TASK 4

[Document subtitle]

LinkedIn Article :

https://www.linkedin.com/pulse/mastering-navigation-properties-entity-framework-core-mohamed-afify-9xuxf/

Self study:

Tracking vs. AsNoTracking

The Core Idea

DbContext in EF Core has a Change Tracker — think of it like a notebook where EF writes down every entity it loads from the database and tracks its state (Unchanged, Modified, Deleted).
This lets EF Core generate UPDATE/DELETE SQL automatically without you writing SQL yourself.

Default Behavior — Tracking

When you load data normally:

var emp = context.Employees.FirstOrDefault(e => e.EmpId == 1);

- The returned entity is tracked automatically.
- This means the DbContext now knows about this entity and is watching for changes.
- If you change any property:

emp.Name = "Ali Updated";

- The entity's state automatically changes to Modified.
- When you call:

context.SaveChanges();

- EF Core generates SQL like:
- UPDATE Employees SET Name='Ali Updated' WHERE EmpId=1;

— without you needing to write SQL.

AsNoTracking — No Tracking

When you use AsNoTracking():

var emp = context.Employees.AsNoTracking()

.FirstOrDefault(e => e.EmpId == 1);

- The returned entity is not tracked (Detached) from the beginning.
- The DbContext doesn't know about it and won't record any changes or deletes.
- If you change a property:

emp.Name = "Ali Updated";

- The entity stays Detached, and no automatic update will happen on SaveChanges().

Updating a Detached Entity

If you want to update an entity that's Detached, you have to tell EF Core manually that it's modified:

context.Entry(emp).State = EntityState.Modified;

context.SaveChanges();

This tells EF Core to generate an UPDATE even though the entity wasn't tracked originally.

Quick Comparison Table

Feature	Tracking (default)	AsNoTracking() (explicit)
Entity tracked?	Yes	No
Default state	Unchanged (→ Modified on property change)	Detached
Automatic updates?	Yes	No (must set state manually)
Performance	Slightly slower (because it tracks)	Faster for read-only scenarios
Best for	Reading + editing/updating	Read-only data or large result sets

---

One-to-One Relationship

A one-to-one means:
Each record in Table A matches only one record in Table B, and vice versa.

Example:

- Employee table contains general employee data.
- EmployeeAddress table contains exactly one address for each employee.

Employee

```
public class Employee
{
    public int EmpId { get; set; }
    public string Name { get; set; }
    // Navigation Property
    public EmployeeAddress Address { get; set; }
}
```

EmployeeAddress

```
public class EmployeeAddress
{
    [Key]
    public int EmpId { get; set; } // Primary Key AND also FK to Employee
    public string Street { get; set; }
    public string City { get; set; }
    // Back navigation
    public Employee Employee { get; set; }
}
```

Fluent API Configuration

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   modelBuilder.Entity<Employee>()
      .HasOne(e => e.Address)          // Employee has one Address
      .WithOne(a => a.Employee)        // Address belongs to one Employee
      .HasForeignKey<EmployeeAddress>(a => a.EmpId); // FK in EmployeeAddress
}
```

Here:

- Each Employee has one Address.
- Each Address belongs to one Employee.
- EmpId in EmployeeAddress is both PK and FK to Employee.

Self-Reference (Table pointing to itself)

A self-reference means a table has a foreign key to itself.
Example:

- Each employee may have a manager who is also in the Employee table.
- One employee can manage many employees.

Employee with Manager/Subordinates

```
public class Employee
{
    public int EmpId { get; set; }
    public string Name { get; set; }
    // FK pointing back to same table
    public int? ManagerId { get; set; }
    // Navigation property to the manager
    public Employee Manager { get; set; }
    // Navigation property to employees under this manager
    public ICollection<Employee> Subordinates { get; set; }
}
```

Fluent API for Self-Reference

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{modelBuilder.Entity<Employee>()
        .HasOne(e => e.Manager)          // Each employee has one manager
        .WithMany(m => m.Subordinates)    // Manager can have many subordinates
        .HasForeignKey(e => e.ManagerId);  // FK in the same table
}
```

Here:

- ManagerId is a FK referencing EmpId in the same table.
- You can navigate Employee.Manager (to find the boss) or Employee.Subordinates (to find all employees under that boss).
