Report: Unit of Work

Understanding the Unit of Work Pattern in C

1. What is Unit of Work?

The Unit of Work (UoW) is a design pattern that serves as a transaction manager to coordinate the writing of changes and ensure atomicity in operations involving multiple repositories or entities. It ensures:

- **Atomic commits**: All operations succeed or none at all.
- **Shared context**: Repositories work with the same DbContext instance.
- **Centralized control** over commit/rollback logic.

In short, UoW helps group a set of operations into one logical "unit of work", so that the system remains consistent even if errors occur.

2. Problem with Repository Pattern Without UoW

When each repository (e.g., `EmployeeRepository`, `ProductRepository`) creates its own `DbContext`, it causes issues:

The Issues:

- **Inconsistent transactions**: One repository may save, the other may fail.
- **Duplicate tracking**: Each context tracks changes independently.
- **Partial updates**: If one operation fails and another commits, the DB is left in an inconsistent state.

Example of Failure:

ProductRepository.SaveChanges() → Succeeds

EmployeeRepository.SaveChanges() → Fails

→ Database is now partially updated, violating integrity.

3. Solution: Introduce a Unit of Work

Instead of giving each repository its own `DbContext`, we create a shared instance managed by the `UnitOfWork` class.

Benefits:

- Shared `DbContext` across all repositories.
- Transaction control in one place.
- Easy rollback on failure.
- Ensures data consistency and transaction integrity.

4. Full Implementation: Based on Habr Article

Step 1: Define `IUnitOfWork` Interface

```
public interface IUnitOfWork : IDisposable
{
    IRepository UserRepository { get; }
    IRepository OrderRepository { get; }

    void Commit();
    void Rollback();
}
```

Step 2: Create `AppDbContext`

```
public class AppDbContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Order> Orders { get; set; }

    public AppDbContext(DbContextOptions options)
        : base(options) { }

    public void BeginTransaction() => Database.BeginTransaction();
    public void CommitTransaction() => Database.CommitTransaction();
    public void RollbackTransaction() => Database.RollbackTransaction();
}
```

Step 3: Implement `UnitOfWork`

```
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;
    private IRepository _userRepository;
    private IRepository _orderRepository;

    public UnitOfWork(AppDbContext context)
    {
        _context = context;
        _context.BeginTransaction(); // Start transaction
    }

    public IRepository UserRepository => _userRepository ??= new Repository<User>(_context);
    public IRepository OrderRepository => _orderRepository ??= new Repository<Order>(_contex

    public void Commit()
    {
        _context.SaveChanges();
        _context.CommitTransaction();
    }

    public void Rollback() => _context.RollbackTransaction();

    public void Dispose() => _context.Dispose();
}
```

###Step 4: Implement `Repository<T>`

```
public class Repository<T> : IRepository<T> where T : class
{
    private readonly AppDbContext _context;
    private readonly DbSet<T> _dbSet;

    public Repository(AppDbContext context)
    {
        _context = context;
        _dbSet = context.Set<T>();
    }

    public void Add(T entity) => _dbSet.Add(entity);
    public void Update(T entity) => _dbSet.Update(entity);
    public void Delete(T entity) => _dbSet.Remove(entity);
    public IEnumerable<T> GetAll() => _dbSet.ToList();
    public T GetById(int id) => _dbSet.Find(id);
}
```

Step 5: Using UoW in Controller

```
public class UserController : Controller
{
    private readonly IUnitOfWork _unitOfWork;

    public UserController(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    [HttpPost]
    public IActionResult CreateUser(UserViewModel model)
    {
        try
        {
            _unitOfWork.UserRepository.Add(new User { Name = model.Name });
            _unitOfWork.Commit();
            return Ok("User created successfully.");
        }
        catch (Exception ex)
```

```
        {
            _unitOfWork.Rollback();
            return BadRequest($"Error: {ex.Message}");
        }
    }
}
```

5. When to Use or Avoid Unit of Work

Use When...	Avoid When...
Multiple related DB operations	Only a single simple DB operation
Multiple repositories are involved	Application is small or the overhead isn't justified
You want clean separation of concerns	You already use TransactionScope or handle transactions manually
You need rollback or transaction grouping	Performance is extremely critical, and every millisecond counts

6. Good Example: Use UoW

Transactional operation across multiple entities.

```
public IActionResult PlaceOrder(OrderViewModel model)
{
    try
    {
        _unitOfWork.OrderRepository.Add(new Order { /* ... */ });
        _unitOfWork.UserRepository.Update(new User { /* ... */ });
        _unitOfWork.Commit();
        return Ok("Order placed.");
    }
    catch
    {
        _unitOfWork.Rollback();
        return BadRequest("Failed to place order.");
    }
```

7. Bad Example: Don't Use UoW

Only touches one entity — overkill to use UoW here.

```
public IActionResult UpdateUserEmail(int userId, string newEmail)
{
    var user = _context.Users.Find(userId);
    if (user == null) return NotFound();

    user.Email = newEmail;
    _context.SaveChanges();

    return Ok("Email updated.");
}
```

8. Summary Table: Benefits of Unit of Work

Feature	Benefit
Atomicity	All changes succeed or none do
Transaction Coordination	Avoids partial updates across repositories
Centralized Commit Logic	Simplifies controller and business logic
Reduced Coupling	Keeps data layer separated from application logic
Easy Rollbacks	Supports `Rollback()` on errors
Shared DbContext	Prevents issues with multiple DbContext instances

9. When to Avoid UoW

Concern	Why UoW Might Be Overkill
Simplicity	One-off queries don't need UoW
Performance	Extra abstraction may slow down performance-critical systems
Microservices	Each service handles a small domain — isolated transactions suffice

Conclusion

The **Unit of Work pattern** is essential for:

- Transactional consistency
- Repository coordination

- Clean architecture

When to use it:

- Modify multiple entities or aggregates
- Need rollback support
- Want to enforce atomic operations

When to avoid it:

- Single-operation scenarios
- Where the built-in transaction support of `DbContext` is sufficient