Repository Pattern

A data persistence abstraction

The Repository Pattern has gained quite a bit of popularity since it was first introduced as a part of Domain-Driven Design (http://bit.ly/PS-DDD) in 2004. Essentially, it provides an abstraction of data, so that your application can work with a simple abstraction that has an interface approximating that of a collection. Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with database concerns like connections, commands, cursors, or readers. Using this pattern can help achieve loose coupling and can keep domain objects persistence ignorant (http://deviq.com/persistence-ignorance/). Although the pattern is very popular (or perhaps because of this), it is also frequently misunderstood and misused. There are many different ways to implement the Repository pattern. Let's consider a few of them, and their merits and drawbacks.

Repository Per Entity or Business Object

The simplest approach, especially with an existing system, is to create a new Repository implementation for each business object you need to store to or retrieve from your persistence layer. Further, you should only implement the specific methods you are calling in your application. Avoid the trap of creating a “standard” repository class, base class, or default interface that you must implement for all repositories. Yes, if you need to have an Update or a Delete method, you should strive to make its interface consistent (does Delete take an ID, or does it take the object itself?), but don’t implement a Delete method on your LookupTableRepository that you’re only ever going to be calling List() on. The biggest benefit of this approach is YAGNI (http://deviq.com/yagni)– you won’t waste any time implementing methods that never get called.

Generic Repository Interface

Another approach is to go ahead and create a simple, generic interface for your Repository. You can constrain what kind of types it works with to be of a certain type, or to implement a certain interface (e.g. ensuring it has an Id property, as is done below using a base class). An example of a generic C# repository interface might be:

```
1  public interface IRepository<T> where T : EntityBase
2  {
3      T GetById(int id);
4      IEnumerable<T> List();
5      IEnumerable<T> List(Expression<Func<T, bool>> predicate);
6      void Add(T entity);
7      void Delete(T entity);
8      void Edit(T entity);
9  }
10
11 public abstract class EntityBase
12 {
13    public int Id { get; protected set; }
14 }
```

The advantage of this approach is that it ensures you have a common interface for working with any of your objects. You can also simplify the implementation by using a Generic Repository Implementation (below). Note that taking in a predicate eliminates the need to return an IQueryable, since any filter details can be passed into the repository. This can still lead to leaking

of data access details into calling code, though. Consider using the Specification pattern (described below) to alleviate this issue if you encounter it.

Generic Repository Implementation

Assuming you create a Generic Repository Interface, you can implement the interface generically as well. Once this is done, you can easily create repositories for any given type without having to write any new code, and your classes the declare dependencies can simply specify IRepository<Item> as the type, and it's easy for your IoC container to match that up with a Repository<Item> implementation. You can see an example Generic Repository Implementation, using Entity Framework, here.

```
public class Repository<T> : IRepository<T> where T : EntityBase
{
    private readonly ApplicationDbContext _dbContext;

    public Repository(ApplicationDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public virtual T GetById(int id)
    {
        return _dbContext.Set<T>().Find(id);
    }

    public virtual IEnumerable<T> List()
    {
        return _dbContext.Set<T>().AsEnumerable();
    }

    public virtual IEnumerable<T> List(System.Linq.Expressions.Expression<Func<T, bool>> predicate)
    {
        return _dbContext.Set<T>()
               .Where(predicate)
               .AsEnumerable();
    }

    public void Insert(T entity)
    {
        _dbContext.Set<T>().Add(entity);
        _dbContext.SaveChanges();
    }

    public void Update(T entity)
    {
        _dbContext.Entry(entity).State = EntityState.Modified;
        _dbContext.SaveChanges();
    }

    public void Delete(T entity)
    {
        _dbContext.Set<T>().Remove(entity);
        _dbContext.SaveChanges();
    }
}
```

Note that in this implementation, all operations are saved as they are performed; there is no Unit of Work being applied. There are a variety of ways in which Unit of Work behavior can be added to this implementation, the simplest of which being to add an explicit Save() method to the IRepository<T> method, and to only call the underlying SaveChanges() method from this method.

IQueryable?

Another common question with Repositories has to do with what they return. Should they return data, or should they return queries that can be further refined before execution (IQueryable)? The former is safer, but the latter offers a great deal of flexibility. In fact, you can simplify your interface to only offer a single method for reading data if you go the IQueryable route, since from there any number of items can be returned.

A problem with this approach is that it tends to result in business logic bleeding into higher application layers, and becoming duplicated there. If the rule for returning valid customers is that they're not disabled and they've bought something in the last year, it would be better to have a method ListValidCustomers() that encapsulates this logic rather than specifying these criteria in lambda expressions in multiple different UI layer references to the repository. Another common example in real applications is the use of "soft deletes" represented by an IsActive or IsDeleted property on an entity. Once an item has been deleted, 99% of the time it should be excluded from display in any UI scenario, so nearly every request will include something like

.Where(foo => foo.IsActive)

in addition to whatever other filters are present. This is better achieved within the repository, where it can be the default behavior of the List() method, or the List() method might be renamed to something like ListActive(). If it's truly necessary to view deleted/inactive items, a special List method can be used for just this (probably rare) purpose.

Specification

Repositories that follow the advice of not exposing IQueryable can often become bloated with many custom query methods. The solution to this is to separate queries into their own types, using the Specification Design Pattern (/specification-pattern/). The Specification can include the expression used to filter the query, any parameters associated with this expression, as well as how

much data the query should return (i.e. “Include()” in EF/EF Core). Combining the Repository and Specification patterns can be a great way to ensure you follow the Single Responsibility Principle (/single-responsibility-principle/) in your data access code. See an example of how to implement a generic repository along with a generic specification in C# (/specification-pattern/).

Reference

DDD Fundamentals (http://bit.ly/PS-DDD) – Pluralsight

Repository (Martin Fowler) (http://martinfowler.com/eaaCatalog/repository.html)

Introducing The CachedRepository Pattern (https://ardalis.com/introducing-the-cachedrepository-pattern)

Building a CachedRepository via Strategy Pattern (https://ardalis.com/building-a-cachedrepository-via-strategy-pattern)

Repository Tip – Encapsulate Query Logic (http://www.weeklydevtips.com/018)

Do I Need a Repository? (http://www.weeklydevtips.com/024)

What Good is a Repository? (http://www.weeklydevtips.com/025)

Specification Pattern (/specification-pattern/)

Live Search

Tags

(https://deviq.com/tag/pattern/) practice (https://deviq.com/tag/practice/) principle (https://deviq.com/tag/principle/) tool (https://deviq.com/tag/tool/) value (https://deviq.com/tag/value/) xp (https://deviq.com/tag/xp/)

Popular Posts

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- Courage (https://deviq.com/courage/)
- Feedback (https://deviq.com/feedback/)
- Communication (https://deviq.com/communication/)
- Simplicity (https://deviq.com/simplicity/)

Recent Posts

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- State Design Pattern (https://deviq.com/state-design-pattern/)
- Guard Clause (https://deviq.com/guard-clause/)
- Bounded Context (https://deviq.com/bounded-context/)
- Specification Pattern (https://deviq.com/specification-pattern/)

RECENT ARTICLES

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- State Design Pattern (https://deviq.com/state-design-pattern/)
- Guard Clause (https://deviq.com/guard-clause/)
- Bounded Context (https://deviq.com/bounded-context/)
- Specification Pattern (https://deviq.com/specification-pattern/)
- Kinds of Models (https://deviq.com/kinds-of-models/)
- Singleton (https://deviq.com/singleton/)
- Antipatterns (https://deviq.com/antipatterns/)
- Design Patterns (https://deviq.com/design-patterns/)

POPULAR ARTICLES

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- Hollywood Principle (https://deviq.com/hollywood-principle/)
- Keep It Simple (https://deviq.com/keep-it-simple/)
- YAGNI (https://deviq.com/yagni/)
- Boy Scout Rule (https://deviq.com/boy-scout-rule/)
- Simplicity (https://deviq.com/simplicity/)
- Communication (https://deviq.com/communication/)
- Feedback (https://deviq.com/feedback/)
- Courage (https://deviq.com/courage/)