (var ctx1 = new CompanyDBContext())
{
  emp = ctx1.Employees.FirstOrDefault();
}
emp.EmployeeName = "Modified Name";
using (var ctx = new CompanyDBContext())
{
  ctx.Update(emp);
  // or the followings are also valid
  // ctx.Employees.Update( emp);
  // ctx.Attach<Employee>(emp ).State = EntityState.Modified;
  // ctx.Entry<Employee>(emp ).State = EntityState.Modified;
}
```

Disconnected Entities – UPDATE

```
var Employee attached_employee1,attached_employee2;
EntityState state1,state2;
using (var context1 = new CompanyDbContext())
{
  attached_employee1=context1.Employees.FirstOrDefault();
  attached_employee2=new Employee{Name="Attached_new"};
  state1=context1.Entry(attached_employee1).State;  //Unchanged
  state2=context1.Entry(attached_employee2).State;   //Detached
}
  attached_employee1.Name="ModName";
using (var context2 = new CompanyDbContext())
{
  context2.Update(attached_employee1);
  context2.Update(attached_employee2);
  state1=context2.Entry(attached_employee1).State; //Modified
  state2=context2.Entry(attached_employee2).State; // Added
}
```

Disconnected Entities – UPDATE

- Add& update
- For Auto generated key
  - Update method could be used instead add or update
  - The Update method normally marks the entity for update, not insert. However, if the entity has an auto generated key, and no key value has been set, then the entity is automatically marked for insert

Disconnected Entities – UPDATE

- Add& update
- Not Auto generated key
  - Use Find (Id)

```
Employee DisconnectedEmployee;
/// code here
using (var ctx = new CompanyDBContext())
{
  var em = ctx.Employees.Find(DisconnectedEmployee.EmployeeId);
  if (em == null)
  {  ctx.Add(DisconnectedEmployee);                }
  else
  {  ctx.Entry(em).CurrentValues.SetValues(DisconnectedEmployee);  }
}
```

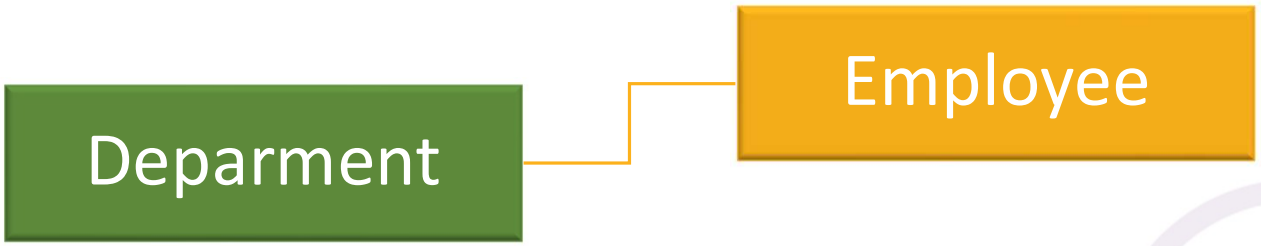
Disconnected Entities - DELETE

DbContext Methods	DbSet Methods	Description
DbContext.Remove	DbSet.Remove	Attaches the specified entity to the DbContext with Deleted state and starts tracking it.
DbContext.RemoveRange	DbSet.RemoveRange	Attaches a collection or array of entities to the DbContext with Deleted state and starts tracking them.

Disconnected Entities – DELETE

```
Employee emp;
using (var ctx1 = new CompanyDBContext())
{
  emp = ctx1.Employees.FirstOrDefault();
}
emp.EmployeeName = "Modified Name";
using (var ctx = new CompanyDBContext())
{
  ctx.Remove(emp);
  // or the followings are also valid
  //ctx.Employees.Remove( emp);
  // ctx.Attach<Employee>(emp ).State = EntityState.Deleted;
  // ctx.Entry<Employee>(emp ).State = EntityState.Deleted;
}
```

Disconnected Entities
Entity Graph (Related Data)



Disconnected _ Graph _Add

- Adding new entity graph with *all new entities*

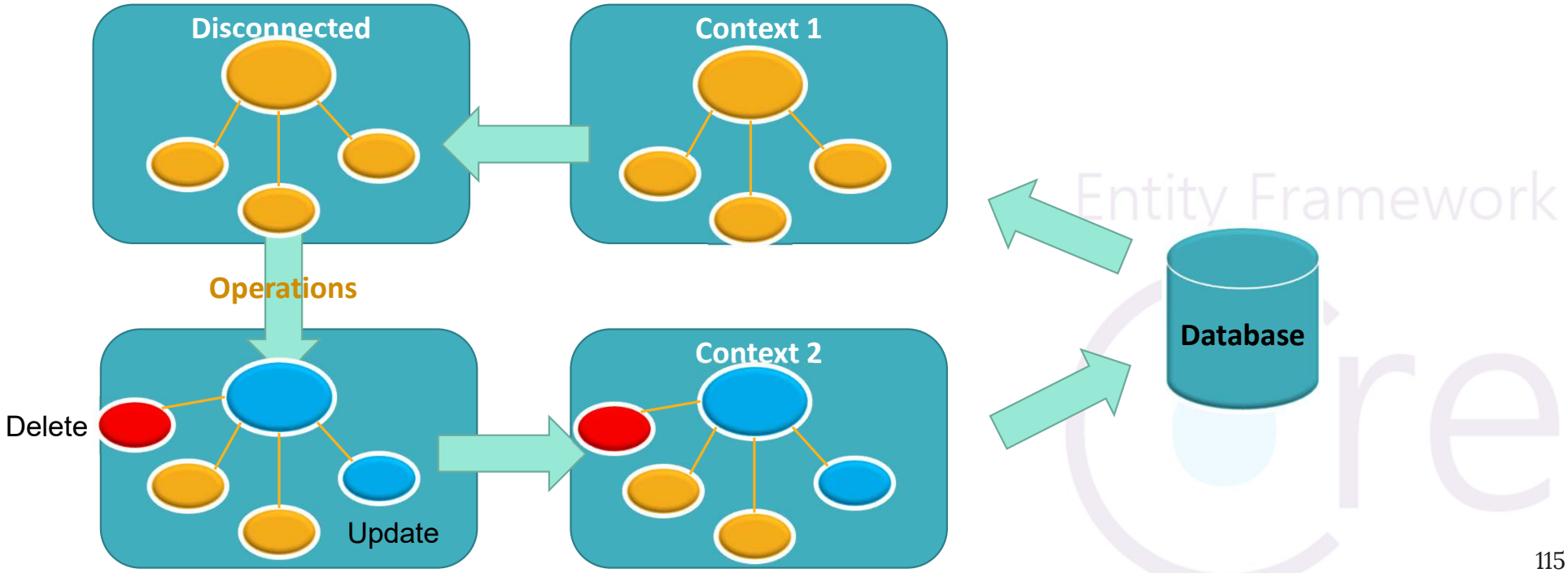
```
public static void Disconected_Graph_Add_AllNew()
    {
        Department dept=new Department{Name="New_Dep"};
        dept.Employees=new List<Employee>();
        dept.Employees.Add( new Employee{Name="New_Ahmed",Salary=1000} );
        dept.Employees.Add( new Employee{Name="New_ALY",Salary=2000} );
        using (var context1 = new CompanyDbContext())
        {
            context1.Add(dept);
            context1.SaveChanges();
        }
    }
```

Disconnected _ Graph _Update

- Update entity graph with *all Exist entities*

```
Department DisconnectedDepartment;
using (var ctx = new CompanyDBContext())
{
 DisconnectedDepartment = ctx.Departments.Include(d=>d.Employees).FirstOrDefault();
 DisconnectedDepartment.Employees.ElementAt(0).EmployeeName = "Modified Employee";
}
using (var context = new CompanyDBContext())
{
  context.Departments.Update(DisconnectedDepartment);
  context.SaveChanges();
}
```

Disconnected _ Graph _Update (MIX)



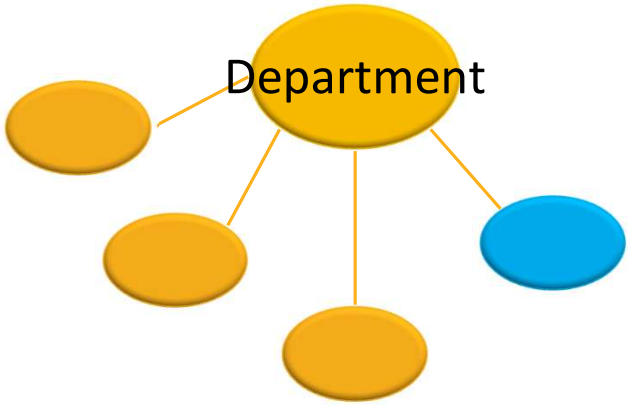
Disconnected _ Graph _Update

- Mix of new and Exist
  - Auto Generated Key
    - Update will work as (Insert- Update) depends on the value of primary Key

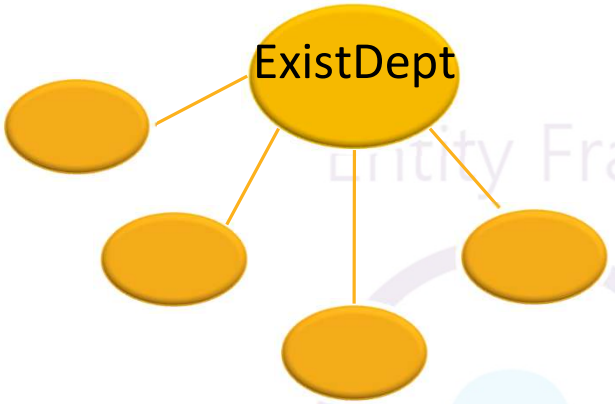
```
Department DisconnectedDepartment;
using (var ctx = new CompanyDBContext())
{
  DisconnectedDepartment = ctx.Departments.Include(d => d.Employees).FirstOrDefault();
  DisconnectedDepartment.Employees.ElementAt(0).EmployeeName = "Modified Employee";
}
DisconnectedDepartment.Employees.Add( new Employee { EmployeeName="New Employee"});
using (var context =new CompanyDBContext())
{
  context.Departments.Update(DisconnectedDepartment);
}
```

Disconnected _ Graph _Update

- Mix of New and Exist
  - Not Auto Generated Key
    - Code in Notes



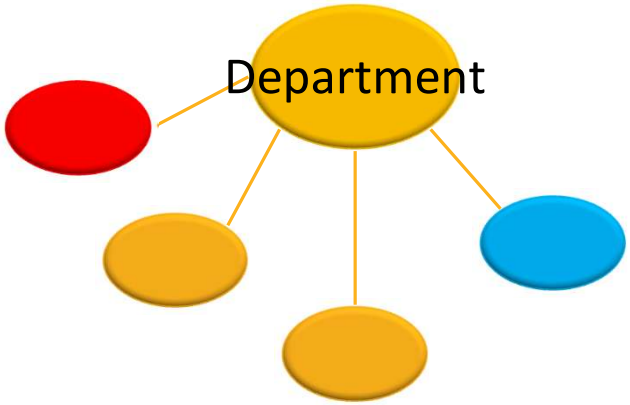
Disconnected
(Detached)



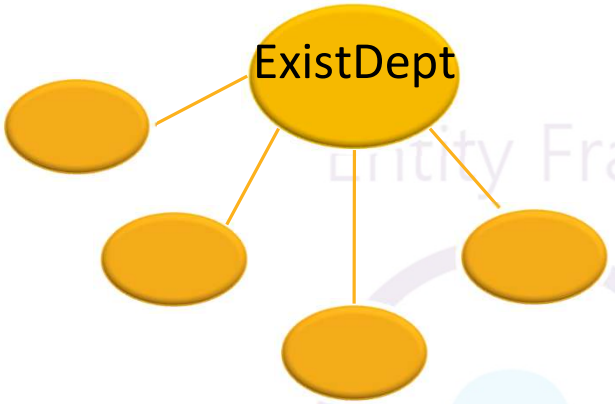
Retrieved from DB
(Tracked)

Disconnected _ Graph _Delete

- Mix of New , Exist and Delete
  - Not Auto Generated Key
    - Code in Notes

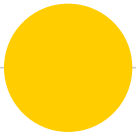


Disconnected
(Detached)

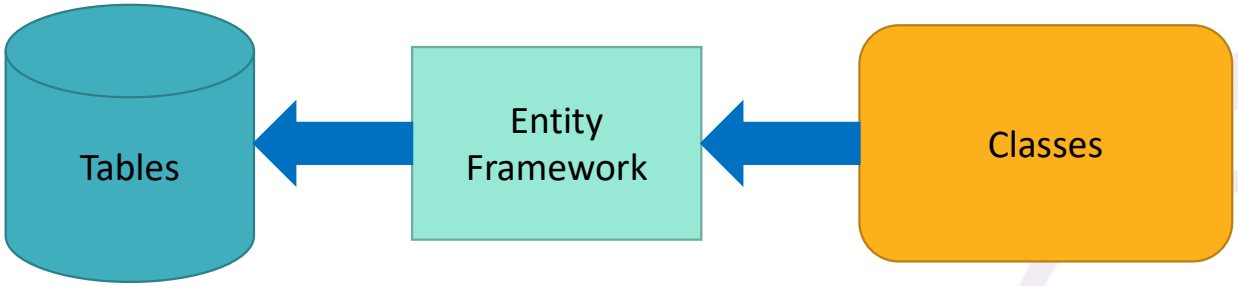


Retrieved from DB
(Tracked)

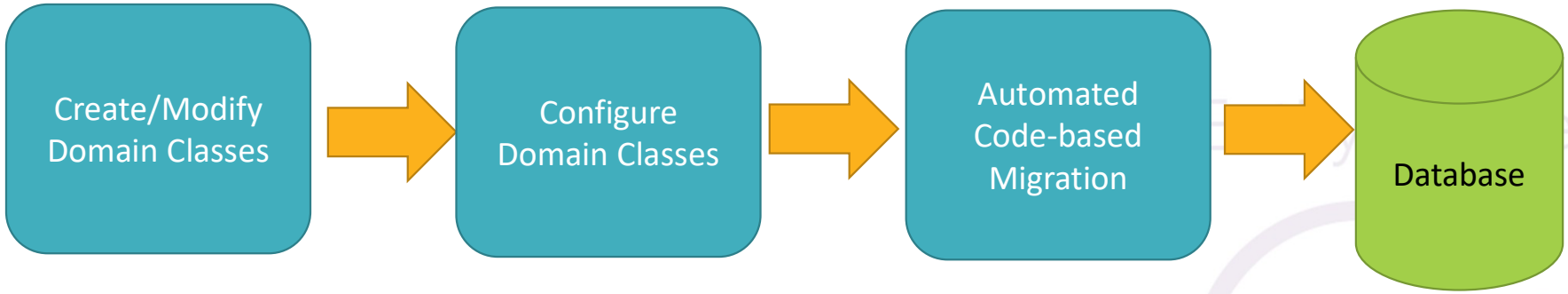
Code First Approach



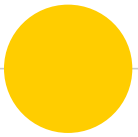
What is Code First



Code First Workflow



Installing Entity Framework Core



Install Data Provider

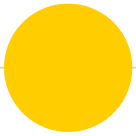
- Data provider
  - plugin libraries used by EF to access many data base
- Add NuGet Package

Database System	NuGet Package
SQL Server or Azure SQL	Microsoft.EntityFrameworkCore.SqlServer
Azure Cosmos DB	Microsoft.EntityFrameworkCore.Cosmos
SQLite	Microsoft.EntityFrameworkCore.Sqlite
EF Core in-memory database	Microsoft.EntityFrameworkCore.InMemory
PostgreSQL*	Npgsql.EntityFrameworkCore.PostgreSQL
MySQL/MariaDB*	Pomelo.EntityFrameworkCore.MySql
Oracle*	Oracle.EntityFrameworkCore

Installing EF Core Tools

- Add NuGet Package *Microsoft.EntityFrameworkCore.Tools*
  - For both (Code first, Database First)
  - Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.
    - Add-Migration
    - Scaffold-DbContext
    - Update-Database
- Add NuGet Package *Microsoft.EntityFrameworkCore.Design*
  - (code first)
  - Used for creating database using migration

EF Code First Demo



Design School application

- Domain classes

  (Business classes)
  - Class Student
  - Class Grade
  - One to many Relationsh

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
    public virtual Grade Grade { get; set; }
}
```

```
public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }
    public virtual ICollection<Student> Students { get; set; }
}
```

DbContext class

- Context Class

```
using Microsoft.EntityFrameworkCore;
```

```
public class SchoolContext:DbContext
{
public SchoolContext():base()
{
}
public virtual DbSet<Student> Students { set; get; }
public virtual DbSet<Grade> Grades { set; get; }
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{ string constr =
@"server=(localdb)\MSSQLLocalDB;Database=SchoolDB2;Trusted_Connection=true;
TrustServerCertificate=True";
 optionsBuilder.UseSqlServer(constr);
}
}
```

DbContext class

- Constructor
  - Default

```
public SchoolContext():base()
{
}
```

    - Creating an object

```
public SchoolContext context= new SchoolContext();
```

DbContext class

- Constructor
  - String parameter (*connection string*)
    - Connection string (Server name, Database name, Username and password)

```
public class SchoolContext:DbContext
{
  readonly string _stringConn;
  public SchoolContext(string constr)
  {
     _stringConn=constr;
  }
```

DbContext class

- Constructor

DbContext class

- Using configuration JSON file
  - To make connection string in config file
  - add a .json file (e.g appconfig.json) at the root of your project and put the following content in it

```
{
  "ConnectionStrings": {
    "myDbConn":"server=(localdb)\\MSSQLLocalDB;Database=CompanyDB;Trusted_Connection=true"
  }
}
```

  - in the solution explorer, right click on the appconfig.json file and select Properties. Set the value of Copy to Output Directory to Copy Always.
  - install the *Microsoft.Extensions.Configuration.Json* package

DbContext class

- modify the `OnConfiguring` method
  - Install nugget package *Microsoft.Extensions.Configuration.json*

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    var config = new ConfigurationBuilder()
        .AddJsonFile("appconfig.json", optional: false).Build();

    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(config.GetConnectionString("myDbConn"));
    }
}
```

DbContext class

- Using appsettings.json (***web application***)
- Modify `ConfigureServices` method on `startup` class

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<pubsContext>(options =>
    {
        options.UseSqlServer(Configuration.GetConnectionString(“myDbConn"));
    });
    services.AddControllersWithViews();
}
```

DbContext class

- `OnModelCreating()` Method
  - Allows us to tell Entity Framework Core more about the entities like:
    - Length of a property of an entitiy.
    - Whether a property is required by default.
    - Relationships between the entities. One-to-Many, One-to-One,
  - allows us to configure the model using ModelBuilder Fluent API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }
```

Migration

- Migration is a way to keep the database schema in sync with the EF Core model by preserving data.
- Migration is a snap shot of database Schema
- Add migration
  - Visual studio

```
add-migration  CreateSchoolDb2
add-migration -Context ProjectsContext2 CreateDBSqlServer
```

  - VS code or CLI (Command Line Interface)

```
dotnet ef migrations add CreateSchoolDb2 –context projectContext2
```

Update or Create Database

- Visual Studio

```
Update-database
Update-database -context ProjectsContext2
```

- Visual studio code

```
dotnet ef database update
```

Remove Migration

- Visual studio

```
remove-migration
```

- Visual studio code

```
dotnet ef migrations remove
```

Other Options

- Drop Database
  - Visual studio

    ```
    Drop-Database
    ```

  - VS Code

    ```
    dotnet ef database drop
    ```

- Generate SQL Script
  - Visual Studio

    ```
    script-migration
    ```

  - Vs code

    ```
    dotnet ef migrations script
    ```

How Database & tables schema Generated

- Code first Conventions

Default Convention For	Description
Schema	By default, EF creates all the DB objects into the **dbo** schema.
Table	• Generate table for each domain class named as the name of DataSet<T> property • Generate table for each class reachable through navigation property but not included as DataSet<T> property
Column	• Generate column for each Scalar Property in Domain Class • Navigation Properties used for generate relationships between tables • Table Column order as domain class property order

How Database & tables schema Generated

- How Data Mapped

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

C# Data Type	Mapping to SQL Server Data Type
long	bigint
double	float
char	No Mapping
sbyte	No Mapping
object	No Mapping

How Database & tables schema Generated

- Code first Conventions

Default Convention For	Description
Null column	EF creates a null column for all reference type properties and nullable primitive properties e.g. string, Nullable<int>, Student, Grade (all class type properties)
Not Null Column	EF creates NotNull columns for Primary Key properties and non-nullable value type properties e.g. int, float, decimal, datetime etc.

How Database & tables schema Generated

- Code first Conventions

Default Convention For	Description
Primary key	Name 1) Id 2) <Entity Class Name> + "Id" (case insensitive) EF will create a primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive).
Foreign key property Name	<Reference Navigation Property Name>Id **EX:GradeId**

How Database & tables schema Generated

- Database Tables Relationships
  - One-to-Many
  - One-One
  - Many-to-Many

How Database & tables schema Generated

- One-to-Many
  - Reference Navigation Property
  - Collection Navigation Property
  - Both Collection and reference Navigation Property

How Database & tables schema Generated

- One-to-One
  - Reference navigation on both Domain classes

How Database & tables schema Generated

- Many to Many Relationship
  - Collection Navigation on both Domain Class
  - Generate join Table

Configuration Domain Classes

- Override Default Convention
  - Data Annotation Attribute
  - Fluent API

Data Annotation Attribute

- *System.ComponentModel.DataAnnotations* namespace

```
[Table("StudentInfo")]
public class Student
 {
    public decimal Height { get; set; }
    [Key]
    public int SID { get; set; }
    [Column("Name", TypeName = "ntext")]
     [MaxLength(20)]
    public string StudentName { get; set;}
    [NotMapped]
    public int? Age { get; set; }
    [ForeignKey("Grade")]
    public int GradeId { get; set; }
    public virtual Grade Grade { get; set;}
 }
```

Data Annotation Attribute

- Common Data Annotation

Attribute	Description
Table	Applied on entity class to give a name to database table
Column	Applied on a property to give column name, order and data type
Key	Sets the property as primary key for the table.
ForeignKey	Applied to a property to mark it as foreign key
NotMapped	Can be applied to entity class or property for not generating a corresponding table or column in the database.
MaxLength	Sets the max length for the table column
Required	Can be applied on properties to make the corresponding column on the table as not null

Fluent API

- Fluent API has higher precedence than conventions and data annotations.
- used to configure domain classes to override conventions
  - Overriding OnModelCreating (...) Method
    - Calling DbModelBuilder Methods

Fluent API

```
public class SchoolContext:DbContext
{
    public SchoolContext():base()
    {

    }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
            //Write Fluent API configurations here
        modelBuilder.Entity<Teacher>()
                .ToTable("TeacherInfo");
    }
}
```

Fluent API

- Model Configuration
  - Configure mapping to database
- Entity Configuration
  - Configure Primary key , Table name , one to many, etc
- Property Configuration
  - Configures property to Column mapping Column Name , Default value, Data Type, etc

Fluent API

- Model Configuration

Fluent API Methods	Usage
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

- Entity Configuration

Fluent API Methods	Usage
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.
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

- Property Configuration

Fluent API Methods	Usage
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.

Fluent API

- Property Configuration

Fluent API Methods	Usage
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.

Fluent API

- Property Configuration

Fluent API Methods	Usage
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.

Fluent API

- Has/with pattern
  - One to Many
  - One to One
  - Many to Many

Fluent API

- One to Many
- Done Using one of two pattern
  - HasOne - WithMany

```
 protected override void OnModelCreating(ModelBuilder modelBuilder)
{
     modelBuilder.Entity<Employee>()
        .HasOne(e=>e.Department) // Each Employee has one Department
        .WithMany(d=>d.Employees) // Each Department has many Employees
        .HasForeignKey(e => e.DeparmentID) // FK on Employee
        .OnDelete(DeleteBehavior.Cascade);
}
```

Fluent API

- HasMany - WithOne

```
 protected override void OnModelCreating(ModelBuilder modelBuilder)
{
     modelBuilder.Entity<Department>()
        .HasMany(d=>d.Employees)
        .WithOne(e=>e.Department)
        .HasForeignKey(e => e.DeparmentID)
        .OnDelete(DeleteBehavior.Cascade);
}
```

Fluent API

- One to One
- Done with pattern
  - HasOne – WithOne

```
public class EmployeeAddress
{
public int EmployeeAddressId { get; set; }
public string Address { get; set; }
public string City { get; set; }
public string State { get; set; }
public string Country { get; set; }
public int AddressOfEmployeeId { get; set; }
public Employee Employee { get; set; }
}
```

```
public class Employee
{
 public int Id { get; set; }
 public string Name { get; set; }
 public EmployeeAddress Address { get; set; }
}
```

```
modelBuilder.Entity<Employee>()
.HasOne(e=>e.)
.WithOne(a=>a.EmpEmployeeAddressloyee)
.HasForeignKey<EmployeeAdress>(a => a.AddressOfEmployeeId);
```

Fluent API

- Many to Many
- Done with pattern
  - HasMany – WithMany
  - UsingEntity

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    // Collection navigation property
    public IList<Teacher> Teachers { get; set; }
}
```

```
public class Teacher
{
    public int Id { get; set; }
    public string Name { get; set; }
    // Collection navigation property
    public IList<Student> Students { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Teacher>()
         .HasMany(t => t.Students)
         .WithMany(s => s.Teachers)
         .UsingEntity(j => j.ToTable("TeacherStudent")); //Specify the join table name
    }
```

Stored Procedure

- Creating Stored Procedure
  1. Create Migration
  2. Modify up method in it

```
//in migration version class -->up method
public override void Up(MigrationBuilder migrationBuilder)
{
     migrationBuilder.Sql(@"
            CREATE PROCEDURE
                GetAllProducts
                AS
                BEGIN
                SELECT * FROM Products;
                END
                ");
}
```

Thank You