The Kentico [continuous integration solution](#) allows you to serialize the data of objects from the database into XML files and then use a source control system to synchronize the files between different instances. To enable continuous integration for the data of custom module classes, you need to set the appropriate [type information](#) when developing your classes. For more information about the basics of module and class development, see the example in [Creating custom modules](#).

To configure continuous integration support for a custom module class:

1. Open the Kentico solution where you are developing the custom module (in Visual Studio).
2. Edit the *Info* class representing your module class.

> ⚠️ **Important**: To use continuous integration, the code of your custom class must be located in a separate assembly. See the [Adding the class code into a separate assembly](#) section for more information and instructions.

3. Set the **ContinuousIntegrationSettings** property in the [initializer](#) of the TYPEINFO object. Use a nested initializer to configure the properties of the *ContinuousIntegrationSettings* class:
   - Set the **Enabled** property to *true.*
   - (Optional) Set the other [ContinuousIntegrationSettings properties](#) according to the requirements of your class's data.

```
// Enables continuous integration for the object type
ContinuousIntegrationSettings =
{
        Enabled = true
},
```

4. (Optional) Set the **SerializationSettings** property of the TYPEINFO object. Use a nested initializer to configure the [SerializationSettings properties](#).
5. Save the changes.

If continuous integration is enabled, the system now tracks all create, update and delete operations for objects of the given custom class, and serializes the data into XML files in the *CIRepository* folder.

To transfer all existing data of the class into your continuous integration repository, run complete serialization for all objects:

1. Disable running of [scheduled tasks](#) (using the **Settings -> System -> Scheduled tasks enabled** setting).
2. Open the **Continuous integration** application in the Kentico administration interface.
3. Click **Serialize all objects**.
4. Wait until the serialization process finishes and then re-enable scheduled tasks.

You can also use your custom object type's name (the value of the *OBJECT_TYPE* constant) when filtering objects for continuous integration within the **repository.config** file. See [Excluding objects from continuous integration](#) to learn more.

Adding the class code into a separate assembly

When [restoring data](#) from the continuous integration repository to the database, the process runs outside of the Kentico web application (as a console application by default). Module classes defined directly within the Kentico web project (in the *App_Code* folder on web site projects or under the *CMSApp* project on web application installations) cannot be detected by external utilities that are not web applications and are ignored by the continuous integration restore process.

To correctly use continuous integration with custom module classes, your class code files *(Info* and *InfoProvider* classes) must be defined within a **separate assembly**. The assembly must also be configured to allow the system to detect classes using the **AssemblyDiscoverable** assembly attribute.

If your module classes are located directly in the Kentico web project, use the following process to move the code to a separate assembly:

> **ℹ Modules with installation package support**
>
> The following steps describe how to create a standard code library (assembly) project for the module code.
>
> If you are developing a module that you plan to deploy using installation packages, create a *web application* project for the module according to the steps in Creating installation packages for modules.

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

```
using CMS;

[assembly:AssemblyDiscoverable]
```

6. Move all of the module's code files *(Info* and *InfoProvider* classes, etc.) into the new project.
7. Save all changes.
8. Build the Kentico solution (or only the module project on *web site* installations).

Placing the class code into a discoverable custom assembly allows the continuous integration solution to work with the class when restoring data from the *CIRepository* folder.

Deleting classes that have continuous integration enabled

Use the following process if you ever need to delete a custom class with existing data and continuous integration enabled:

1. First delete the given class's code files and rebuild the module project in Visual Studio (or remove the entire module project if it is no longer needed).
2. Delete the class in the **Modules** application in the Kentico administration interface.
3. Manually delete the folder representing the class's data from your *CIRepository* folder (the system automatically removes the data from the database, but cannot clear the continuous integration data).

References

> **ℹ** The reference lists the **ContinuousIntegrationSettings** and **SerializationSettings** properties that are intended for public use. The classes also contain other members that are used internally or handled automatically by the system. We do not recommend working with any of the undocumented members.

ContinuousIntegrationSettings

The **ContinuousIntegrationSettings** property of *ObjectTypeInfo* determines how classes work with the continuous integration solution.

Assign an instance of the *ContinuousIntegrationSettings* class and set its properties:

Property	Type	Description
Enabled	bool	Indicates whether the class is included in the system's continuous integration solution (when serializing all objects to the file system and tracking object changes). Child classes are included only if the property is also enabled for the parent class.
DependencyColumns	ICollection<string>	Defined as an *ICollection* of field names. When the values of the specified fields are changed for an object, the continuous integration solution reserializes the given object and all of its child or otherwise dependent objects. Only needs to be set for classes that use non-standard fields to determine the identity of objects. The object type's code name, GUID, parent ID, site ID and group ID fields are included by default.
ObjectFileNameFields	ICollection<string>	Overrides the names of the XML files containing the serialized data of the class's objects within the *CIRepository* folder. By default, the XML files are named according to the **CodeNameColumn** value of the corresponding object. For object types without a code name field, the **GuidColumn** value is used instead. The property is defined as an *ICollection* of field names that the system combines into the names of the continuous integration XML files. Underscores (_) are used as separators between the values of the specified fields. **Note**: For continuous integration to work correctly, the combination of fields must be unique within the appropriate scope (within a given site for site-related objects, within the objects under a specific parent, etc.).
ObjectTypeFolderName	string	Sets the name of the folder that stores the XML files containing the serialized data of the class's objects within the *CIRepository* folder. If not specified, the object type name (value of the **ObjectType** type information property) is used by default.
SeparatedFields	IEnumerable<SeparatedField>	Identifies fields that the continuous integration solution stores into separate files when serializing objects to the file system. The separated file is created within the same folder as the main XML file holding the remaining serialized data of the object. To define the separated fields, assign an *IEnumerable* collection of **SeparatedField** objects into *SeparatedFields*. For each *SeparatedField* object, specify the name of the field as the constructor parameter. You can configure additional settings for the separated file through the properties of the *SeparatedField* object. **Note**: If the class has a field that stores binary data (specified in the **BinaryColumn** type information property), the system includes the field in *SeparatedFields* by default. Manually setting the *SeparatedFields* property overrides the default separation settings for the *BinaryColumn* field, but you can define a custom *SeparatedField* object for the given field.

Example

```
ContinuousIntegrationSettings =
{
        Enabled = true,

        // Adds a field that the continuous integration solution serializes into a
separate file (in addition to the BinaryColumn field)
        SeparatedFields = new List<SeparatedField>
        {
                new SeparatedField("SeparatedField")
                {
                        FileName = "CustomFileName",
                        FileExtension = ".txt"
                }
        }
}
```

SerializationSettings

The **SerializationSettings** property of *ObjectTypeInfo* determines how the system serializes class objects from the database into XML data. The serialization is used by the system's continuous integration solution.

Assign an instance of the *SerializationSettings* class and set its properties:

Property	Type	Description
ExcludedFieldNames	ICollection<string>	Allows you to limit which fields are included when serializing objects of the given class to XML. Defined as an *ICollection* of field names that you wish to exclude from the serialization (blacklist). By default, the system automatically excludes the class's **IDColumn**, **TimeStampColumn**, **Version GUIDColumn** and **BinaryColumn**.
Structured Fields	IEnumerable<IStructuredField>	Identifies fields that contain structured data, and cannot directly be serialized as inner text values of an XML element (for example fields with XML values). Ensures that the system correctly formats the value, for example as nested XML without a CDATA section. To work with custom data formats, you need to create an **IStructuredField** implementation that defines how the serialization occurs.

Example

```
SerializationSettings =
{
        // Adds two fields to the default collection of fields that are excluded from
the serialized data
        ExcludedFieldNames = { "ClassField", "ClassSecondField" },

        // Ensures that the system processes "ClassXmlField" as nested XML in the
serialized data
        StructuredFields = new IStructuredField[]
        {
                new StructuredField("ClassXmlField")
        }
}
```