To comply with the requirements of personal data regulations, such as the GDPR, you need to provide a way for administrators (data protection officers) to collect personal data stored within the system. This is necessary to resolve personal data queries from data subjects, and requests to transfer personal data to another system or application.

Kentico does not provide any personal data collection functionality by default. This requires exact knowledge of how your website gathers, processes and stores personal data. You need to implement the data collection based on the specifics of your website and the nature of the legal requirements that you wish to fulfill.

> See the Personal data in Kentico reference to learn how the system gathers, stores and uses personal data by default. The information may be helpful when planning the collection functionality for your website.

Use the following process to develop the personal data collection:

> The interfaces, registers and other related classes that you need to implement personal data collection are available in the **CMS.DataProtection** namespace of the Kentico API.

1. Open your Kentico solution in Visual Studio.
2. Create custom classes that implement collector interfaces:
   - Identity collectors – map real-world identifiers, such as email addresses or names, to corresponding Kentico objects that represent data subjects.
   - Data collectors – process the objects added by identity collectors, retrieve related personal data, and format the results into a string containing either human-readable text or machine-readable data (such as XML).

     > **Class location**
     >
     > We recommend adding custom classes as part of a new assembly (*Class library* project) in your Kentico solution. You need to add the appropriate references to both the assembly and the main Kentico web project.
3. Add a custom module class and register your collector implementations within the module's **OnInit** method:
   - You can register any number of collectors.
   - To register identity collectors, call the **IdentityCollectorRegister.Instance.Add** method. The registration order is significant – the identities added by a collector can be accessed by identity collectors that are registered after.
   - To register data collectors, call the **PersonalDataCollectorRegister.Instance.Add** method.

The registered collectors allow users to search for personal data in the **Data protection** application. To give users the option to delete the collected data from the system (or specific parts of it), you also need to implement erasure functionality – see Implementing personal data erasure.

Identity collectors

When searching for personal data in the **Data protection** application, users submit real-world identifiers of data subjects (email addresses, names, etc.). To collect personal data, you first need to create *Identity collectors* that convert these identifiers into Kentico objects representing matching data subjects, such as users (*UserInfo*), contacts (*ContactInfo*) or customers (*CustomerInfo*).

Identity collectors are classes that implement the **IIdentityCollector** interface. Every implementation must contain the **Collect** method, which processes the following parameters:

- **IDictionary<string, object>** – a dictionary holding the submitted identifiers and other filtering parameters. The default identifier input only provides an email address value, which is available under the **"email"** key.
- **List<BaseInfo>** – a list of Kentico objects representing data subjects. Contains all objects added by previously registered *IIdentityCollector* implementations.

To create your own *IIdentityCollector* implementations:

1. Use the ObjectQuery API to load Kentico objects (instances of *Info* classes) that match the submitted identifier values (available in the *Collect* method's first parameter).

2. Add the retrieved objects to the list of identity objects (the *Collect* method's second parameter).

See the Example - Creating contact data collectors section to view a code example.

Customizing the identifier inputs

By default, the **Data protection** application allows users to search for personal data based on an email address value. If you wish to set up personal data collection according to other types of identifiers or add filtering options, you need to create and register a custom control:

1. Open your Kentico solution in Visual Studio.
2. Create a new Web User Control (.ascx file) in the Kentico web project (*CMSApp* or *CMS*).
3. Add components to the control's markup that allow users to input the required identifier values and/or set filtering parameters.
4. Switch to the code behind and make the control class inherit from **DataSubjectIdentifiersFilterControl** (available in the **CMS.UIControls** namespace).
5. Override the following methods:
   - **GetFilter** – return an *IDictionary<string, object>* collection containing all identifier values and filtering parameters that users input through the control.

     Example

     ```
     public override IDictionary<string, object> GetFilter(IDictionary<string,
     object> filter)
     {
         filter.Add("email", txtEmail.Text);

         return filter;
     }
     ```

   - **IsValid** – return a *bool* value that indicates whether the control's input is valid. Call the *AddError* method to display messages to users in cases where the input is not valid.

     Example

     ```
     public override bool IsValid()
     {
         if (!CMS.Helpers.ValidationHelper.IsEmail(txtEmail.Text))
         {
             AddError("Please enter a valid email address.");

             return false;
         }

         return true;
     }
     ```

6. Edit the module class where you register your collector implementations, and register your identifier input control within the module's **OnInit** method:
   - Call the **DataProtectionControlsRegister.Instance.RegisterDataSubjectIdentifiersFilterControl** method (available in the **CMS.UIControls** namespace).
   - Specify the path of the user control file in the method's parameter, for example: *~/CMSModules /CustomDataProtection/DataSubjectIdentifiers.ascx*

The system now uses your custom control to display the identifier inputs in the **Data protection** application. The keys and values added via the control's **GetFilter** method are available as the second parameter of the **Collect** method in your **IIdentityCollector** implementations.

Data collectors

After you implement Identity collectors, you need to create *Data collectors*. These collectors retrieve personal data related to the identity objects provided by the registered identity collectors, and process the results into a suitable format. We recommend creating separate data collectors for different object types, depending on the types of personal data that you process on your website.

Data collectors are classes that implement the **IPersonalDataCollector** interface. Every implementation must contain the **Collect** method.

- The method's **IEnumerable<BaseInfo>** parameter provides the identity objects added by your *IIdentityCollector* implementations. You can convert the *BaseInfo* objects to specific types, such as *UserInfo, ContactInfo,* etc.
- The method's second parameter is a string that specifies the requested output format. By default, the following fixed values are used for the output format:
  - **machine** – when searching for data on the *Data portability* tab of the *Data protection* application. The value is available in the API under the *PersonalDataFormat.MACHINE_READABLE* constant. By default, the system assumes that machine-readable data is in XML format and adds a *<PersonalData>* root tag around the final output. To return data in another format, such as JSON, you need to Customize the data output.
  - **human** – when searching for data on the *Right to access* and *Right to be forgotten* tabs in the *Data protection* application. The value is available in the API under the *PersonalDataFormat.HUMAN_READABLE* constant.

To create your own *IPersonalDataCollector* implementations:

1. Collect all required personal data related to the provided identity objects.
2. Format the data into a string.

   We recommend creating a dedicated writer class for every required output format. You can reuse the API of general writer classes to build the personal data string for different types of objects.

   You may also need to define transformation functions that convert internally stored values into more understandable equivalents. For example, the system uses integer values for some fields that have a fixed set of possible options (e.g. the user gender field).

3. Return a **PersonalDataCollectorResult** object in the **Collect** method, with the resulting personal data string assigned into the **Text** property.

See the Example - Creating contact data collectors section to view a code example.

The text provided by registered data collectors is displayed when searching for personal data in the **Data protection** application.

Customizing the data output

By default, the overall data returned by the collection process is composed according to the registration order of your data collectors (starting from the first to the last). Additionally, the system assumes that machine-readable data is in XML format and adds a *<PersonalData>* root tag around the final output of all data collectors.

If you wish to return machine-readable data in another format, such as JSON, or otherwise adjust the data composition process, use the following customization approach:

1. Open your Kentico solution in Visual Studio.
2. Create a custom class that inherits from **PersonalDataHelper** (available in the **CMS.DataProtection** namespace).
3. Override the **JoinPersonalDataInternal** method of the helper class:

   - The method's **IEnumerable<string>** parameter provides a collection of personal data strings added by your registered *IPersonalDataCollector* implementations.
   - The method's second parameter is a string that specifies the requested output format.
   - Compose the final data output according to your custom requirements and return it as a string.
4. Register your helper class implementation using the **RegisterCustomHelper** assembly attribute.

Example

```
using System;
using System.Linq;
using System.Text;
using System.Collections.Generic;

using CMS;
using CMS.DataProtection;

// Registers the CustomPersonalDataHelper class to replace the default
PersonalDataHelper
[assembly: RegisterCustomHelper(typeof(CustomPersonalDataHelper))]

public class CustomPersonalDataHelper : PersonalDataHelper
{
    // Customizes the output of personal data in machine-readable format
    // Replaces the default <PersonalData> XML root tag with curly brackets for JSON
data
    protected override string JoinPersonalDataInternal(IEnumerable<string>
personalData, string outputFormat)
    {
        // Performs custom handling for personal data in machine-readable format
        if (outputFormat.Equals(PersonalDataFormat.MACHINE_READABLE, StringComparison.
OrdinalIgnoreCase))
        {
            string indentation = "  ";
            string indentedNewLine = Environment.NewLine + indentation;

            var resultBuilder = new StringBuilder();

            // Adds the opening bracket to the result
            resultBuilder.AppendLine("{");
            resultBuilder.Append(indentation);

            // Updates all new line characters in the returned personal data to
include the added indentation
            var modifiedPersonalData = personalData.Select(data => data.Replace
(Environment.NewLine, indentedNewLine));

            // Adds the data provided by all registered personal data collectors
            resultBuilder.AppendLine(String.Join(indentedNewLine,
modifiedPersonalData));

            // Adds the closing bracket to the result
            resultBuilder.Append("}");

            return resultBuilder.ToString();
        }

        // Runs the default method for human-readable or undefined output formats
        return base.JoinPersonalDataInternal(personalData, outputFormat);
    }
}
```

The system now uses your custom data joining implementation when displaying results in the **Data protection** application.

Example – Creating contact data collectors

The following example demonstrates how to implement personal data collection for basic contact data. The sample collectors work with the default email address identifier that users can submit in the **Data protection** application, and produce personal data output in plain text or XML format.

> This example only shows the basic implementation concepts and collects a very limited set of personal data. You can find a more extensive code example in your Kentico program files directory (by default *C:\Program Files\Kentico\<version>*) under the **CodeSamples\CustomizationSamples\DataProtection** subfolder.
>
> Keep in mind that you always need to adjust your own implementation based on the personal data processing used on your website and the legal requirements that you wish to fulfill.

Adding a custom class library

Start by preparing a separate project for custom classes in your Kentico solution:

1. Open your Kentico solution in Visual Studio.
2. Create a new *Class Library* project in the Kentico solution (or reuse an existing custom project).
3. Add references to the required Kentico libraries (DLLs) for the new project:
    a. Right-click the project and select **Add -> Reference**.
    b. Select the **Browse** tab of the **Reference manager** dialog, click **Browse** and navigate to the ***Lib*** folder of your Kentico web project.
    c. Add references to the following libraries (and any others that you may need in your custom code):
        - **CMS.Base.dll**
        - **CMS.ContactManagement.dll**
        - **CMS.Core.dll**
        - **CMS.DataEngine.dll**
        - **CMS.DataProtection.dll**
        - **CMS.Helpers.dll**
4. Reference the custom project from the Kentico web project *(CMSApp* or *CMS)*.
5. Edit the custom project's **AssemblyInfo.cs** file (in the *Properties* folder).
6. Add the **AssemblyDiscoverable** assembly attribute:

```
using CMS;

[assembly:AssemblyDiscoverable]
```

You can now add your collector implementations and other related classes under the custom project.

Creating the identity collector

To create an identity collector for contacts, add a new class implementing the **IIdentityCollector** interface under the custom project:

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

using CMS.DataEngine;
using CMS.DataProtection;
using CMS.ContactManagement;

public class ContactIdentityCollector : IIdentityCollector
{
    public void Collect(IDictionary<string, object> dataSubjectFilter, List<BaseInfo>
identities)
    {
        // Does nothing if the identifier inputs do not contain the "email" key or if
its value is empty
        if (!dataSubjectFilter.ContainsKey("email"))
        {
            return;
        }
        string email = dataSubjectFilter["email"] as string;
        if (String.IsNullOrWhiteSpace(email))
        {
            return;
        }

        // Finds contacts with a matching email address
        List<ContactInfo> contacts = ContactInfoProvider.GetContacts()
                                            .WhereEquals(nameof(ContactInfo.
ContactEmail), email)
                                            .ToList();

        // Adds the matching contact objects to the list of collected identities
        identities.AddRange(contacts);
    }
}
```

The collector loads all contact objects (*ContactInfo*) that match the submitted email address identifier, and adds them to the list of collected identities.

Implementing writer classes

Continue by creating writer classes that convert Kentico objects into the required output formats (plain text and XML in this example).

For plain text, add the following writer class under the custom project:

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

using CMS.DataEngine;

public class TextPersonalDataWriter
{
    private readonly StringBuilder stringBuilder;
    private int indentationLevel;

    public TextPersonalDataWriter()
```

```
    {
        stringBuilder = new StringBuilder();
        indentationLevel = 0;
    }

    // Writes horizontal tabs based on the current indentation level
    private void Indent()
    {
        stringBuilder.Append('\t', indentationLevel);
    }

    // Writes text representing a new section of data, and increases the indentation
level
    public void WriteStartSection(string sectionName)
    {
        Indent();

        stringBuilder.AppendLine(sectionName + ": ");
        indentationLevel++;
    }

    // Writes the specified columns of a Kentico object (BaseInfo) and their values
    public void WriteObject(BaseInfo baseInfo, List<Tuple<string, string>> columns)
    {
        foreach (var column in columns)
        {
            // Gets the name of the current column
            string columnName = column.Item1;
            // Gets a user-friendly name for the current column
            string columnDisplayName = column.Item2;

            // Filters out identifier columns from the human-readable text data
                        if (columnName.Equals(baseInfo.TypeInfo.IDColumn,
StringComparison.Ordinal) ||
                columnName.Equals(baseInfo.TypeInfo.GUIDColumn, StringComparison.
Ordinal))
            {
                continue;
            }

            // Gets the value of the current column for the given object
            object value = baseInfo.GetValue(columnName);

            if (value != null)
            {
                Indent();
                stringBuilder.AppendFormat("{0}: ", columnDisplayName);
                stringBuilder.Append(value);
                stringBuilder.AppendLine();
            }
        }
    }

    // "Closes" a text section by reducing the indentation level
    public void WriteEndSection()
    {
        indentationLevel--;
    }

    // Gets a string containing the writer's overall text
```

```
    public string GetResult()
    {
        return stringBuilder.ToString();
    }
}
```

For machine-readable data in XML format, add the following writer class under the custom project. The sample XML writer is an IDisposable implementation that uses the standard .NET XmlWriter class to create the required output.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;

using CMS.DataEngine;
using CMS.Helpers;

public class XmlPersonalDataWriter : IDisposable
{
    private readonly StringBuilder stringBuilder;
    private readonly XmlWriter xmlWriter;

    public XmlPersonalDataWriter()
    {
        stringBuilder = new StringBuilder();
        xmlWriter = XmlWriter.Create(stringBuilder, new XmlWriterSettings { Indent =
true, OmitXmlDeclaration = true });
    }

    // Writes an opening XML tag with a specified name
    public void WriteStartSection(string sectionName)
    {
        // Replaces period characters in object names with underscores
        sectionName = sectionName.Replace('.', '_');

        xmlWriter.WriteStartElement(sectionName);
    }

    // Writes XML tags representing the specified columns of a Kentico object
(BaseInfo) and their values
    public void WriteObject(BaseInfo baseInfo, List<string> columns)
    {
        foreach (string column in columns)
        {
            object value = baseInfo.GetValue(column);

            if (value != null)
            {
                xmlWriter.WriteStartElement(column);
                xmlWriter.WriteValue(XmlHelper.ConvertToString(value));
                xmlWriter.WriteEndElement();
            }
        }
    }

    // Writes a closing XML tag for the most recent open tag
    public void WriteEndSection()
    {
        xmlWriter.WriteEndElement();
```

```
    }

    // Gets a string containing the writer's overall XML data
    public string GetResult()
    {
        xmlWriter.Flush();

        return stringBuilder.ToString();
    }

    // Releases all resources used by the current XmlPersonalDataWriter instance.
    public void Dispose()
    {
        xmlWriter.Dispose();
    }
}
```

Creating the data collector

To create a data collector for contacts, add a class implementing the **IPersonalDataCollector** interface under the custom project:

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

using CMS.ContactManagement;
using CMS.DataEngine;
using CMS.DataProtection;

public class ContactDataCollector : IPersonalDataCollector
{
    // Prepares a list of contact columns to be included in the personal data
    // Every Tuple contains a column name, and user-friendly description of its content
    private readonly List<Tuple<string, string>> contactColumns = new
List<Tuple<string, string>> {
        Tuple.Create("ContactFirstName", "First name"),
        Tuple.Create("ContactMiddleName", "Middle name"),
        Tuple.Create("ContactLastName", "Last name"),
        Tuple.Create("ContactJobTitle", "Job title"),
        Tuple.Create("ContactAddress1", "Address"),
        Tuple.Create("ContactCity", "City"),
        Tuple.Create("ContactZIP", "ZIP"),
        Tuple.Create("ContactMobilePhone", "Mobile phone"),
        Tuple.Create("ContactBusinessPhone", "Business phone"),
        Tuple.Create("ContactEmail", "Email"),
        Tuple.Create("ContactBirthday", "Birthday"),
        Tuple.Create("ContactGender", "Gender"),
        Tuple.Create("ContactNotes", "Notes"),
        Tuple.Create("ContactGUID", "GUID"),
        Tuple.Create("ContactLastModified", "Last modified"),
        Tuple.Create("ContactCreated", "Created"),
        Tuple.Create("ContactCampaign", "Campaign"),
        Tuple.Create("ContactCompanyName", "Company name")
    };

    public PersonalDataCollectorResult Collect(IEnumerable<BaseInfo> identities,
string outputFormat)
    {
```

```
        // Gets a list of all contact objects added by registered IIdentityCollector
implementations
        List<ContactInfo> contacts = identities.OfType<ContactInfo>().ToList();

        // Uses a writer class to create the personal data, in either XML format or as
human-readable text
        string contactData = null;
        if (contacts.Any())
        {
            switch (outputFormat.ToLowerInvariant())
            {
                case PersonalDataFormat.MACHINE_READABLE:
                    contactData = GetXmlContactData(contacts);
                    break;
                case PersonalDataFormat.HUMAN_READABLE:
                default:
                    contactData = GetTextContactData(contacts);
                    break;
            }
        }

        return new PersonalDataCollectorResult
        {
            Text = contactData
        };
    }

    private string GetXmlContactData(List<ContactInfo> contacts)
    {
        using (var writer = new XmlPersonalDataWriter())
        {
            // Wraps the contact data into a <OnlineMarketingData> tag
            writer.WriteStartSection("OnlineMarketingData");

            foreach (ContactInfo contact in contacts)
            {
                // Writes a tag representing a contact object
                writer.WriteStartSection(ContactInfo.OBJECT_TYPE);
                // Writes tags for the contact's personal data columns and their values
                writer.WriteObject(contact, contactColumns.Select(t => t.Item1).
ToList());
                // Closes the contact object tag
                writer.WriteEndSection();
            }

            // Closes the <OnlineMarketingData> tag
            writer.WriteEndSection();

            return writer.GetResult();
        }
    }

    private string GetTextContactData(List<ContactInfo> contacts)
    {
        var writer = new TextPersonalDataWriter();

        writer.WriteStartSection("On-line marketing data");

        foreach (ContactInfo contact in contacts)
        {
```

```
            writer.WriteStartSection("Contact");
            // Writes user-friendly descriptions of the contact's personal data
columns and their values
            writer.WriteObject(contact, contactColumns);
            writer.WriteEndSection();
        }

        return writer.GetResult();
    }
}
```

The sample data collector processes the contact objects provided by the identity collector, and then uses the writer classes to create personal data text in the requested output format.

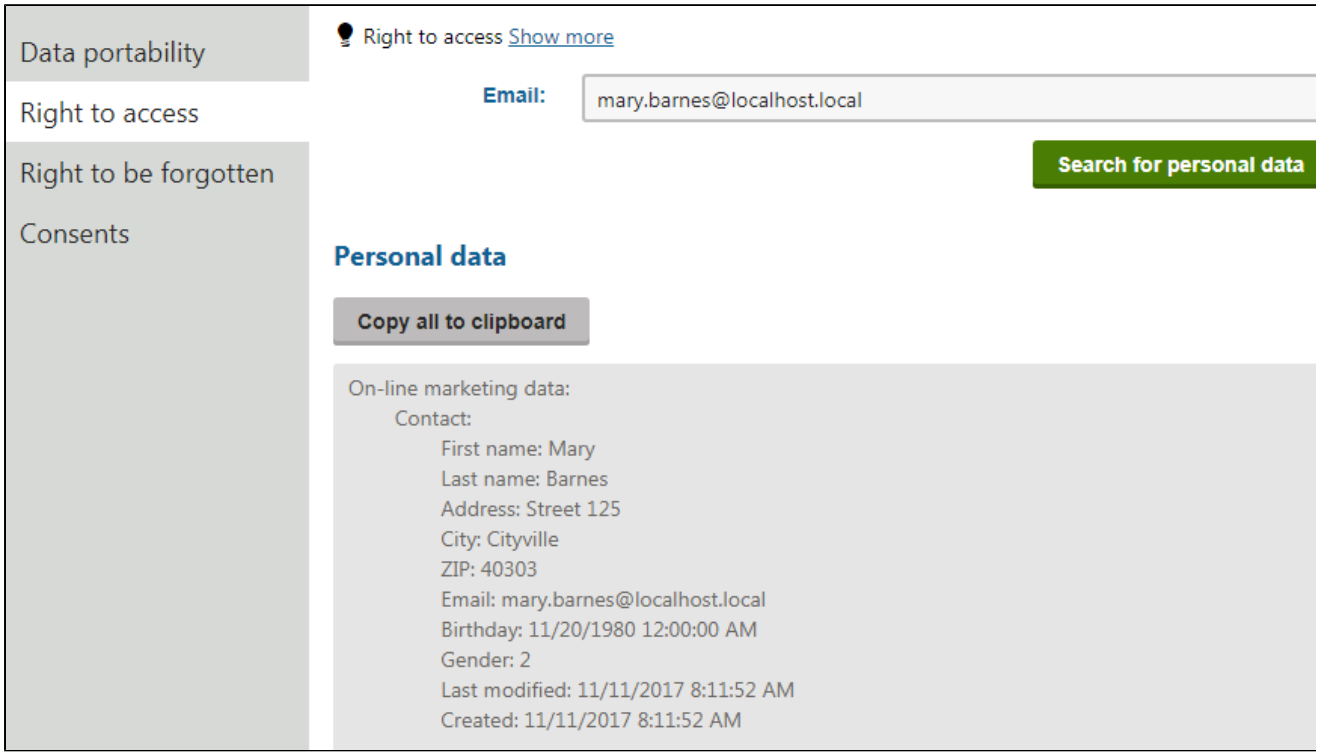
Registering the collectors

To register the identity and data collectors, add a module class under the custom project, and run the required initialization code:

```
using CMS;
using CMS.DataEngine;
using CMS.DataProtection;

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

internal class CustomDataProtectionModule : Module
{
    // Module class constructor, the system registers the module under the name
"CustomDataProtection"
    public CustomDataProtectionModule()
        : base("CustomDataProtection")
    {
    }

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

        // Adds the ContactIdentityCollector to the collection of registered identity
collectors
        IdentityCollectorRegister.Instance.Add(new ContactIdentityCollector());

        // Adds the ContactDataCollector to the collection of registered personal data
collectors
        PersonalDataCollectorRegister.Instance.Add(new ContactDataCollector());
    }
}
```

Save all changes and **Build** the custom project.

You can now search for the email addresses of contacts on the **Data portability** and **Right to access** tabs in the **Data protection** application. If matching contacts exist in the system, the registered collectors return their data in XML or plain text format.


Data portability
Right to access
Right to be forgotten
Consents
Right to access Show more
Email:
mary.barnes@localhost.local
Search for personal data
Personal data
Copy all to clipboard
On-line marketing data:
Contact:
First name: Mary
Last name: Barnes
Address: Street 125
City: Cityville
ZIP: 40303
Email: mary.barnes@localhost.local
Birthday: 11/20/1980 12:00:00 AM
Gender: 2
Last modified: 11/11/2017 8:11:52 AM
Created: 11/11/2017 8:11:52 AM