The system provides a way to customize the content that smart search indexes store for pages or objects. You can add external data into the index, or parse and otherwise modify the text that the smart search adds to the index by default. Such customizations allow you to extend and change how the search finds matching results.

To customize the content of indexes, implement event handlers for the following events:

- **DocumentsEvents.GetContent.Execute** – occurs before the system writes a page's data into smart search indexes.
- **ObjectEvents.GetContent.Execute** – occurs before the system writes the data of objects into search indexes. You can also use *<name>Info.TYPEINFO.Events* instead of *ObjectEvents* to handle the GetContent events for specific object types.

You need to assign handlers for the events at the beginning of the application's life cycle – create a custom module class and override the module's **OnInit** method.

The system triggers **GetContent** events when a page or object is updated and when rebuilding search indexes.

Inside the GetContent event handler methods, you can access the search index content via the **Content** property of the handler's **DocumentSearchEventArgs** or **ObjectEventArgs** parameter (string type).

Search index content

Within search indexes, the content managed by the **GetContent** events is stored in a system field that combines values from a large number of other fields (all fields that have the *Content* flag enabled in the search settings of Kentico object fields). The name of the field within indexes depends on the type of the index:

- Locally stored indexes: **_content**
- Azure Search indexes: **sys_content**

Index rebuilding requirements

If you modify the external data that your GetContent event handlers add to the index, the system does not automatically update the content of the corresponding indexes. To ensure that your indexes are up to date after making changes to the external data, you need to manually **rebuild** the indexes (or set up automatic updating of the search indexes via additional custom code).

For example, if you use GetContent handlers to add **user** data into **page** indexes:

- Saving a page covered by the index updates the content of the index, including the external user data.
- Saving user objects directly does NOT update the index – a rebuild is required.

Example - Adding data from a user field to page indexes (DocumentEvents)

Pages in Kentico can have a user assigned as the owner. The following example shows how to add the **Description** value from the user settings of a page's owner into the search index content of each page.

1. Open your Kentico web project in Visual Studio (using the **WebSite.sln** or **WebApp.sln** file).
2. Create a custom module class. For example, name the class **CustomSmartSearchModule.cs**.
    - Either add the class into a custom project within the Kentico solution (recommended) or directly into the Kentico web project (into a custom folder under the **CMSApp** project for *web application* installations, into the **App_Code** folder for *web site* installations).
3. Override the module's **OnInit** method and assign a handler to the **DocumentEvents.GetContent.Execute** event.

```
using CMS;
using CMS.DataEngine;
using CMS.DocumentEngine;
using CMS.Membership;

// Registers the custom module into the system
[assembly: RegisterModule(typeof(CustomSmartSearchModule))]

public class CustomSmartSearchModule : Module
{
    // Module class constructor, the system registers the module under the name
"CustomSmartSearch"
    public CustomSmartSearchModule()
        : base("CustomSmartSearch")
    {
    }

    // Contains initialization code that is executed when the application starts
    protected override void OnInit()
    {
        base.OnInit();

        // Assigns a handler to the GetContent event for pages
        DocumentEvents.GetContent.Execute += OnGetPageContent;
    }

    private void OnGetPageContent(object sender, DocumentSearchEventArgs e)
    {
        // Gets an object representing the page that is being indexed
        TreeNode indexedPage = e.Node;

        // Checks that the page exists
        if (indexedPage != null)
        {
            // Gets the user object of the page owner
            UserInfo pageOwner = UserInfoProvider.GetUserInfo(indexedPage.
NodeOwner);

            if (pageOwner != null)
            {
                // Adds the value of the "Description" field from the owner's
user settings into the indexed content
                // Spaces added as separators to ensure that typical search index
analyzers can correctly tokenize the index content
                e.Content += " " + pageOwner.UserDescription + " ";
            }
        }
    }
}
```

4. Save the **CustomSmartSearchModule.cs** file.
5. Sign in to the Kentico administration interface.
6. Open the **Smart search** application and **Rebuild** your page search indexes.

The search now returns results for pages if the owner's description field matches the search text.

Example - Indexing personal category names for users (ObjectEvents)

Users in Kentico can create personal categories for organizing pages. The following example demonstrates how to customize the search so that the display names of personal categories are included in the content of user indexes.

1. Open your Kentico web project in Visual Studio (using the **WebSite.sln** or **WebApp.sln** file).
2. Create a custom module class. For example, name the class **CustomSmartSearchModule.cs**.

    - Either add the class into a custom project within the Kentico solution (recommended) or directly into the Kentico web project (into a custom folder under the **CMSApp** project for *web application* installations, into the **A pp_Code** folder for *web site* installations).

3. Override the module's **OnInit** method and assign a handler to the **UserInfo.TYPEINFO.Events.GetContent.Execute** event.

```
using CMS;
using CMS.DataEngine;
using CMS.Membership;
using CMS.Taxonomy;

// Registers the custom module into the system
[assembly: RegisterModule(typeof(CustomSmartSearchModule))]

public class CustomSmartSearchModule : Module
{
        // Module class constructor, the system registers the module under the
name "CustomSmartSearch"
        public CustomSmartSearchModule()
                : base("CustomSmartSearch")
        {
        }

        // Contains initialization code that is executed when the application
starts
        protected override void OnInit()
        {
                base.OnInit();

                // Assigns a handler to the GetContent event for user
objects
                UserInfo.TYPEINFO.Events.GetContent.Execute += OnGetUserContent;
        }

        private void OnGetUserContent(object sender, ObjectEventArgs e)
        {
                // Gets the indexed user object
                UserInfo indexedUser = (UserInfo)e.Object;

                // Checks that the object exists
                if (indexedUser != null)
                {
                        // Gets the personal page categories of the indexed user
                        ObjectQuery<CategoryInfo> personalCategories =
CategoryInfoProvider.GetCategories().WhereEquals("CategoryUserID", indexedUser.
UserID);

                        // Loops through the categories
                        foreach (CategoryInfo category in personalCategories)
                        {
                                // Adds the display name of the category to the
user search index
                                // Spaces added as separators to ensure that
typical search index analyzers can correctly tokenize the index content
                                e.Content += " " + category.CategoryDisplayName +
" ";
                        }
                }
        }
}
```

4. Save the **CustomSmartSearchModule.cs** file.
5. Sign in to the Kentico administration interface.

6. Open the **Smart search** application and **Rebuild** your user indexes.

The search results now include users if the name of at least one of the user's personal page categories matches the search text.

Example - Indexing the names of assigned product options for product pages (DocumentEvents)

The following example shows how to add the names of e-commerce product options into the indexed content for the relevant product pages. Only affects product options in categories of the **Products** type.

1. Open your Kentico web project in Visual Studio (using the **WebSite.sln** or **WebApp.sln** file).
2. Create a custom module class. For example, name the class **CustomSmartSearchModule.cs**.
   - Either add the class into a custom project within the Kentico solution (recommended) or directly into the Kentico web project (into a custom folder under the **CMSApp** project for *web application* installations, into the **A pp_Code** folder for *web site* installations).
3. Override the module's **OnInit** method and assign a handler to the **DocumentEvents.GetContent.Execute** event.

```
using CMS;
using CMS.DataEngine;
using CMS.DocumentEngine;
using CMS.Ecommerce;

// Registers the custom module into the system
[assembly: RegisterModule(typeof(CustomSmartSearchModule))]

public class CustomSmartSearchModule : Module
{
        // Module class constructor, the system registers the module under the
name "CustomSmartSearch"
        public CustomSmartSearchModule()
                : base("CustomSmartSearch")
        {
        }

        // Contains initialization code that is executed when the application
starts
        protected override void OnInit()
        {
                base.OnInit();

                // Assigns a handler to the GetContent event for pages
                DocumentEvents.GetContent.Execute += OnGetProductPageContent;
        }

        private void OnGetProductPageContent(object sender,
DocumentSearchEventArgs e)
        {
                // Gets an object representing the page that is being indexed
                TreeNode indexedPage = e.Node;

                // Checks that the page exists and represents a product (SKU)
                if (indexedPage != null && indexedPage.HasSKU)
                {
                        // Gets the ID of the SKU
                        int skuId = indexedPage.NodeSKUID;

                        // Checks that the SKU has at least one enabled product
option
                        if (SKUInfoProvider.HasSKUEnabledOptions(skuId))
                        {
                                // Gets the SKU's enabled product option 
categories of the "Products" type
                                ObjectQuery<OptionCategoryInfo> categories = 
OptionCategoryInfoProvider.GetProductOptionCategories(skuId, true, 
OptionCategoryTypeEnum.Products);

                                // Loops through the product option categories
                                foreach (OptionCategoryInfo category in 
categories)
                                {
                                        // Gets a list of enabled options in the 
product option category
                                        ObjectQuery<SKUInfo> options = 
SKUInfoProvider.GetSKUOptionsForProduct(skuId, category.CategoryID, true);

                                        // Loops through the product options
                                        foreach (SKUInfo option in options)
                                        {
                                                // Adds the name of the product 
option into the indexed content for the product page
                                                // Spaces added as separators to 
ensure that typical search index analyzers can correctly tokenize the index 
content
                                                e.Content += " " + option.SKUName 
+ " ";
                                        }
                                }
                        }
                }
        }
}
```

4. Save the **CustomSmartSearchModule.cs** file.
5. Sign in to the Kentico administration interface.
6. Open the **Smart search** application and **Rebuild** your page search indexes.

The search now returns product pages if the search text matches the name of one of the page's product options.