Eliassen - Search Query Middleware

Summary

This enables for a common means to query anything using the `IQueryable<T>` model. General features provided by this engine are a common reusable pattern for writing queries that support searching, filtering, sorting and paging. This may be used directly in .Net or transparently as part of ASP.Net Core MVC Actions.

Class Registration

ASP.Net MVC

If you want all Eliassen ASP.Net Core extensions using the `.AddAspNetCoreExtensions()` from the `Eliassen.AspNetCore.Mvc` name space. If you only want the SearchQuery extensions use `.TryAddAspNetCoreSearchQuery()` from `Eliassen.AspNetCore.Mvc.Extensions`

This functionality will be automatically enabled for any ASP.Net MVC Controller/Action that returns `IQueryable<T>`.

```
[Route("api/[controller]")]
[ApiController]
public class UserManagementController : ControllerBase
{
    private readonly IUserManagementManager _usersManager;

    public UserManagementController(
        IUserManagementManager usersManager
        )
    {
        _usersManager = usersManager;
    }

    [Authorize]
    [ApplicationRight(Rights.UserManagement.Manager)]
    [HttpPost("Query")]
    public IQueryable<User> ListUsers() => _usersManager.QueryUsers();
}
```

DotNet in General

The require classes are in the `Eliassen.System` assembly in the `Eliassen.System.Linq.Search` name space. You may choose to either directly instantiate `QueryBuilder<T>` or use the static methods provided on the `QueryBuilder` class but if you are using the `Microsoft.Extensions.DependnecyInjection` library you may prefer to use the `.TryAddSearchQueryExtensions()` with your IOC registration. Using this registration you

retrieve and instance of `IQueryBuilder<TModel>` from your `IServiceProvider` (or constructor injection) where `TModel` is your query model class.

Invoking the `.ExecuteBy(IQueryable<TModel>, ISearchQuery)` method will build and execute you query.

Search Query Model

The search query provides for a common means to define your query parameters. The parameters for the search will be noted as `SearchQuery` or *Search Query* while the data source that is being queried will be referred to as `IQuerable<T>`, or *query model*

Search

Searching uses the `SearchTerm` property on the `SearchQuery` model. This will compose a query using the input value against the predefined properties on the given query model. These predicates will be joined together with `OR` operations allowing for the result to match at least one of these generated expressions.

Searching works along with *Filtering* and the two will be joined with an `AND` operation if both are provided.

Filter

Filtering is a more targeted query. On the `SearchQuery` there is a `Filter` property that is a dictionary of `string` and `FilterParameter`. The `string` is the related property or target name that is used to select the matched data property or map. The `FilterParameter` is a class that allows for discrete operations. All filters are joined with an `AND` operator meaning that the result must be true for all provided predicate values.

Filtering works along with *Searching* and the two will be joined with an `AND` operation if both are provided.

Filter Parameters

The `FilterParameter` has multiple possible operations. These are defined as follows.

Equal To

`Equal To`: pass in the value to match for a given property

If you are using string values you may also use wild cards
*bc -> Ends with
b -> Contains
ab* -> Starts with

Not Equal To

`Not Equal To`: pass in the value to match for a given property

If you are using string values you may also use wild cards
*bc -> Ends with
b -> Contains
ab* -> Starts with

In Set

This allows for providing a set of values where the value from the queries data must match at least one of provided values

Greater than

This is an exclusive greater than where the data value must be greater than but not equal to the provided values

Greater than or Equal To

This is an inclusive greater than where the data value must be greater than or equal to the provided values

Less than

This is an exclusive less than where the data value must be less than but not equal to the provided values

Less than or Equal To

This is an inclusive less than where the data value must be less than or equal to the provided values

Sort

Sorting will control the resulting order of values returned by the underlying query provider. Multiple fields may be provided. The property name is set by the key of the dictionary, the priority of the fields is set by the order they are added to the dictionary and the direction of the order is determined by the value.

Page

Paging is provided by setting the current page and page size. Current Page is zero base being the first page is the value `0`. If the page size is set to `0` the default page size of `10` records will be used by the query.
If all records are desired setting the page size to `-1` will skip the paging functionality returning all matched rows.

The system will also try to count the total possible rows and pages and will include the values in the result set unless paging is disabled or `ExcludePageCount` is set to true. This may be necessary in cases where query performance is negatively effected by trying to count matched rows.

Query Model Features and Extensibility

To help with custom mappings and extensions to the query model some attributes have been included. by referencing the *Eliassen.System* library and importing the *Eliassen.System.ComponentModel.Search* name space you will have access to Attributes that may be used on your query models.

Attributes

Here is an example query model that has been annotated with some of the attributes as an example.

```
[Searchable(FirstNameLastName)]
[Searchable(LastNameFirstName)]
[Filterable(Module)]
[Filterable(UserStatus)]
[SearchTermDefault(SearchTermDefaults.StartsWith)]
public class User
{
    public const string FirstNameLastName = nameof(FirstNameLastName);
    public const string LastNameFirstName = nameof(LastNameFirstName);
    public const string Module = nameof(Module);
    public const string UserStatus = nameof(UserStatus);

    [ExcludeCaseReplacer]
    public string? UserId { get; set; }
    public string? UserName { get; set; }

    [Searchable]
    [DefaultSort(priority: 2, order: OrderDirections.Ascending)]
    public string? EmailAddress { get; set; }

    [Searchable]
    [DefaultSort(priority: 1, order: OrderDirections.Ascending)]
    public string? FirstName { get; set; }

    [Searchable]
    [DefaultSort(priority: 0, order: OrderDirections.Ascending)]
    public string? LastName { get; set; }

    public bool Active { get; set; }

    public List<UserModule>? UserModules { get; set; }

    public DateTimeOffset? CreatedOn { get; set; }

    public static Expression<Func<User, object>>? PropertyMap(string key) =>
        key switch
        {
            FirstNameLastName => e => e.FirstName + " " + e.LastName,
            LastNameFirstName => e => e.LastName + " " + e.FirstName,
            _ => null
        };
```

```
        public static Expression<Func<User, bool>>? PredicateMap(string key, object
value) =>
            key switch
            {
                Module => e => e.UserModules.Any(um => um.Code.Equals(value)),
                UserStatus => e => value.Equals("-1") || e.Active ==
value.Equals("1"),
                _ => null
            };
    }
```

DefaultSort

To control the default sort order for your model you may use the `DefaultSort` attribute. If this attribute is assigned to a property without a target name the related property will be included in the default order set.

The *priority* value will be used to create the precedence order for multi-field sort.

order sets the direction for the sort of `Ascending` or `Descending`

targetName applies when you want to include one of the *Property Map* values. Assign the attribute to the model class and set the *targetName* to the assigned key value. The other values still apply as described able.

Filterable

This attribute is intended only as meta-data to allow the *OpenAPI/Swagger UI* extensions to enumerate a list of *PropertyMap* fields. This does not effect functionality of the query generation engine.

IgnoreStringComparisonReplacement

Use this attribute to disable the `StringComparisonReplacementExpressionVisitor` functionality. Examples might include when the underlying query provided does not support `StringComparison` for a given field.

NotFilterable

This attribute is primarily intended to meta-data to prevent the *OpenAPI/Swagger UI* extensions from enumerating a field as part of the filterable fields. It will also exclude related fields from being included in the built query predicates.

NotSearchable

This attribute is primarily intended to meta-data to prevent the *OpenAPI/Swagger UI* extensions from enumerating a field as part of the search term related fields. It will also exclude related fields from being included in the built query predicates.

NotSortable

This attribute is primarily intended to meta-data to prevent the *OpenAPI/Swagger UI* extensions from enumerating a field as part of the sortable fields. It will also exclude related fields from being included in the built query ordinal.

Searchable

This attribute is primarily intended to meta-data to allow the *OpenAPI/Swagger UI* extensions to including a field as part of the search term enumeration list. It will also include related fields while generating the search term expression.

SearchTermDefault

The `SearchTermDefault` attribute may be used to override the default search term for a given query. By default all *Search Term* queries are generate as an *Equals To* meaning the value is a direct comparison. It is possible to use the asterisk `*` in your query expression to convert it to *Starts With* `Term*` , *Ends With* `*Term`, or *Contains* `*Term*`. When using this attribute if an asterisk is not in your expression it will be converted to the selected equivalent form by the query engine. It is still possible to convert back the *Equals To* form by quoting `"` the search term.

Mapped Extensions

Some times it is necessary to access child fields or compose siblings as part of your predicate or ordinal expressions. These *Maps* allow for creating additional reusable expressions as part of you model definition.

Property Map

Property maps may be used to generated visual composition properties. These may be used by *Order By*, *Search Term* or *Filter* expressions. This should be a `public static` method named `PropertyMap` that has the input of a `string` and return an `Expression<Func<YourModel, object` where `YourModel` is the current query model.

Predicate Map

Predicate maps are similar to *Property Maps* except they are only used by *Search Term* and *Filter* expressions. These are primarily intended for exposing nested lookup queries within your model such as when you need to match on at least one item in a child collection. This should be a `public static` method named `PredicateMap` that has the inputs of a `string` and `object` with a return of `Expression<Func<YourModel, bool` where `YourModel` is the current query model.

Post Build Visitors

Some times you need the ability to modify queries with global rules. The `IPostBuildExpressionVisitor` may be used to modify the expression tree with an `ExpressionVisitor` before the query is executed by the querying/paging engine.

StringComparisonReplacementExpressionVisitor

The `StringComparisonReplacementExpressionVisitor` is an implementation of the `IPostBuildExpressionVisitor`.
This visitor class replaces string functions with their `StringCompare` equivalents. Is using default configurations this process will ensure string functions are add and configured to ignore character casing.

This may be disabled per property with the `IgnoreStringComparisonReplacement` attribute.

Search Query Intercept

It is possible to create extensions to intercept and modify the SearchQuery. This is useful in case where you want to change or ensure formatting for default values or add additional filter properties dynamically at runtime. To use this functionality implement the `ISearchQueryIntercept` interface on a custom attribute and apply that attribute to your query model class.

The following is a simplified example to remove quotes `"` from the beginning and end of a search term. You would use this by adding `[Unquote]` to your query class definition.

```
[AttributeUsage(AttributeTargets.Class)]
public class UnquoteAttribute : Attribute, ISearchQueryIntercept
{
    public ISearchQuery Intercept(ISearchQuery searchQuery)
    {
        if (searchQuery is SearchQuery query &&
            query.SearchTerm != null &&
            query.SearchTerm.StartsWith('"') &&
            query.SearchTerm.EndsWith('"'))
        {
            return query with { SearchTerm = query.SearchTerm[1..^1] };
        }
        return searchQuery;
    }
}
```