Repository Pattern

# Definition:

Repository: It mediates between the domain and data mapping layers, acting like and in-memory collection of domain object. Reference (patterns of Enterprise Application Architecture- By Martin Fowler’s)

# Benefits:

* **Minimizes duplicate query language**.
  + E.g.: Imagine in a few different places in our application we need to get the first 3 top-scoring students in Math. Without the repository pattern we will end you duplicating this query logic over & over in different places.
  + var topStudents = context.Students.Where(s => s.Sub == “Math”).OrderByDescending(s => s.Score).Take(3);
  + In situation like this, we can encapsulate this logic in repository and simply call a method like get top scoring students in Math.
  + var students = repostiry.GetTopScroringStudentInMath(student);
* **Decouples your application from persistence frameworks**.
  + In future if you decide to switch to a persistence framework you can do so with minimal impact on the rest of your application.
  + Now the first argument that comes here is that some developers say, “who would change the persistence framework of an application?”
  + The fact that this does not happen very often is not because it’s not desirable.
  + Persistence frameworks are constantly changing and evolving and if you want to have the freedom to explore a different persistence framework with minimal impact on your application that’s when you should use **Repository pattern.**

# In a Nutshell

* The theory behind, what is the repository in practice? In a nutshell, this is what your repositories should look like. They should look like a collection of objects in memory.
  + Add(obj)
  + Remove(obj)
  + Get(id)
  + GetAll()
  + Find(predicate)
* Note that here we do not have a method called update because with collections in memory if we want to update an object, we simply get it from the collection and change it. We do not tell the collection to update an object and that’s one of the areas that a lot of developers get wrong.
* A repository should not have the semantics of your database. It should be like a collection of objects in memory and should not have methods like Update and Save.

Now, with this, the first question that pops into a lot of people’s head, including myself is: “If the repository acts as a collection of objects in memory, how are we going to save these objects and changes to them to database?” That is when the unit of work pattern comes into the picture.

# A Unit Of Work

* Definition: Maintains a list of objects affected by a business transaction and coordinates the writing transaction and coordinates writing out of changes.
* Some Developers argue that both the repository and unit of work pattern are already implemented in entity framework and there is no need to recreate them as this would lead to unnecessary complexity.
* The argument says that DBSet Has collection like interfaces so it has methods like Add, Remove, find, where and as you see it doesn’t have methods like update and Save.
* Also argument claims that the DBContext acts as the unit of work. It keepks track of changes in the objects and coordinates the writing out of changes. DbContext has references to one or more DBSets and a save methods actually implemented here.It make sense because ,the save method should not be in the repository. Plus, chances are in a business transaction we may work with more than one repository. So that is the reason the save method should be in the unit of work.
* So it will coordinates persisting changes across multiple repositories in one transaction.
* SO, on the surface, yes Entity framework DbSets and DbContext indeed look like repositories and unit of work.
* But there is a hidden problem here with the implementation of these patterns in entity framework.
* According to benefits of Repository pattern, “Repository minimizes duplicates” query logic”. Now, the problem with DBSets and LINQ extension methods implements on them is that they return IQueryable and this means if you want to get top scoring students in math you will end up repeating those few lines in a few different places in you application.
* So, implementation of DBSet does not really help with minimizing duplicate query logic.
* You can argue that we can use extension methods to encapsulate this logic into a method on a DBSet. This solution treats the symptoms, not actual problem. Because even if you use extension methods, all the existing LINQ extension methods still return IQueryable.
* So in this case ideally we want a repository that has a method like get top scoring student in math and all the querying logic is encapsulated their there.
* DBContext should be a private field in the repository and not exposed to the outside.
* It is the implementation detail.
* Now the 2nd benefit of repository pattern is that it decouples your application from persistence frameworks.
  + When you are using DBConetext and DBSets directly in your application, your application is tightly coupled to Entity Framework if you want to switch to a different ORM you need to modify your application code directly.
  + However, if all this is behind a repository and your application code relies on the repository to return the right data, it does not matter what is inside the repository. Today, you can use entity framework in repository, tomorrow you may decide for optimization reason to replace it with store procedures. Your application code will not affect in any way.
* So, with all these, while DBSet and DBContext in entity framework, do look like the implementation of repository and unit of work, they do not really bring the architectural benefit that we can get from these patterns.
* Architecture should be independent of any frameworks.

Application

Repository

Entity Framework

GetTopScoringStudents()

# Implementing the Repository Pattern

