Databashantering

Dapper vad är det?

Code first? Database first?

Normaliseringsformer

- Första 1NF
    a. Primär nyckel är ett måste, id eller email.
    b. Inga flervärdiga attribut i en kolumn/fält
- Andra - Separera ut dublett liknande fält i andra relations tabeller
- Tredje - Kolumnerna måste tillhöra namnet av tabellen, t.ex Customers får ha, namn lastname, age, men inte email, streetname, cat etc.

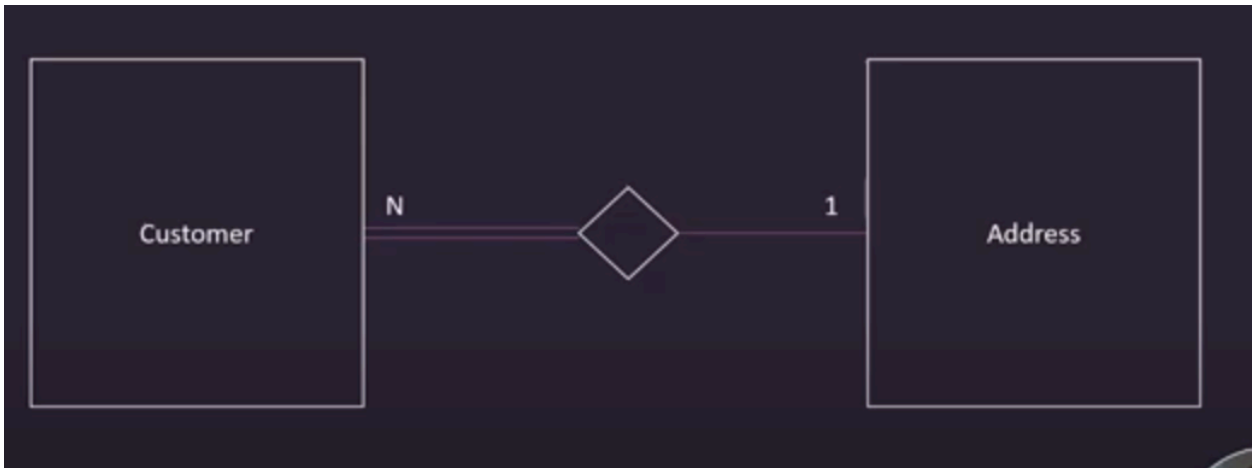
ERD - Entity-Relationship diagram

Konceptuell modellering

Användning av Entiteter.
Oftast ID, Primär Nyckel. PK
Relations tabeller har Foreign Key FK, Främmande nyckel

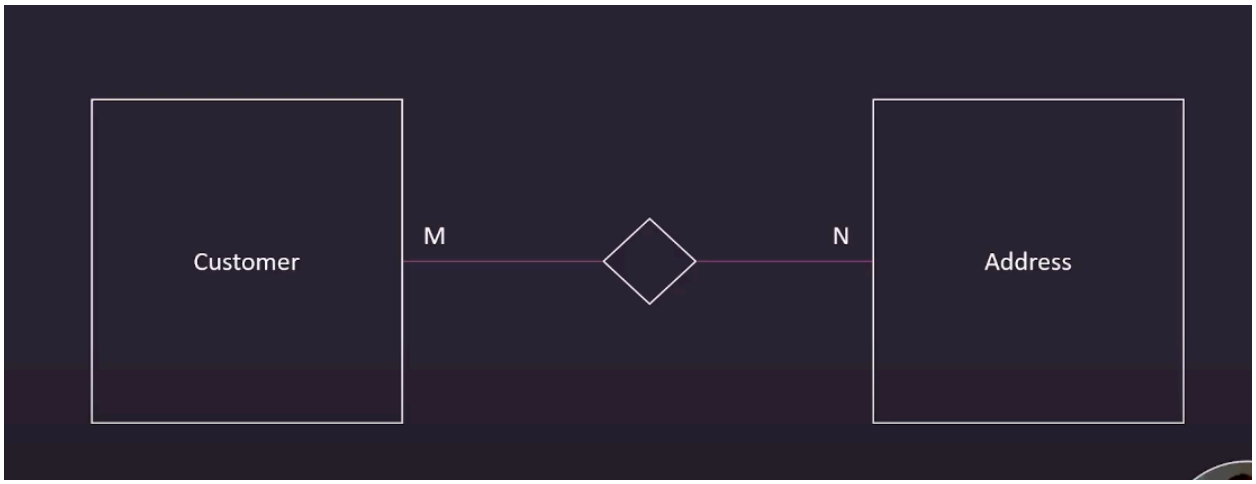
Must have, not must have, enkel linja mellan relationen, eller vid måste så har man dubbel

1:an betyder att Customer måste ha 1 adress. N (Numerous) betyder en eller flera. Så adress kan ha flera customers.



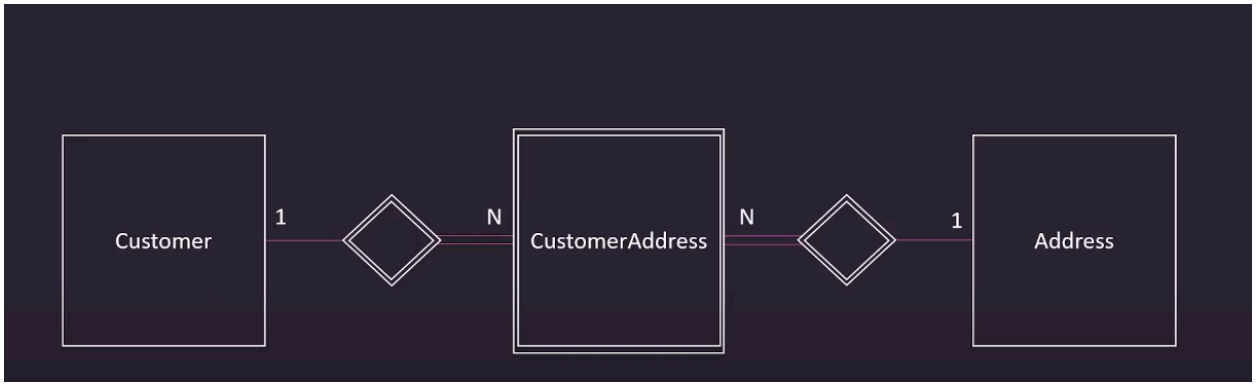
Bilder tyds: Customer måste ha relation till minst 1 adress, adress kan ha relation till en eller flera customers.

Many to many
En kund kan ha flera adresser, flera kunder kan ha samma adress.

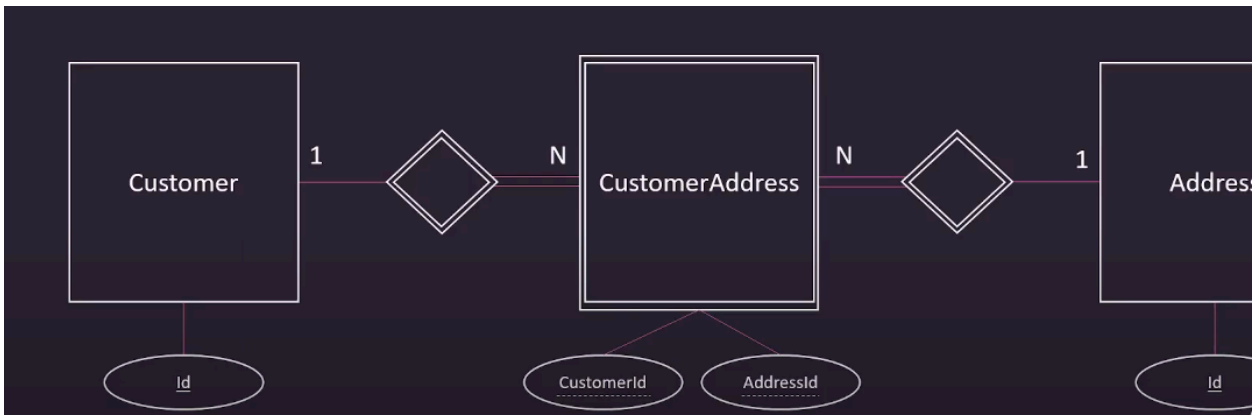


Bilden tyds: Inga måsten men både kan ha flera av varandra

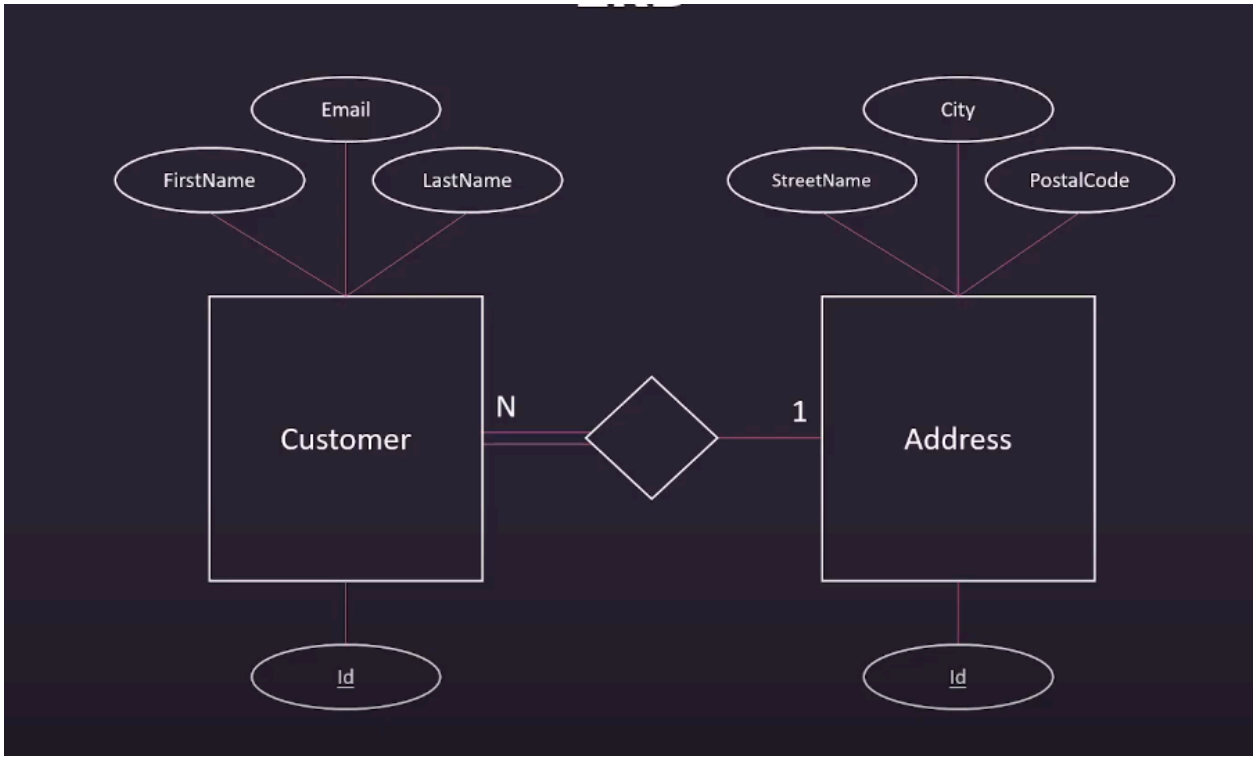
.Viktigt. Men vi ska inte använda many-to-many. Utan då ska vi skapa en mellan relations tabell mellan dessa.



Dessa kopplings/länk tabeller behöver vi då skapa för att ta bort many to many relationerna, utan här istället får vi CustomerAdress måste ha en adress och måste även ha en Customer som sammasätts i samma fält eller kolumn. På så sätt ha vi en tabell med customers och en med adresser men vi ha en Customeradress som är en tabell som visar vilka customers som har vilka adresser. **Many-to-many = Kopplingstabell,** CustomerAdress måste ha en kund och måste ha en adress.



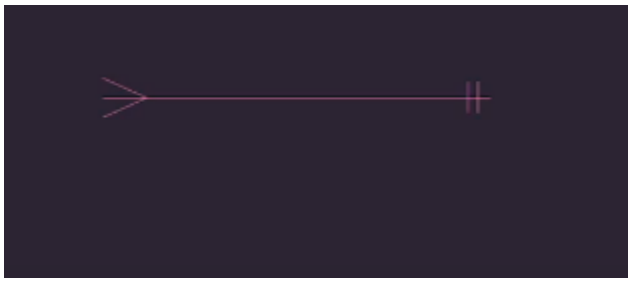
Inuti kopplingstabellerna använder vi oss av namngivna ID’s så vi kan separera dem.



När det sen inte bhövs en many to many utan kunden bara berhöver en adress eller bara ska ha en adress, så fungerar detta utmärkt.vilket då kallas ERD

Chrow’s foot

Chrow’s foot visar relationerna mellan erd tabellerna och man kan ha chen notations. Detta är då strecken mellan tabellerna som man gör i antigen chrows eller chen.



Sammanfattningsvis använd ERD med chrow's foot notations.

Textbaserade lagrings värden.

- **Char(n) n = 1 byte :** fixerat värde man väljer hur många tecken som får användas. Perfekt vid t.ex landskoder med mera. Char(2) t.ex. Enbart engelska tangettecken och använder oftast utf 8 pga minskad bytes eller bitar.

- **nChar(n) n = 2 bytes :** Unicode värde, detta ger möjlighet till specialtecken till skillnad från char, men fortfarande fixat värde nChar(2) vid landskoder t.ex men åäö går nu också. Utf 16 denna är då två bytes per tecken så landskoder kostar 4 här vs 2 i char.

- **varChar(n) n = 1 byte + var = 1 :** Dessa kostar 1 bit per tecken + en bit för att köra den dynamiskt, vilket gör denna ganska billig ändå, dock inga special tecken men olika längder som sagt.

- **nVarChar(n) n = 2 byte + var = 2 bytes :** Denna kostar 2 bit men även 2 extra för dynmiskt, då går alla special tecken också att använda och är den flexiblaste vid t.ex namn eller efternamn osv.

	Värde	Antal Tecken	Antal Bytes
char 6	123 57	6	6
nchar 6	123 57	6	12
varchar 6	123 57	6	7
nvarchar 6	123 57	6	14

- **varBinary(n) n = 1 byte + var = 2 bytes. :** Lagrar bilder, filer och up till 2gb data. Men rekomenderas att använda bild servrar och sedan lagra sökvägarna till serverna.

Bool, Heltal och decimaltalsvärden

Bit - 1 byte : 0 till 1
Tinyint - 1 byte : 0 till 255
Smallint - 2 bytes : -32768 till 32767
Int - 4 bytes : -2147483648 till 2147483647
Bigint - 8 bytes : -9,223,372,036,854,775,808 till 9,223,372,036,854,775,807

Float(p) - 4-8 bytes : om p är mellan 1-4 så 4 bytes, 25-54 så 8 bytes
Real - 4 bytes : Samma som float(24)
Decimal(p, s) - 5-17bytes : A: **p = 1-9**: 5 bytes
B: **p=10-19**: 9 bytes,
C: **p=20-28**: 13 bytes,
D: **p=29-38**: 17bytes
Money : -922337203685477.5808 till 922337203685477.5807

Notiser
1. 0 går som positivt värde
2. I relationstabeller viktigt at samma datatyp används i alla.

One to many
dbdiagram.io

SQL Tutorial

SQL ramverk(presistence framework) - SqlConnection, SqlCommand, SqlDataReader

Database first, code first, !model first

Val av Databas : Microsoft SQL

Viktigt att ha med

Chen, chrows ritningar, välj mssql, ge dig på svårare sql syntaxer och sorteringar, och döå dem till, sortead efter x, eller UsersCity, CustomersPhoneAtId osv.

Försvann kraven på databas first/Code first? Behöver enbart välja? Punkt 5 blev Code-first, men ingen databas first i väl godkänd, utan databas logik på sql blev det mer fokus på och transaktionshantering med rollback. Kolla mer på detta. Databas först är bra att följa när man vet vad som ska finnas sorterat i databasen,

Men Kundbeskrivning > Flödesdiagram > klass diagram > databas diagram är agilt?

Kundbeskrivning > Flödediagram > Databasinnehåll > klass diagram känns också trevligt, klass diagrammet blir lättare att skriva.

CodeFirst , Ritning av entiteter av i klasser, ritning i tabeller, Klassdiagram över service methoder, CRUD baserade methoder. Kör dynmaisk MsSqlRepository

Använd Async på databas anrop alltid.
Använda Dispacher? Hur, när, värt?
WhenAll methoden vid Async.

Nugets - Tools, design, Sqlserver

Business - CoreFiles + Logic (Using .NET)
Data - Entities + Database communication (Using Entity Framework +- core)
Presentation - Frontend (Web, WPF, Maui)

Använd Sammansatta nycklar

OncreationModel overide på DbContext

Så vi har ProjectService som har tabellen prekt, tsm med tänk att man ska logga in Admin, Vustomer, Employee

Admin ska kunna se Projekten och alla information från projektet alla customers och alla emplyees, alla tjänster erbjudna och deras status vem som beställde osv.

En customer ska logga in, se de propjekt hen är delaktig i, kunna beställa tjänster och får statusen ändrad av en employee,

Så Admin ahr ProjectService Employee har EmployeeService och Customer har CustomerService

Dock behöver vi ServiceSerrvice? Osv

Så mer eller mindre skapar Admin Projektet t.ex Bygget vid Midgård. Som är en skola.
Kan finnas flera kunder relaterat till detta projekt, men oftast används nog en.
Dom kan under projektets gång göra beställningar på saker att ta bort/lägga till eller ändra på bygget.

Employees kan finnas Snickaren, Eljöänsterna, Städarna, osv, när någon customer lägger till tjänst eller beställning kan employee ändra statusen, start/slutdatum sätts èller ändras av Admin.

Cirka struktur
Så vi kommer typ behövar en ProjectService som har ProjectRepo, CustomerRepo,EmployeeRepo som kommunicerar med DatabasComFile(DatabasRepo)

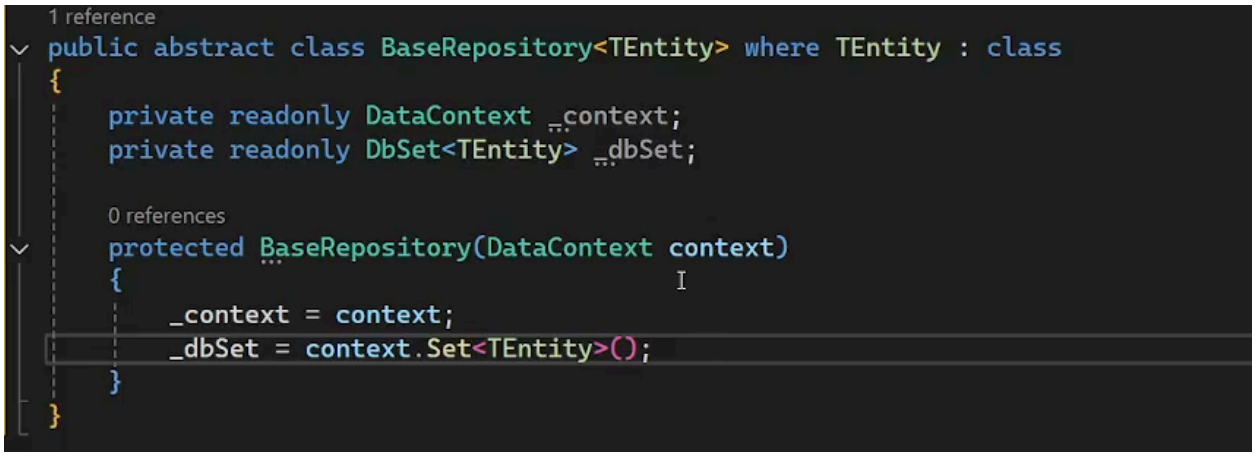
Vi använder en basrepo för Project,Customer,Employee, sedan gör vi en en generisk version av databasrepo som kan ta värden oavsett från vilken repo de kommer ifrån,

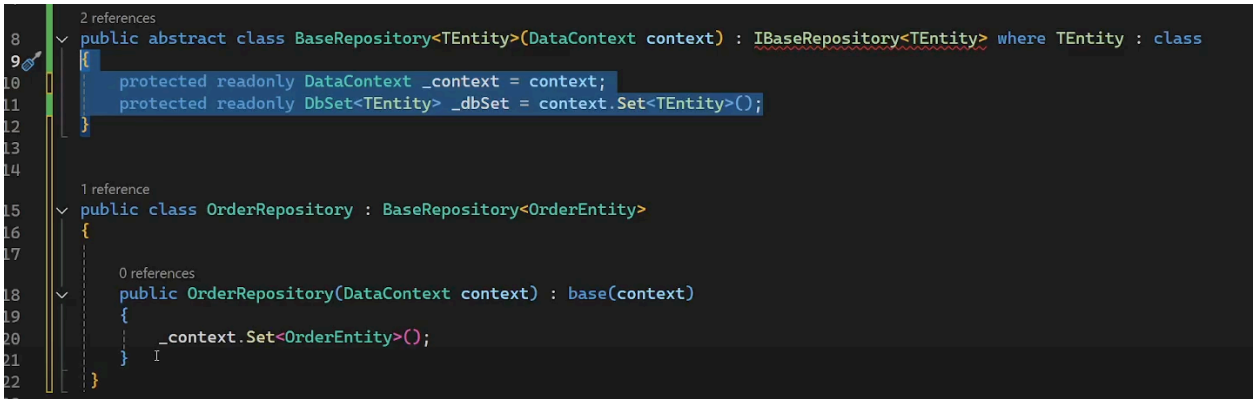
Då får vi 1 service med 3 repos oc hen databasfil för admin

Vi har sedan

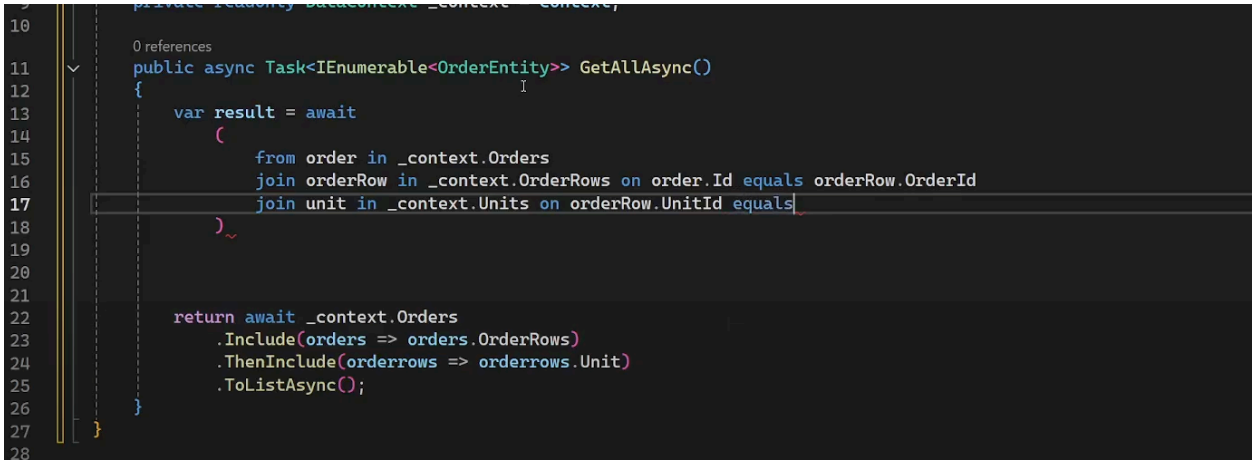
CustomerService > OrderRepository > DatabasComFile
Denna kommunicerar customers med för att påverka databasen

EmployeeService > ConfirmationRepository > DatabasComfile
Denna kommunicerar employees med för att kommunicera med databasem





Basklass



```
0 references
public async Task<IEnumerable<OrderEntity>> GetAllAsync()
{
    var result = await
        (
            from order in _context.Orders
            join orderRow in _context.OrderRows on order.Id equals orderRow.OrderId
            join unit in _context.Units on orderRow.UnitId equals
        )


    return await _context.Orders
        .Include(orders => orders.OrderRows)
        .ThenInclude(orderrows => orderrows.Unit)
        .ToListAsync();
}
}
```

Eager Loading betyder att den **förladdar relationstabellerna** i samma databasfråga. Det innebär att när du hämtar huvudentiteten (t.ex. en order), laddar den också **ivrigt** relaterade data (t.ex. kunder och orderrader) direkt, utan att vänta tills du faktiskt använder dem. Detta ökar laddningstiden men kan de mer information samtidigt.

Lazy loading latmans varianten, tillskilland från eager struntar i inkluderingar och gör det auto.

Lazyloading entiteter som inkuderas i andra entiteter måste vara virtual vid lazyloading

N+1-query-problem

Modell/Order har oftast en klass, Entitierna i databsen är oftast uppdelat i flera klarr som entitieter per tabell som ska skapas.

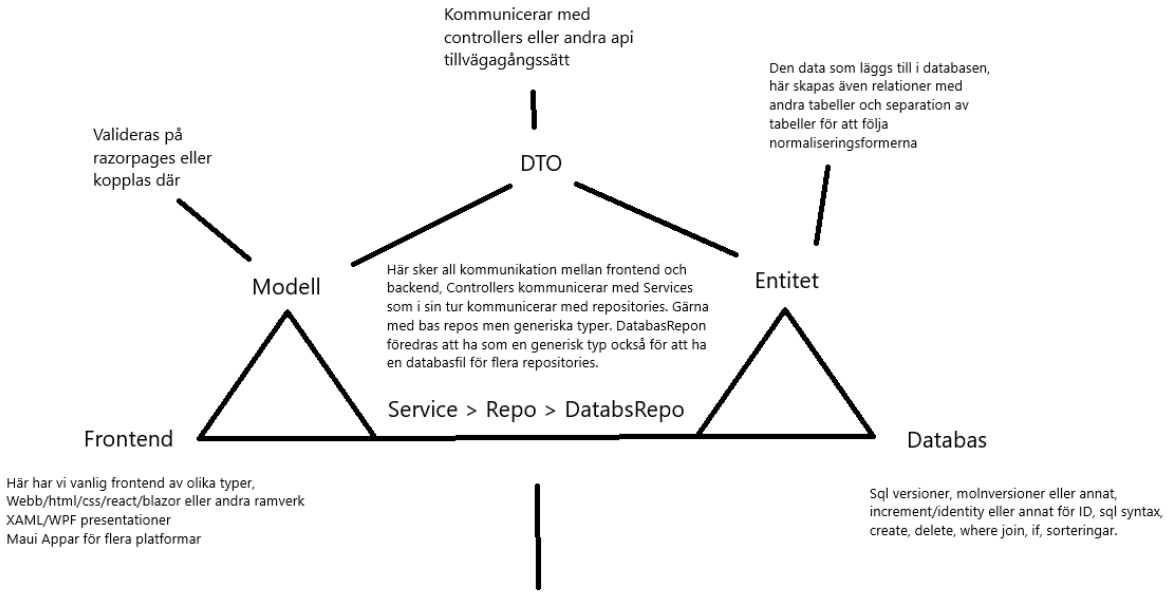
IResult - Generisk respons meddelende. Använda bool på alla methoder i basrepon? Utförlogik men retunerar ett bool värde? Agerar void i en return method.

Alltid try catch, tänker det är felmedeladet som följer med,

Sha256 kryptering, saltning är viktigt så inte 2 personers lösenord blir samma kryptering.

Så en clean aktiktektur som kommunicerar med databasen och som använder try/catch och IResult för att hantera meddelanden vid fel.

IQueryable/IEnumerable



Applikations lösning

Applikations lösningen innehåller oftast projekt typerna **Presentation, Business, Data.**

Presentation har vi frontend delarna de innehåller antigen webb/wpf/Maui här ahr vi modeller som valideras direkt på sidan men även controllers som ska kommunicera med projektets andra delar.

Business har vi logiken, där har vi DTO, Services, Repositories, Factories etc.

Data innehåller entiteter, migartioner med databas first, relationskopplingar och DatabasRepositoryn/DatabasCommunicationFile

Projekt strukturen.

Nuvarande planering är att få gjort en sqldatabas i verktyget som vi uppdaterar under tiden. Vi behöver gör sedan en Chen och en chrows notation av databs modelleringen. Vi ska följsa

normalisering reglerna i sql strukturen.

Vi bygger en aktiketur som kan följa uppgiftens projekt, ProjectService, UserService, CustomerService med tillhörande logik sen vidare till data som som har repositories som kommunicerar med databsfilen i data.

Vi kommer ha modeller, frontend, controllers i Presentation, dtos, repositories, baserepository i Business och entities, migration, dbcontext i Data.

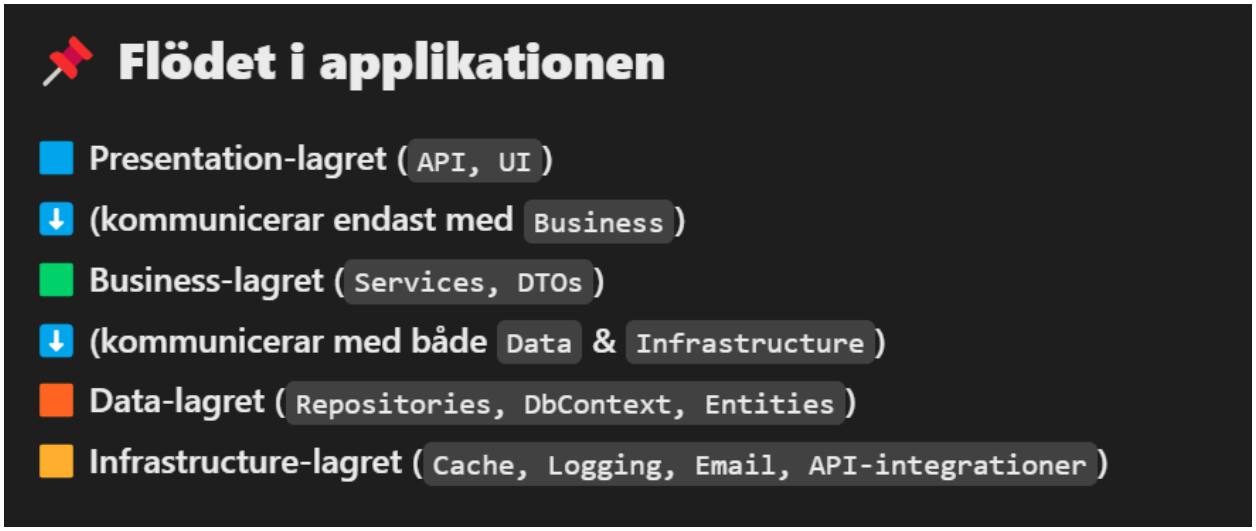
Viktiga noteringar är att vi ska använda oss av imemory chaching, eagerloading, try/catch och IResult,

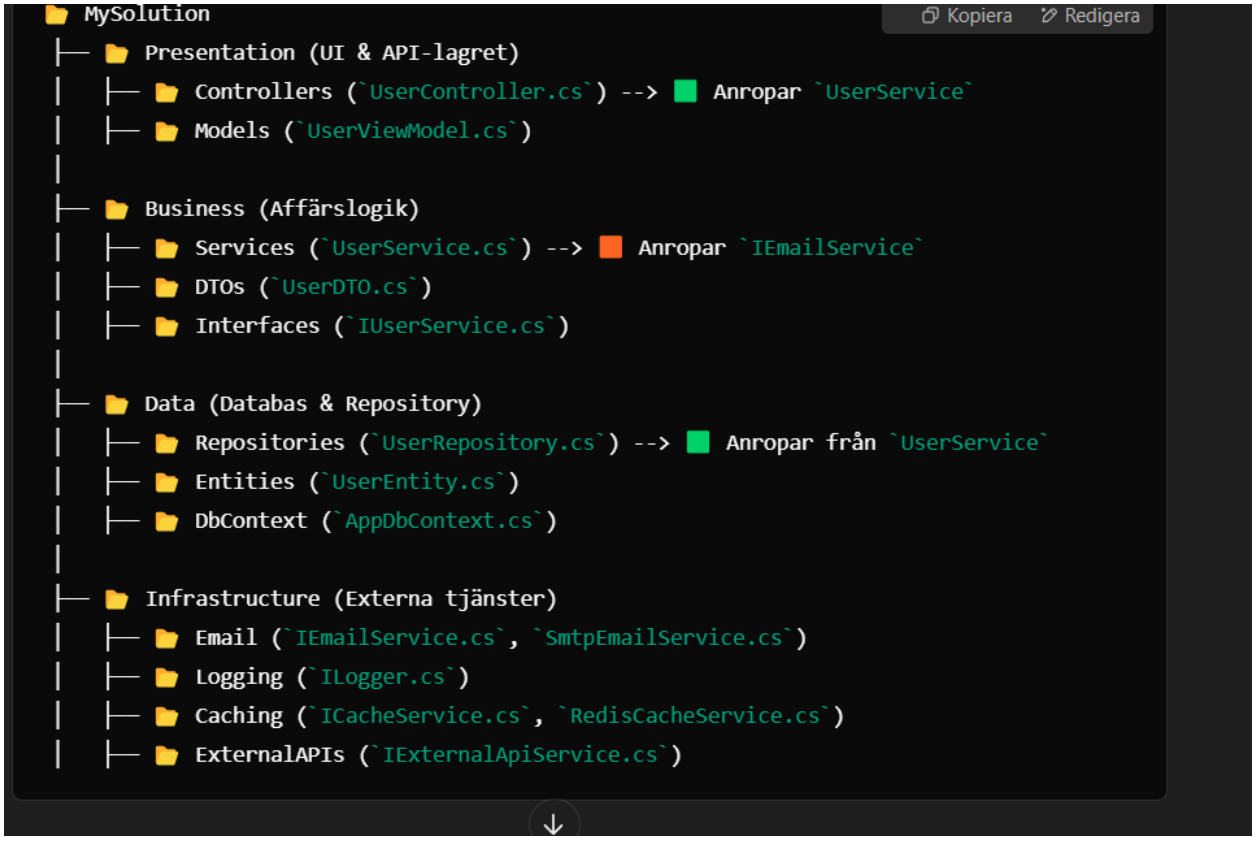
Som frontend lösning? Kör vi på bootstrap, razor pages med modeller och controllers?
Vi kör nog aspo.net kommer i nästa kurs.
Asp.net core web app eller`mvc?
Vi kör på mvc och bootstrap simpel ui

Ännu inte tittat på
enhetstester oc htitta på det där med InMemeryDatabase för test och vi måste göra applikationen i databas först. Dapper och sql client

Vi ska även titta på transaktionshantering med rollback och lösning av konkurrenskonflikter, för att säkerställa datakonsistens och hög prestand

Cache, inmemory chache, spara i lista efter hämtning i **X** tid. Hämta informationen från chache om existeetrar och annars dtatbasen + lägga till i chache.





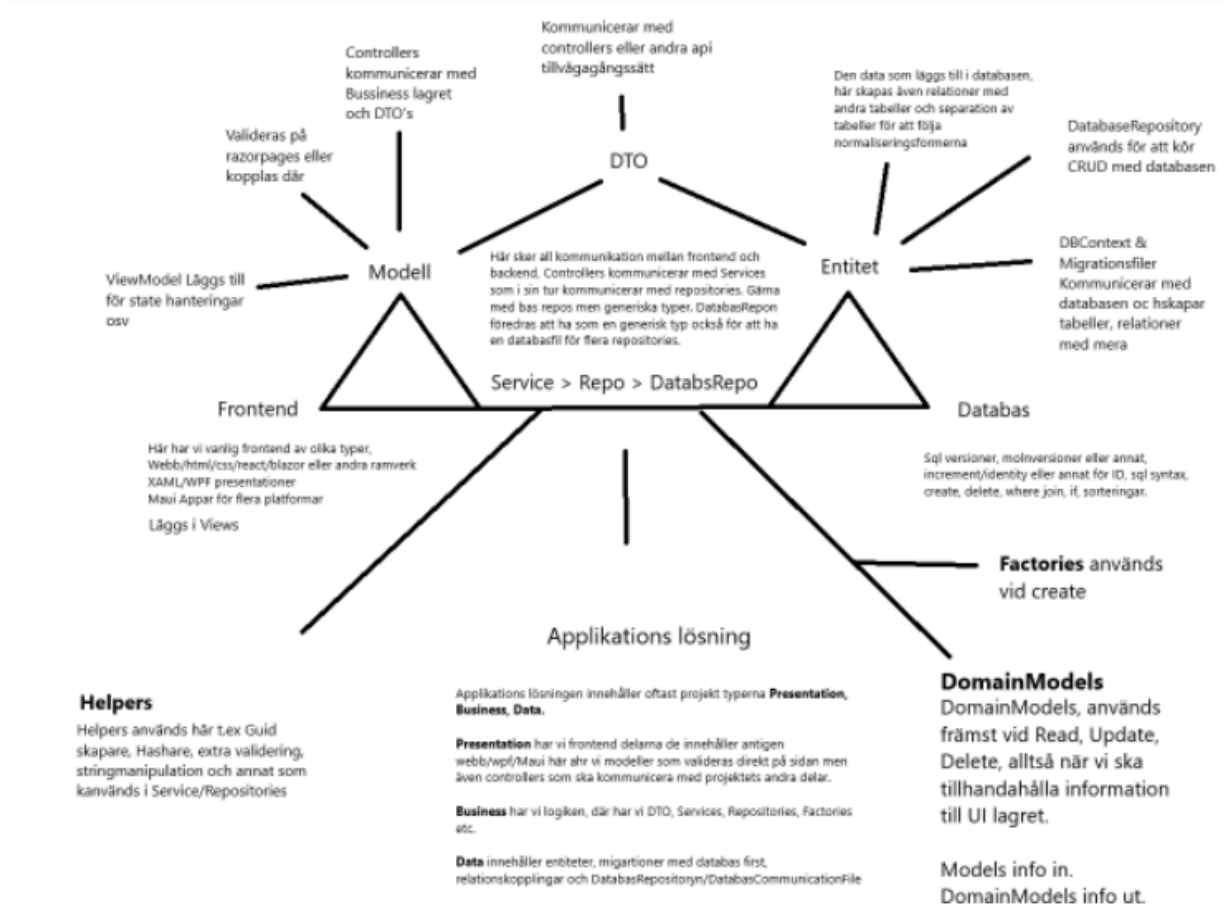
Så har vi IResult i Infrastructure som används i repo och skickar infon till services vidare till presentations lagret så chachar vi appen, loggar, skickar med result.

LINQ och **Dapper** som databas förfrågningstekniker.

Utvecklare flöde > Kolla på kraven från bilden, skapa databas tabeller utifrån det som följer normalisering formerna, “skapa de entiteter och skapa migration” så vi kan kontrollera databasen så de finns rätt saker osv även bra att fixa dbset.

Sen gör vi repositoryn? Så att vi enkelt bara gör DI på denb i servicen.

Vi ritar databasmodellenerna, sedan ritar vi upp de pages som ska finnas, sen börjar vi entititer > repositoryes med base > services sedan bygger vi pages och sätter ihopa allt.



Uppdaterad arkitektur.

1. Presentation Layer (Presentation/UI)

Presentation-lagret är ansvarigt för användargränssnittet och hur data presenteras för användaren. Här ingår alla vyer, formulär och kontroller.

- **Vad ingår här?**
  - **Controllers (MVC):** Hanterar användarens begärningar och skickar svar.
  - **Views (Razor Pages/Views):** Innehåller HTML, CSS och JavaScript för att visa data och interagera med användaren.
  - **ViewModels:** Representation av data som används i vyerna, ofta med specifika metoder eller hjälpmetoder som är användbara för visningen.
  - **JavaScript (och frameworks som React/Angular om det används):** Hantering av dynamiska användarinteraktioner och kommunikation med backend-API:er.
  - **Client-Side Validation:** Validering av användardata innan den skickas till backend.
  - **Styling (CSS, SCSS, TailwindCSS):** Gör applikationen estetiskt tilltalande.

    - **API-consumers (om det finns externa API:er):** Anslutningar till andra system eller externa tjänster.

Exempel:
I Presentation-lagret har du `ProjectController` som tar emot begärningar för att skapa, läsa, uppdatera och ta bort projekt. Den använder ViewModels för att kommunicera med vyerna och validerar data innan de skickas till Business-lagret.

2. Business Layer (Business/Services)

Business-lagret (eller Service-lagret) innehåller all affärslogik och är den mellanliggande nivån mellan presentation och datalager. Det ansvarar för att processera data och logik innan de skickas till datalagret eller tillbaka till presentation-lagret.

- **Vad ingår här?**
    - **Services:** Hanterar affärslogik och utför operationer som involverar en eller flera entiteter (t.ex. `ProjectService`).
    - **Factories:** Skapar instanser av objekt, särskilt när komplexa objekt måste byggas med specifik affärslogik.
    - **Interfaces:** Används för att definiera kontrakt mellan lagren, vilket underlättar enhetstester och abstraktion.
    - **DTOs (Data Transfer Objects):** Skickar data mellan olika lager eller mellan applikation och externa system utan att exponera interna detaljer.
    - **Business Rules:** All affärslogik som kontrollerar och processar data, exempelvis beräkning av priser, verifiering av användardata, hantering av projektstatus etc.
    - **Asynkrona processer:** Hantering av långvariga processer som nätverksanrop, databasoperationer, filöverföringar och liknande.

Exempel:
`ProjectService` kommer att hantera alla operationer relaterade till projekt (skapa, uppdatera, ta bort, lista) och använda olika metoder för att interagera med datalagret, till exempel att validera om ett projekt kan skapas eller om ett projekt har alla nödvändiga data innan det sparas.

3. Data Layer (Data/Database)

Data-lagret är där all data lagras och hämtas från. Det kommunicerar direkt med databasen eller andra datakällor (som externa API:er eller filer).

- **Vad ingår här?**
    - **DbContext (Entity Framework Core):** Används för att interagera med databasen via LINQ och generera SQL-frågor.

- **Repositories:** Hanterar CRUD (Create, Read, Update, Delete) operationer på databasnivå. Repositories kan vara generiska eller specifika för entiteter (som `ProjectRepository`).
- **Entities (Models):** De objekt som representerar databastabellerna i applikationen. Exempel: `Project`, `Customer`, `Order`, `Service`.
- **Migrations:** För att hantera förändringar i databasschemat.
- **Stored Procedures/Queries:** Hantering av specifika och optimerade SQL-frågor för komplexa operationer.
- **External Data Sources:** Om du hämtar eller skickar data till externa källor (API:er, externa databaser) kan detta hanteras här.

ViewModel > Controller > Service > DTO > Repository > Entity > Database

Så vi kan alltså göra hämntning via service i baserepository oc hgöra en sortering direkt i servicen eller så kan vi då göra nonbase repositories för specifika sorteringar med mera där vi då kan använda dapper eller linq för att göra specifika förfråganingar till databasen, så Read/Update separeras från baserepository så länge det inte är Updateall eller ReadAll.

Så även om det går att göra sorteringarna eller specifika ändringar direkt i servicen så gör vi de i repositorsen med dapper eller linq

```
/// <summary>
/// Hämtar alla projektledare som en lista av DTO:er.
/// </summary>
public async Task<IEnumerable<ProjectLeaderDTO>> GetAllProjectLeadersAsync()
{
    var leaders = await _projectLeaderRepository.GetAllAsync();
    return leaders.Select(l => new ProjectLeaderDTO
    {
        ProjectLeaderID = l.ProjectLeaderID,
        Name = l.Name,
        Email = l.Email,
        Phone = l.Phone,
        Department = l.Department
    }).ToList();
}
```

Vi flyttar alltså denna method från ProjectService in i ProjectRepository istället, kanske mer validerings logik isåfall.

Hmm…. vi kan alltså inte använda dapper och entity framewrok core samtidigt. Så om vi gör en EF anslutning i baserepository kan vi inte köra dapper i de repositories som arver från base då den behöver egen connection **IDbConnection** än **DbContext**

Föratt uppnå

1. Redogöra för logiska strukturer inom SQL.
2. Skapa SQL-kod genom att använda vanliga SQL-syntaxer

Så behöver vi skriva vanlig sql kod, detta gör EF automatiskt när vi kör SaveChanges(), FirstOrDefault(), Where(), Remove(), Update() osv.

Det vi kan göra är att göra en nonbase repo så det körs med dapper, eller så skriver vi basrepos med SqlRaw så vi kan skriva sql syntaxer med EF istället.

Val: SqlRaw med vanliga syntaxer så vi kan behålla O:et i solid, så vi kan behåla arven på repositoriesen men ändå använda Sql syntax.

Nytt val, skriv sqlraw so mkommentarer brevid

Preics som baserepositor ykan vi ha basecontext och koppla till flera databser samtidigt, möjliga situationer där man inte bara ska byta kan vara att appen är byggd med en viss databas men vi vill ändå ska ny, så man kopplar bara sqllite och mysql t.ex.

Viktigt ! Vid Database first.
Scaffold database, install-page sqlserve och tools som vanligt.
Scaffold-DbContext “” Microsfot.EntityFrameworkCore. SqlServer -ContextDir Contexts -Context AppDbContext - OutputDit Entities
Start “connection strings” > directy namn på mapp, name på fil, map för entity,

Pga Mvc versions val av hemsida så ActionResult ist för IResult.-

InMemoryCaching i Business lagret tsm med Services.