Repository Pattern

The Repository design pattern in C# abstracts the data access layer from the business logic, providing a centralized and consistent way to interact with data sources. This pattern promotes loose coupling, testability, and maintainability in applications

What is a Repository?

As previously discussed, a repository is a class defined for an entity with all the possible database operations. For example, a repository for an Employee entity will have the basic CRUD operations and any other possible operations related to the Employee entity. The Repository Pattern can be implemented in two ways, i.e., using One Repository Per Entity and One Repository for all Entities. It is also possible to include both of them in our application.

One Repository Per Entity (Non-Generic Repository):

In this case, we must create a separate repository for each entity. For example, if we have two entities, Employee and Customer, in our application, we must create two repositories. The Employee Repository will have operations related to the Employee Entity, and the Customer Repository will only have operations related to the Customer Entity.

Generic Repository (One Repository for All Entities):

A Generic Repository can be used for all entities. In other words, the Generic Repository can be used for Employee Entity, Customer Entity, or any other entity. So, all the standard operations of all the entities will be put inside the Generic Repository. The Generic Repository contains the methods that are common for all the entities. But if you want some specific operation for some specific entities. Then, you need to create a specific repository with the required operations.

```csharp
public interface IBaseRepository<T> where T : class
{
   Task<bool> Add(T entity);

    bool AddWithRelatedEntites(T entity);

    Task<T> GetById(int id);

    Task<IEnumerable<T>> GetAll();

    bool Update(T entity);

    bool Deatach(T entity);

    Task<bool> Delete(int id);

    Task<IEnumerable<T>> FindExpression(Expression<Func<T, bool>> expression);
  }
```

Unit of Work pattern
The Unit of Work pattern in C# is a design pattern that groups one or more operations (typically database operations like create, update, and delete) into a single, atomic transaction. It ensures that all operations within this unit are either successfully committed to the database or, if any operation fails, the entire transaction is rolled back, preventing partial updates and maintaining data consistency.