This page describes how to use code generators to prepare wrapper classes for working with specific object types (Page types, Custom tables and Forms).

Code generators are available for the following objects, in their respective applications:

- Page types – allow you to generate a class with the page type's fields as properties and a provider to retrieve the individual pages.
  - Generate code for individual page types on the **Code** tab of the specific types.
  - Generate code for all site page types on the **Code** tab of the **Page types** application.

  > **Product page types**
  >
  > Code generators also work for product page types. However, unlike page types, which inherit from the *TreeNode* class, product page types inherit from the *SKUTreeNode* class.

- Custom tables – allow you to generate a class with the custom table's fields as properties.
  - Generate code for individual custom tables on the **Code** tab of the specific custom tables.

- Forms – allow you to generate class with the form's fields as properties.
  - Generate code for individual forms on the **Code** tab of the specific forms.

Code generators allow you to create classes for working with specific objects types (Page types, Custom tables, Forms) in the API.

Working with page type code generators

The page type code generators allow you to generate both properties and providers for your custom page types. The generated properties represent the page type fields. You can use the providers to work with published or latest versions of a specific page type.

Generating code for page types

1. Choose whether you want to generate code for a single or all site page types.
   - For a single page type:
     a. In the **Page types** application, edit (✎) a page type.
     b. Navigate to the **Code** tab.
   - For all site page types:
     a. In the **Page types** application, navigate to the **Code** tab.
     b. Select a **Site**.
     c. (Optional) If you want to generate code for container page types, enable the **Include container page types** check box.
2. (Optional) Select a different default folder to place the code files in.
3. Click **Save code**.

   > **Note**: The generated class names are not guaranteed to be unique. The system can in certain cases (usually when also generating code for container page types) generate multiple classes with the same name. In that case, you need to rename the classes manually.

4. Copy the generated code files to your MVC application project. By default, the generated code files are stored in *~/App_Code/CMSClasses* folder of your Kentico project.
5. If you want to use the code files in a separate library or external application, allow the system to detect the generated classes in a custom assembly.
   Add the 'AssemblyDiscoverable' assembly attribute to the library/application project's *AssemblyInfo.cs* file:

   ```
   using CMS;
   ...
   [assembly: AssemblyDiscoverable]
   ```

6. Use the generated properties in your code.
7. Build your MVC application solution.

You can find examples of working with the generated classes on Retrieving content on MVC sites, where you can learn how to retrieve pages and how to access their individual fields. And how to retrieve and filter user form submissions, custom table data, and objects associated with custom modules.

Customizing generated classes

In most scenarios, we do **not recommended directly modifying the generated classes**.

The classes are generated as partial classes. This means that you can extend them in a separate code file. Using this approach, you avoid the need to merge custom changes made to the generated code every time the class is generated again after the object's properties are changed.

Example of customizing generated classes

Articles.generated.cs - generated class

```
namespace CMS.DocumentEngine.Types.Custom
{
        public partial class Article : TreeNode
        {
            ...

                /// <summary>
                /// Article name.
            /// </summary>
                [DatabaseField]
            public string ArticleTitle
                {
                    get
                {
                            return ValidationHelper.GetString(GetValue("ArticleTitle"),
"");
                    }
                    set
                {
                          SetValue("ArticleTitle", value);
                    }
            }
                ...
        }
}
```

Articles.cs - extending the generated class

```
namespace CMS.DocumentEngine.Types.Custom
{
        /// <summary>
        /// Custom article page type class.
        /// </summary>
        public partial class Article : TreeNode
        {
            /// <summary>
            /// Loads sample data into this object.
            /// </summary>
            public void LoadSampleData()
            {
                ArticleTitle = "Sample article title";
                ArticleSummary = "Sample article summary";
                ArticleText = "Sample article text";
            }
        }
}
```

Using nullable properties for non-required fields

Page type, custom table or form fields whose **Required** setting is disabled may have *null* values in the database. However, the code generators create properties with non-nullable types for such fields. Instead, a default value is typically loaded in cases where the value is null, for example 0 for numeric types.

If you wish to have properties with Nullable types in the code representing your page, custom table or form items, you need to manually adjust the code after generating.

1. Change the type of the given properties to a nullable one (using the **T?** syntax) in the **Properties** region of the generated class.
2. Adjust the get property accessor for each property:
    - Remove the default **ValidationHelper.Get** call (*GetInteger*, *GetBoolean*, etc.), which replaces null with a default value.
    - Cast the result of the **GetValue** method to the appropriate nullable type.
3. Change the type of the matching properties within the nested ***Fields** class.

Example - Nullable integer property for a non-required field

```
// Example of a nullable integer field
[DatabaseField]
public int? IntegerField
{
        get
        {
                // Adjusted loading of the value (without replacement of null with a
default value)
                return GetValue("IntegerField") as int?;
        }
        set
        {
                SetValue("IntegerField", value);
        }
}

...

[RegisterAllProperties]
public partial class ItemFields : AbstractHierarchicalObject<ItemFields>
{

        ...

        // Adjusted nullable property within the nested *Fields class
        public int? IntegerField
        {
                get
                {
                        return mInstance.IntegerField;
                }
                set
                {
                        mInstance.IntegerField = value;
                }
        }

        ...
```

Note: Even though we generally recommend using partial classes to extend generated code in separate files, this cannot be done for modifications of property types. As a result, your code adjustments may be overwritten if you generate the code of the given object again at a later time. To preserve your changes, you need to manually update the re-generated code files.