Anti-Pattern:

EF itself is a repository implementation and UoW. Anti-pattern means on top of this we use repository and UoW. Size of application becomes bigger.

For simple app, the above is an overkill.

Benefits in larger application.

* Avoids injecting DbContext to entire application.

No repository – direct access to db

Web server (kernel, IIS) -> DbContext (application layer) -> EF (DB)

With Repository (abstraction between controller and database context. Unit test can mock data to facilitate testing)

Web server -> app layer -> repo -> Uow -> EF

Web-> app llayer -> Mock repository.

[Domain Driven Design]

Domain Model:

Entities represent domain objects. In DDD, entities must not be “anemic”

Anemic entity objects are not real objects (thin person – anemic)

Domain entity pattern:

They should have properties (to maintain the state) and behavior - > attributes and methods. Fat objects :D

Order class, Order service to contain the logic and behavior (encapsulated)

Instead keep it in Order itself (Add method)

Aggregates ->

Describes a group of entites and behavior that can be trated as a cohesive item.

Composed of one entity called the root and multiple child entities.

Ensures consistency of the entire aggregate.

Customer and address -> address can’t be floating around. Address is a part of customer aggregate.

Look at your business how these can be implemented. Logical grouping of entities that need to work together.

Add method works on Order

Order item is child of Order.

Add method should not be allowed to work on OrderItem (child entities). It should work on Order aggregate.

We should never expose repository to work on child entities.

One aggregate root per directory.

Domain Model Layer -> Infrastructure persistence layer -> Data tier

Buyer aggregate and Order Aggregate -> Buyer Repo and Order Repo -> (same UoW layer for both) -> Tables in Db (Order, Buyer,

Best Practices:

* Have identity column as ID for each table. So Id can be kept in the bass class.
* Root element should be decorated with a marker interface (empty interface). It can even be an empty abstract class. Why -> To identify the role that certain objects are going to play in our application.

Order and OrderItem should be in a single aggregate and Order should be the root of that aggregate.

Marker is used to identify the root entities.

4 entities, 3 of which are aggregate roots.

Developers can work directly on Order, Product and customer but not on orderItem.

Now at the repository level, we can have T only accepts AggregateRoot. -> generic repository, InutiofWork, UnitOfWork. Through code the restrictions have been made for developers to use the aggregate root.

Domain driven design says that public setter should not be there. Private setters.

Public int k {get; set;}

Public Order k {get; set;} -> navigation property inside OderItem.

Do not expose collection navigation property as publicly modifyiable. (accessible lit type).

Order.orderitems = new list<orderitems> this throws an error. But order.orderitems.Add () can still be done. Or remove everything from this collection. We don’t want that to be allowed to the developer.

That’s why we need IReadOnlyCollection property with a get.

<http://quartzsystems.com/downloads/core3/patterns.txt>

For adding to the OrderItem collection -> AddOrderItem

Has been exposed.

All the rules related to the Order are under the Order class. (cancel, add)

AddOrder

* Takes care of creating a new Order.

Lets create a WebAPI -> for retrieving the data and adding to the Order.

Async add method because that’s the best practice. Infact everything should be Async. For now only Add.

<https://localhost:44359/api/orders/> -> will call both Get and Get(int id) ambiguity.

Hence to resolve this ambiguity – we have route attribute. [Route("{id}")]

CQRS :-

Based on CQS = Command query segregation

All methods that go ahead and act on the entities-> method is either responsible to return a state or change the state of the object. Not both.

Domain model should not be exposed to the front end or accept from there.

Order (give and take some properties need to be produced to the front end) – Use dynamic view mmodel or a view model.

Dynamic View model.

Include("Customer") 🡪 Include is a method in EF core. This can be done in demos, small apps but in enterprise level project have proper view model. Create a class and assign the domain values properties to this view model for the same of front end.

Difference between a command and event -> command if issued only a single receiver.

Events when published can have multiple receiver whoever is listening for that event.

UI App -> (Command – I want to create a new order)-> API interface (web api) -> (Command forwarded) -> Command Handler (It is responsible for calling the add method in Order) -> Domain Model and Infrastructure -> Infrastructure connected to Database.

We just need customerid and Orderitems to create a order. OrderItems need productid, qty. rate should be from the catalogue. This is the command issued from the UI. Information necessary for the processing the command for adding the order.

Each command should have a command handler -> class that is responsible for executing the command. It implements the single responsibility principle.

Command object as input.

CH Should not contain any business logic. Just enough logic for what to do when order is placed.

Result of CH is status of the operation or an exception.

Trigerring Command Handlers -> identifying the correct command handlers.

Manually its very easy. We need to wire up the command and the commandhandler -> this job is of a mediators -> they also link event and event handlers (these mediators).

UI App -> (Command)-> API interface (web api) -> Mediator-> Command forwarded to the appropriate CH -> Command Handler -> Domain Model and Infrastructure -> Infrastructure connected to Database.

Lot of Mediators available. Open source MediatR available for.net

Mediatr and Mediatr.DependencyInjection to the MVC application should be added. Nuget.

IRequestHandler and IRequest helps with the binding of the CH and command.

This mediatr services should be injected so that it is available to our Asp.net application. In start.cs – configureservices.

services.AddMediatR(Assembly.GetExecutingAssembly());

Assembly needs to be passed, or collection of assemblies. GetExecutingAssemply scans the MVC application what all are using the IRequest.

<https://localhost:44359/api/order/> - Post

{

"CustomerId":1,

"OrderItems": [

{

"ProductId" : 1,

"Quantity" : 2

},

{

"ProductId" : 3,

"Quantity" : 2

}

]

}