SAMPLE CODE:

appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

imports.razor

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using Examine.Web.Demo
@using Examine.Web.Demo.Shared
```

Examine.test

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Library</OutputType>
    <SccProjectName>
    </SccProjectName>
    <SccLocalPath>
    </SccLocalPath>
    <SccAuxPath>
    </SccAuxPath>
    <SccProvider>
    </SccProvider>
  </PropertyGroup>
  <PropertyGroup>
    <TargetFrameworks>net6.0;</TargetFrameworks>
    <IsPackable>false</IsPackable>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  </PropertyGroup>
  <ItemGroup>
    <Content Include="App_Data\StringTheory.pdf">
      <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </Content>
  <None Update="App_Data\PDFStandards.PDF">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Update="App_Data\TemplateIndex\segments_2">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Update="App_Data\TemplateIndex\_0.cfs">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Update="App_Data\umbraco.config">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    <SubType>Designer</SubType>
  </None>
  <None Update="App_Data\TemplateIndex\segments.gen">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <Content Include="App_Data\VS2010CSharp.pdf">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </Content>
  <None Update="App_Data\UmbracoContour.pdf">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
<ItemGroup>
  <ProjectReference Include="..\Examine.Core\Examine.Core.csproj" />
  <ProjectReference Include="..\Examine.Host\Examine.csproj" />
  <ProjectReference Include="..\Examine.Lucene\Examine.Lucene.csproj" />
</ItemGroup>
<ItemGroup>
  <Content Include="App_Data\media.xml">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </Content>
</ItemGroup>
<ItemGroup>
  <PackageReference Include="Azure.Storage.Blobs" Version="12.13.1" />
  <PackageReference Include="Lucene.Net.Spatial">
    <Version>4.8.0-beta00016</Version>
  </PackageReference>
  <PackageReference Include="Microsoft.Extensions.Hosting" Version="6.0.1" />
  <PackageReference Include="Microsoft.Extensions.Logging" Version="6.0.0" />
  <PackageReference Include="Microsoft.Extensions.Logging.Console" Version="6.0.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.3.2" />
  <PackageReference Include="Moq" Version="4.18.2" />
  <PackageReference Include="NUnit">
    <Version>3.13.3</Version>
  </PackageReference>
  <PackageReference Include="Nunit3TestAdapter" Version="4.2.1" />
  <PackageReference Include="System.Data.DataSetExtensions" Version="4.5.0" />
  <PackageReference Include="System.Configuration.ConfigurationManager" Version="6.0.1" />
</ItemGroup>
<ItemGroup>
```

```
    <Compile Remove="DataServices\TestDataService.cs" />
    <Compile Remove="DataServices\TestLogService.cs" />
    <Compile Remove="Search\MultiIndexSearch.cs" />
  </ItemGroup>
</Project>
```

Examine.lucene

```
<?xml version="1.0" encoding="utf-8"?>
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <GeneratePackageOnBuild>false</GeneratePackageOnBuild>
    <Description>A Lucene.Net search and indexing implementation for Examine</Description>
    <PackageTags>examine lucene lucene.net lucenenet search index</PackageTags>
  </PropertyGroup>
  <ItemGroup>
    <None Remove="AspExamineManager.cs.bak" />
  </ItemGroup>
  <ItemGroup>
    <AssemblyAttribute Include="System.Runtime.CompilerServices.InternalsVisibleToAttribute">
      <_Parameter1>Examine.Test</_Parameter1>
    </AssemblyAttribute>
  </ItemGroup>
  <ItemGroup>
    <Content Include="..\..\assets\logo-round-small.png" Link="logo-round-small.png" Pack="true"
PackagePath="" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Lucene.Net.QueryParser">
      <Version>4.8.0-beta00016</Version>
    </PackageReference>
    <PackageReference Include="Lucene.Net.Replicator" Version="4.8.0-beta00016" />
    <PackageReference Include="System.Threading">
      <Version>4.3.0</Version>
    </PackageReference>
    <PackageReference Include="System.Threading.AccessControl">
      <Version>4.7.0</Version>
    </PackageReference>
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\Examine.Core\Examine.Core.csproj" />
  </ItemGroup>

</Project>
```

query.cs

```
using System;
using System.Collections.Generic;
```

```
namespace Examine.Search
{
    /// <summary>
    /// Defines the query methods for the fluent search API
    /// </summary>
    public interface IQuery
    {
        /// <summary>
        /// Passes a text string which is preformatted for the underlying search API. Examine will not
modify this
        /// </summary>
        /// <remarks>
        /// This allows a developer to completely bypass and Examine logic and comprise their own
query text which they are passing in.
        /// It means that if the search is too complex to achieve with the fluent API, or too dynamic to
achieve with a static language
        /// the provider can still handle it.
        /// </remarks>
        /// <param name="query">The query.</param>
        /// <returns></returns>
        IBooleanOperation NativeQuery(string query);

        /// <summary>
        /// Creates an inner group query
        /// </summary>
        /// <param name="inner"></param>
        /// <param name="defaultOp">The default operation is OR, generally a grouped query would
have complex inner queries with an OR against another complex group query</param>
        /// <returns></returns>
        IBooleanOperation Group(Func<INestedQuery, INestedBooleanOperation> inner,
BooleanOperation defaultOp = BooleanOperation.Or);

        /// <summary>
        /// Query on the id
        /// </summary>
        /// <param name="id">The id.</param>
        /// <returns></returns>
        IBooleanOperation Id(string id);

        /// <summary>
        /// Query on the specified field for a struct value which will try to be auto converted with the
correct query
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="fieldName"></param>
        /// <param name="fieldValue"></param>
        /// <returns></returns>
        IBooleanOperation Field<T>(string fieldName, T fieldValue) where T : struct;

        /// <summary>
        /// Query on the specified field
        /// </summary>
        /// <param name="fieldName">Name of the field.</param>
        /// <param name="fieldValue">The field value.</param>
        /// <returns></returns>
        IBooleanOperation Field(string fieldName, string fieldValue);

        /// <summary>
        /// Query on the specified field
        /// </summary>
        /// <param name="fieldName">Name of the field.</param>
        /// <param name="fieldValue">The field value.</param>
        /// <returns></returns>
        IBooleanOperation Field(string fieldName, IExamineValue fieldValue);

        /// <summary>
        /// Queries multiple fields with each being an And boolean operation
        /// </summary>
        /// <param name="fields">The fields.</param>
        /// <param name="query">The query.</param>
        /// <returns></returns>
        IBooleanOperation GroupedAnd(IEnumerable<string> fields, params string[] query);

        /// <summary>
        /// Queries multiple fields with each being an And boolean operation
        /// </summary>
        /// <param name="fields">The fields.</param>
        /// <param name="query">The query.</param>
        /// <returns></returns>
        IBooleanOperation GroupedAnd(IEnumerable<string> fields, params IExamineValue[] query);

        /// <summary>
        /// Queries multiple fields with each being an Or boolean operation
        /// </summary>
        /// <param name="fields">The fields.</param>
        /// <param name="query">The query.</param>
        /// <returns></returns>
        IBooleanOperation GroupedOr(IEnumerable<string> fields, params string[] query);

        /// <summary>
        /// Queries multiple fields with each being an Or boolean operation
        /// </summary>
```

```
    /// <param name="fields">The fields.</param>
    /// <param name="query">The query.</param>
    /// <returns></returns>
    IBooleanOperation GroupedOr(IEnumerable<string> fields, params IExamineValue[] query);

    /// <summary>
    /// Queries multiple fields with each being an Not boolean operation
    /// </summary>
    /// <param name="fields">The fields.</param>
    /// <param name="query">The query.</param>
    /// <returns></returns>
    IBooleanOperation GroupedNot(IEnumerable<string> fields, params string[] query);

    /// <summary>
    /// Queries multiple fields with each being an Not boolean operation
    /// </summary>
    /// <param name="fields">The fields.</param>
    /// <param name="query">The query.</param>
    /// <returns></returns>
    IBooleanOperation GroupedNot(IEnumerable<string> fields, params IExamineValue[] query);

    /// <summary>
    /// Matches all items
    /// </summary>
    /// <returns></returns>
    IOrdering All();

    /// <summary>
    /// The index will determine the most appropriate way to search given the query and the fields
provided
    /// </summary>
    /// <param name="query"></param>
    /// <param name="fields"></param>
    /// <returns></returns>
    IBooleanOperation ManagedQuery(string query, string[] fields = null);

    /// <summary>
    /// Matches items as defined by the IIndexFieldValueType used for the fields specified.
    /// If a type is not defined for a field name, or the type does not implement
IIndexRangeValueType for the types of min and max, nothing will be added
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="min"></param>
    /// <param name="max"></param>
    /// <param name="fields"></param>
    /// <param name="minInclusive"></param>
```

```
        /// <param name="maxInclusive"></param>
        /// <returns></returns>
        IBooleanOperation RangeQuery<T>(string[] fields, T? min, T? max, bool minInclusive = true,
bool maxInclusive = true) where T : struct;
    }
}
```

Lucenequerysearch.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Examine.Lucene.Indexing;
using Examine.Search;
using Lucene.Net.Analysis;
using Lucene.Net.Search;

namespace Examine.Lucene.Search
{
    /// <summary>
    /// This class is used to query against Lucene.Net
    /// </summary>
    [DebuggerDisplay("Category: {Category}, LuceneQuery: {Query}")]
    public class LuceneSearchQuery : LuceneSearchQueryBase, IQueryExecutor
    {
        private readonly ISearchContext _searchContext;
        private ISet<string> _fieldsToLoad = null;

        public LuceneSearchQuery(
            ISearchContext searchContext,
            string category, Analyzer analyzer, LuceneSearchOptions searchOptions, BooleanOperation
occurance)
            : base(CreateQueryParser(searchContext, analyzer, searchOptions), category, searchOptions,
occurance)
        {
            _searchContext = searchContext;
        }

        private static CustomMultiFieldQueryParser CreateQueryParser(ISearchContext searchContext,
Analyzer analyzer, LuceneSearchOptions searchOptions)
        {
            var parser = new ExamineMultiFieldQueryParser(searchContext,
LuceneInfo.CurrentVersion, analyzer);

            if (searchOptions != null)
    {
        if (searchOptions.LowercaseExpandedTerms.HasValue)
        {
            parser.LowercaseExpandedTerms = searchOptions.LowercaseExpandedTerms.Value;
        }
        if (searchOptions.AllowLeadingWildcard.HasValue)
        {
            parser.AllowLeadingWildcard = searchOptions.AllowLeadingWildcard.Value;
        }
        if (searchOptions.EnablePositionIncrements.HasValue)
        {
            parser.EnablePositionIncrements = searchOptions.EnablePositionIncrements.Value;
        }
        if (searchOptions.MultiTermRewriteMethod != null)
        {
            parser.MultiTermRewriteMethod = searchOptions.MultiTermRewriteMethod;
        }
        if (searchOptions.FuzzyPrefixLength.HasValue)
        {
            parser.FuzzyPrefixLength = searchOptions.FuzzyPrefixLength.Value;
        }
        if (searchOptions.Locale != null)
        {
            parser.Locale = searchOptions.Locale;
        }
        if (searchOptions.TimeZone != null)
        {
            parser.TimeZone = searchOptions.TimeZone;
        }
        if (searchOptions.PhraseSlop.HasValue)
        {
            parser.PhraseSlop = searchOptions.PhraseSlop.Value;
        }
        if (searchOptions.FuzzyMinSim.HasValue)
        {
            parser.FuzzyMinSim = searchOptions.FuzzyMinSim.Value;
        }
        if (searchOptions.DateResolution.HasValue)
        {
            parser.SetDateResolution(searchOptions.DateResolution.Value);
        }
    }

    return parser;
}
```

```
    public virtual IBooleanOperation OrderBy(params SortableField[] fields) =>
OrderByInternal(false, fields);

    public virtual IBooleanOperation OrderByDescending(params SortableField[] fields) =>
OrderByInternal(true, fields);

    public override IBooleanOperation Field<T>(string fieldName, T fieldValue)
      => RangeQueryInternal<T>(new[] { fieldName }, fieldValue, fieldValue, true, true,
Occurrence);

    public override IBooleanOperation ManagedQuery(string query, string[] fields = null)
      => ManagedQueryInternal(query, fields, Occurrence);

    public override IBooleanOperation RangeQuery<T>(string[] fields, T? min, T? max, bool
minInclusive = true, bool maxInclusive = true)
      => RangeQueryInternal(fields, min, max, minInclusive, maxInclusive, Occurrence);

    protected override INestedBooleanOperation FieldNested<T>(string fieldName, T fieldValue)
      => RangeQueryInternal<T>(new[] { fieldName }, fieldValue, fieldValue, true, true,
Occurrence);

    protected override INestedBooleanOperation ManagedQueryNested(string query, string[] fields
= null)
      => ManagedQueryInternal(query, fields, Occurrence);

    protected override INestedBooleanOperation RangeQueryNested<T>(string[] fields, T? min, T?
max, bool minInclusive = true, bool maxInclusive = true)
      => RangeQueryInternal(fields, min, max, minInclusive, maxInclusive, Occurrence);

    internal LuceneBooleanOperationBase ManagedQueryInternal(string query, string[] fields,
Occur occurance)
    {
      Query.Add(new LateBoundQuery(() =>
      {
        //if no fields are specified then use all fields
        fields = fields ?? AllFields;

        var types = fields.Select(f => _searchContext.GetFieldValueType(f)).Where(t => t !=
null);

        //Strangely we need an inner and outer query. If we don't do this then the lucene syntax
returned is incorrect
        //since it doesn't wrap in parenthesis properly. I'm unsure if this is a lucene issue (assume
so) since that is what
        //is producing the resulting lucene string syntax. It might not be needed internally within
Lucene since it's an object
```

```
            //so it might be the ToString() that is the issue.
            var outer = new BooleanQuery();
            var inner = new BooleanQuery();

            foreach (var type in types)
            {
                var q = type.GetQuery(query);

                if (q != null)
                {
                    //CriteriaContext.ManagedQueries.Add(new KeyValuePair<IIndexFieldValueType,
Query>(type, q));
                    inner.Add(q, Occur.SHOULD);
                }
            }

            outer.Add(inner, Occur.SHOULD);

            return outer;
        }), occurance);

        return CreateOp();
    }

    internal LuceneBooleanOperationBase RangeQueryInternal<T>(string[] fields, T? min, T?
max, bool minInclusive, bool maxInclusive, Occur occurance)
        where T : struct
    {
        Query.Add(new LateBoundQuery(() =>
        {
            //Strangely we need an inner and outer query. If we don't do this then the lucene syntax
returned is incorrect
            //since it doesn't wrap in parenthesis properly. I'm unsure if this is a lucene issue (assume
so) since that is what
            //is producing the resulting lucene string syntax. It might not be needed internally within
Lucene since it's an object
            //so it might be the ToString() that is the issue.
            var outer = new BooleanQuery();
            var inner = new BooleanQuery();

            foreach (var f in fields)
            {
                var valueType = _searchContext.GetFieldValueType(f);

                if (valueType is IIndexRangeValueType<T> type)
                {
                    var q = type.GetQuery(min, max, minInclusive, maxInclusive);

                    if (q != null)
                    {
                        //CriteriaContext.FieldQueries.Add(new KeyValuePair<IIndexFieldValueType,
Query>(type, q));
                        inner.Add(q, Occur.SHOULD);
                    }
                }
#if !NETSTANDARD2_0 && !NETSTANDARD2_1
                else if(typeof(T) == typeof(DateOnly) && valueType is
IIndexRangeValueType<DateTime> dateOnlyType)
                {
                    TimeOnly minValueTime = minInclusive ? TimeOnly.MinValue :
TimeOnly.MaxValue;
                    var minValue = min.HasValue ? (min.Value as
DateOnly?)?.ToDateTime(minValueTime) : null;

                    TimeOnly maxValueTime = maxInclusive ? TimeOnly.MaxValue :
TimeOnly.MinValue;
                    var maxValue = max.HasValue ? (max.Value as
DateOnly?)?.ToDateTime(maxValueTime) : null;

                    var q = dateOnlyType.GetQuery(minValue, maxValue, minInclusive, maxInclusive);

                    if (q != null)
                    {
                        inner.Add(q, Occur.SHOULD);
                    }
                }
#endif
                else
                {
                    throw new InvalidOperationException($"Could not perform a range query on the
field {f}, it's value type is {valueType?.GetType()}");
                }
            }

            outer.Add(inner, Occur.SHOULD);

            return outer;
        }), occurance);

        return CreateOp();
    }
```

```
        /// <inheritdoc />
        public ISearchResults Execute(QueryOptions options = null) => Search(options);

        /// <summary>
        /// Performs a search with a maximum number of results
        /// </summary>
        private ISearchResults Search(QueryOptions options)
        {
            // capture local
            var query = Query;

            if (!string.IsNullOrEmpty(Category))
            {
                // rebuild the query
                IList<BooleanClause> existingClauses = query.Clauses;

                if (existingClauses.Count == 0)
                {
                    // Nothing to search. This can occur in cases where an analyzer for a field doesn't return
                    // anything since it strips all values.
                    return EmptySearchResults.Instance;
                }

                query = new BooleanQuery
                {
                    // prefix the category field query as a must
                    { GetFieldInternalQuery(ExamineFieldNames.CategoryFieldName, new
    ExamineValue(Examineness.Explicit, Category), true), Occur.MUST }
                };

                // add the ones that we're already existing
                foreach (var c in existingClauses)
                {
                    query.Add(c);
                }
            }

            var executor = new LuceneSearchExecutor(options, query, SortFields, _searchContext,
    _fieldsToLoad);

            var pagesResults = executor.Execute();

            return pagesResults;
        }

        /// <summary>
        /// Internal operation for adding the ordered results
        /// </summary>
        /// <param name="descending">if set to <c>true</c> [descending].</param>
        /// <param name="fields">The field names.</param>
        /// <returns>A new <see cref="IBooleanOperation"/> with the clause appended</returns>
        private LuceneBooleanOperationBase OrderByInternal(bool descending, params
SortableField[] fields)
        {
            if (fields == null) throw new ArgumentNullException(nameof(fields));

            foreach (var f in fields)
            {
                var fieldName = f.FieldName;

                var defaultSort =  SortFieldType.STRING;

                switch (f.SortType)
                {
                    case SortType.Score:
                        defaultSort = SortFieldType.SCORE;
                        break;
                    case SortType.DocumentOrder:
                        defaultSort = SortFieldType.DOC;
                        break;
                    case SortType.String:
                        defaultSort = SortFieldType.STRING;
                        break;
                    case SortType.Int:
                        defaultSort = SortFieldType.INT32;
                        break;
                    case SortType.Float:
                        defaultSort = SortFieldType.SINGLE;
                        break;
                    case SortType.Long:
                        defaultSort = SortFieldType.INT64;
                        break;
                    case SortType.Double:
                        defaultSort = SortFieldType.DOUBLE;
                        break;
                    default:
                        throw new ArgumentOutOfRangeException();
                }

                //get the sortable field name if this field type has one
                var valType = _searchContext.GetFieldValueType(fieldName);
                if (valType?.SortableFieldName != null)
                    fieldName = valType.SortableFieldName;

                SortFields.Add(new SortField(fieldName, defaultSort, descending));
            }

            return CreateOp();
        }

        internal IBooleanOperation SelectFieldsInternal(ISet<string> loadedFieldNames)
        {
            _fieldsToLoad = loadedFieldNames;
            return CreateOp();
        }

        internal IBooleanOperation SelectFieldInternal(string fieldName)
        {
            _fieldsToLoad = new HashSet<string>(new string[] { fieldName });
            return CreateOp();
        }

        public IBooleanOperation SelectAllFieldsInternal()
        {
            _fieldsToLoad = null;
            return CreateOp();
        }

        protected override LuceneBooleanOperationBase CreateOp() => new
LuceneBooleanOperation(this);

  }
}
```