A Unit of work is used for a single purpose: to ensure that when there are multiple repository components which need to be invoked or processed for a single request share a common database context. That way we can reduce the number of times a database connection is made for transaction when these repository components are used separately.

Then how the add/update/delete will affect the data source? Here the UOW plays that role. UOW knows about each repository. This helps to achieve multiple transactions at a time.

The UnitOfWork pattern is a design for grouping a set of tasks into a single group of transactional work. The UnitOfWork pattern is the solution to sharing the Entity Framework data context across multiple managers and repositories.

As mentioned, Unit Of Work pattern helps developers work with multiple repositories share single database context. This way, when a unit of work is complete, you can call the savechanges method of dbcontext, which will make sure all the changes associated with the context is saved to the database.

```
public interface IUnitOfWork
    {
         IAccountRepository AccountRepository { get; }

        public void ChangeTrackerDetectChanges();

        Task<bool> SaveContext();
    }


    public class UnitOfWork : IUnitOfWork
    {
        AppDbContext appDbContext;
        private IRecruiterRepository recruiterRepository;

        public UnitOfWork(AppDbContext appDbContext)
        {
            this.appDbContext = appDbContext;
        }


        public IRecruiterRepository RecruiterRepository => recruiterRepository =
recruiterRepository ?? new RecruiterRepository(appDbContext);

        public void ChangeTrackerDetectChanges()
        {
             appDbContext.ChangeTracker.DetectChanges();
        }
       //Database assosiate errors when issuing commands
        public async Task<bool> SaveContext()
        {

            try
            {
                // Returns:
                //     The number of state entries written to the database.
                return await appDbContext.SaveChangesAsync()>0;
            }
            catch (Exception e)
            {
                var message = e.Message;
                return false;
            }
        }
    }
```

Since EF Core already implement the repository pattern and unit of work behind the scenes, we don't have to care about a rollback method.

" - What? So why do we have to create all these interfaces and classes?"

Separating the persistence logic from business rules gives many advantages in terms of code reusability and maintenance. If we use EF Core directly, we'll end up having more complex classes that won't be so easy to change.

The Repository Design Pattern

As we already discussed in our previous articles, a repository is nothing but 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:

One repository per entity (non-generic):

This type of implementation involves the use of one repository class for each entity. For example, if you have two entities, Employee, and Customer, each entity will have its own repository.

Generic repository:
A generic repository is the one that can be used for all the entities. In other words, it can be either used for Employee or Customer or any other entity.

```
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);


    }
```

```
  //sekade kadesto ima T se zamenuva vo zavisnost koj repository go povikal
    public class BaseRepository<T> : IBaseRepository<T> where T : class
    {
        protected AppDbContext dbContext;


        // add update sets the entry State when saveContext called it will issue db
command
        // errors are assosiate with the EntryState Context
        public BaseRepository(AppDbContext dbContext)
        {
            this.dbContext = dbContext;
        }




        /// <summary>
        /// Sets the EntityState to Add and strats tracking the entity
        /// //After the SaveChangesAsync, you will have the ID populated on objects which
is in the database
        /// </summary>
        /// <param name="entity"></param>
        /// <returns>bool</returns>
        public async Task<bool> Add(T entity)
        {

            try {
                await dbContext.Set<T>().AddAsync(entity);
                return true;
            }
            catch (Exception e) {

                return false;
            }

        }



        public bool AddWithRelatedEntites(T entity)
        {

            try {

                dbContext.ChangeTracker.DetectChanges();

                dbContext.Attach(entity);//all object maeked Unchanged Object with no key
added

                return true;
            }
            catch (Exception e) {
                return false;
            }

        }

        public async Task<T> GetById(int id)
        {
            return await dbContext.Set<T>().FindAsync(id); //null or entity
        }

        public async Task<IEnumerable<T>> GetAll()
        {
            return await dbContext.Set<T>().ToListAsync();
        }




        public bool Update(T entity)
        {

            try
            {
                dbContext.Entry(entity).State = EntityState.Modified;//updates only the
entety not his related ddata
                return true;
            }
            catch (Exception e)
            {
                var a = e.Message;

                return false;
            }


        }

        public async Task<bool> Delete(int id)
        {

            T entity = await dbContext.Set<T>().FindAsync(id);

            try {

                dbContext.Set<T>().Remove(entity);
                return true;
            }
            catch (Exception e)
            {
                return false;
            }

        }


        public bool Deatach(T entity)
        {

            try {

                dbContext.Entry(entity).State = EntityState.Detached;

                return true;
            }
            catch (Exception e) {
                return false;
            }

        }


        public async Task<IEnumerable<T>> FindExpression(Expression<Func<T, bool>>
expression)
        {
            var query = dbContext.Set<T>().Where(expression);
            // where prima Expression<Func<T, bool>> ili Func<T, bool>
            //ako go primi prvoto vraka IQueryable ako vtoroto IEnumberable
            //IQueryable nema async

            var results = await query.ToListAsync();

            return results;
        }


    }
```