Sitecore CMS 6.0 to 6.5

Sitecore Search and Indexing

Developers Guide to Using Sitecore.Search and Lucene Search Indexes

Table of Contents

Chapter 1

Introduction

This guide is for Sitecore partners and developers who want to implement search functionality in Sitecore CMS based on Lucene indexes. It includes updated information on Lucene and the Sitecore.Search API.

- Overview
- Maintaining Search Indexes in Sitecore
- Sitecore Search
- Sitecore.Search API

1.1 Overview

Lucene is an open source search engine used in Sitecore CMS for indexing and searching the contents of a Web site. Sitecore implements a wrapper for the Lucene engine which has its own API. The original API (`Lucene.Net`) and the Sitecore API (`Sitecore.Search`) are both accessible to developers that want to extend their indexing and search capabilities.

However, before you start to use *Lucene.Net or* the *Sitecore.Search* API, it is important to understand some key concepts.

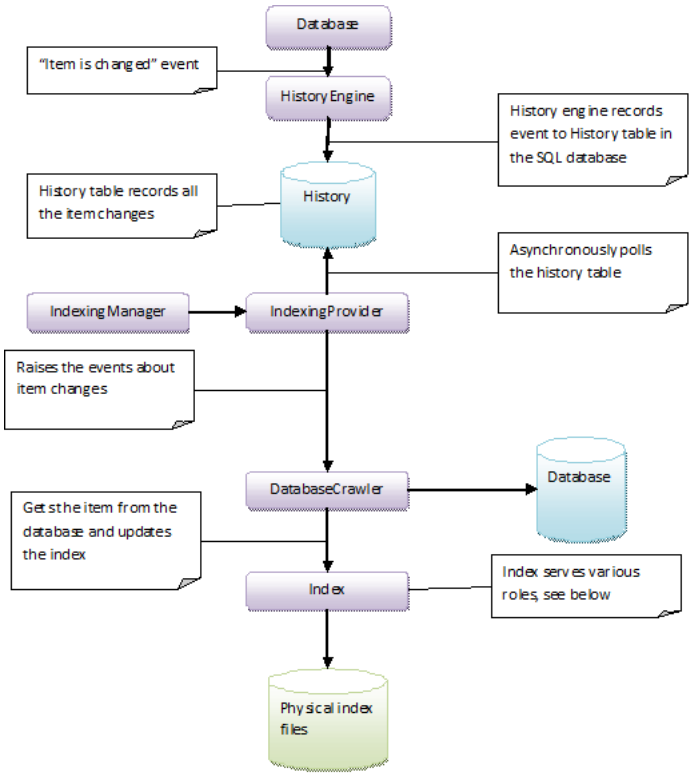
Important Note
The `Sitecore.Data.Indexing` API was deprecated in Sitecore CMS 6.5 and in 7.0 it will be completely removed. Developers should use the `Sitecore.Search` API when configuring Sitecore Search or Lucene search indexes.

1.2 Maintaining Search Indexes in Sitecore

Sitecore maintains indexes by scanning items in Sitecore databases. Every time you update, create or delete an item Sitecore runs a job that updates the indexes. The process is usually complete by the time you have saved or published an item. So, if for example, you create a new item in the Content Editor, it is added to the index straight away.

Indexing in Sitecore

Sitecore Indexing flow diagram.



- A Sitecore item is changed — alter, create or delete. Database class fires an item changed event.
- The History engine executes the appropriate event handler, and sends a record to the History table. The History table records all the item changes.
- Indexing Manager and Index Provider scan the History table, raise an event that an item has changed and invoke the Database Crawler.
- Database Crawler reads the appropriate item from the database and uses the Index class as an interface to Lucene to store item data in the index.
- Index class maintains physical files in the data folder. There are four index folders: `_system`, `core`, `master`, and `web`.

Index Definition

An index is a physical file populated by a crawler and stored in the data folder of the Web site. In Sitecore, an 'index' can serve a number of different purposes:

- From a configuration perspective, an index is a searchable database with a collection of crawlers that populate a database. An Index is identified by its name and stores data in a physical location in the data folder.

- From crawler's perspective, an index is the interface to Lucene.
- From end-user perspective, an index is a searchable database of documents which links to actual Sitecore items.

Lucene Search Index Classes

`IndexWriter`, `Analyzer`, `Document`, and `Field`

Lucene uses these classes during the indexing process and in the retrieval of data from indexes regardless of which API is used.

Class	Description
`IndexWriter`	Used to create a new index and writes documents to the index.
`Analyzer`	Extracts tokens (words) from a stream or string of text. You must use the same analyzer for both indexing and searching as it processes the search string and the index in the same way.
`Document`	Defines a single searchable entry in the index. *Document* is like a row in a database. Each document consists of one or more fields.
`Field`	*Field* is a single named value in the document. For example, *name*, *content* and *creation date*. Unlike columns in a database, fields are not required to be present in all documents. In other words, every document can have an individual set of fields.

Configuring Indexes

Index values that can be set in the `web.config` file:

Index Element	Description
`<configuration>`	Contains index definitions and their configuration settings.
`<indexes>`	The section under configuration where you define indexes. For example, 'system' is one index defined here.
`<analyzer>`	Specify what type of analyzer to use by default in the indexes. An index can use a custom analyzer or refer to this default analyzer definition. Search indexes use the same analyzer both for indexed data (documents) and for the search queries.
`<locations>`	Specify which database you want to add to the index. You can also define the following settings under locations: • `database` • `root` • `tags` • `boost`
`<database>`	Specify which database you want to index, for example 'master'.
`<root>`	Specify the root node of the content tree to be included into the index. The indexing crawler will index content below this location.

Index Element	Description
<tags>	You can attach a string tag to items from this location making it possible to filter or categorize results during a search.
<boost>	Use boost to adjust the priority of results relative to results from other locations.
<IndexAllFields>	Tells the crawler to put a copy of all the fields in the item into the index. This makes fine-grained filtering possible but creates a performance overhead. Default setting = true
<Monitor Changes>	Tells the crawler to subscribe to item changes and updates the index automatically. Default setting = true

Note
New configuration setting was introduced started from the Sitecore 6.5.0 rev. 120427 (Update-4) - `Indexing.IndexStandardTemplateFields`. By default it's set to false. In this case item's standard fields won't be indexed even if <indexAllFields> is true. To change this you should add the `Indexing.IndexStandardTemplateFields` setting to your web.config file and change its value to true, like shown below:

```
<settings>

<!--INDEX STANDARD FIELDS
If true - all item's standard fields will be added to the index.
Default: false-->
<setting name="Indexing.IndexStandardTemplateFields" value="true"/>
```

Rebuilding Lucene Indexes

In certain situations, such as when deploying a site to a production environment or when indexes are out of sync or corrupted, it may be necessary to fully rebuild Lucene indexes.

Rebuilding Search Indexes in Sitecore

To rebuild Lucene indexes from the Sitecore Client:

1. Log in to the Sitecore Desktop.
2. Open the Control Panel.
3. Click *Database* and then click *Rebuild the Search Index.*
4. Use the wizard to select the indexes you want to rebuild and click **Rebuild**.

Rebuilding Search Indexes using Custom Code

To rebuild Lucene indexes from custom code, run one of the following scripts from a custom `.ASPX` page:

For "new" search indexes `Sitecore.Search` namespace, `<search>` section in the `web.config`:

```
        // rebuild "new" search indexes, use this part of code for every "new" index
definition from <search> section in web.config
                Sitecore.Search.Index index = SearchManager.GetIndex("system");
                index.Rebuild();
```

For "old" search indexes `Sitecore.Data.Indexing` namespace, `<indexes>` section in the `web.config`:

```
                // rebuilding all "old" indexes
```

```
Sitecore.Data.Database db = Sitecore.Configuration.Factory.GetDatabase("web");
for (int i = 0; i < db.Indexes.Count; i++)
{
    db.Indexes[i].Rebuild(db);
}
```

1.3 Sitecore Search

You can use the Sitecore.Search API to search the content of your Web site in a number of different ways.

Desktop Search

There are three types of search customizations possible in the Sitecore Desktop:

- Content Editor — the search box above the content tree (has the same functionality as Quick Search).
- Quick Search — the search box to the bottom right of the Sitecore Desktop (has the same functionality as Content Editor Search).
- Classic Search — this is available from the Sitecore Start button and in the Navigate tab on the ribbon.

Content Editor and Quick Search invoke the Search Pipeline found in the `web.config` file to access the Index class and retrieve the search results displayed in an AJAX control.

Extending Desktop Search

- Create your own database crawler or file crawler.
- Implement custom indexing. For example, to include index PDF or MS Word files.
- Extend the search pipeline to include results from locations other than Sitecore or customize the search results.

Web Site Search

On your Web site, you have two options when using Lucene to search indexes and present results:

- Use the Sitecore Lucene Module — you can install the Lucene search module enabling visitors to search your Web site or adapt the source code in the search results page.
- Create your own custom search result controls from scratch — Use Lucene indexes to find Sitecore items as an alternative to using Sitecore queries or the links database.

Extending Web Site Search

- Create filters to exclude certain items from the search results.
- Create categories to present search results grouped by location or other criteria.
- In a bidirectional relationship, display items from the reverse side of the relationship.
- Create faceted navigation from items in a bidirectional relationship.

1.4 Sitecore.Search API

The Sitecore.Search API acts as a wrapper for Lucene.Net. It provides flexible integration of Sitecore with Lucene and a set of .NET-friendly wrappers around Lucene classes related to search.

1.4.1 Overview of Sitecore.Search

The classes in the Sitecore.Search namespace can be grouped in the following way:

Group	Classes	Brief description
Entry Point	`SearchManager`	Entry point to the search subsystem.
Configuration	`SearchConfiguration`	Used in the `web.config` file.

Lucene Abstraction	`ILuceneIndex`
	`Index`
	`IndexSearchContext`
	`IndexUpdateContext`
	`IndexDeleteContext`
		Provides a friendly API to Lucene.NET.

Field Query Object Model	`CombinedQuery`
	`FieldQuery`
	`FullTextQuery`
	`QueryBase`
	`QueryClause`
	`QueryOccurance`
		Provides an object model to create complex queries.

Search Context	`ISearchContext` `SearchContext`	Simple framework to attach context information to the search, such as user identity or user preferences.

Handling Results	`SearchHit`
	`SearchHits`
	`SearchResult`
	`SearchResultCollection`
	`SearchResultCategoryCollection`
		Wrapper for Lucene API enabling grouping, paging, and categorization of search results.

Constants and Enumerations	`SearchType` `BuiltinFields`	`BuiltinFields` contains the names of the fields maintained by Sitecore by default in the system index. Search or filter these fields to refine search results.

1.4.2 Class Descriptions

The following table contains a description of each class in the Sitecore.Search API:

Class Name	Description
`BuiltinFields`	A collection of string constants that refer to fields maintained by Sitecore in the Lucene index. When any item is indexed, the following fields are added to the Lucene document: • `_content` • `_id` • `_path` • `_language` • `_name` • `_group` • `_url` Other classes use `BuiltinFields` to access the content in an index via queries.
`CombinedQuery`	Enables you to combine more than one query clause. Each clause is translated into a simple Lucene Query nested in a Boolean Query.
`FieldQuery`	Searches on a specific field in the index. You can use the Field Name and Field Value to point to a specific field in an item.
`FullTextQuery`*	Similar to a field query, represents a more convenient way of performing full text search against a composite “content” field that includes all tokenized fields. Behind the scenes it is translated into Lucene calls to the Query Parser for full text search.
`ILuceneIndex`	An interface that any index implementation class needs to inherit from. See `Sitecore.Search.Index` class.
`Index`*	This is the Search Index implementation that encapsulates a single Lucene index. When searching, it is used to instantiate the `IndexSearchContext` object which runs the search query. This class implements the `ILuceneIndex` interface and provides two basic functions – reading and writing to the index. This class also provides maintenance functions for the search index.

Class Name	Description
`IndexDeleteContext`	Every time an index is updated via a Database Crawler, this class is instantiated providing `DeleteDocument` (writes a Lucene document) and `Commit` (commits index writer transaction) methods. The class also expects an instance of `ILuceneIndex` interface for example, `Sitecore.Search.Index` class. This type implements the `IDisposable` interface. The `IDisposable` interface enforces certain programming practices, such as using the `try/finally` pattern and manually calling the `Dispose()` method, or instantiating the `IDisposable` objects within the `Using` statement. If you do not correctly dispose of related objects you may experience index corruption and file locking related errors. For more information on the `IDisposable` interface visit MSDN (Microsoft Developers Network).
`IndexSearchContext`*	The main purpose of this class is to run search queries and pass them to Lucene. The Search method returns a `SearchHits` collection. It expects an instance of `ILuceneIndex` interface for example `Sitecore.Search.Index` class. This type implements the `IDisposable` interface. The `IDisposable` interface enforces certain programming practices, such as using the `try/finally` pattern and manually calling the `Dispose()` method, or instantiating the `IDisposable` objects within the `Using` statement. If you do not correctly dispose of related objects you may experience index corruption and file locking related errors. For more information on the `IDisposable` interface visit MSDN (Microsoft Developers Network).
`IndexUpdateContext`	Every time an index is updated either through Rebuild or via a Database Crawler, this class is instantiated providing `AddDocument` (writes a Lucene document) and `Commit` (commits index writer transaction) methods. The class also expects an instance of `ILuceneIndex` interface for example, `Sitecore.Search.Index` class. This type implements the `IDisposable` interface. The `IDisposable` interface enforces certain programming practices, such as using the `try/finally` pattern and manually calling the `Dispose()` method, or instantiating the `IDisposable` objects within the `Using` statement. If you do not correctly dispose of related objects you may experience index corruption and file locking related errors. For more information on the `IDisposable` interface visit MSDN (Microsoft Developers Network).

Class Name	Description
`ISearchContext`	An interface to extend or control what a user searches for. Also used by `IndexSearchContext`. This class provides more information about the context of the search and the site visitor. For example, visitor search terms, search and navigation history or information from behavior tracking.
`PreparedQuery`	Internal system class used by Sitecore. PreparedQuery allows you to pass a Lucene query in a way that disables its preprocessing. It makes low-level queries of the index possible and is used internally by the Database Crawler.
`QueryBase`*	The following query classes inherit from `QueryBase`: • `CombinedQuery` • `FieldQuery` • `FullTextQuery`
`QueryClause`	A single entry in a combined query.

`QueryOccurance`	An enumeration that defines a logical operator in a combined or Boolean query. When you add a query, you need to specify which type of logical condition Lucene should use to evaluate the expression.
	
	Lucene uses the following operators for the search terms in complex queries:
	
	• **Must** – the search term must occur in the document to be included into the search results.
	• **Should** – the search term may occur in the document but is not necessary, and the document may be included in search results based on other criteria. However, the documents containing the search term are ranked higher than equivalent documents that do not contain the search term.
	• **Must not** – the search term must not occur in the document in order to be included in the search results. Documents with the search term will be excluded from the results.
	
	When a Sitecore Combined query is translated into a Lucene query, the `QueryOccurance` is converted into a Lucene Boolean `BooleanClause.Occur` class:
	
	`QueryOccurance.Must = Lucene.Net.Search.BooleanClause.Occur.MUST`
	
	`QueryOccurance.MustNot = Lucene.Net.Search.BooleanClause.Occur.MUST NOT`
	
	`QueryOccurance.Should = Lucene.Net.Search.BooleanClause.Occur.SHOULD`

`SearchConfiguration`	Internal class used by Sitecore. Provides a way to define indexes in the config file. You can also create your own elements in the config using this class.

Class Name	Description
SearchContext	Inherits from and is the default implementation of ISearchContext.
SearchHit*	A single search hit. Use this class to access the content in a hit. The Url field uniquely identifies the resource, item, or page.
SearchHits	SearchHits represents a collection of search hits. Use the FetchResults method to return a SearchResultCollection from the collection of hits. You can specify the start and end position in the set of search hits, for example, to get the first 10 results in a SearchHits collection.
SearchManager*	This is the entry point to Sitecore.Search. It encapsulates the search functionality in Sitecore. It includes the following three functions: • It loads the configuration in the <search> section of web.config. • It provides access to the configured instances of the Index class for searching and modification. • The GetObject method enables you to get an item from the SearchResult class. The SystemIndex property refers to the default Sitecore index named 'system'.
SearchResult*	Represents a single search result. Properties Name, Icon and Url provide access to key attributes of the result. SubResults collection contains lower-ranked results in the same group. SearchResult uses BuiltinFields to access specific fields such as Name, Url and Icon in the document.
SearchResultCategoryCollection	Works in conjunction with SearchResultCollection. Makes it possible to put search results into different categories.
SearchResultCollection*	Used for categorizing search results. You can collapse results by language or version as sub results. More modifiable than SearchHits. Default categories are set in the config file under the <search> section but it is possible to use custom categories at run time.

*Classes used in the bidirectional example: Lucene Search and Bidirectional Relationships

Chapter 2

Sitecore Lucene Search Module

Sitecore Lucene Search is a Shared Source module intended as example code for developers learning how to use Sitecore Search. While the code samples may not be applicable to the specifics of your implementation, they will teach you the basics of the Search API.

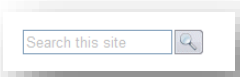
- Overview
- Installation and Configuration
- Lucene Search Module Source Code

2.1 Overview

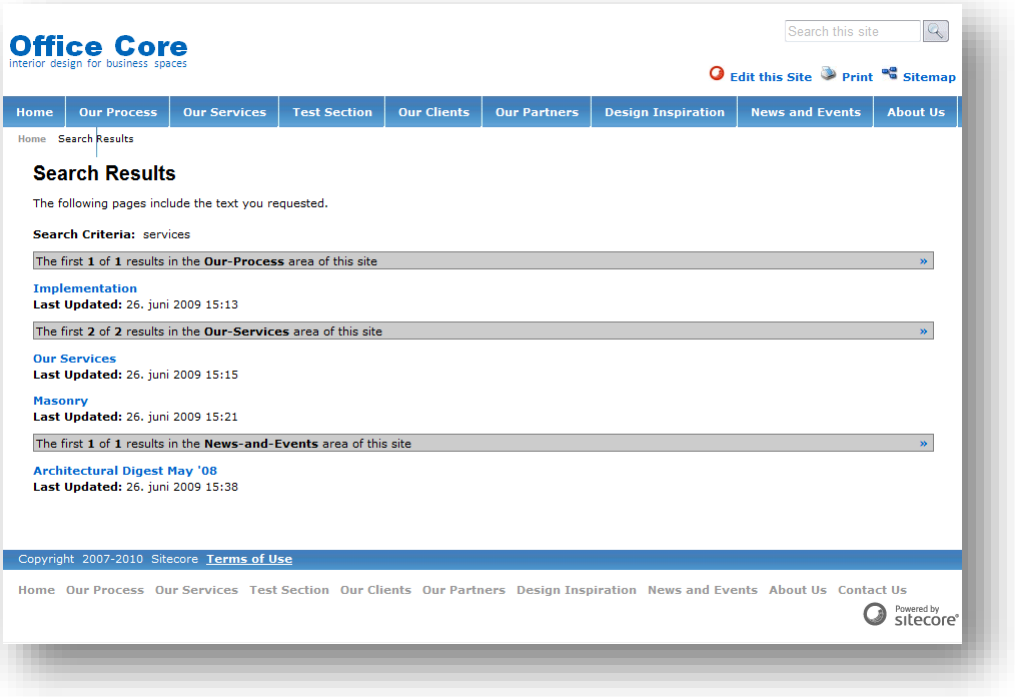
The Lucene search module uses the Sitecore starter kit. It includes the following basic components for creating Web site search:

- Search Box sublayout
- Search Results sublayout
- Relevant classes and source code

Search Box rendering



Search Results rendering



The Lucene search module utilizes the new *Sitecore.Search* API.

If you want to customize this module further, you can download the source code in this module.

2.2 Installation and Configuration

You can download the Lucene Search module from the TRAC Web site:

http://trac.sitecore.net/LuceneSearch

This page contains the following resources:

- LuceneSearch-1.1.zip
- TRAC documentation

2.2.1 Installing the Lucene Search Module

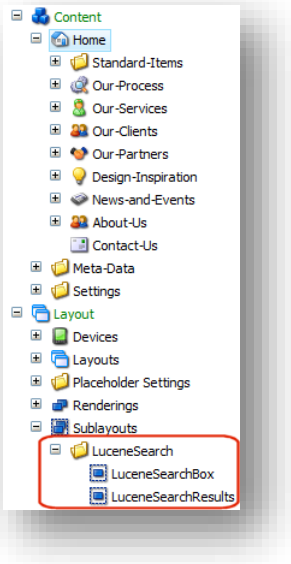
You use the Sitecore Installation Wizard to install the Lucene Search module. The Lucene Search Module can run on CMS 6.0 or later.

In the following example, we also use the Sitecore Office Core Web site to demonstrate the search module.

Using the Sitecore Installation Wizard, browse to the location where you downloaded `LuceneSearch-1.1.zip`. Ensure that you browse to the correct version of the Lucene Search module contained in the zip file (Lucene 1.1).

2.2.2 Configuring the Search Box and Search Results Sub Layouts

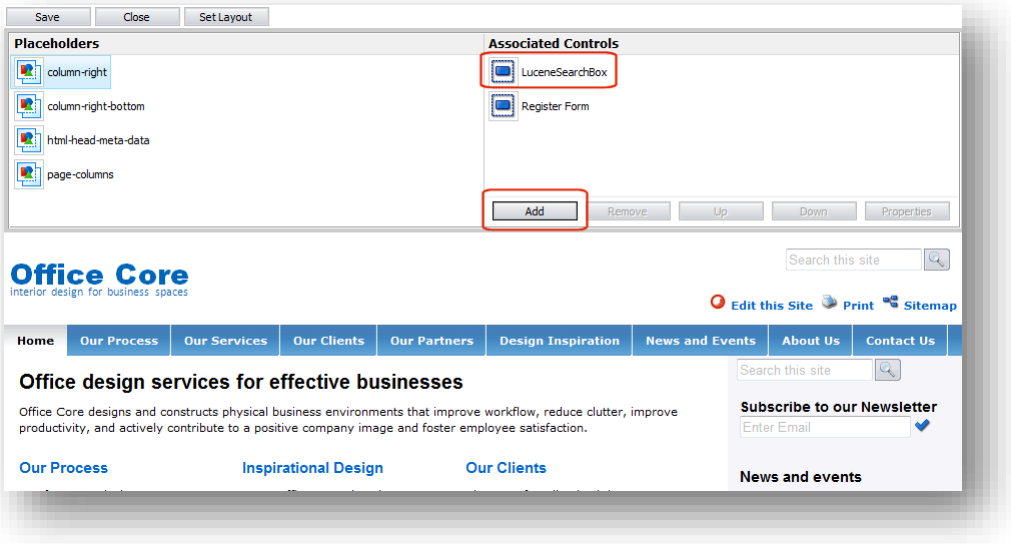
After you have installed the Lucene Search module, you can see two Lucene sub layouts in the content tree; `LuceneSearchBox` and `LuceneSearchResults`.



To add the Search Box to the Office Core Web site:

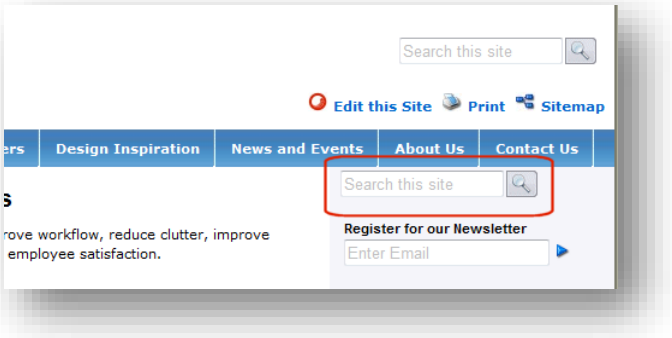
1. Open the Page Editor.
2. Click Design.

3. In design mode, navigate to the *Home* page of the Office Core Web site. Select the column-right placeholder and add the LuceneSearchBox control to the associated controls.



4. Save your changes and close the Page Editor.

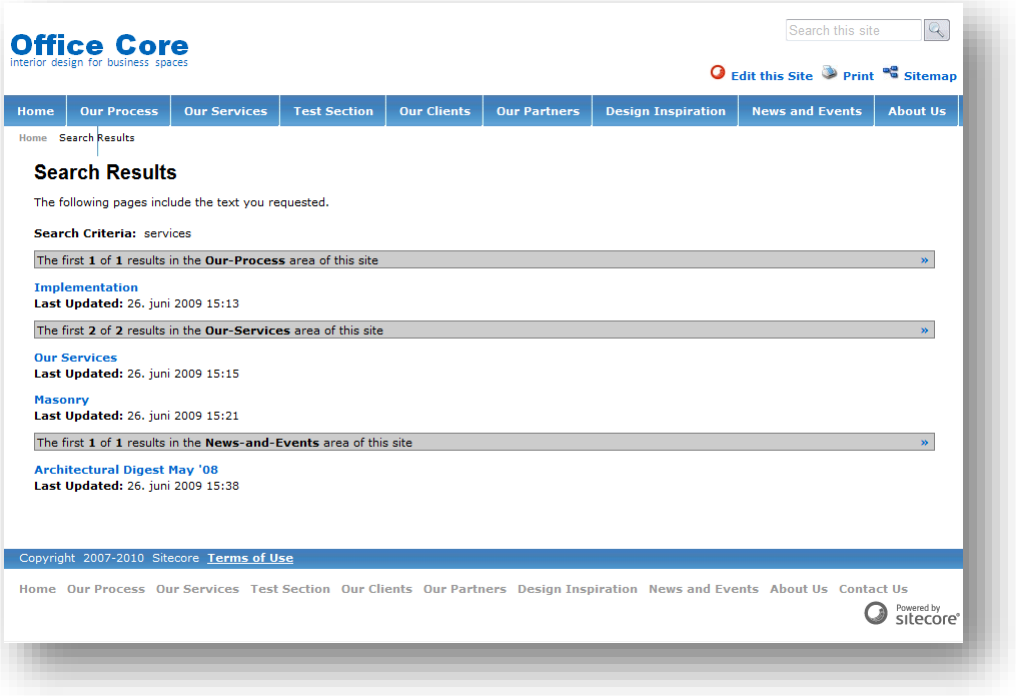
   You can add the Search Box control to any page on your site. In this example, I have added it to the *Home* page and have left the existing search box on the page. You can then compare the results you get using the Lucene search module and the old Lucene search built into the Starter Kit.



   You do not need to configure the `SearchResults` control. This works ‘out of the box’.

   To test the search results page, create a new content item, for example, a new site section. Lucene will add this item to the index before you make a search. Enter a search term in the

Search box and you can see your results displayed in the SearchResults control.

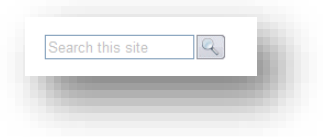


2.3 Lucene Search Module Source Code

This section provides a simple overview of the source code used to create this module. You can use the source code as the basis for any search customizations you want to make.

2.3.1 Search Box Control

`LuceneSearchBox.ascx.cs` is a sublayout that consists of a search box and a submit button.



Include the following namespaces:

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.Xml.XPath;
using Sitecore.Configuration;
using Sitecore.Data;
using Sitecore.Data.Items;
using Sitecore.Links;
using Sitecore.Xml.XPath;
```

```
namespace LuceneSearch.LuceneSearch
```

Name this class `LuceneSearchBox`.

```
{
    public partial class LuceneSearchBox : System.Web.UI.UserControl
    {
        private Database masterDB = null;
```

Open a connection to the master database.

```
        protected void Page_Load(object sender, EventArgs e)
        {
            masterDB = Factory.GetDatabase("master");
            if (!IsPostBack)
                txtCriteria.Text = CommonText.get("search");
        }
```

Take the search term entered in the search box (`txtCriteria.Text`) and perform a search when the user clicks the search button.

'*Search this site*' is the default text displayed in the search box before a user enters a search term.

`-!= CommonText.get("search"))` = If the search term entered into the search box is different from the default text, perform a search.

`performSearch();` = perform search

```
        protected void btnSearch_Click(object sender, ImageClickEventArgs e)
        {
            if (txtCriteria.Text != CommonText.get("search"))
                performSearch();
        }
```

This part of the code handles any changes made to the text in the search box.

```
        protected void txtCriteria_TextChanged(object sender, EventArgs e)
        {
            if (txtCriteria.Text != CommonText.get("search"))
                performSearch();
        }
```

Use XPATH to iterate through Sitecore items in the content tree.

```
        private void performSearch()
        {
            Database database = Factory.GetDatabase("master");
            var item = database.GetRootItem();

            ItemNavigator nav = Factory.CreateItemNavigator(item);
            XPathNodeIterator iterator = nav.Select(".");

            while (iterator.MoveNext())
            {
                string itemID = iterator.Current.GetAttribute("id", string.Empty);
                Item home = masterDB.Items[itemID];

                if (home != null)
                {
                    Item standard items = home.Children["standard-items"];
                    Item results = standard items.Children["Search-Results"];
                    if (results != null)
                    {

                        string results url = LinkManager.GetItemUrl(results) +
                        "?searchStr=" + txtCriteria.Text;
```

Sends the search results to the `LuceneSearchResults.ascx.cs` using `"?searchStr="`

```
                        Response.Redirect(results_url);
                    }
```

If it finds the Home node or no results can be found, display one of two alternate messages.

```
                    Else-
                    {
                        txtCriteria.ForeColor = Color.Red;
                        txtCriteria.Text = "Unable to find results item";
                    }
                }
                else
                {
                    txtCriteria.ForeColor = Color.Red;
                    txtCriteria.Text = "Unable to find home!";
                }
            }
        }
    }//class
}//namespace
```

2.3.2 Search Manager

`SearchManager.cs` retrieves the search results from the index.

Start by including the following namespaces:

```
using System;
using Sitecore.Data;
using Sitecore.Search;
using Sitecore.Xml.XPath;
using Version = Sitecore.Data.Version;
using Sitecore.Data.Items;
using Sitecore.Configuration;
using Sitecore;
using System.Xml.XPath;
using Sitecore.Globalization;

namespace LuceneSearch.LuceneSearch
{
```

Give this class the name `SearchManager`.

```
            public class SearchManager
        {
            public const string OtherCategory = "Other";
```

The item associated with the site root.

```
            private readonly Item SiteRoot;
```

This specifies the search index for the current database, usually the master database in Sitecore. This is set in the `web.config` file, for example, `system`.

```
            private readonly string SearchIndexName;


            private SearchResultCollection _searchResults;

            public SearchResultCollection SearchResults
            {
                get { return _searchResults;  }
            }

            public SearchManager(string indexName)
            {
                SearchIndexName = indexName;
```

Iterate through the Sitecore content tree. Start at the site root before traversing the tree.

```
            Database database = Factory.GetDatabase("master");

            SiteRoot = database.GetRootItem();
            }
```

Get the appropriate category (`Item item`) is the item that matches the criteria of the search. Then get the category name, for example *Products*. Every item that has site root as a parent becomes a category

```
            private string GetCategoryName(Item item)
            {
                Item workItem = item;

                if (!SiteRoot.Axes.IsAncestorOf(workItem))
                {
                    return string.Empty;
                }
```

When category = Home. Put the result in the 'Other' category if the item is site root or does not have an ancestor that is site root.

```
              if (workItem == SiteRoot)
                {
                    return OtherCategory;
                }

                while (workItem.ParentID != SiteRoot.ID)
                {
                    workItem = workItem.Parent;
                }

                return workItem.Name;
            }
            public SearchResultCollection Search(string searchString)
            {
```

Lucene retrieves the index and other settings from the `web.config` file.

```
                var searchIndex = Sitecore.Search.SearchManager.GetIndex(SearchIndexName);
                using(IndexSearchContext context = searchIndex.CreateSearchContext())
                {
                    SearchHits hits = context.Search(searchString, new
                SearchContext(SiteRoot));
```

Limit results to 100.

`var results` = Variable containing results.

```
                    var results = hits.FetchResults(0, 100);
                    foreach(SearchResult result in results)
                    {
                        try
                        {
                            Item item = result.GetObject<Item>();
                            if(item != null)
                            {
                                string categoryName = GetCategoryName(item);

                                if(item.Language.Name != Context.Language.Name ||
                                categoryName == string.Empty)
                                {
                                    continue;
                                }
```

This section of code concatenates the category with the search results.

```
                                results.AddResultToCategory(result, categoryName);
                            }
                        }
                        catch
                        {
                            continue;
                        }
                    }
                    return _searchResults = results;
                }
            }
        }//class
    }//namespace
```

2.3.3 Search Results Rendering

`LuceneSearchResults.ascx.cs` is a sublayout that controls the presentation of search results on the Web site. It controls the number of results displayed, the categories shown and the formatting of the text to display.

Include the following namespaces.

```
    using System;
    using System.Collections;
    using System.Configuration;
    using System.Data;
    using System.Linq;
    using System.Web;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.UI.HtmlControls;
    using System.Web.UI.WebControls;
    using System.Web.UI.WebControls.WebParts;
    using System.Xml.Linq;
    using Sitecore.Data.Fields;
    using Sitecore.Data.Items;
    using Sitecore.Links;
    using Sitecore.Search;
    using Sitecore.Web;
    using Sitecore;

    namespace LuceneSearch.LuceneSearch
```

Give this class the name `LuceneSearchResults`.

```
    {
        public partial class LuceneSearchResults : System.Web.UI.UserControl
        {
```

Gather search results from the Lucene search index.

```
    SearchManager searchMgr;
            private string baseURL;

            protected void Page_Load(object sender, EventArgs e)
```

Call `SearchManager` in relation to a specific index name, for example '`system`'.

```
                {
                    string indexName = StringUtil.GetString(IndexName, CommonText.get("Search
                Index"));
                    searchMgr = new SearchManager(indexName);
```

Decode the search string query string '`searchStr`'.

```
                    string searchStr = Server.UrlDecode(WebUtil.GetQueryString("searchStr"));
```

If a visitor wants to see all the results for a given area of the site, get the category. The following code creates a category on the results page.

`{0}?searchStr={1}&categoryStr=` determines the format of the text that will appear on the results page.

```
                    string categoryStr =
Server.UrlDecode(WebUtil.GetQueryString("categoryStr"));
```

It takes the search string and the category string and formats a URL, for example:

`http://<your site>/Standard-Items/Search-Results.aspx?searchStr=partners`

```
            baseURL = string.Format("{0}?searchStr={1}&categoryStr=",
LinkManager.GetItemUrl(Sitecore.Context.Item), searchStr);
```

If no results can be found, apply the following code. Check that we have an empty search string at very least (avoid null).

```
                    if (searchStr == null) searchStr = string.Empty;
```

If the visitor provides no criteria, do not search

```
                    if (searchStr == string.Empty)
                        lblSearchString.Text = CommonText.get("search criteria") +
                        CommonText.get("search no criteria");
                    else
                    {
```

Remind the visitor what they provided as search criteria

```
                        lblSearchString.Text = CommonText.get("search criteria") + searchStr;
```

Perform the actual search. This invokes the `SearchManager` class which is also part of this project.

```
                        searchMgr.Search(searchStr);
```

Display the search results

```
                        DisplayResults(searchMgr.SearchResults, categoryStr);
                    }
                }

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

Initially, this method is called with a blank category name.

In that case, display initial results for all areas of the site.

If the user wants more results for a given category, use this method to call the appropriate category name.

```
                private void DisplayResults(SearchResultCollection results, string
                categoryName)
                {
                    pnResultsPanel.Controls.Clear();
```

If no results are found, give an appropriate message.

```
                    if (results.Count == 0)
                    {
                        string noResultsMsg = CommonText.get("Search No Results");
                        LiteralControl noResults = new
                        LiteralControl(string.Format("<p>{0}</p>", noResultsMsg));
                        pnResultsPanel.Controls.Add(noResults);
                    }
                    else
                    {
```

The category name will be empty if this is an initial search.

If this is the case, show up to `maxInitialResults` in each site section.

```
                    if (categoryName == string.Empty)
                    {
```

Loop through all the search result categories

```
                        foreach (var category in results.Categories)
                        {
                            string maxResultsStr = CommonText.get("Search Max Initial
                            Results");
                            int maxResults = int.Parse(maxResultsStr);
```

Check if there are any hits in this category.

```
                            if (category.Count > 0)
                            {
```

If there are any results, build the category title.

```
                                string categoryTitleFormat = CommonText.get("Search
                                Category Partial Title");
                                int resultsShown = Math.Min(category.Count, maxResults);
                                int totalResults = category.Count;
                                string categoryTitle =
                                    string.Format(categoryTitleFormat, resultsShown,
                                    totalResults, category.Name);
                                string titleDiv =
                                    string.Format("<div class='search-results-
                    category'><div class='link'><a href='{1}{2}'>»</a></div><div
                    class='title'>{0}</div></div>", categoryTitle, baseURL, category.Name);

                                LiteralControl catTitle = new LiteralControl(titleDiv);
                                pnResultsPanel.Controls.Add(catTitle);
```

Now iterate over the number of results to display.

```
                                foreach (var result in category)
                                {
                                    Item hit = result.GetObject<Item>();
                                    if (hit != null)
                                    {
                                        string hitText = GenerateHitText(hit);
                                        LiteralControl hitControl = new
                                        LiteralControl(hitText);
                                        pnResultsPanel.Controls.Add(hitControl);
                                    }
                                }
                            }
                        }
                    }
                    else
                    {
                        SearchResultCategoryCollection category =
                        results.GetCategory(categoryName);
```

If there are results, build the category title.

```
                        string categoryTitleFormat = CommonText.get("Search Category Full
                        Title");
                        string categoryTitle =
                            string.Format(categoryTitleFormat, category.Count,
                        category.Name);
                        string titleDiv =
                            string.Format("<div class='search-results-category'><div
                        class='title'>{0}</div></div>", categoryTitle);
                        LiteralControl catTitle = new LiteralControl(titleDiv);
                        pnResultsPanel.Controls.Add(catTitle);
```

Now iterate over all the results for this category.

```
                        foreach (var result in category)
                        {
                            var hit = result.GetObject<Item>();
                            if (hit != null) {
```

```
                                string hitText = GenerateHitText(hit);
                                LiteralControl hitControl = new LiteralControl(hitText);
                                pnResultsPanel.Controls.Add(hitControl);
                            }
                        }
                    }
                }
            }
```

Format the search results on the results page.

Generate text to display for each item that matches the search criteria.

```
            private static string GenerateHitText(Item hit)
            {
```

Set the name of the link to a non-blank value, for example use one of the title fields.

```
                string hitName;
                if (hit["menu title"] != string.Empty) hitName = hit["menu title"];
                else if (hit["breadcrumb title"] != string.Empty) hitName =
                hit["breadcrumb title"];
                else if (hit["banner title"] != string.Empty) hitName = hit["banner
                title"];
                else if (hit["title"] != string.Empty) hitName = hit["title"];
                else hitName = hit.Name;
```

Set the description to the start of one of the description fields.

```
                string hitDescription;
                if (hit["menu abstract"] != string.Empty) hitDescription = hit["overview
                abstract"];
                else if (hit["text"] != string.Empty && hit["text"].Contains("<") ==
                false) hitDescription = hit["text"];
                else hitDescription = string.Empty;
```

If the description is too long, shorten it. If the description is not blank, add a line break after it.

```
                if (hitDescription.Length > 150) hitDescription =
                string.Format("{0}...<br/>", hitDescription.Substring(0, 150));
                else if (hitDescription.Length > 0) hitDescription += "<br/>";

                DateField lastUpdatedField = hit.Fields["  updated"];
                string lblLastUpdated = CommonText.get("Search Last Updated");
                string lastUpdated = (lastUpdatedField != null
                                      ? lastUpdatedField.ToString()
                                      : "unknown");
                string hitText =
                    string.Format("<div class='search-results-hit'><p><a
                href='{0}'>{1}</a><br/>{2}<strong>{3}</strong>{4}</p></div>",
                                  LinkManager.GetItemUrl(hit), hitName, hitDescription,
                lblLastUpdated, lastUpdated);
                return hitText;
            }
        }//class
    }//namespace
```

Chapter 3

Sitecore.Search Examples

This chapter examines several problems that you can solve using Lucene search.

- Lucene Search and Bidirectional Relationships
- Creating a File Crawler

3.1 Lucene Search and Bidirectional Relationships

Definition – The term bidirectional refers to navigating the *one to many* relationships that are common in relational databases. Sitecore has difficulty finding content items in both directions in a bidirectional relationship.

The example of lawyers and practices illustrates this concept in more detail.

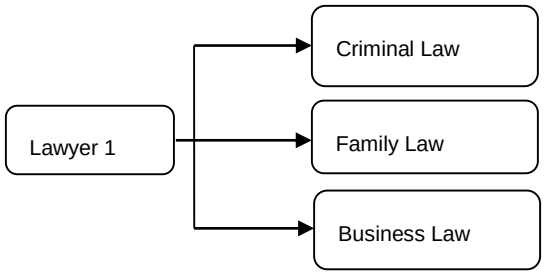
A Web site could have content items that represent lawyers and their associated areas of practice.

Each lawyer could have one or more areas of practice related to them. For example, a single lawyer might have three areas of practice such as Criminal law, Family law or Business law and another might only have one practice area such as Family law.

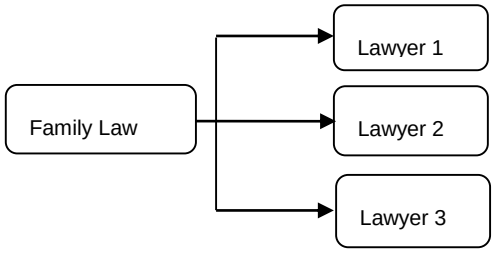
You want to create a rendering or Web control that displays all lawyers and their associated area of practice and another that displays areas of practice and all associated lawyers.

For example:

Rendering 1: Displays a list of Lawyers and their related areas of practice:



Rendering 2: Displays a list of Practice areas and associated lawyers:



In Sitecore, there are two suitable field types that you could use in the Lawyer template:

- Multilist field — If the related items all have the same parent item
- Treelist field — If the items are in a hierarchy

When conducting a search or a query using these field types, Sitecore might not have too much difficulty finding a small number of related items in both directions but when there are more than 100 items to find, performance is significantly affected. This is when another, more effective method of searching for items is preferable.

3.1.1 Possible Solutions

In Sitecore, there are currently three possible ways to solve the problem of bidirectional relationships. Each method has its own advantages and disadvantages.

Solutions	Advantages/Disadvantages
Links database	• Can handle bidirectional relationships. All links are stored in a table called *Links*. • Only works with certain field types • You must filter for languages and versions • You need to implement a solution to maintain the links in the links database
Sitecore Queries	• Can be used effectively to query small numbers of items • If there are too many items to find, performance can be affected.
Lucene Search	• Fast • Flexible • Can handle a large number of items • Can filter results • No adverse effect on performance

Lucene Search is the most effective solution as it is fast without having a negative effect on performance. You can use *Lucene.Net* and the *Sitecore.Search* API to create a crawler to scan indexes and find all the relevant items required in a bidirectional relationship.

3.1.2 Bidirectional Example: Training Sample Site

This example uses the Office Products section in the Sitecore Training Sample Web site to demonstrate how *Lucene.Net* and *Sitecore.Search* can navigate bidirectional relationships.

The example consists of two Web controls.

The full solution is available as a Sitecore shared source module. You can download the module and study the source code or use it to create your own solution following the steps below.

Example 1: Office Products Web Control

Use the *Sitecore.Search* API to list the product items that link to the currently selected product by creating a control in C#. This control must find items in both the forward and reverse direction of the bidirectional relationship.

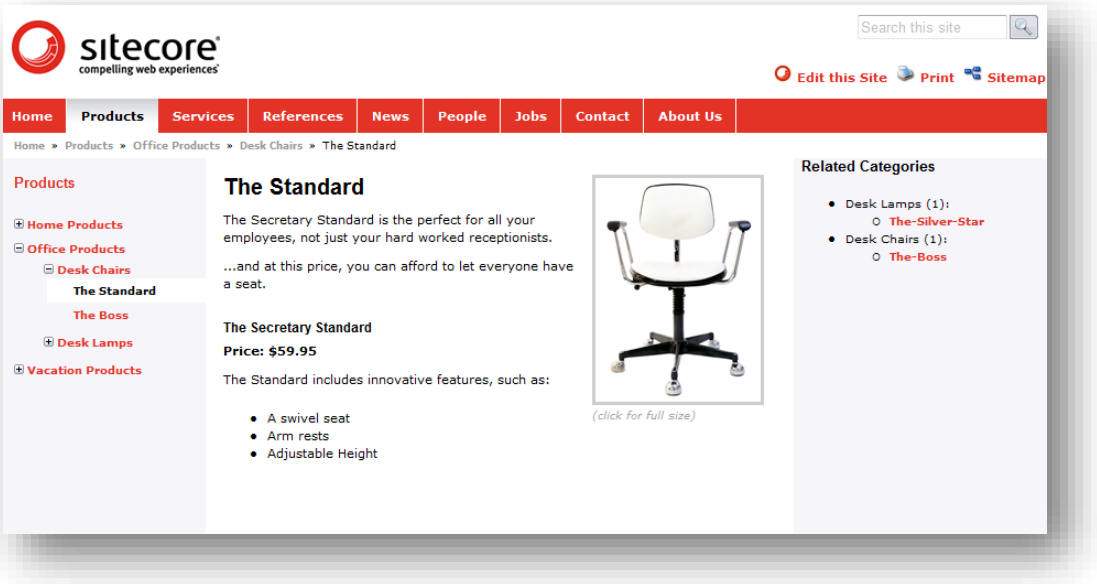
Give the control a name such as `RelatedItems.cs`.

Requirements:

- This control must display all products that have the current item selected in the **Related Items** field.
- It should display related items by category in the side bar to the right of the page.

- It should display a number in brackets that indicates the number of products found in each category.

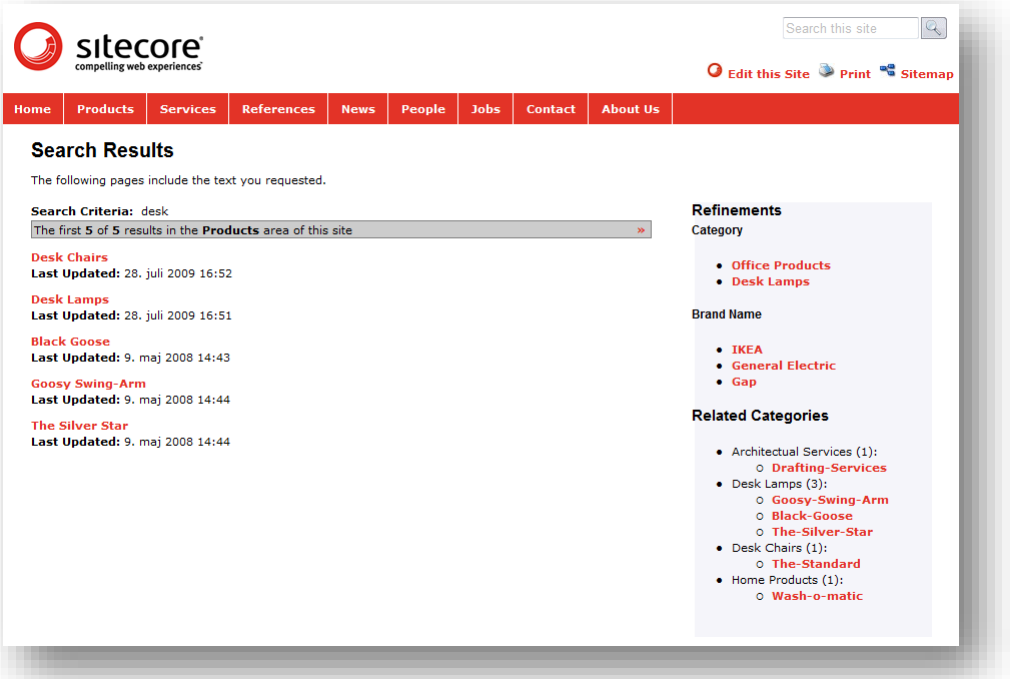
The Standard office chair after implementing the Related Items control:



Example 2: Search Results Web Control

The Search Results page has the same requirements as the first example but adds related categories and refinements to the search results in the side bar. To implement this, you need to add some extra code to the `SearchResults.cs` control.

After implementing these changes, a search for ‘desk’ should return all related items, brands, and categories.



3.1.3 How to Create the Bidirectional Example

The following steps explain how the main classes in this bidirectional example work together.

Implement a Searcher Class

Create a Searcher and a Query Runner that invokes a database crawler to search the Lucene search indexes for related items, depending on which Web control is used. The Web control then processes the results and displays them in the side bar on the Web site.

`Searcher.cs` contains multiple methods that query the indexes in several different ways. The full code for each method is contained in the Query Runner; it is not visible in the Searcher. The Searcher simply contains a list of methods used.

The Searcher can only execute one method at a time. When the Related Items Web control invokes a method in the Searcher, it calls the Runner. The Runner searches the indexes and sends a list of items to the Searcher used by the Related Items Web control to display related items to the site visitor.

Searcher.cs

To create a Searcher:

1. First include the following namespaces:

```
namespace Sitecore.Training.Index
{
    using System;
    using System.Collections.Generic;

    using Search;
    using Data;
    using Data.Items;
    using Collections;
```

2. Name the class *Searcher*: `public class Searcher`
3. Create a new instance of the Query Runner property to be used by the Searcher.

```
protected QueryRunner Runner { get; set; }
```

4. Create two constructors that you trigger when you run the Searcher. Only one of these constructors should accept parameters to specify which index to use when you conduct a search.

Constructors	Parameters
`public Searcher() { Runner = new QueryRunner(Constants.Index.Name); }`	`()` No parameters No index id specified, so the query runner tries to find the index by calling `index.name`
`public Searcher(string indexId) { Runner = new QueryRunner(indexId); }`	`(string indexId)` is specified as the parameter to use to fetch the search index.

5. Define a new region in the code for search methods and then add each method that you want to use. Each search method is basically a query. This example shows the first method in `Searcher.cs: GetItemsByFullTextQuery`

```
  #region Searching Methods

public virtual List<Item> GetItemsByFullTextQuery(string query)
        {
            var results = new
```

```
                List<SearchResult>(Runner.GetItemsByFullTextQuery(query));
                return GetItemsFromSearchResult(results);
            }
```

The Related Items control triggers each search method. When it calls a search method it creates a new instance of the Query Runner in the Searcher. It calls the new instance *Runner*.

```
var results = new List<SearchResult>(Runner.GetItemsByFullTextQuery(query));
```

Runner then searches the indexes and sends back all relevant search results to the method.

```
return GetItemsFromSearchResult(results);
```

The search results list returned to the Searcher is then converted by the `GetItemsFromSearchResult` method into a result set containing each item. This method iterates through the list of results and finds a url for each item.

It takes the search results extracted from the index and then stores them in a `resultingSet` list.

6. Next create a `GetItemsFromSearchResult` method. This method returns the set of items stored in the `resultingSet` with the corresponding `URL` so that you can click a link to get the item.

```
public static List<Item> GetItemsFromSearchResult(IList<SearchResult>
searchResults)
        {
            var resultingSet = new List<Item>();

            foreach (var result in searchResults)
            {
                var uriField = result.Document.GetField("_url");
                if (uriField != null && !String.IsNullOrEmpty(uriField.StringValue()))
                {
                    var itemUri = new ItemUri(uriField.StringValue());
                    resultingSet.Add(Context.Database.GetItem(new DataUri(itemUri)));
                }
            }

            return resultingSet;
        }
```

Methods and Parameters used in the Searcher Class

Searcher Methods	Parameters
`GetItemsByFullTextQuery(string query)`	- full text query on a string
`GetRelatedItemsByMultipleFields(string query, SafeDictionary<string> refinements)`	- full text query on a string - the second parameter searches for refinements using *SafeDictionary*, then creates a list of dictionary refinements – brand = IKEA, Category = Desk etc
`GetRelatedItems(string ids)`	– Ids = returns the Ids of all items
`GetRelatedItemsByField(string ids, string fieldName, bool partial)`	- Ids = Id of the product selected in the Related Items field. For example, the ‘Standard’ product. - fieldname – the related items field name - partial = boolean
`ContainsItemsByFields(string ids, string fieldName, string fieldValue)`	- Applies an ID filter - fieldName, - string `fieldValue` = Brand e.g. IKEA
`GetItemsFromSearchResult(IList<SearchResult> searchResults)`	- Search result list – for each item in the search result list this method gets the url and creates another list called resultingSet.backup on DVD

Implement a Query Runner Class

Searcher contains a list of all query methods available to interrogate the search indexes. Query Runner contains the processing logic for each query.

For every method run in `Searcher.cs`, it calls Query Runner behind the scenes. To see how each query is constructed, look in `QueryRunner.cs`. This is also where it runs each query.

QueryRunner.cs

To create a query runner:

1. First add the following namespaces:

```
namespace Sitecore.Training.Index
{
    #region Usings

    using System;
    using System.Collections.Generic;
    using System.Linq;

    using Lucene.Net.Search;
    using Lucene.Net.Index;
    using Lucene.Net.QueryParsers;
    using Lucene.Net.Analysis.Standard;
    using Search;
    using Diagnostics;
    using Data;
    using Collections;
```

2. Name the class `QueryRunner`:

```
public class QueryRunner : IDisposable
```

3. Create the following constructor:

```
public QueryRunner(string indexId)
        {
            Index = SearchManager.GetIndex(indexId);
        }
```

This constructor gets the appropriate search index. It calls the `SearchManager` class with the `GetIndex` method and uses *index* as a parameter. The `IndexId` parameter can refer to either the `training master` or `training web` indexes.

4. Create a property called index:

```
    public Index Index { get; set; }
```

5. Create two query runner methods. Both of these use the same parameter called `query` but the type is different:

   `QueryBase` = Sitecore.Search

   `Query` = Lucene.Net

Query Method	Description
`SearchResultCollection RunQuery(QueryBase query)`	Using Sitecore.Search API – this method expects Sitecore queries
`SearchResultCollection RunQuery(Query query)`	Using Lucene.Net API – this method expects Lucene queries

6. Create your search methods (queries). These are the same as the methods listed in `Searcher.cs`. The following table describes each of these methods in more detail. View the bidirectional shared source module to see the full source code for each method.

Method	Description
`GetItemsByFullTextQuery(string query)`	Searches through all the text contained in the item fields, including Rich Text fields and meta-data.

`GetRelatedItemsByMultipleFields(string query, SafeDictionary<string> refinements)`	Called from the `SearchManager.Search()` method via `Searcher.GetRelatedItemsByMultipleFields`:
	Get the refinements from the query string:
	`var refinements = WebUtil.ParseQueryString(WebUtil.GetQueryString()); refinements.Remove("search");`
	Pass them on to the searcher:
	`var results = searcher.GetRelatedItemsByMultipleFields(searchString, refinements);`
	This method links to a list of dictionary refinements and adds category and brand as refinements to the search.

`GetRelatedItems(string ids)`	Gets all relations without specifying a field. Get related items by list of Ids and returns a pipe separated list of ids.
`GetRelatedItemsByField(string ids,string fieldName, bool partial)`	Extends the `GetRelatedItems` method by adding more parameters. This method returns items in bidirectional relationships.
`ContainsItemsByFields(string ids, string fieldName, string fieldValue)`	For refinements to work — to check if a category relates to any of the items returned in the search results.

7. Define the following Clause Construction Helper classes:

```
AddFieldValueClause(BooleanQuery query, string fieldName, string fieldValue, BooleanClause.Occur occurance)
```

```
AddPartialFieldValueClause(BooleanQuery query, string fieldName, string fieldValue)
```

```
AddFullTextClause(BooleanQuery query, string searchText)
```

```
ApplyIdFilter(BooleanQuery query, string fieldName, string filter)
```

```
ApplyRelationFilter(BooleanQuery query, string ids)
```

The Helper classes perform some extra validation on GUIDs before passing them to the search indexes. It converts all GUIDs to lowercase and then gives them short IDs. Lucene also checks for and filters out any stop words. It is necessary to normalize GUIDs in this way to avoid unexpected results when querying.

8. Return a search index specified by ID.

```
public static Sitecore.Search.Index GetIndex(string indexId)
        {
            return SearchManager.GetIndex(indexId);
        }
```

Create a Web Control to Display Related Items

Create a Web control class to display search results on your Web site. This is one of three Web controls used in this example:

- RelatedItems.cs
- SearchResults.cs
- Facet.cs

RelatedItems.cs

The purpose of this control is to trigger a query that fetches content from the search indexes. It uses the Searcher and the Query Runner classes to return a list of items and then formats the list and displays the results as related items in the side bar of the appropriate product pages.

To create a related items control:

1. Include the following namespaces:

```
namespace Starterkit.WebControls
{
    using System;
    using System.Web.UI;
    using Sitecore.Data.Items;
    using Sitecore.Links;
    using System.Collections.Generic;
    using System.Linq;
    using Sitecore.Data;
    using Sitecore.Data.Fields;
    using Sitecore.Diagnostics;
    using Sitecore.Training.Index;
```

2. Name this Web control `RelatedItems`.

```
    public class RelatedItems : Sitecore.Web.UI.WebControl
```

3. When you create a class that inherits from a Web control, you always need a `DoRender` method. Create a `DoRender` method to control the presentation of the related items list. The following code extract shows the entire method.

```
        protected override void DoRender(HtmlTextWriter output)
        {
            var categories = from relatedItem in GetAllRelations()
                             group relatedItem by
                             GetDisplayNameById(relatedItem.ParentID) into g
                              select new { CategoryName = g.Key, Items = g };

            if (categories.Count() > 0)
            {
                //<div id="relatedItems">
                output.AddAttribute(HtmlTextWriterAttribute.Id, "relatedItems");
                output.RenderBeginTag(HtmlTextWriterTag.Div);

                //<h2>
                output.RenderBeginTag(HtmlTextWriterTag.H2);
                output.Write("Related Categories");
                //</h2>
                output.RenderEndTag();

                //<ul>
                output.RenderBeginTag(HtmlTextWriterTag.Ul);

                foreach (var category in categories)
                {
                    //<li>
                    output.RenderBeginTag(HtmlTextWriterTag.Li);
                    output.Write(String.Format("{0} ({1}):", category.CategoryName,
                    category.Items.Count()));
                    //</li>
                    output.RenderEndTag();
```

```
                        //<ul>
                        output.RenderBeginTag(HtmlTextWriterTag.Ul);

                        foreach (var item in category.Items)
                        {
                            //<li>
                            output.RenderBeginTag(HtmlTextWriterTag.Li);
                            //<a href='/item.aspx'
                            output.AddAttribute(HtmlTextWriterAttribute.Href,
                            LinkManager.GetItemUrl(item));
                            output.RenderBeginTag(HtmlTextWriterTag.A);
                            output.Write(item.DisplayName);
                            //</a>
                            output.RenderEndTag();
                            //</li>
                            output.RenderEndTag();
                        }
                        //</ul>
                        output.RenderEndTag();
                    }

                    //</ul>
                    output.RenderEndTag();
                    //</div>
                    output.RenderEndTag();
                }
            }
```

4. Create a method called `GetAllRelations`.

```
protected IEnumerable<Item> GetAllRelations()
```

This method goes to the multi-list field on the item and fetches all items that are marked as related by reading the value of the multi-list field.

It then creates an instance of the Searcher and calls the Searcher `GetRelatedItems byField` method which triggers the `Searcher` and `Query Runner`.

5. In the `GetAllRelations` method, use comments to select a Searcher method. Add or remove comments so that only one method can be run at any one time:

```
//relatedItems.AddRange(searcher.GetItemsByFullTextQuery(ItemIds));
relatedItems.AddRange(searcher.GetRelatedItemsByField(ItemIds, fieldName, true));
//relatedItems.AddRange(searcher.GetRelatedItems(ItemIds));
```

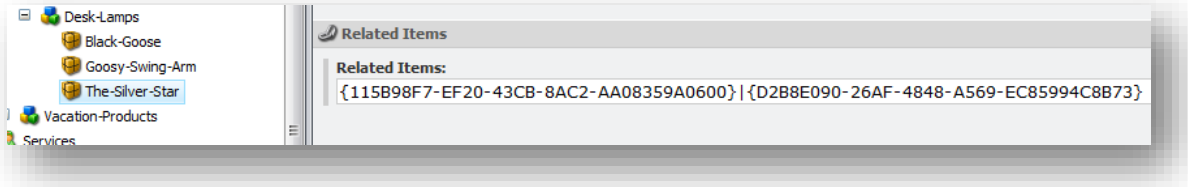
This makes it easy change the way the control interacts with the Searcher.

6. Create a method called `GetDirectItemRelations(string fieldName)`

```
        protected List<Item> GetDirectItemRelations(string fieldName)
        {
            var directItemRelations = new List<Item>();

            foreach (var itemId in ItemIds.Split(new[] { "|" },

            StringSplitOptions.RemoveEmptyEntries))
            {
                var item = Sitecore.Context.Database.GetItem(itemId);

                if (item != null &&
                    item.Fields[fieldName] != null &&
                    (FieldTypeManager.GetField(item.Fields[fieldName]) is
                     MultilistField))
                {
                    MultilistField relatedItemsField = item.Fields[fieldName];
                    directItemRelations.AddRange(relatedItemsField.GetItems());
                }
            }

            return directItemRelations;
        }
```

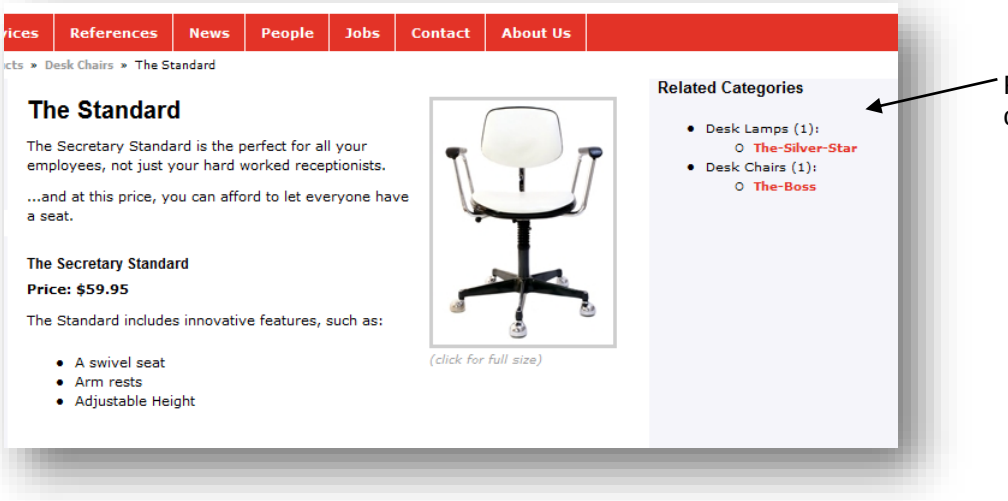
Get all related items and return list of GUIDS that are pipe separated. Sitecore uses pipe separated lists in the **Related Items** field to separate GUIDS.

In the Content Editor, **View** tab, if you select the raw values field, you can see the pipe separated GUIDS in the **RelatedItems** field.



7. Create a method called `GetDisplayNameById(ID itemId).`

   This method gets the title or display name for the selected item from the title field and displays it in the control.



8. Save and compile your new Web control.

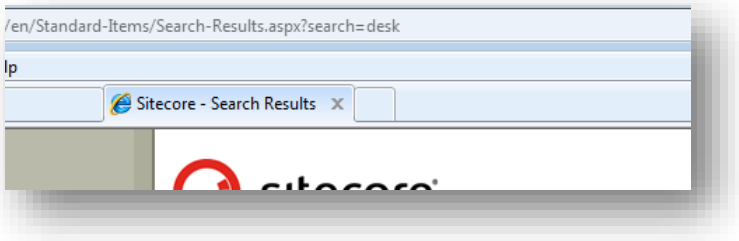
Facet.cs

This control adds the refinements that display *Category* and *Brand Name* above the related items control.

The Facet Web control fetches items from the Content Editor, content tree and then gets the URLs of each item. Then it constructs a query in the address bar of the browser.

For example:

`http://<your site>/en/Standard-Items/Search-Results.aspx?search=desk`



Code extract to show how the refinements query is constructed:

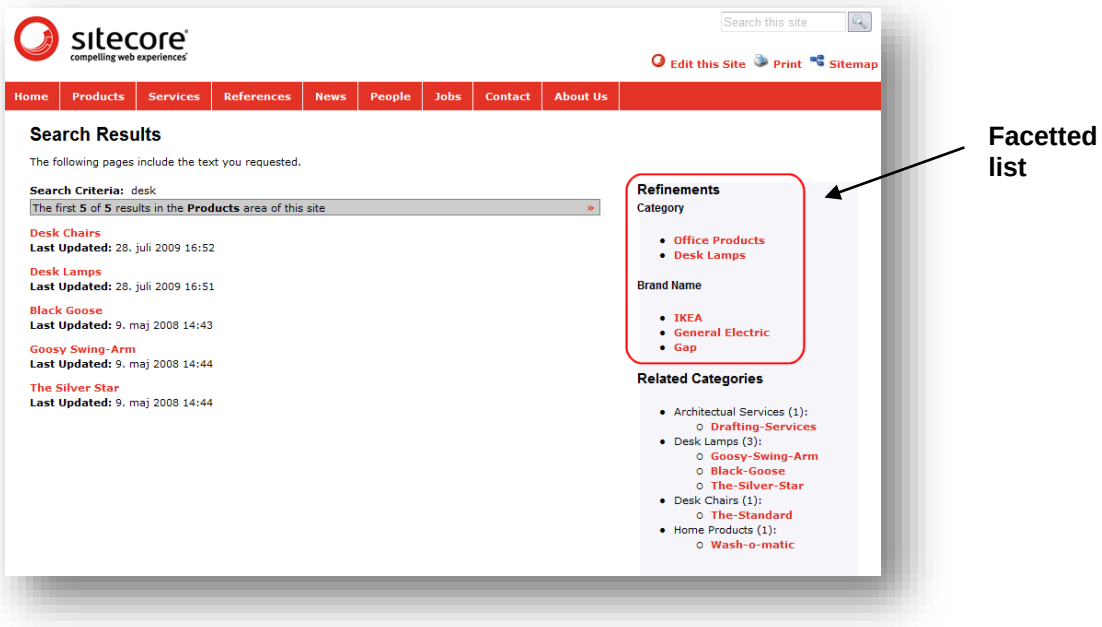
```
                {
                    string url = LinkManager.GetItemUrl(Sitecore.Context.Item);

                    var queryStringDictionary =
        WebUtil.ParseQueryString(WebUtil.GetQueryString());
```

```
                if(queryStringDictionary.ContainsKey(FieldName))
                    queryStringDictionary.Remove(FieldName);

                return WebUtil.AddQueryString(url + "?" +
        WebUtil.BuildQueryString(queryStringDictionary, false), new[] { FieldName, FieldValue
        });
            }
```

The following code calls `AppendFacetURL` that controls the presentation of the refinements:

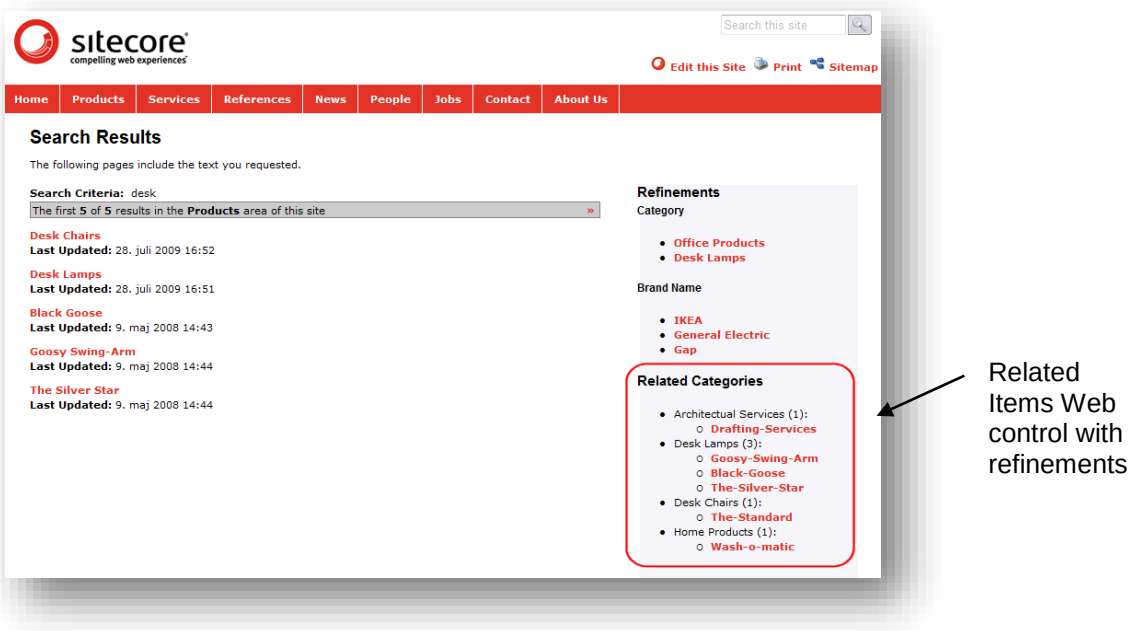
```
            {
                //<div id="relatedItems">
                output.AddAttribute(HtmlTextWriterAttribute.Class, "facet");
                output.RenderBeginTag(HtmlTextWriterTag.Div);

                //<a href='/item.aspx'
                output.AddAttribute(HtmlTextWriterAttribute.Href, AppendFacetUrl());
                output.RenderBeginTag(HtmlTextWriterTag.A);
                output.Write(Name);
                //</a>
                output.RenderEndTag();

                //</div>
                output.RenderEndTag();
            }
```

Facet Web control:



Search Results.cs

The Search Results Web control uses the same classes as `RelatedItems.cs` to display related items and other refinements in the side bar. See the full solution source code to find out how this control has been adapted to display the same related items and facetted list in the side bar.



3.2 Creating a File Crawler

Database crawlers play an important role in the Lucene search indexing process. To demonstrate how to implement a custom crawler, we will create one that indexes the file system.

3.2.1 Introduction

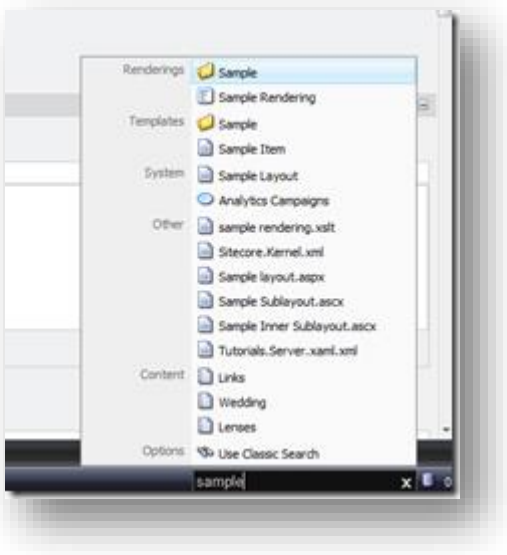
Sitecore.Search Crawler is a component in Sitecore that scans a specific storage system such as a database or file system, extracting information and storing it in a search index, making it available to Sitecore Search. It performs several roles:

- **Indexer** — Extracts data from a specified document requested by the crawler or monitor. The data extracted consists of metadata and content.
  i) *Metadata* — The Indexer extracts metadata that is understood by the system. You can filter and prioritize the metadata. For example, using the _name or _template field.
  ii) *Content* — The Indexer also extracts body content and prioritizes it. You can prioritize content in the document by using boost. This is usually only applied to a single field, giving the document a single prioritization.
- **Crawler** — Traverses a storage system and uses the indexer to populate the search index.
- **Monitor** — Monitors changes in a storage and updates the search index (not implemented in this example).

Search crawlers implement the `Sitecore.Search.Crawlers.ICrawler` interface.

This section explains how to extend your Sitecore Desktop search functionality to include files from your Web site root as well as Sitecore items in its search results. It explains how to create a *Sitecore.Search Crawler* and integrate it with Sitecore.Search framework to get results in Sitecore Desktop Search UI.

The following screenshot shows a typical set of search results after implementing a *Sitecore.Search* crawler for the file system:



Search crawlers provide a way for *Sitecore.Search* to consume data from various sources to make search results more comprehensive. You can design a crawler to index any of the following:

- Files contained in a specific folder in the file system. For example the Web site file system
- Tables in an external database
- Another folder containing Word or PDF documents
- Content from an external system, such as a CRM

- An external Web site

Note
If there are some protected Microsoft Office documents in your media library, the index rebuilding may not work for these files.

Task

- Implement a simple crawler for files in a specific folder on a Web site. The crawler makes XML and text files in the folder searchable by their content. All other files are searchable by name.
- Create a processor to integrate with the Quick Search UI so a user can perform actions on the search. For example, open and view a file that appears in the search results.
- Create a `crawler.config` file that integrates the crawler with the *Quick Search* index and installs the processor for the Quick Search UI. This solution can retrieve all Sitecore items and files that match the search criteria without any significant performance overheads.

Implement this solution in C# using Visual Studio.

Prerequisites

- Web site built on Sitecore CMS 6 or later
- Visual Studio 2008 or later

Summary of Steps

To create a *Sitecore.Search Crawler* complete the following steps:

1. Create the `Sitecore.Search` Crawler. Implement the `ICrawler` interface to index the file system.
2. Display search results in the Desktop. Integrate with the Quick Search UI so that users can perform actions based on the search results.
3. Create a File Crawler config file – sample configuration of the crawler component and UI integration.
4. Test the `Sitecore.Search` Crawler.

3.2.2 Create the Sitecore.Search Crawler

Implement a crawler that indexes files in a specific folder. In this example, the crawler will index the Web site root by default. It must crawl through the contents of the file system and add specific data to the Lucene search index.

Crawler.cs

To create a crawler:

1. Open Visual Studio. Create a new Web Application Project and copy the project files to the Sitecore Web site root.
2. Create a C# class and include the following namespaces:

```
namespace FileCrawler.FileSystem
{
  using System;
  using System.IO;
                        using System.Text;
  using System.Xml;
```

```
using Lucene.Net.Documents;
using Sitecore;
using Sitecore.Search;
using Sitecore.Search.Crawlers;
```

3. Think of a suitable name for your class, such as `Crawler`. Ensure that your class inherits from the `BaseCrawler` class and the `ICrawler` interface.

```
public class Crawler : BaseCrawler, ICrawler
```

4. Declare a property called `Root`. This property defines the root folder to index.

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

5. The `Add` method is the entry point for the crawler. Its purpose is to populate the search index. `IndexUpdateContext` passed to the `Add` method provides ways to add content to the search index.

```
public void Add(IndexUpdateContext context)
```

6. `Add` performs sanity checks and calls a recursive scanner to traverse the file system: `AddRecursive`

```
string path = MainUtil.MapPath(this.Root);
  var info = new DirectoryInfo(path);
     if (!info.Exists)
     {
       return;
     }
      this.AddRecursive(context, info);
```

7. The `Add Recursive` method traverses the file system tree by navigating through all the folders and subfolders. For each file it finds, it creates an index entry (a document) using `CreateFileEntry` method and adds it to the index using the `AddDocument` method.

```
  protected virtual void AddRecursive(IndexUpdateContext context, DirectoryInfo info)
   {
     foreach (FileInfo file in info.GetFiles())
     {
              context.AddDocument(this.CreateFileEntry(file));
     }
     foreach (DirectoryInfo subfolder in info.GetDirectories())
    {
      this.AddRecursive(context, subfolder);
     }
   }
```

8. The `CreateFileEntry method(file)` creates the file entry to be stored in the index. It calls two functions to populate the entry:

    - `AddCommonFields` adds the common metadata fields of the `Sitecore.Search` infrastructure.
    - `AddContent` extracts text content from the file that used in the search.

```
protected virtual Document CreateFileEntry(FileInfo file)
   {
    var document = new Document();
     this.AddCommonFields(document, file);
     this.AddContent(document, file);
     return document;
   }
```

Note
`Document` is a term used to define the structure of a Lucene index. Each document consists of one or more fields and behaves like a row in a database.

9. `AddCommonFields` adds the common fields supported by `Sitecore.Search`.

```
    protected virtual void AddCommonFields(Document document, FileSystemInfo info)
 {
    document.Add(this.CreateTextField(BuiltinFields.Name, info.Name));
    document.Add(this.CreateDataField(BuiltinFields.Name, info.Name));
```

```
            document.Add(this.CreateTextField(BuiltinFields.Path, info.FullName));
            document.Add(this.CreateDataField(BuiltinFields.Url, "file://" + info.FullName));
            document.Add(this.CreateTextField(BuiltinFields.Tags, this.Tags));
            document.Add(this.CreateDataField(BuiltinFields.Tags, this.Tags));
                   document.Add(this.CreateDataField(BuiltinFields.Icon,
                   "Applications/16x16/document.png"));
            document.SetBoost(this.Boost);
          }
```

Description of common fields added to the Document:

Field	Description
`BuiltinFields.Name`	Used to prioritize search by file name. This is the value to display when presenting search results to the user.
`BuildinFields.Path and BuiltinFields.Tags`	Narrows down the search results. Not fully implemented in the current UI.
`BuiltinFields.Url`	Important! Used to identify and open a file associated with the result. Notice that "file://" is the prefix to help identify results from the file system. This field is vital for integration with Quick Search because the UI relies on a URL value to identify and activate specific results. URL value must be sufficient to identify a specific result from this location among all results returned by Sitecore.Search.
`BuiltinFields.Icon`	Used to display an icon next to each of the search results.
`Boost`	Used to adjust the priority of results from the file system relative to other results.

Description of functions used to create fields in a Document:

Function	Description
`CreateTextField(name, value)`	Creates a field optimized for full-text search. The content of the field cannot be retrieved from the index.
`CreateValueField(name, value)`	Creates a field optimized for value search (such as dates, GUIDs etc). The content of the field cannot be retrieved from the index.
`CreateDataField(name, value)`	Creates a field returned in the search result. It is not possible to search for values in such fields.

Note
These functions are just helpers, and it is also possible to use the Lucene.Net API here.

10. The `AddContent` function extracts text from the indexed files.

    It relies on two example functions for getting the content:

    - `AddXmlContent` to extract information from XML files (XAML controls, layouts or configuration files)
    - `AddTextContent` to load content from UTF-8 encoded text files (ASPX, CSS and JS files)

```
protected virtual void AddContent(Document document, FileInfo file)
{
  if (this.AddXmlContent(document, file))
  {
    return;
  }
  if (this.AddTextContent(document, file))
  {
    return;
  }
}
```

11. The `AddTextContent` function converts the indexed content into text. It reads the file as text in UTF-8 and puts the content into the search index. Notice that `BuiltinFields.Content` stores information in the document which is the default destination for search queries performed by the `Sitecore.Search` framework.

```
protected virtual bool AddTextContent(Document document, FileInfo file)
    {
       try
      {
        using (var reader = new StreamReader(file.FullName, Encoding.UTF8))
        {
          document.Add(this.CreateTextField(BuiltinFields.Content,
          reader.ReadToEnd()));
        }
        return true;
      }
      catch
      {
      }
      return false;
    }
```

12. The `AddXmlContent` function extracts content from XML files.

    It parses the file as XML and adds the text content to the index. It does not add markup elements and attribute names to the index. This function also detects and boosts files with "`TODO`" markers in the comments and text nodes.

```
protected virtual bool AddXmlContent(Document document, FileInfo file)
    {
       try
      {
        using (var reader = new StreamReader(file.FullName))
        {
          var xreader = new XmlTextReader(reader);
          while (xreader.Read())
          {
            if (xreader.NodeType == XmlNodeType.Text ||
                xreader.NodeType == XmlNodeType.Attribute ||
                xreader.NodeType == XmlNodeType.CDATA ||
                xreader.NodeType == XmlNodeType.Comment)
            {
              float boost = 1.0f;
              if (xreader.Value.IndexOf("TODO",
              StringComparison.InvariantCultureIgnoreCase) >= 0)
              {
                boost = 5.0f;
              }
              document.Add(this.CreateTextField(BuiltinFields.Content, xreader.Value,
              boost));
```

```
                    }
                  }
                }
                return true;
              }
              catch
              {
              }
              return false;
            }
```

You have now successfully created a `Sitecore.Search` Crawler.

3.2.3 How to Display Search Results in the Desktop

Create a class called `Launchresult.cs`. This class allows user to open files found in Quick Search in the Developer Center.

Launchresult.cs

Include the following namespaces:

```
namespace FileCrawler.FileSystem
{
  using System.IO;
  using Sitecore;
  using Sitecore.Shell.Framework;
  using Sitecore.Shell.Framework.Pipelines;
  using Sitecore.Web.UI.Sheer;
```

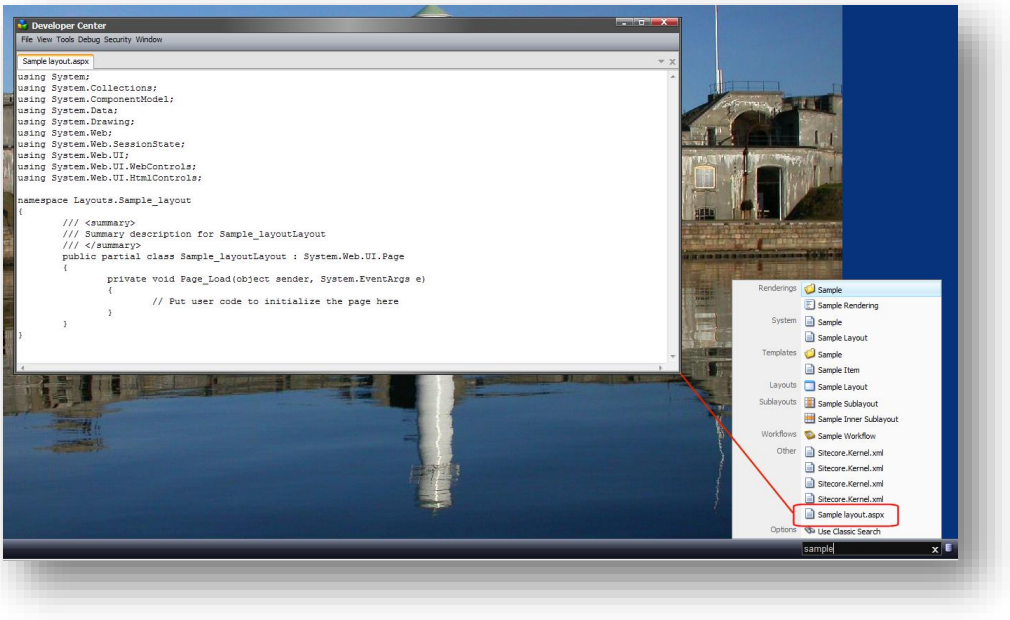
When a user selects a search result that points to a file in the file system, the following function processes the request and opens the file in the Developer Center:

```
public class LaunchResult
  {
    public void Process(LaunchSearchResultArgs args)
    {
      if (args.HasResult)
      {
        return;
      }
      if (!args.Url.StartsWith("file://"))
      {
        return;
      }

      string path = args.Url.Substring("file://".Length);
      if (File.Exists(path))
      {
        Windows.RunApplication("Layouts/IDE", "fi=" + path);
      }
      args.AbortPipeline();
    }
  }
}
```

This section of code contains the path to the file that the *Sitecore.Search Crawler* has found.

```
string path = args.Url.Substring("file://".Length);
      if (File.Exists(path))
```

This function opens the developer center, using the file path as a parameter.



When you see a file name in the search results, click the file name, and open it directly in the Developer Center. The `fi` query string parameter allows you to open the developer center to view or edit the specific file.

```
        Windows.RunApplication("Layouts/IDE", "fi=" + path);
```

3.2.4 Create a File Crawler Config File

Include the `App_Config\Include` folder in the project and create a new file called `FileCrawler.config`. This is an include file that integrates the crawler and UI handler with the `Sitecore.Search` framework. Copy and paste the content below into the `FileCrawler.config` file:

FileCrawler.config

```
        <configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
          <sitecore>
            <processors>
              <uiLaunchSearchResult>
                <processor type="FileCrawler.FileSystem.LaunchResult,FileCrawler"
                 patch:before="*[1]"/>
              </uiLaunchSearchResult>
            </processors>
            <search>
              <configuration>
                <indexes>
                  <index id="system">
                    <locations>
                      <filesystem type="FileCrawler.FileSystem.Crawler, FileCrawler">
                        <Root>/</Root>
                        <Tags>filesystem</Tags>
                        <Boost>0.5</Boost>
                      </filesystem>
                    </locations>
                  </index>
                </indexes>
              </configuration>
            </search>
          </sitecore>
        </configuration>
```

`LaunchResult` processor integrates into the `uiLaunchSearchResult` pipeline which handles user requests to open a search result:

```
        <uiLaunchSearchResult>
                <processor type="FileCrawler.FileSystem.LaunchResult,FileCrawler"
patch:before="*[1]"/>
              </uiLaunchSearchResult>
```

It adds the new `FileSystem.Crawler` as a location in the system index. Sitecore Quick Search uses the system index:

```
                  <index id="system">
                    <locations>
                      <filesystem type="FileCrawler.FileSystem.Crawler, FileCrawler">
                        <Root>/</Root>
                        <Tags>filesystem</Tags>
                        <Boost>0.5</Boost>
                      </filesystem>
                    </locations>
                  </index>
```

`Root` specifies the root folder of the Web site which is the starting point for the crawler.

```
                        <Root>/</Root>
```

Tags are words or text that can be associated with search results. When performing a search, tags enable you to return results with a specific tag by appending `+_tags:tagname` to the search query.

The current Sitecore UI does not support tags. However, when implementing a Web site search or other custom search UI, it is possible to include support for tags.

```
                           <Tags>filesystem</Tags>
```

Enter a value in *Boost* to adjust the priority of files compared to other search results in Sitecore, such as items, control panel options, or applications. A value of 0.5 gives a usable list of search results.

```
                        <Boost>0.5</Boost>
```

Rules for Boost

- Value must be greater than zero.
- Default value for Boost is 1.0
- Values greater than 1.0 push the results to the top.
- Values less than 1.0 drag the results to the bottom (this is not often used).

3.2.5 Test the Sitecore.Search Crawler

Now that you have created a *Sitecore.Search Crawler* and a class that displays your results, it is time to test what you have created.

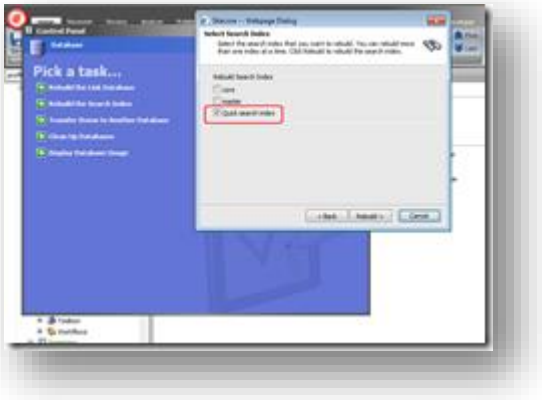
There are two simple tests that you can perform.

Test Scenario 1

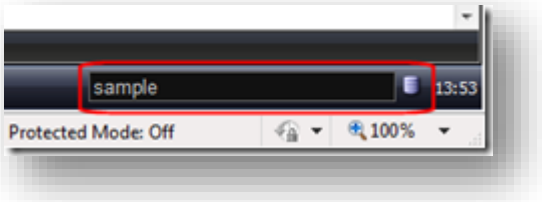
1. Compile your *Sitecore.Search Crawler* solution.
2. Open Sitecore and rebuild the search index.

To rebuild the search index:

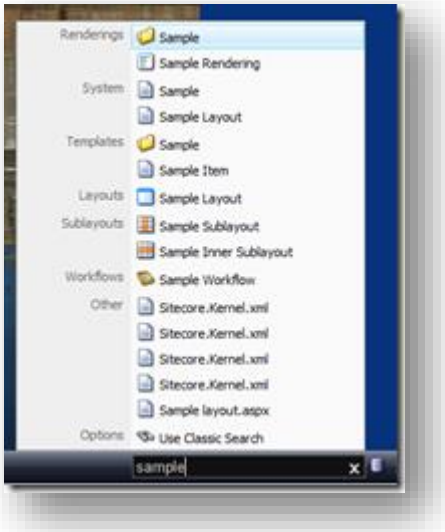
3. In the Sitecore Desktop, open the Control Panel, click **Database**, and then click **Rebuild the Search Index**.
4. In the **Select Search Index** dialog box, select *Quick search index* and then click *Rebuild.*



5. In the Sitecore Desktop, enter part of the file name in the search box. The system will show matching file names and allow you to open the files in the editor.

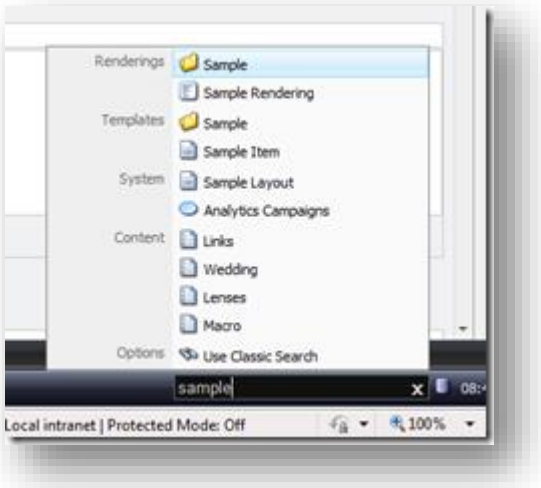


Search results found:

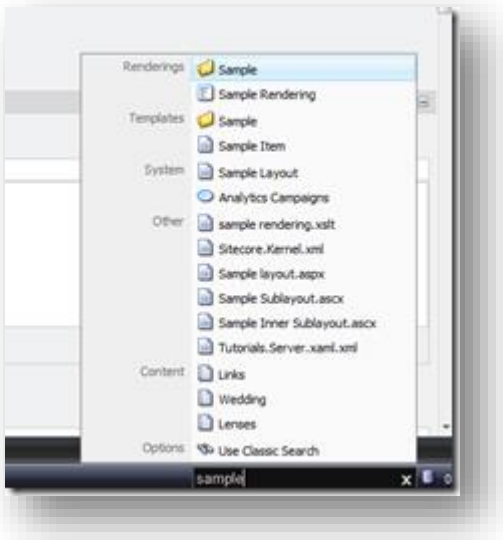


Before and after implementing the Sitecore.Search Crawler for the File System:

Quick Search results using the search term *sample* in the Quick Search box.



Then the same search results after implementing the *Sitecore.Search* crawler for the file system. This search returns more files from the Web site root including XML and `.ascx` files under the *Other* category.



Test Scenario 2

1. Enter some text into a file `ASPX` layout, `ASCX` sublayout, `XSL, CSS, JS, or CS` file. This could be a comment in the source code that describes certain functionality or a "`TODO`" marker left for later. Quick search will look for this term.
2. Rebuild the search index (see Test Scenario 1).
3. Type the same or a similar term into the Quick search box. The system will display the name of the file and allow you to open it in the developer center.

Extending the Sitecore.Search Crawler

This example of a `Sitecore.Search` Crawler demonstrates how it is possible to create your own crawler. This is a relatively simple example.

Once you have created a `Sitecore.Search` Crawler there are many other ways in which you can extend and improve its functionality.

For example:

- Extend the class to monitor and update search indexes automatically when a file changes.
- Exclude certain files or items from search results.
- Conduct filtering at the time of crawling.
- Add metadata and custom fields to search by. For example, author, size, and age.

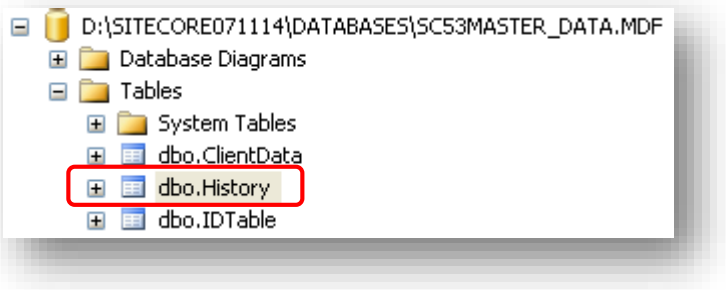
3.3 Appendix

This section contains information taken from the Lucene Search Engine document on how to configure Lucene search indexes using the `Sitecore.Data.Indexing` API. Some of this information may still be useful to developers working with search indexes.

Important Note
In Sitecore CMS 6.5, the `Sitecore.Data.Indexing` API was deprecated and in Sitecore CMS 7.0 it will be completely removed. To configure Sitecore Search or Lucene search indexes wherever possible use the `Sitecore.Search` API.

3.3.1 Updating Indexes

Whenever you change or modify a content item in Sitecore, the History Engine adds a record to the History table.



As a result, this fires the `IndexingManager.HistoryEngine_AddedEntry` event and starts the `UpdateIndexAsync` job, which in turn updates the Lucene indexes. The `UpdateIndexAsync` job only indexes newly added or changed items.

The first time you create an index the History table is empty, so it is important to rebuild the search indexes either manually or programmatically before allowing Sitecore to update the indexes.

The `Properties` table contains a record with the key `IndexingProvider_LastUpdate`. This record allows you to set a date to use as the starting point for item indexing. Any items modified before this date are not updated by the `UpdateIndexAsync` job. Therefore after you have defined an index for a database it is important to rebuild the indexes.

In the `web.config` file, you can set an update interval parameter to a specific time value so that Sitecore can update the Lucene indexes on a regular basis.

If an item has not been indexed after a modification has been made, then it will be indexed when the update interval expires. In the following example, the interval is set to update the indexes every five minutes:

```
<setting name="Indexing.UpdateInterval" value="00:05:00" />
```

This feature has been implemented for safety purposes – if the index update process breaks, the appropriate items will be indexed within the `Indexing.UpdateInterval`.

The History table is updated by the `Engines.HistoryEngine.Storage` engine, which is defined for every database that you want to index.

Example index definition for the master database:

```
        <!-- master -->
        <database id="master" singleInstance="true" type="Sitecore.Data.Database,
Sitecore.Kernel">
        ...
        <Engines.HistoryEngine.Storage>
              <obj type="Sitecore.Data.$(database).$(database)HistoryStorage, Sitecore.Kernel
">
                  <param desc="connection" ref="connections/$(id)">
```

```
                </param>
                <EntryLifeTime>30.00:00:00</EntryLifeTime>
              </obj>
        </Engines.HistoryEngine.Storage>
        <Engines.HistoryEngine.SaveDotNetCallStack>false</Engines.HistoryEngine.SaveDotNetCall
Stack>
        ...
        </database>
```

Note
If you are using an Oracle database you need to refer to the `Sitecore.$(database).dll` as it is not part of the Sitecore.Kernel.

The `SqlServerHistoryStorage` class adds the history of changing, adding and deleting items to the current database `History` table. Each addition starts the index update job, and another job runs at a specified interval to check if there is anything else left to index.

The process described in this section updates the indexes automatically. You can also rebuild the indexes manually. To do this, in the Sitecore desktop, click *Control Panel*, *Database*, then click *Rebuild the Search* Index.

It is also possible to index items based on a specific template. This is done by adding the `<templates hint="list:AddTemplate">` definition to an index.

For example, the following `<index>` definition for the archive database contains the `<templates>` definition:

```
        <index id="archive" singleInstance="true" type="Sitecore.Data.Indexing.Index,
Sitecore.Kernel">
                <param desc="name">$(id)</param>
                <templates hint="list:AddTemplate">
                  <!-- Archived item template -->
                  <template>{BF2B8DA2-3CBA-485D-8F85-3788B8AFBDBF}</template>
                </templates>
                <fields hint="raw:AddField">
                  <field target="name">@name</field>
        ...
```

3.3.2 Lucene Search in Staged Environments

You can use Lucene search on a slave server in a staged environment with multiple web farm servers if the slave web farm servers share the same production (Web) database.

To configure Lucene in a staged environment, first open the `web.config` file and set the `Indexing.ServerSpecificProperties` parameter to `true`.

```
<!--  INDEX PROPERTIES PER SERVER
      Indicates if server specific keys should be used for property values (such as
'last updated').
      Default value: false
-->
<setting name="Indexing.ServerSpecificProperties" value="true" />
```

Then follow these steps:

1. Add the following entries on the Master server for production (web) database which is shared between the Slave and the Master servers

```
            <!-- Production -->
              <database id="production" singleInstance="true" type="Sitecore.Data.Database,
Sitecore.Kernel">
                <param desc="name">$(id)</param>
                <securityEnabled>true</securityEnabled>
                <dataProviders hint="list:AddDataProvider">
                  <dataProvider ref="dataProviders/main" param1="$(id)">
                    <disableGroup>publishing</disableGroup>
                    <prefetch hint="raw:AddPrefetch">
                      <sc.include file="/App_Config/Prefetch/Common.config" />
                      <sc.include file="/App_Config/Prefetch/Web.config" />
```

```
                    </prefetch>
                  </dataProvider>
                </dataProviders>
                <proxiesEnabled>false</proxiesEnabled>
                <proxyDataProvider ref="proxyDataProviders/main" param1="$(id)" />
                <!-- Add for Lucene -->
                <Engines.HistoryEngine.Storage>
                  <obj type="Sitecore.Data.$(database).$(database)HistoryStorage,
Sitecore.Kernel ">
                    <param desc="connection" ref="connections/$(id)">
                    </param>
                    <EntryLifeTime>30.00:00:00</EntryLifeTime>
                  </obj>
                </Engines.HistoryEngine.Storage>

<Engines.HistoryEngine.SaveDotNetCallStack>false</Engines.HistoryEngine.SaveDotNetCallStack>
                <!-- End Lucene -->
                <cacheSizes hint="setting">
```

2. Add these entries for the same production database on the Slave server:

```
              <!-- Production -->
              <database id="production" singleInstance="true" type="Sitecore.Data.Database,
Sitecore.Kernel">
                <param desc="name">$(id)</param>
                <securityEnabled>true</securityEnabled>
                <dataProviders hint="list:AddDataProvider">
                  <dataProvider ref="dataProviders/main" param1="$(id)">
                    <disableGroup>publishing</disableGroup>
                    <prefetch hint="raw:AddPrefetch">
                      <sc.include file="/App Config/Prefetch/Common.config" />
                      <sc.include file="/App_Config/Prefetch/Web.config" />
                    </prefetch>
                  </dataProvider>
                </dataProviders>
                <proxiesEnabled>false</proxiesEnabled>
                <proxyDataProvider ref="proxyDataProviders/main" param1="$(id)" />
                 <!-- Add for Lucene -->
                 <indexes hint="list:AddIndex">
                   <index path="indexes/index[@id='system']" />
                 </indexes>
                 <Engines.HistoryEngine.Storage>
                   <obj type="Sitecore.Data.$(database).$(database)HistoryStorage, Sitecore.
Kernel">
                     <param desc="connection" ref="connections/$(id)">
                     </param>
                     <EntryLifeTime>30.00:00:00</EntryLifeTime>
                   </obj>
                 </Engines.HistoryEngine.Storage>

<Engines.HistoryEngine.SaveDotNetCallStack>false</Engines.HistoryEngine.SaveDotNetCallStack>
                 <!-- End Lucene -->
```

3. Make the full publishing from Master to Slave server. This will update the production (web) database that is shared between the Master and the Slave servers.
4. On the Slave server create the Lucene index for the production (web) database.
5. To create the index for the production database, in the Sitecore Desktop, click **Control Panel**, **Database**, and then click **Rebuild Search Indexes**.

   This will create the index for the production database. In case the Sitecore back-end is not available on the slave server, you can use the following code snippet to rebuild the indexes:

```
        Database database = Factory.GetDatabase("web");
        if (database != null)
           {
           for (int i = 0; i < database.Indexes.Count; i++)
              {
              database.Indexes[i].Rebuild(database);
              }
           }
```

3.3.3 Troubleshooting Sitecore Search in a Distributed Environment

In a distributed system, where there is at least one Content management and one Content delivery instance. The content delivery instance maintains its own search index and may not be aware of changes made to the content management instance.

To ensure that each instance is fully aware of all changes made to search indexes on other instances:

1. Ensure that the application pool account has read/write/modify access to the `/data/indexes` folder or any other location where your indexes are stored.
2. Enable the `HistoryEngine` on the web database:

```
          <database id="web"
              <Engines.HistoryEngine.Storage>
                 <obj type="Sitecore.Data.$(database).$(database)HistoryStorage,
Sitecore.Kernel">
                    <param connectionStringName="$(id)"/>
                    <EntryLifeTime>30.00:00:00</EntryLifeTime>
                 </obj>
              </Engines.HistoryEngine.Storage>
              …
         </database>
```

   Repeat this on both the Content Management and Content Delivery instances.

   You may also have a number of *web* databases configured, for example, *stage-web*, *pub-web* or *prod-web*. As the names of the databases may be different from one environment to another, you need to apply this step to each of the *web* databases you use to deliver content in production.

3. Do not set the update interval setting to “00:00:00” as this disables the live index rebuild functionality:

   For example, if you set the default value to *5*, this means that the remote server will check if it needs to add anything to the index every 5 minutes.

```
<setting name="Indexing.UpdateInterval" value="00:05:00"/>
```

   Even if everything is running ok, you may still experience some delay. Adjust the update interval depending on your environment and frequency of content change but 30 seconds should be the minimum value.

4. In the `web.config`, enable `Indexing.ServerSpecificProperties`:

```
<setting name="Indexing.ServerSpecificProperties" value="true"/>
```

   In most installations this is set to *true*.

   Set `Indexing.ServerSpecificProperties` to true if:

   - You have more than one Content Delivery server in a web farm
   - Your Content Management environment points to the same physical web database as the Content Delivery environment.

   In a clustered Content Management environment this setting is overridden and set to *true* automatically using the `EventQueues` functionality.

   If this setting is set to *false* and you have one of the configurations mentioned above, the Content Delivery server will not know that it needs to update the indexes.

   After each index update operation, Sitecore writes a timestamp to the Properties table of the currently processed database. This helps the `IndexingProvider`, that is responsible for the index update process to know which items to extract from the history table when performing an index update. With `Indexing.ServerSpecificProperties` set to *false*, the timestamp is not unique to the environment, so you may have an issue when Content

Delivery is confused regarding which items to process from the history table.

The instance name can either be explicitly set in the `web.config` or created from a combination of the machine name and the site name. This grants the uniqueness of the key within an environment.

5. Check your index configuration:
    a. Your search index configuration in Content Database may be referencing the *master* database instead of the *web* database.
    b. Check that the root index is configured to point to an item that actually exists in the *web* database:

```
<search>
        <configuration>
          <indexes>
            <index id="test" type="Sitecore.Search.Index, Sitecore.Kernel">
               <param desc="name">$(id)</param>
               <param desc="folder">test</param>
               <Analyzer ref="search/analyzer" />
               <locations hint="list:AddCrawler">
                   <master type="...">
                       <Database>master</Database>
                     <Root>/sitecore/content/test</Root>
```

    c. If you are using template filters within the configuration, make sure that every tag in the `<include />` section is unique as they are used as keys:

```
<include hint="list:IncludeTemplate">
         <residential>{71D42CF2-CE89-4030-9EB1-0065B35B78C4}</residential>
         <business>{ED9F466B-D436-4A3F-B22F-EA6E8097085D}</business>
         <industry>{78166FE4-EDFB-4B0D-A3ED-860AEB44CD40}</industry>
</include>
```

Otherwise only the last item will get into the filter if you define it like this:

```
<include hint="list:IncludeTemplate">
                <template>{71D42CF2-CE89-4030-9EB1-0065B35B78C4}</template>
                <template>{ED9F466B-D436-4A3F-B22F-EA6E8097085D}</template>
                <template>{78166FE4-EDFB-4B0D-A3ED-860AEB44CD40}</template>
</include>
```

Further troubleshooting steps to take if the previous steps fail:

- Rule out *live indexing* functionality that relies on the history table, update interval and index configuration settings.
- To find out if you have properly configured the index and to run a full index rebuild on the Content Delivery side:

1. Download the `RebuildDatabaseCrawlers.zip` script and copy it to the `/layouts` folder of your Content Delivery instance. Download the script from Alex Shyba's blog website:

   http://sitecoreblog.alexshyba.com

2. Execute the script by entering the following URL in your browser:

   `http://<your_site>/layouts/RebuildDatabaseCrawlers.aspx`

3. Toggle to the index you want to rebuild and click rebuild.

When you execute the script it launches a background process. You must view the log files to confirm whether the script worked or not. If your custom list does not appear in the log files then your index is not correctly registered in the system. Review your configuration and ensure that your index is in the configuration files.

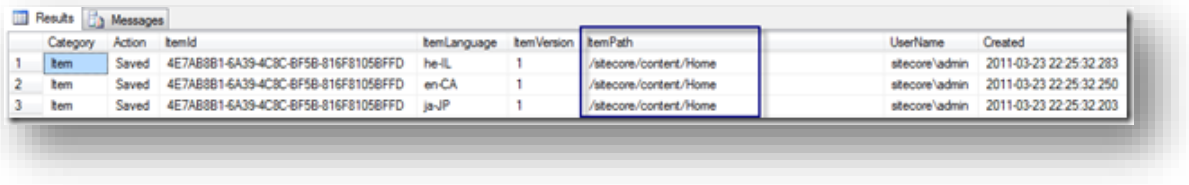
After you have rebuilt the index, if you start getting hits and the search index contains the expected number of documents, then the index itself is configured properly. To confirm this use *IndexViewer* or *Luke*.

To check if you have configured *live indexing* correctly:

1. Login to the Content Management instance.
2. Modify a content item that you know is included in the search index. For example, change the **Title** field in an item.
3. On the ribbon, click **Save**.
4. Check to see if the item change appears in the index on the Master/Content Management side.
5. Publish the item that you changed.
6. Verify that the item change was published and then clear the cache.
7. Open SQL Management Studio and query the History table of the web database. Use the following SQL query:

```
SELECT Category, Action, ItemId, ItemLanguage, ItemVersion, ItemPath, UserName, Created
FROM [Sitecore_web].[dbo].[History] order by created desc
```

Your query results display all modified items:

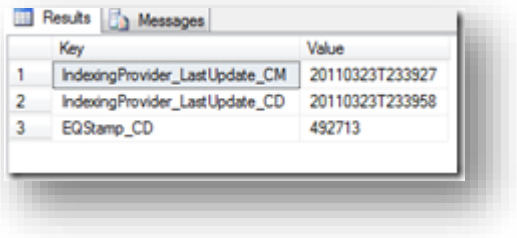


	Category	Action	ItemId	ItemLanguage	ItemVersion	ItemPath	UserName	Created
1	Item	Saved	4E7AB8B1-6A39-4C8C-BF5B-816F81058FFD	he-IL	1	/sitecore/content/Home	sitecore\admin	2011-03-23 22:25:32.283
2	Item	Saved	4E7AB8B1-6A39-4C8C-BF5B-816F81058FFD	en-CA	1	/sitecore/content/Home	sitecore\admin	2011-03-23 22:25:32.250
3	Item	Saved	4E7AB8B1-6A39-4C8C-BF5B-816F81058FFD	ja-JP	1	/sitecore/content/Home	sitecore\admin	2011-03-23 22:25:32.203

8. Now query the **Properties** table of the Web database using the following query:

```
SELECT [Key], [Value] FROM [Sitecore_web].[dbo].[Properties]
```

Your query results should display two *IndexProvider* related entries for each of the environments.



	Key	Value
1	IndexingProvider_LastUpdate_CM	20110323T233927
2	IndexingProvider_LastUpdate_CD	20110323T233958
3	EQStamp_CD	492713

Note
The actual key names can vary depending on your configuration.

When performing updates, UTC based timestamps help the `IndexingProvider` to know which items to retrieve from the history table.

Therefore, the timestamp for the Content Delivery environment should be later than the timestamp for Content Management.

If you do not see an entry for the Content Delivery environment, then your configuration may be incorrect. Check the `UpdateInterval` setting, history table and index configuration.

9. Open up the most recent log file on the Content Delivery instance and look for the following entries:
    - *ManagedPoolThread #12 16:39:58 INFO Starting update of index for the database 'web' (1 pending).*
    - *ManagedPoolThread #12 16:39:58 INFO Update of index for the database 'web' done.*

    These entries indicate that the `IndexingProvider` ran correctly and processed the changed items (1 item in this example). The item should now be in the physical index file as a document.

    If you do not see these messages, then there is an error with the crawler. Look for any exceptions that appear in this timeframe. The `DatabaseCrawler` component may not be processing your items properly. Therefore, you may need to override the crawler and step into the code to fix the problem.
10. As a final check, take a deeper look into the search index files themselves.

    The following tools will help you to browse the contents of the index and search:
    - IndexViewer
    - Luke

After reviewing all these steps, your search index should now be working correctly.

3.3.4 General Troubleshooting Steps

When you index website content using Lucene, if you have problems finding the information you have indexed, try performing the following troubleshooting steps:

1. Check that the information you want to index is in the database you are indexing. If you use two databases, such as a master and web, make sure that this information is available in the appropriate database.
2. Check that you are using the correct search index.
3. If you are using your own indexing class, make sure that content is stored and can be searched using the available indexing fields.
4. Rebuild indexes for databases.