public User() { Answers = new HashSet<Answer>();}

This guarantees that the Answers collection is never null. Without this constructor, you'd get a NullReferenceException when trying to use:

var user = new User();

user.Answers.Add(new Answer()); *// Would throw without constructor*

The constructor ensures your domain objects are always in a valid state and ready to use!

If the DbSet<Role> is missing from your DbContext while the Role entity exists in your domain model,

-EF Core Cannot Create/Migrate the Roles Table

-Cannot Directly Query Roles

**Add the DbSet** unless you have a specific reason not to: This makes your intent clear and provides direct access to query and manage roles.

Task<User> vs User - Async vs Sync

**Task<User>** GetByCompanyIdAsync(int companyId);

* **Returns**: A Task that will eventually contain a User
* **Usage**: var user = await GetByCompanyIdAsync(123);
* **Benefit**: Non-blocking, better for scalability (web apps, APIs)
* **Ideal for**: Database calls, HTTP requests, file I/O

**User** GetByCompanyId(int companyId);

* **Returns**: A User object directly
* **Usage**: var user = GetByCompanyId(123);
* **Risk**: Can block the thread while waiting for database
* **Ideal for**: In-memory operations, simple data access

public interface IUserRepository : IRepository<User>

{ *// Only User-specific methods*

Task<User> GetByCompanyIdAsync(int companyId);

Task<User> GetUserWithRoleAsync(int userId);}

**Benefits:**

* **DRY Principle**: Common CRUD operations in one place
* **Consistency**: All repositories have same base interface
* **Maintainability**: Change base interface, affect all repositories
* **Less Boilerplate**: No need to redefine common methods

public interface IUserRepository

{ *// Must define ALL methods needed*

User GetByCompanyId(int companyId);

User GetById(int id);

User GetByOU(string ou);

int Insert(User user);

*// Missing: Update, Delete, GetAll, etc.*}

**Drawbacks:**

* **Code Duplication**: Every repository redefines common methods
* **Inconsistency**: Different method names/signatures across repositories
* **More Maintenance**: Changes require updating every repository interface

*// Base repository with async operations*

public interface IRepository<T> where T : class

{

Task<T> GetByIdAsync(int id);

Task<IEnumerable<T>> GetAllAsync();

Task<T> SingleOrDefaultAsync(Expression<Func<T, bool>> predicate);

Task AddAsync(T entity);

void Update(T entity);

void Remove(T entity);

}

*// Specialized user repository*

public interface IUserRepository : IRepository<User>

{

Task<User> GetByCompanyIdAsync(int companyId);

Task<User> GetUserWithRoleAsync(int userId);

Task<User> AuthenticateAsync(int companyId, string fullName, string password);

Task<string> GetUserOUAsync(int userId);

}

**Why This is Better:**

1. **Performance**: Async doesn't block threads
2. **Scalability**: Essential for web applications
3. **Consistency**: All repositories follow same pattern
4. **Separation of Concerns**: Generic CRUD vs business-specific operations
5. **Testability**: Easy to mock and test

Generic Repository Inheritance: Repository<User>

**What this means:**

* **Inherits ALL CRUD implementation** from Repository<T>
* **Only writes custom User-specific methods**
* **DRY**: No repetitive CRUD code

**Benefits:**

* Less code to write/maintain
* Consistent CRUD implementation across all repositories
* Faster development

No Repository Inheritance: BaseRepository Only

**What this means:**

* **Only shares DbContext** via BaseRepository
* **Must implement every method** from scratch
* **More boilerplate** code

**Use Generic Repository Inheritance** (Repository<User>) for:

* Most business applications
* Teams of any size
* Long-term maintainability
* Rapid development
* **Code Reduction:**
* csharp
* *// Without inheritance: 50+ lines of boilerplate per repository*
* *// With inheritance: Only custom methods (5-10 lines)*
* **Consistency:**
* All repositories behave the same way
* Bug fixes in base class apply to all repositories
* Standardized error handling, logging, etc.
* **Maintainability:**
* Change CRUD logic in one place
* Easier to add cross-cutting concerns (caching, auditing)

The generic approach gives you the best balance of consistency, maintainability, and development speed.