The Query Specification pattern can be applied to the [Domain Driven Design](https://www.udemy.com/course/domain-driven-design-in-net-core/?referralCode=5316BA92BB1321CBEF72) Architecture that is intended to be used to define logic for optional **sorting**, **filtering**and **paging**logic. This is where you use Entity Framework “includes” in the generic repository.

If your DDD application already exists and works without the Query Specification Pattern, it can be tricky to apply it because you have to modify existing code and can also use it in CQRS and [Domain Events](https://medium.com/@cizu64/using-mediatr-for-domain-events-in-net-core-9750a3406444) as well.

The Query Specification pattern is beneficial because it enables **Reusability**and avoids**cluttering and improves the repository pattern**and it is well suited for a Domain Driven Design Projects.

**The ISpecification<T> Interface**

In order to implement the [Specification pattern](https://www.udemy.com/course/domain-driven-design-in-net-core/learn/lecture/38855828#overview), we have to define the **ISpecification<T>** generic interface that **T** inherits from the Entity class that defines the identity for all Entities and some [domain event](https://www.udemy.com/course/domain-driven-design-in-net-core/learn/lecture/37778592#overview) code. The code below defines two methods that takes the Criteria and the Includes properties. The Criteria can be seen as the Where clause or filter logic.

using School.Domain.SeedWork;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Linq.Expressions;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace School.Domain.Specifications  
{  
 public interface ISpecification<T> where T: Entity  
 {  
 Expression<Func<T, bool>> Criteria { get; }  
 List<Expression<Func<T, object>>> Includes { get; }  
 }  
}

**The BaseSpecification Class**

The **BaseSpecification<T>** Class implements the **ISpecification**interface. The Criteria is assigned in the constructor so other concrete classes that extends the **BaseSpecification<T>** class can pass the Criteria in the constructor which is a lambda for that entity.

using School.Domain.SeedWork;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Linq.Expressions;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace School.Domain.Specifications  
{  
 public abstract class BaseSpecification<T> : ISpecification<T> where T : Entity  
 {  
 public BaseSpecification(Expression<Func<T, bool>> criteria)   
 {  
 Criteria = criteria;  
 }  
   
 public Expression<Func<T, bool>> Criteria { get; }  
  
 public List<Expression<Func<T, object>>> Includes { get; } = new();  
  
 protected void AddInclude(Expression<Func<T, object>> includeExpression)  
 {  
 Includes.Add(includeExpression);   
 }   
 }  
}

Let’s define a class that extends the **BaseSpecification<T>** class. The **GetActiveCourseDepartment**is a class that returns all active course and its departments is included.

using School.Domain.Aggregates;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Linq.Expressions;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace School.Domain.Specifications  
{  
 public class GetActiveCourseAndDepartment : BaseSpecification<Course>  
 {  
 public GetActiveCourseAndDepartment() : base(c => c.IsActive) //the criteria passed in the base parameter  
 {  
 AddInclude(c => c.Department); //including deparment entity  
 //how it looks like  
 //var course = \_set.Include(c=>c.Department).Where(c=>c.IsActive)   
  
 }  
 }  
}

**The GenericRepository<T> Without The Query Specification Pattern**

using Microsoft.EntityFrameworkCore;  
using School.Domain.SeedWork;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Linq.Expressions;  
using System.Threading.Tasks;  
  
namespace School.Infrastructure.Repositories  
{  
 public class GenericRepository<T> : IGenericRepository<T> where T : Entity, IAggregateRoot  
 {  
 protected readonly SchoolContext \_context;  
 internal DbSet<T> \_set;  
  
 public IUnitOfWork UnitOfWork => \_context;  
  
 public GenericRepository(SchoolContext context)  
 {  
 \_context = context;  
 \_set = \_context.Set<T>();  
 }  
 public async Task<T> AddAsync(T entity)  
 {  
 await \_set.AddAsync(entity);  
 return entity;  
 }  
  
 public async Task<T[]> AddRangeAsync(T[] entity)  
 {  
 await \_set.AddRangeAsync(entity);  
 return entity;  
 }  
  
 public async Task<bool> AnyAsync(Expression<Func<T, bool>> predicate, string include)  
 {  
 return await \_set.Include(include).AnyAsync(predicate);  
 }  
  
 public Task DeleteAsync(T entity)  
 {  
 \_set.Remove(entity);  
 return Task.CompletedTask;  
 }  
  
 public async Task<T> Get(Expression<Func<T, bool>> predicate, params string[] includes)  
 {  
 var data = await \_set.FirstOrDefaultAsync(predicate);  
 if (includes.Any())  
 {  
 data = Include(includes).FirstOrDefault();  
 }  
 return data;  
 }  
  
 public async Task<IReadOnlyList<T>> GetAll(Expression<Func<T, bool>> predicate = null, params string[] includes)  
 {  
  
 var data = predicate == null ? \_set : \_set.Where(predicate);  
 if (includes.Any())  
 {  
 data = Include(includes).AsQueryable();  
 }  
 return await data.ToListAsync();  
 }  
  
 IEnumerable<T> Include(params string[] includes)  
 {  
 IEnumerable<T> query = null;  
 foreach (var include in includes)  
 {  
 query = \_set.Include(include);  
 }  
  
 return query ?? \_set;  
 }  
  
  
 public async Task<T> GetByIdAsync(int Id, params string[] includes)  
 {  
 var data = await \_set.FindAsync(Id);  
 if (includes.Any())  
 {  
 data = Include(includes).FirstOrDefault();  
 }  
 return data;  
 }  
  
 public Task UpdateAsync(T entity)  
 {  
 \_context.Entry(entity).State = EntityState.Modified;  
 return Task.CompletedTask;  
 }  
 }  
}

In the code above, the Get methods defines a params of string[] includes that also call the defined private **Include()** method. This is the basic way to add includes in the repository pattern because all related objects are not returned as we are using EF eager loading.

Implementing the Query Specification Pattern suggest that the logic for sorting and paging and also code include related entities should be defined. Therefore, all the Includes defined in this class in the Get methods and the private Include() method is removed and two new methods are defined that implements the specification pattern in the DDD project as shown below.

**The GenericRepository<T> With The Query Specification Pattern**

using Microsoft.EntityFrameworkCore;  
using Microsoft.EntityFrameworkCore.Metadata.Internal;  
using School.Domain.SeedWork;  
using School.Domain.Specifications;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Linq.Expressions;  
using System.Threading.Tasks;  
  
namespace School.Infrastructure.Repositories  
{  
 public class GenericRepository<T> : IGenericRepository<T> where T : Entity, IAggregateRoot  
 {  
 protected readonly SchoolContext \_context;  
 internal DbSet<T> \_set;  
  
 public IUnitOfWork UnitOfWork => \_context;   
  
 public GenericRepository(SchoolContext context)  
 {  
 \_context = context;  
 \_set = \_context.Set<T>();  
 }  
 public IEnumerable<T> Specify(ISpecification<T> spec)  
 {  
 var includes = spec.Includes.Aggregate(\_context.Set<T>().AsQueryable(),  
 (current, include) => current.Include(include));  
   
 return includes  
 .Where(spec.Criteria)  
 .AsEnumerable();  
 }  
   
 //can be optional  
 public IEnumerable<T> SpecifyWithPagination(ISpecification<T> spec, int pageSize = 10, int pageIndex = 0)  
 {  
 var includes = spec.Includes.Aggregate(\_context.Set<T>().AsQueryable(),  
 (current, include) => current.Include(include));  
  
 return includes  
 .Where(spec.Criteria).Skip(pageSize \* pageIndex).Take(pageSize)  
 .AsEnumerable();  
 }  
  
 public async Task<T> AddAsync(T entity)  
 {  
 await \_set.AddAsync(entity);  
 return entity;  
 }  
 public async Task<T[]> AddRangeAsync(T[] entity)  
 {  
 await \_set.AddRangeAsync(entity);  
 return entity;  
 }  
 public Task DeleteAsync(T entity)  
 {  
 \_set.Remove(entity);  
 return Task.CompletedTask;  
 }  
 public async Task<T> Get(Expression<Func<T, bool>> predicate)  
 {  
 var data = await \_set.FirstOrDefaultAsync(predicate);  
 return data;  
 }  
 public async Task<IReadOnlyList<T>> GetAll()  
 {  
 var result = await \_set.ToListAsync();  
 return result;  
 }  
 public async Task<T> GetByIdAsync(int Id)  
 {  
 var data = await \_set.FindAsync(Id);  
 return data;  
 }  
 public Task UpdateAsync(T entity)  
 {  
 \_context.Entry(entity).State = EntityState.Modified;  
 return Task.CompletedTask;  
 }  
 }  
}

Then in the controller, it can be called in an action method this way:

[HttpGet, Route("courses/{pageSize:int}/{pageIndex:int}")]  
 [ProducesResponseType(typeof(PaginatedItemsViewModel<Course>), (int)HttpStatusCode.OK)]  
 public async Task<IActionResult> GetCourses(int pageSize = 10, int pageIndex = 0)  
 {  
 //var courses = await courseRepository.GetAll(c => c.IsActive, "Department");  
 //var courses = courseRepository.Specify(new GetActiveCourseAndDepartment());  
 var courses = courseRepository.SpecifyWithPagination(new GetActiveCourseAndDepartment(), pageSize, pageIndex);  
  
 int totalItem = courses.Count();  
 var model = new PaginatedItemsViewModel<Course>(pageIndex, pageSize, totalItem, courses);  
 return Ok(new ApiResult  
 {  
 Message = "Retrieved successfully",  
 Result = model  
 });  
 }

The Query Specification Pattern is not too complex, you just need to know when to call it and how to use it in the repository pattern and other DDD patterns. To better understand how it works, enroll for the complete DDD practical course on [**Udemy**](https://www.udemy.com/course/domain-driven-design-in-net-core/?referralCode=5316BA92BB1321CBEF72). This course covers CQRS, Domain Events, MediatR, Repository Patterns and Specifications and how to structure and build a Domain Driven Design from ground up using this [**link**](https://www.udemy.com/course/domain-driven-design-in-net-core/?referralCode=5316BA92BB1321CBEF72).