Agenda - Searching & Indexing with Examine

We have a great day ahead of us, and these are the things we will be working on.

Solution Setup

- Local solution

Exercise 1

- Implementing a simple search

Exercise 2

- TransformingIndexValues event handler

Exercise 3

- Wildcard searches

Exercise 4

- Split terms

Exercise 5

- Sorting by default properties
- Sorting by custom property
- Fallback values during index time

Exercise 6

- Filtering by document types

Exercise 7

- PDF index

Exercise 8

- Multisearchers
- Multi index search

Exercise 9

- Custom indexing

Optional exercise

- Searching on custom index

Appendix

- Examine on Azure

Solution setup

Local Environment

- Umbraco 10
- SQLite
- .NET 5 or .NET 6

Bundled items:

- Document Types
- Basic content structure
- Templates
- JavaScript and CSS

Solution setup

1. Follow the instructions here to create a new blank Umbraco project locally:

https://our.umbraco.com/documentation/Fundamentals/Setup/Install/install-umbraco-with-templates

2. Run the project and install Umbraco before moving onto the next step.

3. Install `UmbTrainingSite.Package`

4. Run the site - on the frontend, you should see a website with sample content.

The package contains the necessary files for most of our courses. If anything else is needed, it will be provided during the course.

Umbraco MVC

Umbraco 9 follows the conventions set by the .NET CORE framework. However, to create the best out-of-box experience for everyone, Umbraco comes with a bunch of presets you should know about. These presets can either be overridden or customized in case you have special requirements, but in most scenarios you can leave them as they are. The presets are divided into 3 groups: the standard model, views and controllers.

Model

By default Umbraco serves a model based on the current request. For example, if a request is made to domain.com/news, Umbraco will find the matching piece of content for the /news page and populate a Model with these values to send to the appropriate View.

Views

Umbraco page templates are MVC Views.

Controllers

Umbraco comes with a collection of base classes that you can inherit when implementing your own controllers. These are the Umbraco wrappers around the standard ASP.NET equivalents we have:

- RenderController: The default Umbraco controller which can be used to override controllers if required.
- SurfaceController: An Umbraco specific controller for handling form post submissions from an Umbraco page or for rendering MVC actions.
- UmbracoApiController: A wrapper around the standard ApiController which provides easy access to Umbraco services and member authentication. Used for creating REST services.

Authorized controllers

An Umbraco Authorized Controller is used when the controller requires member or user authentication (authN) and/or authorization (authZ). If either the authN or authZ fail, the controller will return a 401 - unauthorized response.

- UmbracoAuthorizedController: Provides services to authenticated users and members. MVC controller that is not auto-routed.
- UmbracoAuthorizedApiController: Inherits from the above controller and is auto-routed, which applies to backoffice routing too.
- UmbracoAuthorizedJsonController: Inherits from the Authorized Api Controller, has some additional filters applied to it to automatically handle anti-forgery tokens for use with AngularJS in the backoffice

Exercise 1: Implement simple Examine search

In this exercise, we will start by creating the foundation for our search functionality. Then, the search logic will be added in a custom service, and we will use a controller to handle and control incoming requests for content pages based on a specific Document Type.

The end goal of this approach is to enrich the view model passed to the template with additional properties before having everything available in the view. In this way, our view will not contain any custom code for our search functionality. Hence, the approach is called *Route Hijacking*.

Once we have the foundation in place for our search, we will implement a simple search that searches initially on the `bodytext` field of our site.

We will then proceed to add more fields to search on.

In this exercise we will:

- Configure the custom models
- Configure the services
- Configure the controller
- Register the services via composers
- Enhance the view with the model

Configure the custom models

When we search for content on the front end, we want to return the name and the URL of those search results. On top of that, Lucene (that sits below Examine) has a scoring mechanism. Each time you search for a piece of content, a score is applied to the results returned by the query.

We will create two models to consider each search result's score and return a collection of strongly typed content items.

Let's configure the first model for the individual score. In Visual Studio, add a new file in **/Models** folder called **SearchResultItem.cs**

In this new class, we'll add two new properties, `PublishedItem` and `Score`. The Published item is of type `IPublishedContent`, so we get all the standard properties for the content node. Thus, our new model class will end up looking like this:

```
using Umbraco.Cms.Core.Models.PublishedContent;
namespace SearchInV10.Models
{
   public class SearchResultItem
   {
       public IPublishedContent PublishedItem { get; init; }
       public float Score { get; init; }
   }
}
```

Moving on, let's create our second model. In Visual Studio, add a new file in **/Models** folder called **SearchModel.cs**.

We need our custom model to build upon the underlying existing `PublishedContent` model for the page. In Umbraco 10, this can be achieved by making our custom model inherit from a special base class called `PublishedContentWrapped`. The `PublishedContentWrapped` will take care of populating all the usual underlying Umbraco properties.

Our new model class will end up looking like this:

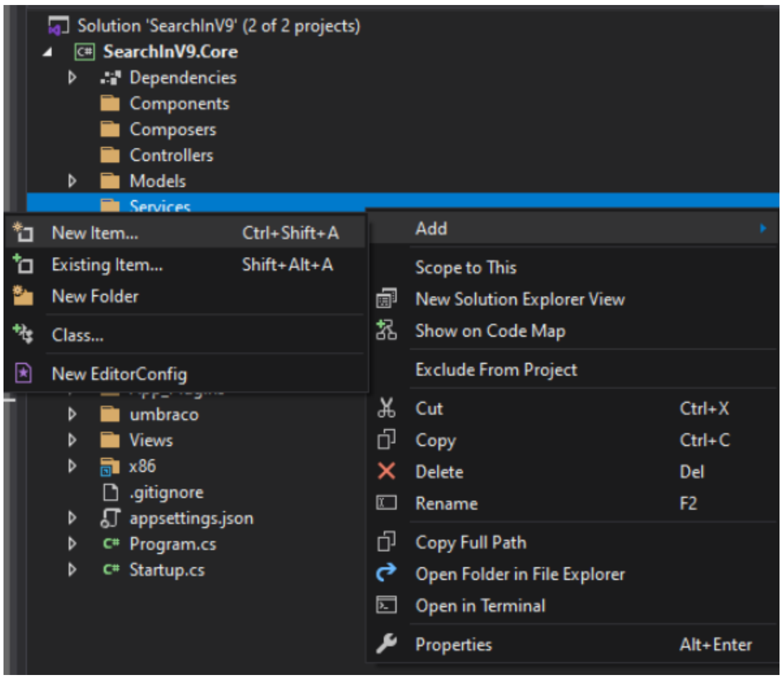
```
using System.Collections.Generic;
using Umbraco.Cms.Core.Models;
using Umbraco.Cms.Core.Models.PublishedContent;
namespace SearchInV10.Core.Models
{
   public class SearchModel : PublishedContentWrapped
   {
       public SearchModel(IPublishedContent content, IPublishedValueFallback publishedValu
eFallback) : base(content, publishedValueFallback)
       {
       }
       public string Query { get; set; }
       public long TotalResults { get; set; }
       public IEnumerable<SearchResultItem> SearchResults { get; set; }
   }
}
```

Great! At this point, we have one model that returns a collection of search results and one that returns the individual score of each search result. #H5YR

However, we are not finished yet. The next step is to create our search criteria. Then, to make it easier to test the code, we will create our service to handle the logic.

Configure the services

Inside your project (in the .Core project if you followed that part of the setup), right click and add a new folder called **Services**. In that folder, create a new file class called **ISearchService.cs**.



In our view, we want to return the number and the name of the search results based on our query, so our new class will be an interface that has two methods.

```
using System.Collections.Generic;
using Examine;
using SearchInV10.Core.Models;
namespace SearchInV10.Core.Services
{
   public interface ISearchService
   {
       IEnumerable<ISearchResult> GetSearchResults(string searchTerm, out long totalItemCo
unt);
       IEnumerable<SearchResultItem> GetContentSearchResults(string searchTerm, out long t
otalItemCount);
   }
}
```

The first method, `GetSearchResults`, returns a collection of `ISearchResult` and has two parameters. The first parameter takes our `SearchTerm` and the second parameter returns the total number of returned search results.

The second method, `GetContentSearchResults`, returns a collection of `SearchResultItem` - that's the model we have created earlier.

Great, now let's create a new class that will implement the methods from our interface.

Create a new class in the **/Services** folder called **SearchService.cs**. We will need to inherit from the `ISearchService` interface to be able to implement the interface..

To inherit from `ISearchService`, write (remember to make it public):

```
public class SearchService : ISearchService
```

At this point, you might notice our service class will be highlighted because it does not implement the interface methods. So, go ahead and implement the interface. Our new class will look like:

```
using Examine;
using SearchInV10.Core.Models;
using System;
using System.Collections.Generic;
namespace SearchInV10.Core.Services
{
   public class SearchService : ISearchService
   {
       public IEnumerable<SearchResultItem> GetContentSearchResults(string searchTerm, out
 long totalItemCount)
       {
           throw new NotImplementedException();
       }
       public IEnumerable<ISearchResult> GetSearchResults(string searchTerm, out long tota
lItemCount)
       {
           throw new NotImplementedException();
       }
   }
}
```

Notice the first method returns a collection of `SearchResultItem`, the model we added to the **/Models** folder. Furthermore, the second method returns an array of `ISearchResult`, an Examine interface that gives access to different fields of the search result such as `Id` and `Score`.

To construct our basic search criteria and get the name of each search result, we need to access `Examine` and the `UmbracoContextAccessor` service.

✍ Note : In Umbraco 9 and higher, we need to use Dependency Injection for each service that we need to access. It's a new workflow compared to Umbraco 8. If you inherit, for example, from the `SurfaceController`, you will get access to many Umbraco base classes such as the `ServiceContext`. On the other hand, in Umbraco 9+, you have to inject them inside the constructor if you need to gain access to the `ServiceContent` or any additional service.

```
public class MyController : SurfaceController
   {
       public MyController(
           IUmbracoContextAccessor umbracoContextAccessor,
           IUmbracoDatabaseFactory databaseFactory,
           ServiceContext services,
           AppCaches appCaches,
           IProfilingLogger profilingLogger,
           IPublishedUrlProvider publishedUrlProvider)
           : base(umbracoContextAccessor, databaseFactory, services, appCaches, profilingL
ogger, publishedUrlProvider)
       {
       }
   }
```

You can find more information about this in the official Umbraco documentation .

Update the class to provide access to `Examine` and the `umbracoContentAccessor` service.

```
using Examine;
using SearchInV10.Core.Models;
using System;
using System.Collections.Generic;
using Umbraco.Cms.Core.Web;
namespace SearchInV10.Core.Services
{
   public class SearchService : ISearchService
   {
     private readonly IExamineManager _examineManager;
     private readonly IUmbracoContextAccessor _umbracoContextAccessor;
    public SearchService(IExamineManager examineManager, IUmbracoContextAccessor umbracoCo
ntextAccessor)
       {
           _examineManager = examineManager;
           _umbracoContextAccessor = umbracoContextAccessor;
       }
       public IEnumerable<SearchResultItem> GetContentSearchResults(string searchTerm, out
 long totalItemCount)
       {
           throw new NotImplementedException();
       }
       public IEnumerable<ISearchResult> GetSearchResults(string searchTerm, out long tota
lItemCount)
       {
           throw new NotImplementedException();
       }
   }
}
```

Great! Now we have everything in place to start building our query.

Let’s start by updating the first method. We are trying to get the `ID`, `Score`, and all of the Umbraco default properties for each search result.

First, inside of the first method, create a new variable that creates an instance of the second method like so:

```
var pageOfResults = GetSearchResults(searchTerm, out totalItemCount);
```

Once that’s in place, let’s add each search result to a list.

```
var items = new List<SearchResultItem>();
```

Then, let’s consider if we get any results back, and if we do, let’s loop through all of the search results and get the `ID` and `Score` of each result.

Update the method to have the following configuration:

```
public IEnumerable<SearchResultItem> GetContentSearchResults(string searchTerm, out long t
otalItemCount)
       {
           var pageOfResults = GetSearchResults(searchTerm, out totalItemCount);
           var items = new List<SearchResultItem>();
           if (pageOfResults != null && pageOfResults.Any())
           {
               foreach (var item in pageOfResults)
               {
                   if (_umbracoContextAccessor.TryGetUmbracoContext(out var umbracoContext
))
                   {
                   }
                   var page = umbracoContext.Content.GetById(int.Parse(item.Id));
                   if (page != null)
                   {
                       items.Add(new SearchResultItem() {
                       PublishedItem = page,
                       Score = item.Score
                       });
                   }
               }
           }
           return items;
       }
```

Excellent, now it’s time to get our hands dirty. Umbraco ships with a preconfigured index. All searching is based on an index, and this index contains all the published content for the site.

In the second method, we can now start to create a search criteria by using Examine's fluent API. For this first attempt, we'll search the Body Text field (the `bodyText` property) for a match (this field is variant, so we need to append the language iso code) and exclude documents where Hide in navigation (the `umbracoNaviHide` property) is set to true.

Finally, we’ll compile the Examine query and perform the search. Update the method to have the following configuration

```
public IEnumerable<ISearchResult> GetSearchResults(string searchTerm, out long totalItemCo
unt)
       {
           totalItemCount = 0;
           if (_examineManager.TryGetIndex(Constants.UmbracoIndexes.ExternalIndexName, out
 var index))
           {
               var searcher = index.Searcher;
              var fieldToSearch = "bodyText" + "_" + CultureInfo.CurrentCulture.ToString()
.ToLower();
               var hideFromNavigation = "umbracoNaviHide";
               var criteria = searcher.CreateQuery(IndexTypes.Content);
               var examineQuery = criteria.Field(fieldToSearch, searchTerm);
               examineQuery.Not().Field(hideFromNavigation, 1.ToString());
               var results = examineQuery.Execute();
               totalItemCount = results.TotalItemCount;
               if (results.Any())
               {
                   return results;
               }
               else
               {
                   Console.WriteLine("Error");
               }
           }
           return Enumerable.Empty<ISearchResult>();
       }
```

```
using Umbraco.Cms.Infrastructure.Examine; using System.Globalization; using Umbraco.Web.Co
mposing; using Umbraco.Cms.Core;
```

👆 Tip: It is common to exclude certain fields set such as `hideFromNavigation`. It can be achieved by adding an additional rule that excludes pages with values of “1” (which corresponds to a checked True/False datatype).

examineQuery.Not().Field(hideFromNavigation, 1.ToString());

Configure the controller

The final bit of the puzzle for our custom search is making Umbraco return a `SearchPageModel` as a model when rendering pages.

In Visual Studio, click the folder **Controllers** and choose “Add class...” and name it **SearchPageController** - the name is essential as it follows a strict naming convention: ***DocumentTypeAliasController***. The Controller name must match the alias of the DocumentType of the content you intend to hijack.

Also, for the Umbraco routing engine to find your new route hijacked controller, we must ensure the controller inherits from the default `RenderController` class:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewEngines;
using Microsoft.Extensions.Logging;
using Umbraco.Cms.Core.Web;
using Umbraco.Cms.Web.Common.Controllers;
namespace MvcCourse.Core.Controllers
{
  public class SearchPageController: RenderController
  {
     public SearchPageController(ILogger<SearchPageController> logger, ICompositeViewEngin
e compositeViewEngine,
     IUmbracoContextAccessor umbracoContextAccessor) : base(logger, compositeViewEngine, u
mbracoContextAccessor)
        {

        }
      public override IActionResult Index()
      {
          return base.Index();
      }
  }
}
```

If we build our project and place a breakpoint inside the `Index` method, we will see it hit each request to any page of the type SearchPage.

We are going to fetch all pages of the type `SearchContentModel` using our custom service and display them on search pages.

First, we are going to inject `ISearchService`, `IVariationContextAccessor`, and `ServiceContext` into our controller via a constructor:

```
private readonly ISearchService _searchService;
private readonly IVariationContextAccessor _variationContextAccessor;
private readonly ServiceContext _serviceContext;

public SearchPageController(
    ILogger<SearchPageController> logger,
    ICompositeViewEngine compositeViewEngine,
    IUmbracoContextAccessor umbracoContextAccessor,
    ISearchService searchService,
    IVariationContextAccessor variationContextAccessor,
    ServiceContext context) : base(logger, compositeViewEngine, umbracoContextAccessor)
       {
           _searchService = searchService;
           _variationContextAccessor = variationContextAccessor;
           _serviceContext = context;
       }
```

```
using SearchInV10.Core.Services; using Umbraco.Cms.Core.Web; using Umbraco.Cms.Web.Common.
Controllers;
```

Rename your `Index` method to `SearchPage` and write the following:

```
 public IActionResult SearchPage(string query)
        {
            //query might be null if we navigate across different language versions of the
 page
            if (query != null)
            {
                var searchResults = _searchService.GetContentSearchResults(query, out var
totalItemCount);
                var searchPageModel = new SearchModel(CurrentPage, new PublishedValueFallb
ack(_serviceContext, _variationContextAccessor))
                {
                    Query = query,
                    SearchResults = searchResults,
                    TotalResults = totalItemCount
                };
                return CurrentTemplate(searchPageModel);
            }
            else
            {
                var searchPageModel = new SearchModel(CurrentPage, new PublishedValueFallb
ack(_serviceContext, _variationContextAccessor))
                {
                    Query = "_",
                    SearchResults = Enumerable.Empty<SearchResultItem>(),
                    TotalResults = 0
                };
                return CurrentTemplate(searchPageModel);
            }
        }
```

```
using Umbraco.Cms.Core.Models; using SearchInV10.Core.Models;
```

We changed the `Index` method to `SearchPage` because on top of the default route, we can route via template. With this we ensure that if there was an alternate template to the Search, the code would only execute on the default SearchPage.cshtml template.

Enhancing the view

To get the benefit of the new model, we will need to change the model declaration in the **SearchPage.cshtml** file; the inherits declaration (in the code below) is how we register the custom model at the top of the view:

```
@using SearchInV10.Core.Models;
@inherits Umbraco.Cms.Web.Common.Views.UmbracoViewPage<SearchModel>
```

Excellent, let's update the template to loop through our search results and get each search result's `name`, `URL`, and `score`. Update the template to have the following configuration:

```
<article>
           <p>Your search for <strong>@Model.Query</strong> returned a number of <strong>
@Model.TotalResults</strong> result(s):</p>
           @if (Model.SearchResults != null && Model.SearchResults.Any())
           {
               foreach (var item in Model.SearchResults)
               {
                   <ul>
                       <li>
                           <a href="@item.PublishedItem.Url()">@item.PublishedItem.Name</a
>

                           <p>Score: @item.Score</p>
                       </li>
                   </ul>
               }
           }
</article>
```

Register the custom service

Umbraco is an application you can easily extend and fit your needs by composing it with different elements such as content finders, controllers, etc. We can use a composer to register our service.

First, make **/Composers** folder in your project, create a new class called **ExamineComposer.cs**.

In our **ExamineComposer** file, we can then register our **ISearchService**.

```
namespace SearchInV10.Core.Composers
{
   public class ExamineComposer : IComposer
   {
       public void Compose(IUmbracoBuilder builder)
       {
           builder.Services.AddSingleton<ISearchService, SearchService>();
       }
   }
}
```

```
using Microsoft.Extensions.DependencyInjection;using SearchInV10.Core.Services;using Umbra
co.Cms.Core.Composing; using Umbraco.Cms.Core.DependencyInjection;
```

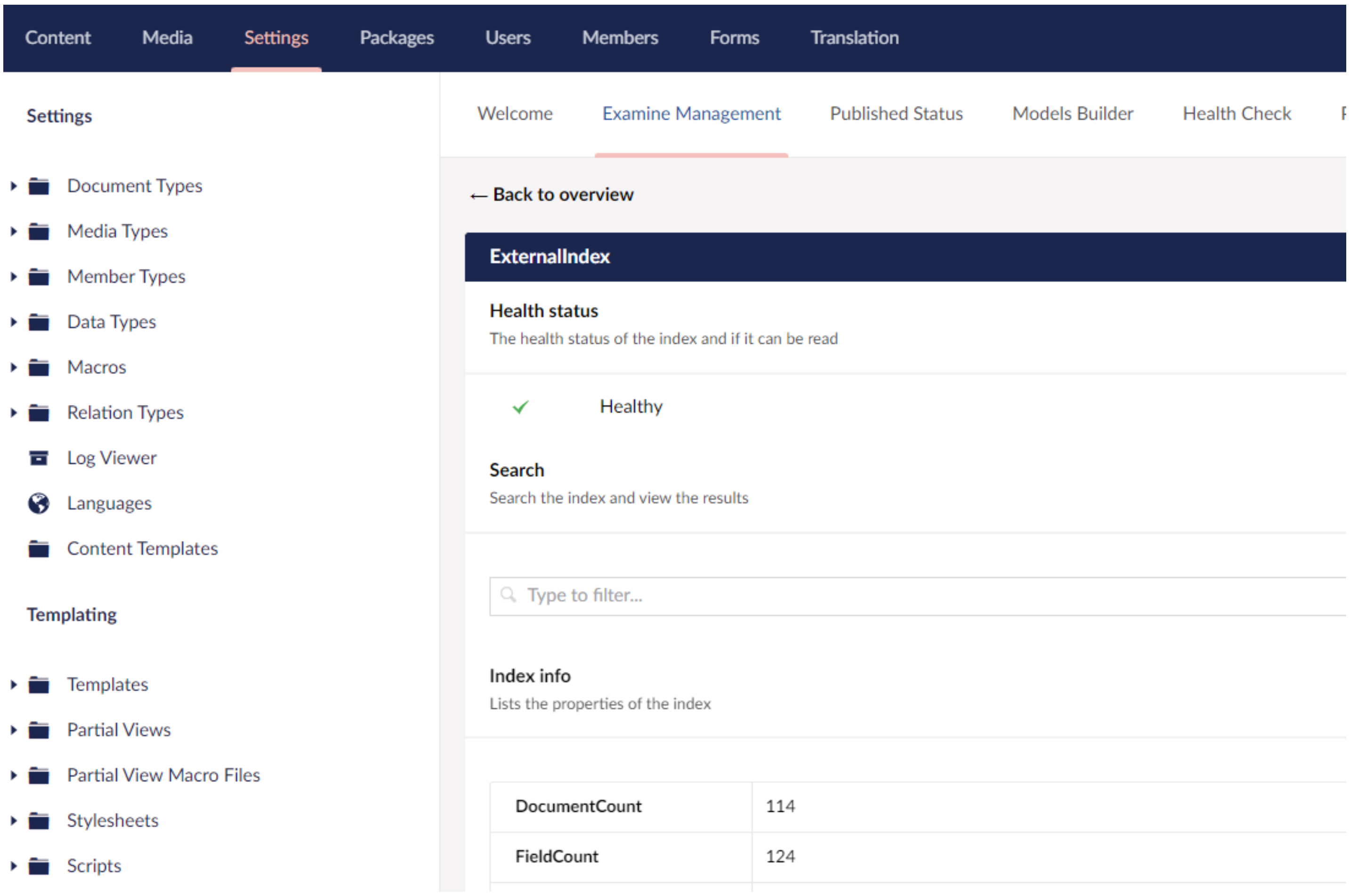
Test it out

With everything in place for our basic search, build your solution, and go to the front of your site and search for text that exists in one of our `bodyText` fields, such as "umbraco". As the site is using Umbraco language variants, also search the Danish version for "friheden" by going to /dk/ version of the page.

You will see that our basic search functionality is set up, however, we are only searching on the `bodyText` property. What if we want to search for more fields?

Rebuilding an Index in the backoffice

Next, login to the backoffice and rebuild your index (Indexes can be rebuilt via the Settings section - see figure below)



ExternalIndex

Health status
The health status of the index and if it can be read

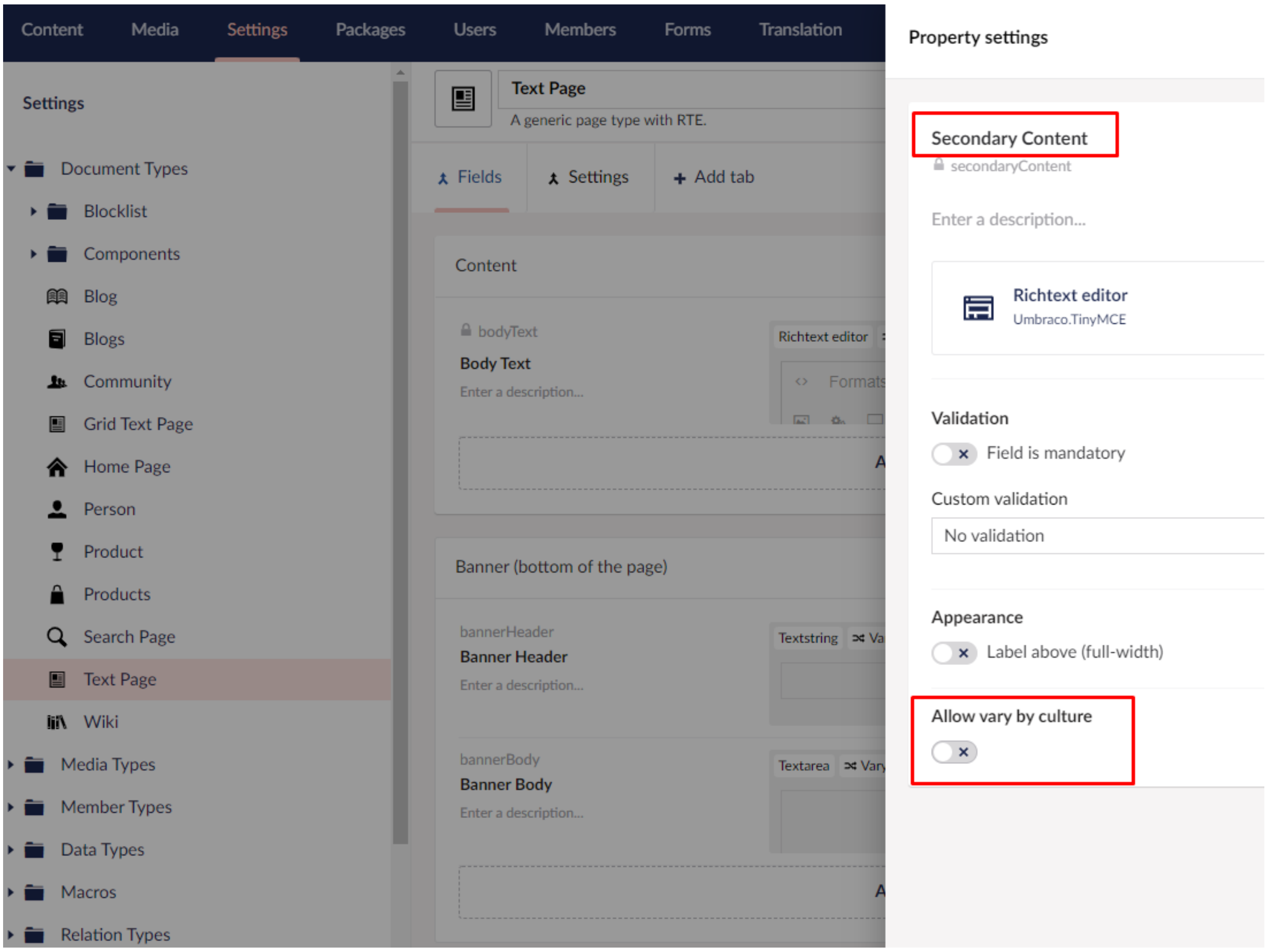
✓ Healthy

Search
Search the index and view the results

Index info
Lists the properties of the index

DocumentCount	114
FieldCount	124

Stay in the 'Settings' section and open the document type Text Page then add an **invariant** extra rich text field. Call it Secondary Content.


Property settings

Secondary Content

secondaryContent

Enter a description...

Richtext editor
Umbraco.TinyMCE

Validation

Field is mandatory

Custom validation

No validation

Appearance

Label above (full-width)

Allow vary by culture

Navigate to the ‘Content’ section, open existing text pages, and enter some content into this new field, (we would suggest keywords like “cat”, “dog”, “house” etc - something that would not exist on our site already) then ‘Save and Publish’.

After entering content, search for one of the words you entered, like ‘cat,’ you should get no results because the word is in the `secondaryContent` field, and we are not currently searching on that field.

Edit the search to now also search on the `secondaryContent` field. Edit the `examineQuery` in **/Services/SearchService** as shown below:

```
var examineQuery = criteria.GroupedOr(new[] {fieldToSearch, "secondaryContent"}, searchTer
m);
examineQuery.Not().Field(hideFromNavigation, 1.ToString());
```

Now search for the word ‘cat’ again. You should now see some search results.

Do you see a problem with this? Every time we add a new field to Umbraco, we will have to update our search code and add this new field.

Exercise 2: TransformingIndexValues

In the previous exercise, we determined that we would need to update our search code when adding new fields. Ideally, we do not want to do that. So in this exercise, using transforming index values, we will update the indexing process and also our search.

1. In Visual Studio add a folder called **Components**
2. Create a file **ExamineComponents.cs** in **Components**
3. Copy the contents of ExamineComponents.txt from the course files into the new class.

The composer is responsible for registering the component within the Umbraco Application. The component `Initialize` method is where we will tie into aspects such as `EventHandlers`. All composers and components need to inherit from `IComposer` and `IComponent`, respectively.

✍ Note : For more documentation on Composing Umbraco, visit: https://our.umbraco.com/documentation/Implementation/Composing/

With our basic layout in place, we will now set up our `TransformingIndexValues EventHandler`. Before we can access the `TransformingIndexValues` event, we need to add some error handling. In the `Initialize` method, add the following:

```
if (!_examineManager.TryGetIndex(Constants.UmbracoIndexes.ExternalIndexName, out var exter
nalIndex))
   throw new InvalidOperationException($"No index found by name {Constants.UmbracoIndexes.
ExternalIndexName}");
```

Next (still in our `Initialize` method) we need to cast because the `BaseIndexProvider` contains the `TransformingIndexValues` event

```
if (!(externalIndex is BaseIndexProvider indexProvider))
   throw new InvalidOperationException("Could not cast");
```

With these in place we can now add our `TransformingIndexValues EventHandler`:

```
indexProvider.TransformingIndexValues += IndexProviderTransformingIndexValues;
```

In the method `IndexProviderTransformingIndexValues` add the following code:

```
  private void IndexProviderTransformingIndexValues(object sender, IndexingItemEventArgs e
)
        {
            if (e.ValueSet.Category == IndexTypes.Content)
            {
                try
                {
                    IEnumerable<ILanguage> languages = _localizationService.GetAllLanguage
s();
                    string variesByCulture = null;


                    if (e.ValueSet.Values.TryGetValue("__VariesByCulture", out var result)
)
                    {
                        variesByCulture = (string)result[0];
                    };

                    var updatedValues = e.ValueSet.Values.ToDictionary(x => x.Key, x => x.
Value.ToList());

                    if (variesByCulture != null && variesByCulture == "y")
                    {
                        foreach (var language in languages)
                        {
                            var languageIsoCode = language.IsoCode.ToLower();

                            var cultureAndInvariantFields = GetCultureAndInvariantFields(u
pdatedValues, languageIsoCode);
                            var combinedFieldsLang = new StringBuilder();


                            foreach (var field in cultureAndInvariantFields.Where(x => !x.
StartsWith("contents") && !x.StartsWith("__Raw")))
                            {
                                updatedValues.TryGetValue(field, out List<object> values);

                                if (values != null)
                                    foreach (var value in values)
                                    {
                                        if (value != null)
                                            combinedFieldsLang.AppendLine(value.ToString()
);

                                    }
                            }

                            updatedValues.Add("contents_" + languageIsoCode, new List<obje
ct> { combinedFieldsLang.ToString() });

                            e.SetValues(updatedValues.ToDictionary(x => x.Key, x => (IEnum
erable<object>)x.Value));
                        }

                    }
                    else
                    {
                        var combinedFields = new StringBuilder();
                        foreach (var fieldValues in e.ValueSet.Values)
                        {
                            foreach (var value in fieldValues.Value)
                            {
                                if (value != null)
                                    combinedFields.AppendLine(value.ToString());
                            }
                        }

                        updatedValues.Add("contents", new List<object> { combinedFields.To
String() });

                        e.SetValues(updatedValues.ToDictionary(x => x.Key, x => (IEnumerab
le<object>)x.Value));
                    }
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Error combining fields for {ValueSetId}", e.Valu
eSet.Id);
                }
            }
        }
```

 If Visual Studio does not recognize IndexTypes AND you are using a .Core project, add a reference to the Umbraco.Examine.dll from the .Web project to your .Core project

Register the ExamineComponents

Once we have the configuration in place, let's register our component inside of the composer file.

Head over to the **Composers** folder and open the **ExamineComposer** file. Update it to include the newly created file as well.

```
public class ExamineComposer : IComposer
   {
       public void Compose(IUmbracoBuilder builder)
       {
           builder.Services.AddSingleton<ISearchService, SearchService>();
           builder.Components().Append<ExamineComponents>();
       }
   }
```

```
using SearchInV10.Core.Components;
```

Update your search code to search on the newly injected `contents` fields as below

```
var fieldToSearchLang = "contents" + "_" + CultureInfo.CurrentCulture.ToString().ToLower()
;
var fieldToSearchInvariant = "contents";
var hideFromNavigation = "umbracoNaviHide";
var fieldsToSearch = new[] {fieldToSearchLang, fieldToSearchInvariant};
var criteria = searcher.CreateQuery(IndexTypes.Content);
var examineQuery = criteria.GroupedOr(fieldsToSearch, searchTerm);
examineQuery.Not().Field(hideFromNavigation, 1.ToString());
```

Now you can add any number of fields to your document type, and your search code will still work.

Exercise 3 - Wildcard Searches

With our indexing process in place, we can now address one of the shortcomings of our current search implementation. In this exercise, we will account for wildcard searches when a word is potentially incomplete.

Try searching for the word “suppo” Do you get any results?

We need to update the query so that we have wildcard searches. So where you have:

```
var examineQuery = criteria.GroupedOr(fieldsToSearch, searchTerm);
```

Add the extension method to `searchTerm` as shown below:

```
var examineQuery = criteria.GroupedOr(fieldsToSearch, searchTerm.MultipleCharacterWildcard
());
```

Query Debugging

To help debug queries, we can write out the generated raw Lucene query by Examine.

Open your **SearchService.cs** file and add the following code inside of the `if` statement:

```
if (results.Any())
```

add:

```
Debug.WriteLine(criteria.ToString());
```

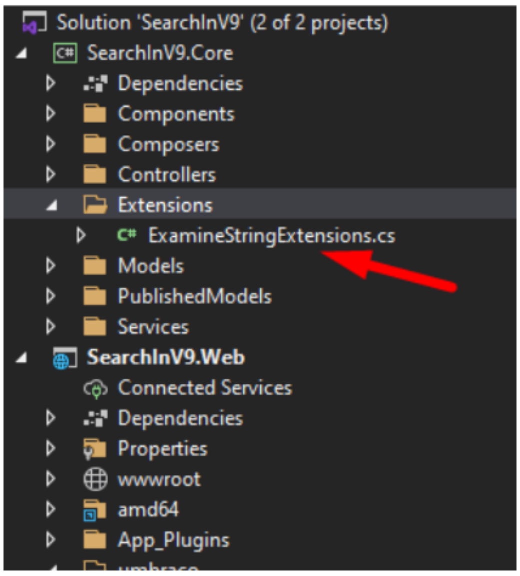
Now try searching for ‘suppo,’ and you should get some results. Take a look at the generated query in the Output window in Visual Studio. You will notice it has *, which is the Lucene query syntax wildcard character.

✪ Going further: edge cases and special characters

Search on * or ? on its own, you should get an error, so wrap the code in a “try catch” and write out the exception. Lucene has reserved characters that have specific meanings in a query that we need to test for and remove.

To do this we can create an extension method to clean query terms.

So in Visual Studio make a new folder called **Extensions** , and inside of it create a `static class` called **ExamineStringExtensions**.



In the class add the following code:

```
public static class ExamineStringExtensions
{
 public static string MakeSearchQuerySafe(this string query)
 {
      return query?.Replace("*", string.Empty).Replace("?", string.Empty);
 }
}
```

```
using System.Text.RegularExpressions;
```

Next, in your **SearchService.cs** update the search term just before trying to get the Examine Searcher.

```
searchTerm = searchTerm.MakeSearchQuerySafe();
if(string.IsNullOrEmpty(searchTerm))
 {
  return Array.Empty<ISearchResult>();
 }
var searcher = index.Searcher;
var fieldToSearchLang = "contents" + "_" + CultureInfo.CurrentCulture.ToString().ToLower()
;
```

```
using SearchCourse.Core.Extensions;
```

Going further: fuzzy search

What if you want to account for possible mistakes, such as entering "umbrako" instead of "umbraco" for search? You can adjust how "fuzzy" the search is by replacing

```
var examineQuery = criteria.GroupedOr(fieldsToSearch, searchTerm.MultipleCharacterWildcard
());
```

with

```
var examineQuery = criteria.GroupedOr(fieldsToSearch, searchTerm.Fuzzy(0.2f));
```

where the `Fuzzy` method takes in a parameter to customize the degree of fuzziness. The higher the number, the more forgiving it will be and - at some point - it might actually lead to inaccurate results if the number is set to higher values.

More searching tricks can be found at the offficial Examine docs at:

https://shazwazza.github.io/Examine/searching

Additional notes to acknowledge

The Examine search results Object implements its own take and skip. Also, the search method has overrides. One override allows you to set the max number of results to send back. Use these methods and not LINQ. Further to this **DO NOT DO FURTHER FILTERING OF RESULTS USING LINQ** for example:

```
//do not do this
var results = examineQuery.Execute();
 results
   .Where(x=>x.Values["nodeName"]=="whatever")
   .OrderBy("nodeName")
   .Take(10);
```

Another way to view this is that searching IS filtering, you should not post filter and don't post sort, because then the Score is irrelevant.

✍ Very important note:

When working with TransformingIndexValues you may need to get content from a node e.g you have multi node tree picker property and you want to get some values from those picked items to inject into your index.

You can try and instantiate an Umbraco helper object in your `TransformingIndexValues` data class however you will get a null exception error.

There is then a temptation to use `ContentService` to get these values. The problem here is that `ContentService` makes hits to the database. So imagine you have 30k blog posts each one has 3 or more tags.

If you use ContentService you will potentially make 2*30k database hits during a full re-index. This will potentially bring your site down. It will definitely give you index rebuild times of >20 minutes.

If you are on Azure this can be a major problem because Azure web apps will swap file stores and this triggers index rebuilds.

So always use `IPublishedContent` - the code sample below illustrates how to do this:

```
private void IndexProviderTransformingIndexValues(object sender, IndexingItemEventArgs e)
{
   using (var umbCtxRef = _umbracoContextFactory.EnsureUmbracoContext())
   {
       var contentCache = umbCtxRef.UmbracoContext?.ContentCache;
       if (contentCache == null)
           throw new InvalidOperationException("Could not acquire content cache");
       AddSearchFields(sender, e, umbCtxRef.UmbracoContext?.ContentCache);
   }
}
private void AddSearchFields(object sender, IndexingItemEventArgs args, IPublishedContentC
ache contentCache)
{
   if (contentCache == null) throw new ArgumentNullException(nameof(contentCache));
   var item = contentCache.GetById(1234);
}
```

Exercise 4 - Split Terms

In the previous exercise, we made our search more user-friendly by allowing search on incomplete keywords. With this in place, our search functionality is starting to take shape. One thing that we haven’t accounted for is how to handle more than one word/value being queried. In this exercise, we will dive into a simple debugging practice to understand the problem at hand and then split the entered terms to account for all results.

Query Debugging

To help debug queries let’s look at our Output dialog in Visual Studio (from the `Debug.WriteLine` part in the last exercise). A query is broken up into terms and operators. There are two types of terms: Single Terms and Phrases. A Single Term is a single word such as "central" or "government". A Phrase is a group of words surrounded by double quotes such as "central government". Multiple terms can be combined together with Boolean operators to form a more complex query.

✍ Note: For more information on Lucene query parsing visit: http://lucene.apache.org/core/2_9_4/queryparsersyntax.html

We have 2 options. We can leave the search as it currently is so that we perform an actual phrase search when more than one word has been entered. Alternatively, we can update the query so that we search on all the terms with an ‘or’ query.

Take your query term and test if it has whitespace, if it does then split it on the space character. We also need to re-write the initial code in **SearchService.cs** to code below:

```
var criteria = searcher.CreateQuery(IndexTypes.Content);
IBooleanOperation examineQuery;

if (searchTerm.Contains(" "))
{
    string[] terms = searchTerm.Split(' ');
    examineQuery = criteria.GroupedOr(fieldsToSearch, terms);
}
else
{
    examineQuery = criteria.GroupedOr(fieldsToSearch, searchTerm.MultipleCharacterWildcard
());
}

examineQuery.Not().Field(hideFromNavigation, 1.ToString());

var results = examineQuery.Execute();
```

Retry your searches from the previous exercise and see how it works!

Exercise 5 - Sorting

By default, search results are sorted by `Score`, and you can see this being written out in your search results list. So let’s change the sort parameter to be `createDate`.

Before updating the sort, create some new pages, then ‘Save and Publish’.

These newly created pages will have today’s date (we are using the default IPublishedContent creation date).

In the **/Views/SearchPage** below the following line:

```
<p>Score - @item.Score</p>
```

Add

```
<br /> <p>date: @(item.PublishedItem.CreateDate)</p>
```

By default, `createDate` is a date in Examine stored as the type Long, so we need to sort based on its real value rather than a string. So, update the query (just before `var results`) to include the `SortType.Long` parameter:

```
examineQuery.OrderByDescending(new SortableField("createDate", SortType.Long));
```

We’re seeing `NaN` for score, but we have successfully sorted on one of the key Umbraco fields. What we will now do is try and sort on a custom date field. On the Blog document type, there is already a custom date property called “Publishing Date” using a datepicker.

Update some of the content items (not all) in the English site and give them a ‘Publishing Date’ then ‘Save and Publish’. Next, update your search code in **/Services/SearchService** to sort on that field.

Change

```
examineQuery.OrderByDescending(new SortableField("createDate", SortType.Long));
```

To

```
examineQuery.OrderByDescending(new SortableField("publishingDate", SortType.Long));
```

Also, in the **/Views/SearchPage** below the following line:

```
<p>Score - @item.Score</p>
```

Add

```
<br /> <p>date: @(item.PublishedItem.Value<DateTime>("publishingDate"))</p>
```

Execute a search - does it sort? Why not?

Before we answer that, it's worth noting that when we add a sort field, relevance/score appears as `NaN`.
It makes sense as we are sorting ourselves and not relying on Lucene's score, hence either Examine or Lucene does not give you a score as it is no longer relevant. Now back to that question, why doesn't our search sort?

Modifying Examine

Our custom Date field is not being used to sort, and the reason is that it has not been added to the index as a sortable field. Configuration of Examine indexes is done by using the .NET's Options pattern. For Examine, this is done with named options: `IConfigureNamedOptions`.

Therefore, to enable sorting, we need to add a field definition for the custom date.

Add a new file called **ConfigureCustomFieldOptions.cs**. Copy the contents of the corresponding code snippet to that new file.

```
    public class ConfigureCustomFieldOptions : IConfigureNamedOptions<LuceneDirectoryIndex
Options>
    {
        public void Configure(string name, LuceneDirectoryIndexOptions options)
        {
            if (name.Equals(Constants.UmbracoIndexes.ExternalIndexName))
            {
                options.FieldDefinitions.AddOrUpdate(new FieldDefinition("publishingDate",
                FieldDefinitionTypes.DateTime));
            }
        }

        public void Configure(LuceneDirectoryIndexOptions options)
        {
            throw new NotImplementedException();
        }
    }
```

```
using Examine; using Examine.Lucene; using Microsoft.Extensions.Options; using Umbraco.Cms
.Core;
```

Tip: There are several options that can be configured, the most common ones are:

- `FieldDefinitions FieldDefinitionCollection` - Manages the mappings between a field name and its index value type
- `Analyzer Analyzer` - The default Lucene Analyzer to use for each field (default = StandardAnalyzer)
- `Validator IValueSetValidator` - Used to validate a value set to be indexed, if validation fails it will not be indexed
- `IndexValueTypesFactory IReadOnlyDictionary<string, IFieldValueTypeFactory>` - Allows you to define custom Value Types

Since we have created a new component, we need to register it with our composer. We could either create a new composer or add this to our existing **ExamineComposer.cs**. For ease, we will add to our existing composer. Go to the **ExamineComposer.cs** and register our new component by adding the following:

```
builder.Services.ConfigureOptions<ConfigureCustomFieldOptions>();
```

The last thing to consider is that when we have no 'publishing date,' which date do we use? Ideally we would fall back to either the last update date, or create date - which has to be done at index time.

The following snippet is an updated search results code that tests for publishing date - put this in our view:

```
<article>
    <p>Your search for <strong>@Model.Query</strong> returned a number of <strong> @Model.
TotalResults</strong> result(s):</p>
        @if (Model.SearchResults != null && Model.SearchResults.Any())
        {
            foreach (var item in Model.SearchResults)
            {
            <ul>
                <li>
                    <a href="@item.PublishedItem.Url()">@item.PublishedItem.Name</a>
                        @{
                        var pDate = item.PublishedItem.Value<DateTime>("publishingDate");
                        if (pDate == DateTime.MinValue)
                            {
                                pDate = item.PublishedItem.CreateDate;
                            }
                            if (item.PublishedItem.GetType().ToString() == "Umbraco.Cms.We
b.Common.PublishedModels.Blog")
                            {
                                <p>@pDate.ToString("dd MMMM yyyy")</p>
                            }
                        }
                </li>
            </ul>
            }
        }
</article>
```

As for the logic, go back to **ExamineComponents.cs** and replace `IndexProviderTransformingIndexValues` method with:

```
private void IndexProviderTransformingIndexValues(object sender, IndexingItemEventArgs e)
    {
        if (e.ValueSet.Category == IndexTypes.Content)
        {
            try
            {
                IEnumerable<ILanguage> languages = _localizationService.GetAllLanguages();
                string variesByCulture = null;


                if (e.ValueSet.Values.TryGetValue("__VariesByCulture", out var result))
                {
                    variesByCulture = (string)result[0];
                };

                var updatedValues = e.ValueSet.Values.ToDictionary(x => x.Key, x => x.Valu
e.ToList());

                if (variesByCulture != null && variesByCulture == "y")
                {
                    foreach (var language in languages)
                    {
                        var languageIsoCode = language.IsoCode.ToLower();

                        var cultureAndInvariantFields = GetCultureAndInvariantFields(updat
edValues, languageIsoCode);
                        var combinedFieldsLang = new StringBuilder();


                        foreach (var field in cultureAndInvariantFields.Where(x => !x.Star
tsWith("contents") && !x.StartsWith("__Raw")))
                        {
                            updatedValues.TryGetValue(field, out List<object> values);

                            if (values != null)
                                foreach (var value in values)
```

```
                                 {
                                     if (value != null)
                                         combinedFieldsLang.AppendLine(value.ToString());
                                 }
                         }

                         updatedValues.Add("contents_" + languageIsoCode, new List<object>
{ combinedFieldsLang.ToString() });

                         e.SetValues(updatedValues.ToDictionary(x => x.Key, x => (IEnumerab
le<object>)x.Value));
                     }

                 }
                 else
                 {
                     var combinedFields = new StringBuilder();
                     foreach (var fieldValues in updatedValues)
                     {
                         foreach (var value in fieldValues.Value)
                         {
                             if (value != null)
                                 combinedFields.AppendLine(value.ToString());
                         }
                     }

                     updatedValues.Add("contents", new List<object> { combinedFields.ToStri
ng() });

                     e.SetValues(updatedValues.ToDictionary(x => x.Key, x => (IEnumerable<o
bject>)x.Value));
                 }
             }
             catch (Exception ex)
             {
                 _logger.LogError(ex, "Error combining fields for {ValueSetId}", e.ValueSet
.Id);
             }

             try
             {
                 var updatedValues = e.ValueSet.Values.ToDictionary(x => x.Key, x => x.Valu
e.ToList());
                 string[] alias = new string[1] { "blog" };
                 string blogTypeId = _contentTypeService.GetAllContentTypeIds(alias).FirstO
rDefault().ToString();
                 updatedValues.TryGetValue("nodeType", out var nType);
                 updatedValues.TryGetValue("publishingDate", out var publishingDate);
                 updatedValues.TryGetValue("createDate", out var fallbackDate);
                 //good to know - dates are saved in DateTime.Ticks
                 //we only do this on Blog doctype
                 if (nType.FirstOrDefault().ToString() == blogTypeId)
                 {
                     if (publishingDate == null)
                     {
                         updatedValues.Remove("publishingDate");
                         //cleared the publishingDate value, let's add one that is filled o
ut
                         updatedValues.Add("publishingDate", fallbackDate );
                     }
                     else
                     {
                         updatedValues.Remove("publishingDate");
                         updatedValues.Add("publishingDate", publishingDate);
                     }
                     e.SetValues(updatedValues.ToDictionary(x => x.Key, x => (IEnumerable<o
bject>)x.Value));
                 }
```

```
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Error setting the fallback for PublishingDate");
            }
        }
    }
```

The above code will check the Blog items for `PublishDate`; if it is empty, it will default to `CreateDate`. We convert both dates to Ticks (C# datetime ticks), to make sure they follow the same convention as `CreateDate` stored in our index.

With all of this done, perform a search on the blog items (search for “ipsum”) and verify the items are sorted. You might need to rebuild the ExternalIndex to see changes.

Exercise 6: Narrowing search down

We have the search working, it sorts our items by date, however we do not really have a way of searching across a specific content type - let’s say, Blog items or Text Pages only. This we can remedy quite easily by making a few minor modifications.

1. Let us start with the model - go back to **SearchModel.cs** file and add a new property:

```
public string DoctypeToSearch { get; set; }
```

2. Next, onto **ISearchService** - in each of the methods we have, add a new parameter

```
string contentType
```

After our changes the methods will look like this:

```
IEnumerable<ISearchResult> GetSearchResults(string searchTerm, string contentType, out lon
g totalItemCount);
IEnumerable<SearchResultItem> GetContentSearchResults(string searchTerm, string contentTyp
e, out long totalItemCount);
```

3. Next up - **SearchService.cs**. Add `string contentType` as a parameter to both methods in the class:

```
GetContentSearchResults(string searchTerm, string contentType, out long totalItemCount)

...

GetSearchResults(string searchTerm, string contentType, out long totalItemCount)
```

We will get the `contentType` string from our form when it submits, it will be passed to the Controller, and end up in our Service.

4. Under `GetContentSearchResults`, change the first line of the method to:

```
var pageOfResults = GetSearchResults(searchTerm, contentType, out totalItemCount);
```

5. In `GetSearchResults` method down below, before you execute the query, add this bit of code:

```
if (contentType != "All")
{
    examineQuery.And().Field("__NodeTypeAlias", contentType);
}
```

The `NodeTypeAlias` here corresponds to the item’s document type alias. We will use a dropdown to pick from “All” document types to search through, and specific ones we will gather via a corresponding Umbraco service.

6. Move to our **SearchPageController** - the `SearchPage` method should accept a new `doctypeToSearch` parameter. Make sure it looks like this:

```
 public IActionResult SearchPage(string query, string doctypeToSearch)
        {
            //query might be null if we navigate across different language versions of the
 page
            if (query != null)
            {
                var searchResults = _searchService.GetContentSearchResults(query, doctypeT
oSearch, out var totalItemCount);
                var searchPageModel = new SearchModel(CurrentPage, new PublishedValueFallb
ack(_serviceContext, _variationContextAccessor))
                {
                    Query = query,
                    DoctypeToSearch = doctypeToSearch,
                    SearchResults = searchResults,
                    TotalResults = totalItemCount
                };
                return CurrentTemplate(searchPageModel);
            }
            else
            {
                var searchPageModel = new SearchModel(CurrentPage, new PublishedValueFallb
ack(_serviceContext, _variationContextAccessor))
                {
                    Query = "_",
                    DoctypeToSearch = "All",
                    SearchResults = Enumerable.Empty<SearchResultItem>(),
                    TotalResults = 0
                };
                return CurrentTemplate(searchPageModel);
            }
        }
```

We are “just” passing this `doctypeToSearch` string along from the submitted form to our Service, and our Model - so we will be able to tell the user which Document Type they searched on.

7. Now for the final touches - on the **SearchPage.cshtml**, where we show our results, replace the text with:

```
 <p>Your search for <strong>@Model.Query</strong> returned a number of <strong> @Model.Tot
alResults</strong> result(s) on item types: <strong>@Model.DoctypeToSearch</strong></p>
```

8. On the Search.cshtml under **Views/Partials** folder, at the very top of the template add these two lines:

```
@using Umbraco.Cms.Core.Services
@inject IContentTypeService ContentTypeService
```

We are injecting a `ContentTypeService` into our views with the `@inject` keyword. It is one of the services we can use in Controllers, as well - we just get to it in a slightly different way in Razor views.

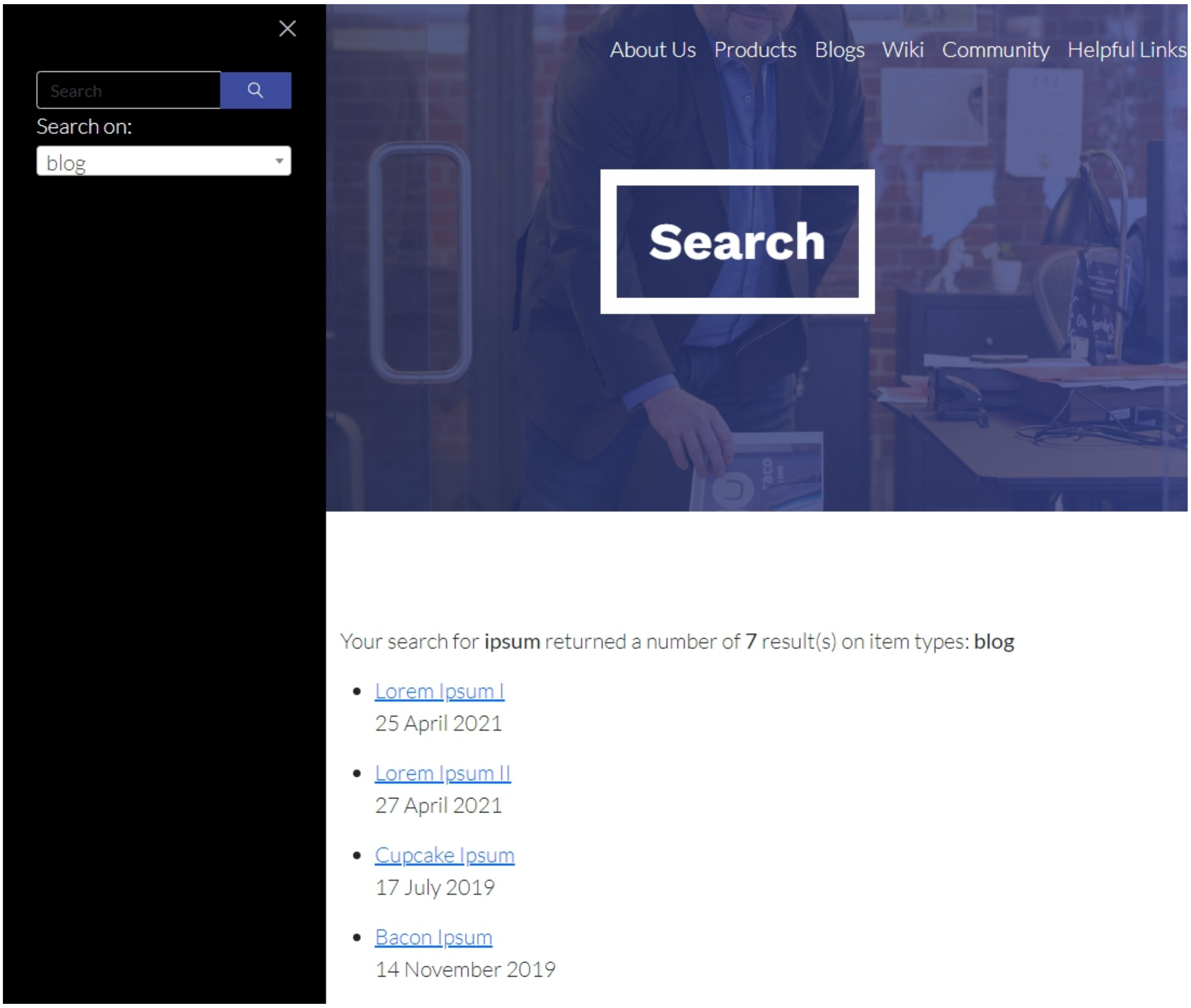
9. In the code block below, make sure to define your doctypes collection as:

```
var docTypes = ContentTypeService.GetAllContentTypeAliases();
```

10. Finally, just before the ending `</form>` tag, add this code to render a dropdown menu:

```
Search on:
<select style="width:100%" id="doctypeToSearch" name="doctypeToSearch">
    <option>All</option>
    @foreach (var docType in docTypes)
    {
        <option>@docType</option>
    }
</select>
```

Then test it out! You should see a result similar to this one:



Your search for **ipsum** returned a number of **7** result(s) on item types: **blog**

- Lorem Ipsum I
  25 April 2021
- Lorem Ipsum II
  27 April 2021
- Cupcake Ipsum
  17 July 2019
- Bacon Ipsum
  14 November 2019

Exercise 7: PDF indexing

Now that we can index and search our text-based content as we would like, it is time to look into indexing and searching for more advanced data, such as PDF files.

With Examine out of the box, you get three indexes:

- External - Contains all published content
- Internal - Contains all published and unpublished content used by the internal search in the Umbraco back office.
- Member - Index of site members

If we want to index PDF content in the media section, we need to use the *Umbraco.ExaminePdf* extension package.

If you are using Visual Studio for this course, you can install the package using Package Manager. Install on the project by typing the following command (if you have a .Core project, make sure to install the package on both projects in the solution):

```
Install-Package Umbraco.ExaminePDF -Version 10.0.0
```

👆 Tip: You can install the package in both .Core and .Web projects by using the .NET CLI. Inside of the root of each project, run the following command

```
dotnet add package Umbraco.ExaminePDF --version 10.0.0
```

After installing the package, building the solution, you'll end up with a new index called PDFIndex. The PDFIndex indexes the content of the PDF to a field called `fileTextContent`.

In the Media section, upload some PDFs - there are two available in the course files. Use the internal Umbraco Examine dashboard to see if you have any content in the pdf index. Searching the contents of this Index requires the creation of a `Searcher` and a `query`. If you want to search over the content, media, and PDFs, you'll need to create a multi-index searcher that spans the PDFIndex and the External one. We will create a multi-searcher in the next exercise.

Exercise 8 - Multi index search

In this exercise, we will update our search code to search over our Umbraco content index contained in 'ExternalIndex' and our media index or specifically PDF index, which we created in the previous exercise.

To use the multisearcher in Umbraco 10, we can register a multi-index searcher with the `ExamineManager` on startup. However, to create the Searcher at startup, we must use dependency injection composers and components.

In the composer, we have a dependency that it must run after ExaminePDF; the component creates the `Searcher` and registers it.

Head over to your **ExamineComposer** class and add the following code above the ExamineComposer public class:

```
[ComposeAfter(typeof(ExaminePdfComposer))]
```

```
using UmbracoExamine.PDF;
```

Open the **Startup.cs** class. In the `AddUmbraco` method update the following code to get both the external and the PDF index.

FROM

```
services.AddUmbraco(_env, _config)
                .AddBackOffice()
                .AddWebsite()
                .AddComposers()
                .Build();
```

TO:

```
services.AddUmbraco(_env, _config)
                .AddBackOffice()
                .AddWebsite()
                .AddComposers()
                .Build();
               services.AddExamineLuceneMultiSearcher("MultiSearcher", new string[] {
                Constants.UmbracoIndexes.ExternalIndexName, PdfIndexConstants.PdfIndexName
            });
```

```
using UmbracoExamine.PDF; using Examine; using Umbraco.Cms.Core;
```

With the above approach, we get both the external, the PDF index and register a multisearcher for the indexes.

Once registered, we can use `TryGetSearcher` in code to perform a query. For example, in Visual Studio, open the **/Services/SearchService.cs** and update the following code.

FROM:

```
if (_examineManager.TryGetIndex(Constants.UmbracoIndexes.ExternalIndexName, out IIndex ind
ex))
{

searchTerm = searchTerm.MakeSearchQuerySafe();
if(string.IsNullOrEmpty(searchTerm))
  {
   return Array.Empty<ISearchResult>();
  }
var searcher = index.Searcher();

var fieldToSearchLang = "contents" + "_" + CultureInfo.CurrentCulture.ToString().ToLower()
;
var fieldToSearchInvariant = "contents";
var hideFromNavigation = "umbracoNaviHide";
var fieldsToSearch = new[] { fieldToSearchLang, fieldToSearchInvariant };
var criteria = searcher.CreateQuery(IndexTypes.Content);
```

TO:

```
if (_examineManager.TryGetSearcher("MultiSearcher", out var multiSearcher))
{
searchTerm = searchTerm.MakeSearchQuerySafe();
if(string.IsNullOrEmpty(searchTerm))
  {
   return Array.Empty<ISearchResult>();
  }
var fieldToSearchLang = "contents" + "_" + CultureInfo.CurrentCulture.ToString().ToLower()
;
var fieldToSearchInvariant = "contents";
var hideFromNavigation = "umbracoNaviHide";
var pdfTextContent = "fileTextContent";
var fieldsToSearch = new[] { fieldToSearchLang, fieldToSearchInvariant, pdfTextContent };
var criteria = multiSearcher.CreateQuery(null, BooleanOperation.Or);
```

✍ Note: `fileTextContent` is the name of the field in the Examine pdf index in which the Pdf indexer stores our pdf content.

Furthermore, to display the search results based on the multisearcher and take the media items into account, we need to make extra adjustments in the `GetContentSearchResults` in the **SearchService** class.

The first thing to consider is that we want to display media items as well. Therefore, add the following markup **below the page variable** to get the media item by id and take error handling into account.

```
//var page = umbracoContext.Content.GetById(int.Parse(item.Id));
var pageMedia = umbracoContext.Media.GetById(int.Parse(item.Id));
if (page != null)
     {
       items.Add(new SearchResultItem()
                   {
                           PublishedItem = page,
                           Score = item.Score
                       });
                   }
                   if (pageMedia != null)
                   {
                       items.Add(new SearchResultItem() {
                       PublishedItem = pageMedia,
                       Score = item.Score
                       });
                   }
```

Now search for the word ‘umbraco’. You should now see some search results.

Next, try searching for the word ‘enterprise’ (word only present in one of the pdfs). You also should see a search result.

Add the word “enterprise” to some content in Umbraco and retry the search, you should see results for both the PDF and the content.

Exercise 9 - Custom indexing

On a day-to-day job, there may be a requirement to search using Examine for non-Umbraco content, e.g., external content in a database table or being pulled from an API. Creating your indexer to index this third-party content is pretty straightforward. In this exercise, we will build our custom database indexer.

The process of creating a custom index requires us to have some form of external data. For simplicity, we are going to use a JSON feed of a simple lorem ipsum ToDo List.

You can see the JSON here: https://jsonplaceholder.typicode.com/todos/

Now that we have our dummy data, we need to create three classes. A **ValueSetBuilder**, an **IndexCreator**, and an **IndexPopulator**. We have already set up the **TodoModel** in the code snippets establishing the properties to account for.

Model

Make a new class called **TodoModel.cs** and paste in this code:

```
public class ToDoModel
{
    public int UserId { get; set; }
    public int Id { get; set; }
    public string Title { get; set; }
    public bool Completed { get; set; }
}
```

ValueSetBuilder

Next, our **ValueSetBuilder**. Make a new class called **TodoValueSetBuilder.cs**.

The completed ValueSetBuilder will contain the following:

```
public class TodoValueSetBuilder : IValueSetBuilder<ToDoModel>
{
   public IEnumerable<ValueSet> GetValueSets(params ToDoModel[] data)
   {
       foreach (var todo in data)
       {
           var indexValues = new Dictionary<string, object>
           {
               ["userId"] = todo.UserId,
               ["id"] = todo.Id,
               ["title"] = todo.Title,
               ["completed"] = todo.Completed
           };
           var valueSet = new ValueSet(todo.Id.ToString(), "todo", indexValues);
           yield return valueSet;
       }
   }
}
```

```
using Examine; using Umbraco.Cms.Infrastructure.Examine; using System.Collections.Generic;
```

Our value set builder is converting the to-do objects coming from the JSON feed into a ValueSet.

IndexPopulator

Next, we need to configure our **IndexPopulator**. This class is responsible for populating our index based on the data that it is pulling from the JSON feed.

Add a new class to the **CustomIndex** directory called **TodoIndexPopulator**. Once created add the following code:

```
public class TodoIndexPopulator : IndexPopulator
{
    private readonly TodoValueSetBuilder _todoValueSetBuilder;
    public TodoIndexPopulator(TodoValueSetBuilder productValueSetBuilder)
    {
        _todoValueSetBuilder = productValueSetBuilder;
        //We're telling this populator that it's responsible for populating only our index
        RegisterIndex("TodoIndex");
    }
    protected override void PopulateIndexes(IReadOnlyList<IIndex> indexes)
    {
        using (WebClient httpClient = new WebClient())
        {
            var jsonData = httpClient.DownloadString("https://jsonplaceholder.typicode.com
/todos/");
            var data = JsonConvert.DeserializeObject<IEnumerable<ToDoModel>>(jsonData);
            if (data != null)
            {
                foreach (var item in indexes)
                {
                    item.IndexItems(_todoValueSetBuilder.GetValueSets(data.ToArray()));
                }
            }
        }
    }
}
```

```
using Examine; using Newtonsoft.Json; using System.Collections.Generic; using System.Linq;
 using System.Net; using Umbraco.Cms.Infrastructure.Examine;
```

CustomIndexOptions

We also need to configure our **CustomIndexOptions**. The **CustomIndexOptions** will handle the implementation of our index and implement our marker interface. It's a similar workflow one of the previous exercises when we added a new field definition.

Add a new class to the **CustomIndex** directory called **ConfigureCustomIndexOptions.cs**. Once created, add the following code:

```
public class ConfigureCustomIndexOptions : IConfigureNamedOptions<LuceneDirectoryIndexOpti
ons>
    {
        private readonly IOptions<IndexCreatorSettings> _settings;
        public ConfigureCustomIndexOptions(IOptions<IndexCreatorSettings> settings)
        {
            _settings = settings;
        }
        public void Configure(string name, LuceneDirectoryIndexOptions options)
        {
            if (name.Equals("TodoIndex"))
            {
                options.Analyzer = new StandardAnalyzer(LuceneVersion.LUCENE_48);
                options.FieldDefinitions = new FieldDefinitionCollection(
                    new FieldDefinition("userID", FieldDefinitionTypes.Integer),
                    new FieldDefinition("id", FieldDefinitionTypes.Integer),
                    new FieldDefinition("title", FieldDefinitionTypes.FullTextSortable),
                    new FieldDefinition("completed", FieldDefinitionTypes.FullTextSortable
));
                options.UnlockIndex = true;
                if (_settings.Value.LuceneDirectoryFactory == LuceneDirectoryFactory.Synce
dTempFileSystemDirectoryFactory)
                {
                    // if this directory factory is enabled then a snapshot deletion polic
y is required
                    options.IndexDeletionPolicy = new SnapshotDeletionPolicy(new KeepOnlyL
astCommitDeletionPolicy());

                }
            }
        }
        public void Configure(LuceneDirectoryIndexOptions options)
        {
            throw new System.NotImplementedException("This is never called and is just par
t of the interface");
        }

    }
```

```
using Examine; using Examine.Lucene; using Lucene.Net.Analysis.Standard; using Lucene.Net.
Index; using Lucene.Net.Util; using Microsoft.Extensions.Options; using Umbraco.Cms.Core.C
onfiguration.Models;
```

You will notice that we set up our field definitions and types in the **ConfigureCustomIndexOptions**.

Initialize index with options

Add a new file **CustomTodoIndex.cs** and fill it out like this:

```
public class CustomToDoIndex : UmbracoExamineIndex
{
    public CustomToDoIndex(
    ILoggerFactory loggerFactory,
    string name,
    IOptionsMonitor<LuceneDirectoryIndexOptions> indexOptions,
    Umbraco.Cms.Core.Hosting.IHostingEnvironment hostingEnvironment,
    IRuntimeState runtimeState)
    : base(loggerFactory, name, indexOptions, hostingEnvironment, runtimeState)
    {
    }

}
```

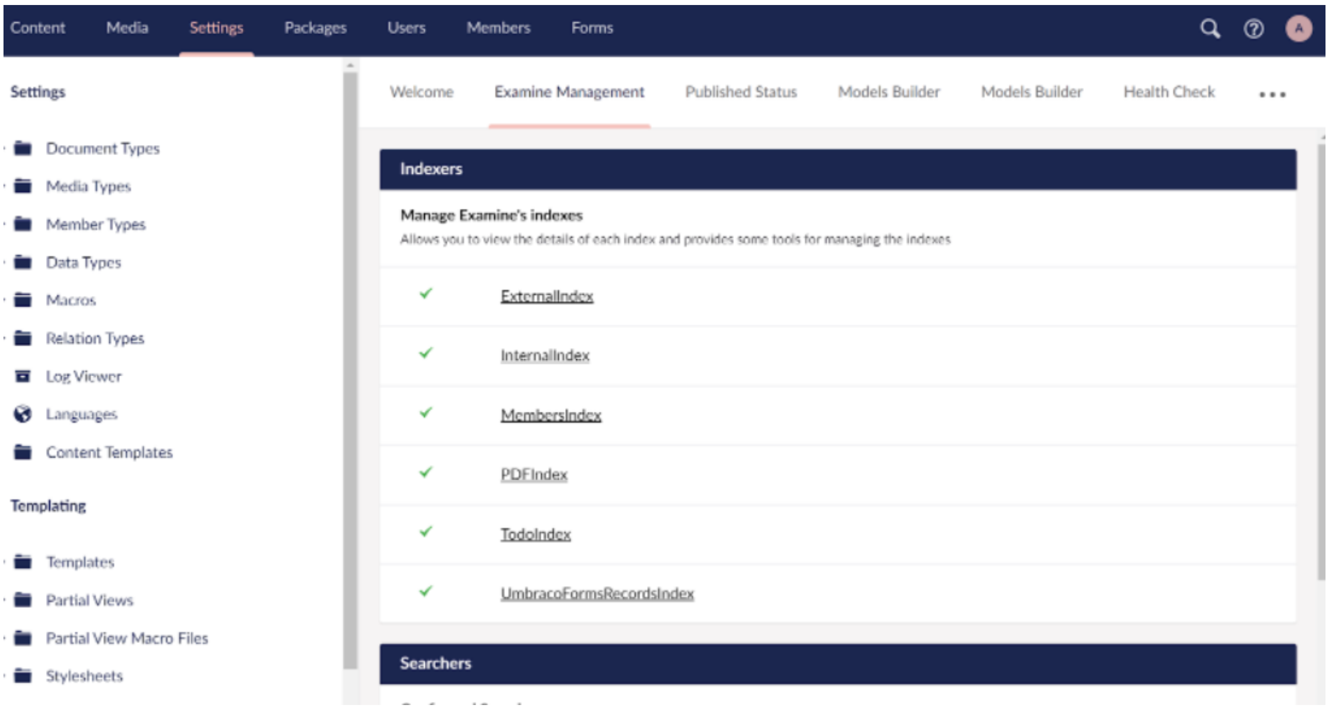
Composer

With our main classes in place, we need to hook them up to our **Composer**. It’s worth noting that you would probably add a new composer in a “real world" scenario, but for the sake of simplicity in the course, we will add to the existing files. Feel free to create your Composer if you are confident with the process!

Go to the **ExamineComposer.cs** and add the following:

```
builder.Services.AddSingleton<TodoValueSetBuilder>();
            builder.Services.AddSingleton<IIndexPopulator, TodoIndexPopulator>();
            builder.Services
        .AddExamineLuceneIndex<CustomToDoIndex, ConfigurationEnabledDirectoryFactory>("Todo
Index");
```

With our components and composers configured, we should now have our custom index. A great way to test to see if it's working is to go to the Umbraco backoffice and head to the Settings section. From here, navigate to the Examine Management dashboard.



Indexers

Manage Examine's indexes

Allows you to view the details of each index and provides some tools for managing the indexes

✓ ExternalIndex
✓ InternalIndex
✓ MembersIndex
✓ PDFIndex
✓ TodoIndex
✓ UmbracoFormsRecordsIndex

Searchers

Optional exercise - custom index search

In case you wanted to allow frontend-based search on this new custom index, there would only be a few extra steps to take.

Multisearcher case

In **Startup.cs** add "TodoIndex" to the multisearcher parameters:

```
services.AddExamineLuceneMultiSearcher("MultiSearcher", new string[] {
    Constants.UmbracoIndexes.ExternalIndexName, PdfIndexConstants.PdfIndexName, "TodoIndex
"
});
```

In **searchService.cs** add a field to search on for ToDo items:

```
var todoTitle = "title";
var fieldsToSearch = new[] { fieldToSearchLang, fieldToSearchInvariant, pdfTextContent, to
doTitle };
```

Update SearchResultItem model to include a ToDoModel item:

```
public class SearchResultItem
{
    public IPublishedContent PublishedItem { get; init; }
    public ToDoModel ToDo { get; init; }
    public float Score { get; init; }
}
```

Back in **searchService.cs** add a case for filling out a ToDo model right after the if check for null pageMedia:

```
if (page == null && pageMedia == null) {
    //we got no content and no media, so we assume the item is a ToDo
                items.Add(new SearchResultItem()
                {
                    PublishedItem = null,
                    ToDo = new ToDoModel() { Title = item.GetValues("title").FirstOrDefaul
t(), Id = Int32.Parse(item.GetValues("id").FirstOrDefault())},
                    Score = item.Score
                });
            }
```

To render the result, in SearchPage.cshtml update the `if` block to

```
@if (Model.SearchResults != null && Model.SearchResults.Any())
                    {
                        foreach (var item in Model.SearchResults)
                        {
                            <ul>
                                <li>
                                    @if (item.PublishedItem != null)
                                    {
                                        <a href="@item.PublishedItem.Url()">@item.Publishe
dItem.Name</a>
                                        var pDate = item.PublishedItem.Value<DateTime>("pu
blishingDate");
                                        if (pDate == DateTime.MinValue)
                                        {
                                            pDate = item.PublishedItem.CreateDate;
                                        }
                                        if (item.PublishedItem.GetType().ToString() == "Um
braco.Cms.Web.Common.PublishedModels.Blog")
                                        {
                                            <p>@pDate.ToString("dd MMMM yyyy")</p>
                                        }
                                    }

                                    @if (item.ToDo != null)
                                    {
                                        <div>
                                            <p>Todo with ID: @item.ToDo.Id</p>
                                            <p>@item.ToDo.Title</p>
                                        </div>
                                    }
                                </li>
                            </ul>
                        }
                    }
```

This should allow you to view search result for content/media items as well as ToDos.

ToDos search only

In the above approach we mix Umbraco content with external data. If, for the sake of simplicity and keeping different data sources separate, you wanted to only search on the custom index, the steps would largely be the same as in Exercise 1.

The most notable changes would be:

-switching out PublishedItem in the SearchResultItem model to a ToDo

-get TodoIndex to search on in the SearchService instead of ExternalIndex

```
if (_examineManager.TryGetIndex("todoIndex", out var index))
```

-making sure you search for "todo" in the SearchService when you create the search query

```
var criteria = searcher.CreateQuery("todo");
```

-adjusting your template to render ToDo items instead of a published Umbraco item

Appendix - Examine on Azure

Examine requires a special configuration when running with Azure Web App to avoid issues with indexes becoming locked and corrupt. To do this you will use a directory factory that will configure Examine to store its indexes in the Environment Temporary folder.

Add the following to **appSettings.json**

```
{
    "Umbraco": {
        "CMS": {
            "Global": {
                "MainDomLock" : "SqlMainDomLock"
            },
            "Hosting": {
                "LocalTempStorageLocation": "EnvironmentTemp"
            },
            "Examine": {
                "LuceneDirectoryFactory": "SyncedTempFileSystemDirectoryFactory"
            }
        }
    }
}
```

For more information refer to the documentation on setting up on Azure web apps: https://our.umbraco.com/Documentation/Fundamentals/Setup/Server-Setup/azure-web-apps