**Requirements:**

*Implement an API in C# which will manage Customers and Orders. No user interface is required, only an*

*API to create, update and delete Customers and manage Orders. Customer entity should contain: name*

*(first and last), address, postal code. Each Order should contain one or more Items, order date and total*

*price. Item should contain Product and quantity of product and Product should contain name and price.*

*▪ For persistence entity framework should be used and repository pattern.*

*▪ Also, unit of work pattern is required to be implemented.*

*▪ In addition to creating/editing/deleting it is required also to support iterating over the customer*

*orders by order date.*

*▪ Implement focusing on domain logic using Domain Driven Design.*

*▪ Segregate commands and queries using CQRS.*

*▪ Having unit tests in your project is a plus (preferably with NUnit)*

*▪ XML documentation of the API is welcome, too*

*NOTE: A short list of the assumptions that you made when designing/implementing the API would be*

*great.*

# Analysis Phase

*Implement an API in C# which will manage Customers and Orders.*

*Implement focusing on domain logic using Domain Driven Design.*

*Customer entity should contain: name*

*(first and last), address, postal code. Each Order should contain one or more Items, order date and total*

*price. Item should contain Product and quantity of product and Product should contain name and price.*

## Find the domain models

The initial step is to find based on requirements the business entities/models of our system.

**Customer entity**:

The Customer entity **should** contain: **name (first and last)**, **address**, and **postal code**.

**Order entity:**

Each Order should contain **one or more Items, order date** and **total price**

**Item entity:**

Order item should contain **Product** and **quantity of product**.

**Product entity**:

Product should contain **name** and **price**.

Also, from requirements we can extract relationships among models.

**An Order belongs to a Customer** / A customer can have multiple orders. (Customer to Orders – one to many)

**Each order contains one or more items** (Order to Items – one to many).

**Each Item refers to a Product.**

Based on these assumptions and build models.

## Persistence Layer (EF Core)

*For persistence entity framework should be used and repository pattern.*

We will use the EF Core 8 for Sqlite for this simple project. This layer will use the AppDbContext class which inherits from Microsoft.EntityFrameworkCore.DbContext.

Here we can configure the entities-models, relationships and Foreign keys.

Also we use the AppDbContext for migration and database creation.

### Repository Pattern

Using this pattern we try to create repository interfaces and their implementations to interact with the database through the DbContext. We will also implement the Unit of Work Pattern to manage transactions and ensure consistency when multiple repositories are involved.

Specifically

* Define the Generic Repository Interface (IRepository<T>).
* Implement the Generic Repository (Repository<T>).
* Create Specific Repositories (for Customer, Order, etc.).
* Integrate with Unit of Work (to ensure transaction management).

## CQRS Implementation (Commands & Queries)

In order to implement the CQRS pattern, we will break it down in several sub-steps:

1. Define Commands and Queries:

* Commands: Actions that modify the state of the system (create, update, delete operations).
* Queries: Actions that read data without modifying the system (fetching customers, orders, etc.).

2.Command Handlers: Handle the commands by performing the required operations using the repositories.

3. Query Handlers: Handle queries by returning data.

4. Add the MediatR library in order to implement the mediator pattern and simplify the communication between Commands, Queries, and Handlers.

## API Controllers

Build API Controllers in order to set up to handle customer and order operations via HTTP requests.

In addition controller utilized MediatR to send commands and queries from the controllers, keeping the logic decoupled and maintainable.