Projektdokumentation: CRUD-baserad lösning med Clean Architecture och SQL Server

1. Inledning

Detta projekt har genomförts inom ramen för kursen i datalagring, där vi har utvecklat en CRUD-baserad applikation kopplad till en **Microsoft SQL Server (MSSQL)**-databas. Projektet följer principerna för **Clean Architecture** för att säkerställa en tydlig separation mellan olika lager i applikationen.

Målet med projektet var att skapa en **välstrukturerad applikation** där datahantering sker på ett **effektivt och skalbart sätt**, samtidigt som vi implementerar **Entity Framework Core (EF Core)** för att hantera databaskommunikation.

---

2. Clean Architecture

Vi har följt principerna för **Clean Architecture** för att skapa en **modulär och lättunderhållen kodbas**. Arkitekturen består av följande lager:

1. **Presentation (UI-lager)** – MVC-baserad frontend där användaren interagerar med applikationen.
2. **Business (Tjänstelager)** – Hanterar affärslogik, validering och datatransformation med **DTOs och Services**.
3. **Data (Datalager)** – Innehåller databaskoden, inklusive **Repositories, DbContext och databasmigreringar**.

Genom att dela upp koden i dessa lager blir den **enklare att testa, underhålla och vidareutveckla**.

---

3. Datalagret (Data Layer)

Datalagret ansvarar för kommunikationen med **MSSQL-databasen** via **Entity Framework Core**. Vi började med att:

- **Skapa en `AppDbContext`** – Detta fungerar som en brygga mellan applikationen och databasen, där vi definierade våra entiteter och relationer.
- **Definiera entiteterna** – Vi skapade entitetsklasser som representerar tabeller i databasen.
- **Migrera databasen** – Genom att använda **EF Core Migrations** genererade vi tabeller i **SQL Server** baserat på våra entiteter.
- **Bygga ett generiskt `BaseRepository<T>`** – Detta repository kan hantera **CRUD-operationer** med hjälp av **LINQ och expressions**.

Efter att grundstrukturen var på plats byggde vi ut repository-lagret med specifika repositories för olika tabeller, exempelvis `ProjectRepository`.

---

4. Affärslagret (Business Layer)

Affärslogiken hanteras i **Services-lagret**, där vi bearbetar data och validerar affärsregler innan informationen skickas till databasen eller presentationen.

DTOs (Data Transfer Objects)

- DTOs används för att **mellanlagra och transformera data** mellan presentation och datalager.
- Vi skapade **DTOs utifrån ViewModels**, bearbetade informationen och mappade sedan om den till entiteter i databasen.

Tjänster (Services)

- Huvudlogiken för CRUD-operationer ligger i services, där vi bl.a. använder en **Factory Pattern** för att skapa och hantera DTOs.
- **ProjektService** är den mest omfattande servicen.
- Vi har även andra **mindre services** som hanterar specifika delar av databasen, exempelvis **CustomerService och OrderService**.

Affärslagret fungerar som en **brygga mellan presentation och data**, och ser till att rätt data skickas vidare i systemet.

---

5. Presentationslagret (Presentation Layer)

Presentationslagret är en **MVC-applikation** där vi har byggt upp en **enkel och responsiv UI** med **Bootstrap**.

- **Controllers** tar emot förfrågningar från användaren och kommunicerar med **Business Layer**.
- **ViewModels** används för att hantera information i gränssnittet och kopplas till våra **Razor Views**.

Sidor vi har byggt:

- **Startsida** – En översikt över systemet.
- **Skapa projekt** – Formulär för att skapa ett nytt projekt.
- **Visa projekt** – Lista med alla projekt i databasen.
- **Redigera projekt** – Sida för att uppdatera befintliga projekt.
- **Admin-sida** – Hanterar tjänster, projektledare och kunder.

All datainhämtning och manipulation sker via **controllers**, som skickar och hämtar data till/från **Business Layer**.

---

6. Sammanfattning och Reflektion

I detta projekt har vi fått en **djup förståelse för Entity Framework Core** och hur databaslagring fungerar i en **Clean Architecture-struktur**.

Vi har lärt oss:

- **Hur EF Core fungerar i praktiken**, från entitetsdefinitioner till migrering och databashantering.
- **Hur man bygger en skalbar arkitektur** där varje lager har ett tydligt ansvar.
- **Hur DTOs används för att separera databasen från presentationen.**
- **Grundläggande SQL-frågor och mer avancerade read-operationer.**

Vi ser att **Read-operationer** ofta blir de mest komplexa och kräver **optimering**. I framtiden skulle vi kunna använda **Dapper** för att hantera tyngre databasfrågor mer effektivt.

Reflektion kring ViewModels

Under projektets gång märkte vi att det kan vara **onödigt att separera ViewModels från entiteterna i alla fall**. **ViewModels passar bättre direkt till formuläret**, eftersom de ofta **liknar entiteterna** och ändå behöver innehålla samma fält.

I framtida projekt skulle vi kunna **förenkla vår arkitektur** genom att använda ViewModels mer direkt, istället för att konvertera dem genom flera lager om det inte är nödvändigt.

Slutsats

Clean Architecture är en **intressant och flexibel arkitektur** som ger en **stabil grund**, men vi ser också att det finns **stora förbättringsmöjligheter**, t.ex. att göra applikationen **mer tillgänglig och effektivare**.

Avslutningsvis inser vi att **moderna applikationer kräver genomtänkt arkitektur** och hantering av **datatyper, strukturer, controllers och visuella aspekter**. Genom att förstå dessa delar kan vi skapa **robusta system** som inte bara fungerar, utan också **bidrar till samhället genom att hantera data på ett effektivt och skalbart sätt**.

Källhänvisning

Den information och delar av koden som använts i detta projekt baseras på **Microsofts officiella dokumentation** samt har helt eller delvis genererats i samarbete med **ChatGPT**.

- Microsoft Docs: https://learn.microsoft.com/en-us/dotnet/
- ChatGPT har använts för att generera och förbättra kod samt dokumentation under projektets gång.

Detta har bidragit till att säkerställa korrekta implementationer och en tydlig struktur i projektet.