Kentico does not provide any built-in components for searching through Azure Search indexes and displaying results on your website's pages.

To allow visitors to search your website using Azure Search, you need to develop and maintain custom components. This way, you can create a search solution that best matches the implementation of your website and your search requirements. You have full control over the output of your search results and other interface elements, and the flexibility to use the wide range of Azure Search features.

Interact with your Azure Search indexes using the Azure Search .NET SDK (provided via the **Microsoft.Azure.Search** NuGet package). For typical search solutions, the development process consists of the following general steps:

- Create interface elements that collect input from users
- Prepare full-text search queries and apply filters (based on the user input)
- Search for matching documents in your index
- Process the response to display search results, faceted navigation, etc.

To get a deeper understanding of how search queries work in Azure Search, refer to the How full text search works in Azure Search article.

Example

The following example demonstrates how to develop a basic search interface that uses an Azure Search index. The example works with the data of coffee products from the *Dancing Goat* sample site, and provides the following:

- An input for search text
- Faceted navigation for filtering based on coffee farms and regions
- Search results containing the names and descriptions of matching coffee products

Start by creating an Azure Search index for the store content of the Dancing Goat sample site:

1. Open the **Smart search** application.
2. On the **Azure indexes** tab, click **New index**.
3. Fill in the index properties:
    - **Display name**: DG store
    - **Code name**: dg-store
    - **Index type**: Pages
    - **Service name**: *the name of your Azure Search service*
    - **Admin key**: *a valid admin key for your Azure Search service*
    - **Query key**: *a valid query key for your Azure Search service*

    To learn more about managing Azure Search services, refer to the Create an Azure Search service in the portal article.
4. Click **Save**.
5. Make sure the index is assigned to the *Dancing Goat* sample site on the **Sites** tab.
6. Switch to the **Cultures** tab, and add at least one culture to the index (for example *English - United States*).
7. Open the **Indexed content** tab and click **Add allowed content**.
8. Set the **Path** value to: /Store/%
9. Click **Save**.

Note: The index includes all product and page types under the website's */Store* section, even though the search interface in this example only works with coffee products. When creating your own search implementations, you can use a single broad index with multiple specialized search components that work with different types of data.

Configure the search field settings required for the scenario:

1. Open the **Modules** application in Kentico.
2. Edit the **E-commerce** module.
3. Select the **Classes** tab, edit the **SKU** class.
4. Open the **Search** tab and click **Customize**.
5. Enable the following search field options in the **Azure** section of the grid:
   - **SKUName** – Content, Retrievable, Searchable
   - **SKUDescription** – Content, Searchable
   - **SKUShortDescription** – Content, Retrievable, Searchable
   - **NodeAliasPath** – Searchable
6. Click **Save**.
7. Open the **Page types** application.
8. Edit the **Coffee (Dancing Goat)** page type.
9. Select the **Search fields** tab.
10. Enable the following search field options in the **Azure** section of the grid:
    - **CoffeeCountry** – Content, Searchable, Facetable, Filterable
    - **CoffeeFarm** – Content, Searchable, Facetable, Filterable
11. Click **Save**.
12. Open the **Smart search** application and **Rebuild** the *DG store* index (on the **Azure indexes** tab).

Next you need to create components that provide a search interface with faceted navigation, perform the search requests, and display search results.

In this example, all interface elements and functions are implemented within a single custom web part.

When developing your own search solutions, you can separate the logic into any number of components according to the specifics of your website and your search requirements.

1. Open your Kentico solution in Visual Studio.
2. Right-click the Kentico web project (CMS or CMSApp) and select **Add -> Reference**.
3. Open the **Assemblies -> Framework** tab in the **Reference manager** dialog, select the **System.Net.Http** assembly and click **OK**.
4. Expand the *CMSWebParts/SmartSearch* folder and add a new **Web User Control** named *AzureSearch.ascx*.
5. Copy the following markup into the *AzureSearch.ascx* file (below the **@ Control** directive):

```
<div class="product-page row">
  <div class="col-md-4 col-lg-3 product-filter">
    <asp:TextBox runat="server" ID="searchBox" CssClass="search-box" />

    <h4>Farms</h4>
    <asp:CheckBoxList runat="server" ID="filterFarm" Visible="true" CssClass="
ContentCheckBoxList" AutoPostBack="True" />

    <h4>Coffee region</h4>
    <asp:CheckBoxList runat="server" ID="filterCountry" Visible="true" CssClass="
ContentCheckBoxList" AutoPostBack="True" />
  </div>

  <div class="col-md-8 col-lg-9 product-list">
    <asp:Label runat="server" ID="lblSearchResults" Visible="true" />
  </div>
</div>
```

Note: The CSS classes used in the example work with the default stylesheets of the Dancing Goat sample site.

6. Copy the following code into the control's code behind (*AzureSearch.ascx.cs*):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web.UI.WebControls;

using CMS.Base.Web.UI;
using CMS.Search;
using CMS.Search.Azure;
using CMS.PortalEngine.Web.UI;

using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;

public partial class CMSWebParts_SmartSearch_AzureSearch : CMSAbstractWebPart
{
    // The SearchIndexClient instance used to interact with the specified Azure
Search index
    private ISearchIndexClient searchIndexClient = InitializeIndex("dg-store");

    // The fields used for faceted navigation
    private const string FACET_COFFEE_COUNTRY = "coffeecountry";
    private const string FACET_COFFEE_FARM = "coffeefarm";
    public readonly List<string> Facets = new List<string>
    {
        FACET_COFFEE_COUNTRY,
        FACET_COFFEE_FARM
    };

    /// <summary>
    /// Returns an initialized SearchServiceClient instance based on the
specified index code name.
    /// </summary>
    private static ISearchIndexClient InitializeIndex(string indexName)
    {
        // Converts the Kentico index code name to a valid Azure Search index
name (if necessary)
        indexName = NamingHelper.GetValidIndexName(indexName);

        SearchIndexInfo index = SearchIndexInfoProvider.GetSearchIndexInfo
(indexName);
        SearchServiceClient client = new SearchServiceClient(index.
IndexSearchServiceName, new SearchCredentials(index.IndexQueryKey));

        return client.Indexes.GetClient(indexName);
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        // Prepares a list of filter queries for the search request
        var filterQueries= new List<string>()
        {
            // Filters the search results to display only pages (products) of the
'dancinggoat.coffee' type
            "classname eq 'dancinggoat.coffee'"
        };

        // Adds filter queries based on the selected options in the faceted
navigation (coffee farm and region)
        var selectedCountries = filterCountry.GetSelectedItems();
```

```
        var selectedFarms = filterFarm.GetSelectedItems();

        if (selectedCountries.Any())
        {
            filterQueries.Add(GetFilterQuery(selectedCountries,
FACET_COFFEE_COUNTRY));
        }
        if (selectedFarms.Any())
        {
            filterQueries.Add(GetFilterQuery(selectedFarms, FACET_COFFEE_FARM));
        }

        // Prepares the search parameters
        var searchParams = new Microsoft.Azure.Search.Models.SearchParameters
        {
            Facets = Facets,
            Filter = String.Join(" and ", filterQueries),
            HighlightPreTag = "<strong>",
            HighlightPostTag = "</strong>",
            // All fields used for text highlighting must be configured as
'searchable'
            HighlightFields = new List<string>
            {
                FACET_COFFEE_COUNTRY,
                FACET_COFFEE_FARM,
                "nodealiaspath",
                "skudescription",
                "skuname"
            }
        };

        // Gets the search text from the input
        string searchString = searchBox.Text;

        // Performs the search request for the specified Azure Search index and
parameters
        DocumentSearchResult result = searchIndexClient.Documents.Search
(searchString, searchParams);

        // Fills or updates the faceted navigation options
        if (result.Facets != null)
        {
            if (!IsPostBack)
            {
                foreach (var resultFacet in result.Facets)
                {
                    foreach (FacetResult value in resultFacet.Value)
                    {
                        AddItemToCheckBoxList(value, resultFacet.Key);
                    }
                }
            }
            else
            {
                foreach (ListItem item in filterCountry.Items)
                {
                    UpdateCountListItem(result.Facets[FACET_COFFEE_COUNTRY],
item);
                }
```

```
                foreach (ListItem item in filterFarm.Items)
                {
                    UpdateCountListItem(result.Facets[FACET_COFFEE_FARM], item);
                }
            }
        }

        // Displays the search results
        GenerateResultView(result);
    }

        /// <summary>
    /// Builds a filter query based on selected faceted navigation options.
    /// </summary>
    private string GetFilterQuery(IEnumerable<ListItem> selectedItems, string
column)
    {
        var queries = selectedItems.Select(item => $"{column} eq '{item.Value.
Replace("'", "''")}'");
        return $"({String.Join(" or ", queries)})";
    }

    /// <summary>
    /// Adds a retrieved FacetResult to the 'listItem' filtering options.
    /// </summary>
    private void AddItemToCheckBoxList(FacetResult facetResult, string
resultFacetKey)
    {
        var item = new ListItem($"{facetResult.Value} ({facetResult.Count})",
facetResult.Value.ToString());
        switch (resultFacetKey)
        {
            case FACET_COFFEE_COUNTRY:
                filterCountry.Items.Add(item);
                break;
            case FACET_COFFEE_FARM:
                filterFarm.Items.Add(item);
                break;
            default:
                break;
        }
    }

    /// <summary>
    /// Updates the counts of matching results for the 'listItem' filtering
options according to the retrieved facet data.
    /// </summary>
    private static void UpdateCountListItem(IEnumerable<FacetResult>
facetResults, ListItem listItem)
    {
        long? count = 0;
        foreach (var items in facetResults)
        {
            if (items.Value.Equals(listItem.Value))
            {
                count = items.Count;
                break;
            }
        }
```

```
            listItem.Text = $"{listItem.Value} ({count})";
        }

        /// <summary>
        /// Displays the search results filtered according to the selected parameters.
        /// </summary>
        private void GenerateResultView(DocumentSearchResult searchResult)
        {
            if (searchResult.Results.Count == 0)
            {
                lblSearchResults.Text = "No search results found.";
                return;
            }

            var resultText = new StringBuilder();

            foreach (SearchResult result in searchResult.Results)
            {
                string resultItemName = $"<div><h4>{result.Document["skuname"]}</h4>";
                string resultItemDescription = $"<p>{result.Document
["skushortdescription"]}</p>";

                // Displays the 'skushortdescription' field's value for result items
without highlights matching the search text
                if (result.Highlights == null)
                {
                    resultText.Append($"{resultItemName}{resultItemDescription}<
/div>");
                }
                // Displays highlights matching the search text from the
'HighlightFields' specified in the SearchParameters
                else
                {
                    string highlightValues = String.Join("<br />", result.Highlights?.
Values.Select(value => String.Join(" || ", value)));
                    resultText.Append($"{resultItemName}{highlightValues}</div>");
                }
            }

            lblSearchResults.Text = resultText.ToString();
        }
    }
```

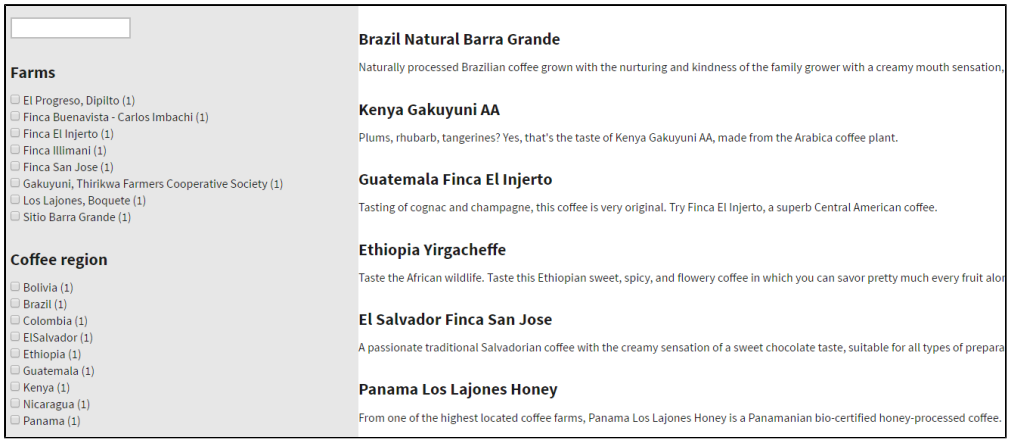
7. Save the new files (if you have a web application installation, build the *CMSApp* project).

Continue by registering the new control as a custom web part:

1. Open the **Web parts** application in Kentico.
2. Select the **Full-text search -> Smart search** category.
3. Click **New web part** and fill in the following values:
    - **Web part**: Create a new
    - **Display name**: Azure Search
    - **Code files**: Use existing file
    - **File path**: SmartSearch/AzureSearch.ascx
4. Click **Save**.

Note: The example only uses a basic web part without any properties. In your own implementations, you can provide configuration options by defining web part properties (for example for the search index name).

You can now place the custom *Azure Search* web part onto a page on the Dancing Goat sample site. The web part provides a search interface that works with the content of the defined Azure Search index.



Logging internal search activities and web analytics

By default, Kentico on-line marketing features cannot track search actions that occur through custom Azure Search components. This includes logging of the **Internal search** activity for contacts and the **On-site search key words** web analytics.

If you wish to use the search tracking features together with Azure Search, you need to manually perform the required logging in the code of your custom search components.

```
using System;

using CMS.DocumentEngine;
using CMS.Localization;
using CMS.SiteProvider;
using CMS.WebAnalytics;

...

// Performs on-line marketing logging only if search text is specified
if (!String.IsNullOrWhiteSpace(searchString))
{
        // Logs the 'Internal search' activity for the current contact
        PagesActivityLogger activityLogger = new PagesActivityLogger();
        activityLogger.LogInternalSearch(searchString);

        // Logs a hit for the 'On-site search keywords' web analytics
        AnalyticsHelper.LogOnSiteSearchKeywords(SiteContext.CurrentSiteName,
DocumentContext.CurrentAliasPath, LocalizationContext.CurrentCulture.CultureCode,
searchString, 0, 1);
}
```

Tip: To ensure that the logging only occurs for valid search requests, add the code **after** you call the *ISearchIndexClient.Documents.Search* method.