

Adaptive Print Studio 1.0

Using Script Objects

Using Script Objects

Table of Contents

Chapter 1	Introduction	3
	1.3 Working with tables	9
	1.5 Page type production	11
Chapter 2	Using tasks	13
Chapter 3	Transformations	16

Chapter 1

Introduction

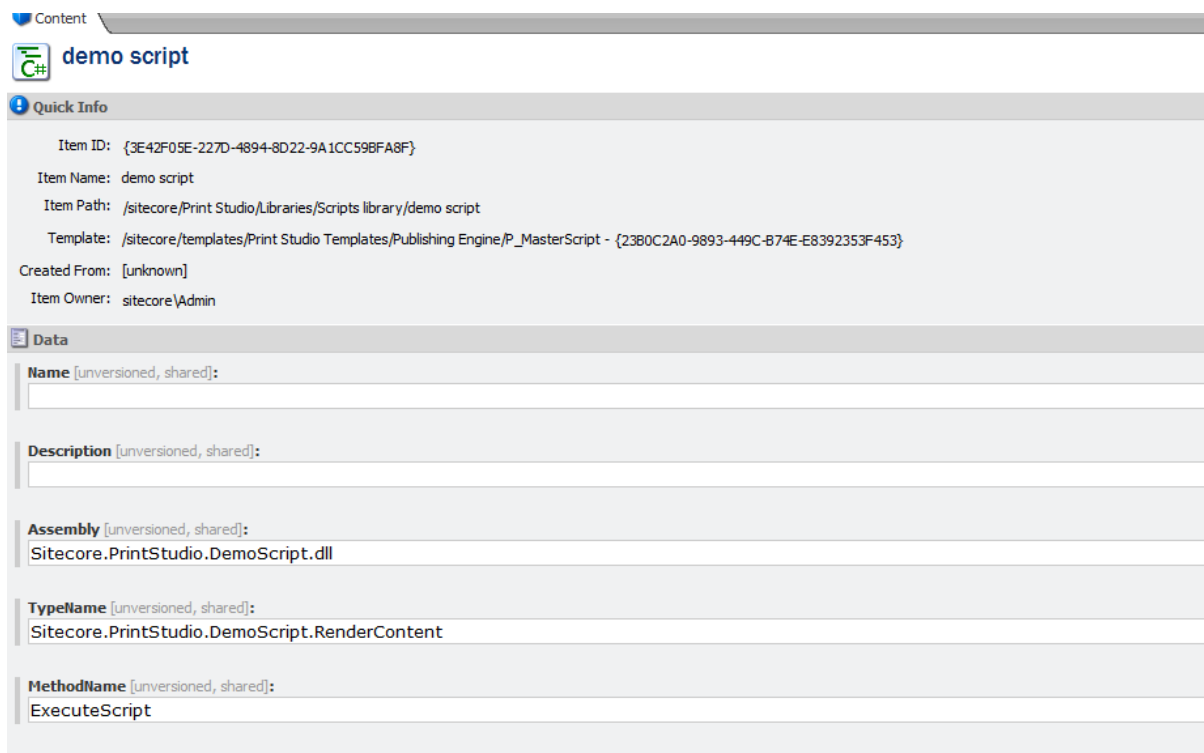
This document describes how to use the Script objects to extend the standard functionality of the Sitecore InDesign connector.

This tutorial assumes that you have good knowledge of Sitecore, InDesign and XML in more general. You will also need the engine XML schema files at hand while creating the logic to publish the XML.

1.1 Using Script Objects

A script object allows the execution of external assembly at runtime during the execution of an InDesign Project. In order to have a script attached to an Item in the Document items hierarchy we need to create a masterscript item (template: /sitecore/templates/Print Studio Templates/Publishing Engine/P_MasterScript) item.

The master script item contains the necessary fields and values for the script to be executed.



The screenshot displays the configuration for a 'demo script' item in the Sitecore Content Editor. The interface is divided into sections: Content, Quick Info, and Data.

Content: demo script

Quick Info:

- Item ID: {3E42F05E-227D-4894-8D22-9A1CC598FA8F}
- Item Name: demo script
- Item Path: /sitecore/Print Studio/Libraries/Scripts library/demo script
- Template: /sitecore/templates/Print Studio Templates/Publishing Engine/P_MasterScript - {23B0C2A0-9893-449C-B74E-E8392353F453}
- Created From: [unknown]
- Item Owner: sitecore\Admin

Data:

- Name** [unversioned, shared]:
- Description** [unversioned, shared]:
- Assembly** [unversioned, shared]: Sitecore.PrintStudio.DemoScript.dll
- TypeName** [unversioned, shared]: Sitecore.PrintStudio.DemoScript.RenderContent
- MethodName** [unversioned, shared]: ExecuteScript

The Assembly field value points to the assembly that should be invoked, it needs a full physical assembly name. If only the assembly name is entered the engine will look for the assembly in the bin folder of the website. If the full path is entered, it will load the assembly from the path location.

The TypeName field points to the class that should be instantiated; it needs the full class name including the namespace.

The MethodName field contains the name of the method that should be executed; it has to be a method in the class declared in TypeName field.

Once we have those values set a dynamic invocation will be performed using Reflection.

A standard list of parameters will be sent to the method being executed, so the method needs to accept either object or Dictionary<string, object> as a parameter.

Code Example:

```
public string ExecuteScript(Dictionary<string, object> dictionary)
{
    StringBuilder resultBuilder = new StringBuilder();
    if (dictionary != null)
    {
        foreach (KeyValuePair<string, object> k in dictionary)
        {
            resultBuilder.AppendFormat("<ParagraphStyle Style=\"plattetekst\">{0} - {1}
</ParagraphStyle>", k.Key, k.Value.ToString());
        }
        Guid itemID = (Guid)dictionary["ItemID"];
        Guid contentItemID = (Guid)dictionary["ContentItemID"];
        int languageIndex = (int)dictionary["LanguageIndex"];
        bool isClient = (bool)dictionary["IsClient"];
        string destinationFolder = dictionary["DestinationFolder"].ToString();
        bool useHighRes = (bool)dictionary["UseHighRes"];
        string database = dictionary["Database"].ToString();
    }
    return resultBuilder.ToString();
}
```

Parameters description:

Key	Type	Description
dictionary["ItemID"]	Guid	The script item ID
dictionary["ContentItemID"]	Guid	The parent item ID
dictionary["LanguageIndex"]	int	Language index
dictionary["IsClient"]	bool	Indication whether the caller is from desktop (true) or server (false)
dictionary["DestinationFolder"]	string	The project folder on client
dictionary["UseHighRes"]	bool	Whether to use High resolution images or not
dictionary["Database"]	string	Item database in sitecore (default is master)
dictionary["CurrentUserName"]	string	username

1.2 Working with images

Working with images in print output is of course different than using images for a website. In print output the images need to be available as physical files to InDesign desktop (when working on the desktop client) or the InDesign server (when working in a web-to-print situation). Besides that we have the need for both small file sizes and high resolution larger file sizes when a high quality PDF production is needed to send to print.

When we work with the InDesign client (InDesign connector), the images loaded in the locally rendered document, need to be available to the InDesign client. Therefore the images will be downloaded to the client and the XML needs to contain paths to those images.

The images need to be available on the server and will be automatically downloaded by the InDesign client. When on the InDesign client the “Use high res images” is switched on, it assumes that the images are accessible on some location on some file storage that is accessible by the InDesign client. In that case the images are not downloaded.

When working on a server based production, the images need to be available to the InDesign application server. In such a case we refer to the images stored on a path accessible by InDesign server, there is no need for downloading images.

Code sample:

```
private static string CreateImageOnServer(MediaItem medItem, bool fromClient, bool
useHighRes, string projectsPathOnServer, string destinationFolderOnClient)
{
    Stream stream = medItem.GetMediaStream();
    string imagePath = string.Empty;

    // first check if a high res production is requested
    if (useHighRes)
    {
        // if yes, use the high path values from the selected image item if available
        // regardless if its from the ID client or front end since both need to use the same

        // the item needs to use the storage item to compile the full path
        Item storeItem =
medItem.Database.GetItem(medItem.InnerItem.Fields["ReferenceStorePaths"].Value);
        string storagePath = storeItem.Fields["HighResFilePath"].Value;
        imagePath = storagePath + medItem.InnerItem.Fields["HighResFilePath"].Value;
    }
    else
    {
        if (fromClient)
        {
            // if its a request from the InDesign client, use image from the database
            // (you could use the low resolution images also of course)

            // the image needs to be available to the InDesign client so we need to use
            // the local path on the desktop machine.
            // the image needs to be created in the projects folder on the server
            // and we use the local path in the XML. When the XML is loaded the image will
            // be downloaded by the client from the server location.
            // projects path + subfolder (project name) + item Id + extension

            try
            {
                // create the image on the server in the projects folder
                // you could use the project name instead of the "temp" folder or some image storage
                CopyStreamToFile(stream, projectsPathOnServer + "temp" + "\\\" + medItem.ID + "." +
medItem.Extension);
            }
        }
    }
}
```

```

        // return the path used in the XML for the ID connector (local path on the
        // client machine)
        imagePath = destinationFolderOnClient + medItem.ID + "." + medItem.Extension;
    }
    catch { }
}
else
{
    // its a request from a front end (server based production), so use the low res path
    values if available
}
}

return imagePath;
}

private static string CopyStreamToFile(Stream stream, string destination)
{
    char[] trimChars = { '.', ' ' };
    string result = destination.TrimEnd(trimChars);
    using (BufferedStream bs = new BufferedStream(stream))
    {
        using (FileStream os = File.OpenWrite(destination))
        {
            byte[] buffer = new byte[2 * 4096];
            int nBytes;
            while ((nBytes = bs.Read(buffer, 0, buffer.Length)) > 0)
            {
                os.Write(buffer, 0, nBytes);
            }

            os.Close();
        }
    }

    return result;
}
}

```

When we then create the actual XML element by using for example:

```

public string ExecuteScriptImageFrame(Dictionary<string, object> dictionary)
{
    StringBuilder resultBuilder = new StringBuilder();
    if (dictionary != null)
    {
        try
        {
            XslHelper appSettings = new XslHelper();
            XmlDataDocument schemaDoc = new XmlDataDocument();
            XmlNamespaceManager nsmanager = new XmlNamespaceManager(schemaDoc.NameTable);
            nsmanager.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema");
            schemaDoc.Load(string.Format("{0}Data\\PrintStudioPublishingEngine.xsd",
            appSettings.AppSettings("APS.Root")));

            string projectsPathOnServer = appSettings.AppSettings("APS.ProjectsPath");

            Guid itemID = (Guid)dictionary["ItemID"];
            Guid contentItemID = (Guid)dictionary["ContentItemID"];
            int languageIndex = (int)dictionary["LanguageIndex"];
            bool isClient = (bool)dictionary["IsClient"];

```

```

string destinationFolder = dictionary["DestinationFolder"].ToString();
bool useHighRes = (bool)dictionary["UseHighRes"];
string database = dictionary["Database"].ToString();
string currentUser = dictionary["CurrentUserName"].ToString();

// get the script items parent (the textframe)
string fieldContent = "none";

Hashtable atts = new Hashtable();
atts.Add("Width", "100");
atts.Add("Height", "200");
atts.Add("X", "100");
atts.Add("Y", "200");
atts.Add("Scaling", "Fill Frame Proportionally and Center");
atts.Add("SitecoreID", "{" + itemID.ToString().ToUpper() + "}");

XmlDataDocument tempDoc = new XmlDataDocument();
XmlNode imageFrame = CreateElement("ImageFrame", schemaDoc, nsmanager, atts,
tempDoc);

// add the image
// fetch some media item (make sure the item ID exists)
Database currentDb = Configuration.Factory.GetDatabase(database);
MediaItem someItem = currentDb.SelectSingleItem("{23A86125-DEA2-48C5-B9BD-
FE0D42233685}");
string imagePath = CreateImageOnServer(someItem, isClient, useHighRes,
projectsPathOnServer, destinationFolder);

atts.Clear();
atts.Add("Width", "50");
atts.Add("Height", "100");
atts.Add("LowResSrc", imagePath);
atts.Add("HighResSrc", imagePath);
XmlNode image = CreateElement("Image", schemaDoc, nsmanager, atts, tempDoc);
imageFrame.AppendChild(image);

resultBuilder.AppendLine(imageFrame.OuterXml);
}
catch { }
}

return resultBuilder.ToString();
}

private static XmlNode CreateElement(string elementName, XmlDataDocument schemaDoc,
    XmlNamespaceManager nsmanager,
    Hashtable atts, XmlDataDocument tempDoc)
{
    // create new element
    XmlNode newElement = tempDoc.CreateElement(elementName);

    // fetch the attributes for this element
    try
    {
        XmlNodeList attributeList = schemaDoc.SelectNodes("//xs:element[@name='" + elementName
+ "']//xs:attribute",
                                                    nsmanager);

        foreach (XmlNode attributeNode in attributeList)
        {
            string attName = attributeNode.Attributes["name"].Value;
            string attDefValue = string.Empty;

```



```

try
{
    attDefValue = attributeNode.Attributes["default"].Value;
}
catch { }

// check if the attribute is in the pattern item and if so, use that value
try
{
    attDefValue = atts[attName].ToString();
}
catch { }

// create the attribute
XmlAttribute newAtt = tempDoc.CreateAttribute(attName);
newAtt.Value = attDefValue;
newElement.Attributes.Append(newAtt);
}
}
catch { }

return newElement;
}

```

The ImageFile parameter is passed to both the LowResSrc and the HighResSrc attribute. Depending on whether it's a low or high res production these attributes will get both the high or low res file path. Its UseHighRes attribute of the Project element that determines which one of them is used. When UseHighRes = true, the HighResSrc attribute is used. When UseHighRes = false, the LowResSrc attribute is used.

1.3 Working with tables

When producing more structured documents, tables are very useful. A basic table in InDesign consists of for example:

```

<Table TableStyle="Basic" HeaderRows="0" RepeatHeader="Once per frame" FooterRows="0"
    ColCount="4" RowCount="1">
<Row RowHeight="" RowMin="" KeepWithNext="" StartRow="" RowMax="">
    <Cell ColWidth="" CellStyle="" HorStradle="" VerStradle="">
        <ParagraphStyle Style="Body">Hello world</ParagraphStyle>
    </Cell>
    <Cell ColWidth="" CellStyle="" HorStradle="" VerStradle=""></Cell>
    <Cell ColWidth="" CellStyle="" HorStradle="" VerStradle=""></Cell>
    <Cell ColWidth="" CellStyle="" HorStradle="" VerStradle=""></Cell>
</Row>
</Table>

```

Always make sure that the table structure is correct. When InDesign shuts down unexpectedly, it is in most cases related to an invalid table structure.

The following code snippet creates a table and adds the “Hello world” paragraph to the second row, second cell.

Sample code:

```
private static XmlNode CreateTable(XmlDataDocument schemaDoc, XmlNamespaceManager
nsmanager, XmlDataDocument tempDoc)
{
    // create the table with 4 columns 5 rows (incl header and footer rows), 1 header row and
0 footer rows
    Hashtable atts = new Hashtable();
    atts.Add("RowCount", "5");
    atts.Add("ColCount", "4");
    atts.Add("TableStyle", "NewTable");
    atts.Add("HeaderRows", "1");
    XmlNode tableNode = CreateElement("Table", schemaDoc, nsmanager, atts, tempDoc);
    // add the rows and cells
    for (int a = 0; a < 5; a++)
    {
        // add a row to the table
        atts.Clear();
        XmlNode newRow = CreateElement("Row", schemaDoc, nsmanager, atts, tempDoc);
        tableNode.AppendChild(newRow);

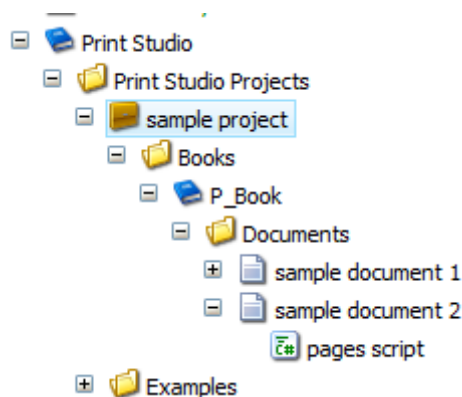
        for (int b = 0; b < 4; b++)
        {
            // add a cell for each column
            atts.Clear();
            XmlNode newCell = CreateElement("Cell", schemaDoc, nsmanager, atts, tempDoc);
            newRow.AppendChild(newCell);

            // add a paragraph to the second row, second cell
            if ((b == 1) && (a == 1))
            {
                // create the hello world paragraph
                atts.Clear();
                XmlNode newPar1 = CreateElement("ParagraphStyle", schemaDoc, nsmanager, atts,
tempDoc);
                newPar1.InnerText = "Some content";
                newCell.AppendChild(newPar1);
            }
        }
    }

    return tableNode;
}
```

1.4 Using script objects to create an entire document

When you want to use a script object to produce an entire InDesign document, you need to attach a script object to the document node as shown below.



This script is now executed as the child of the document node. According to the engine XML schema, the child of “Document” needs to be “Pages” or “Flows”. Therefore your script object should start with one of those elements.

1.5 Page type production

You use a page type production if the lay-out is “page oriented” and you want to add pages with the logic. The objects on the page (text frames or image frames) in most cases have fixed positions and more or less fixed sizes. In general most InDesign productions tend to be page oriented productions where the design is the key factor. The XML structure looks like:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE Pages SYSTEM "PrintStudioPublishingEngine.dtd">
3  <Pages>
4    <Page Number="1" MasterPrefix="B" MasterBase="Master">
5      <TextFrame Width="100" Height="100" X="50" Y="50" LayerName="text">
6        <ParagraphStyle Style="Body">Some content here</ParagraphStyle>
7      </TextFrame>
8    </Page>
9  </Pages>
10
```

1.6 Flow type production

You use a “Flow” type production if you want to add pages depending on the content.

This is a “data-driven” lay-out where the amount of pages is depending on the amount of data and the formatting rules set in the master document. In general you could say that the longer textual documents (like manuals, technical catalogues/pricelist, etc.) are flow type productions. The XML structure looks like:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE Pages SYSTEM "PrintStudioPublishingEngine.dtd">
3 <Pages>
4   <Flow MasterPrefix="A" MasterBase="Master">
5     <ParagraphStyle Style="Body">some content here</ParagraphStyle>
6   </Flow>
7 </Pages>
8
```

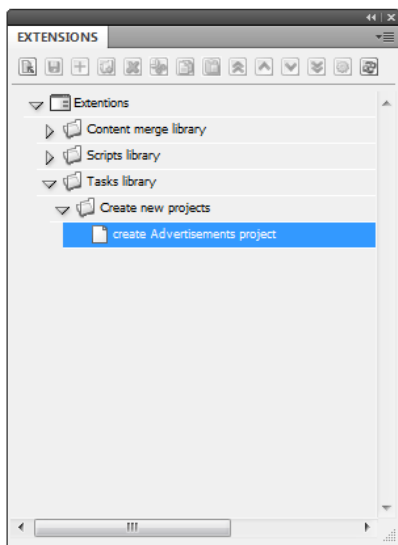
Chapter 2

Using tasks

To automate certain tasks from the InDesign connector interface, a task item can be created. A sample of a task could be “create a new project and add certain pages” or “rename all snippets in a project”. This allows adding specific tasks and executing those tasks from the InDesign connector interface.

2.1 Creating tasks


The tasks are shown in the “Extensions” panel in the “Tasks library” as shown below.



To create a task, add a task item somewhere in the task library inside Sitecore using the template / sitecore/templates/Print Studio Templates/InDesign connector/Tasks/Task.

Fill in the fields Assembly, TypeName and MethodName.

Content

 **create Advertisements project**

Quick Info

Item ID: {B8A54706-831E-4469-85EB-15BAD83F3654}

Item Name: create Advertisements project

Item Path: /sitecore/Print Studio/Libraries/Tasks library/Create new projects/create Advertisements project

Template: /sitecore/templates/Print Studio Templates/InDesign connector/Tasks/Task - {36BCD061-CE6D-416C-8891-0B4344D2D8FC}

Created From: [unknown]

Item Owner: sitecore\Admin

Data

Name [unversioned, shared]:

Description [unversioned, shared]:

Assembly [unversioned, shared]:

Sitecore.PrintStudio.DemoScript.dll

TypeName [unversioned, shared]:

Sitecore.PrintStudio.DemoScript.RenderContent

MethodName [unversioned, shared]:

ExecuteTask

The following parameters are passed:

Key	Type	Description
dictionary["ItemID"]	Guid	The task item ID
dictionary["LanguageIndex"]	int	Language index
dictionary["CurrentUserName"]	string	username
dictionary["ci_projectPanel"]	string	Selected item project panel
dictionary["ci_contentBrowser"]	string	Selected item content browser panel
dictionary["ci_libraryBrowser"]	string	Selected item library browser panel
dictionary["ci_imageViewer"]	string	Selected item image viewer panel
dictionary["ci_workBox"]	string	Selected item worbox panel

For example:

```
public string ExecuteTask(Dictionary<string, object> dictionary)
{
    string result = string.Empty;
    if (dictionary != null)
    {
        foreach (KeyValuePair<string, object> k in dictionary)
        {
            result += " key: " + k.Key + " value: " + k.Value + " ";
        }
    }

    return result;
}
```

The result of the called method is returned and shown as a message in the InDesign connector.

Chapter 3

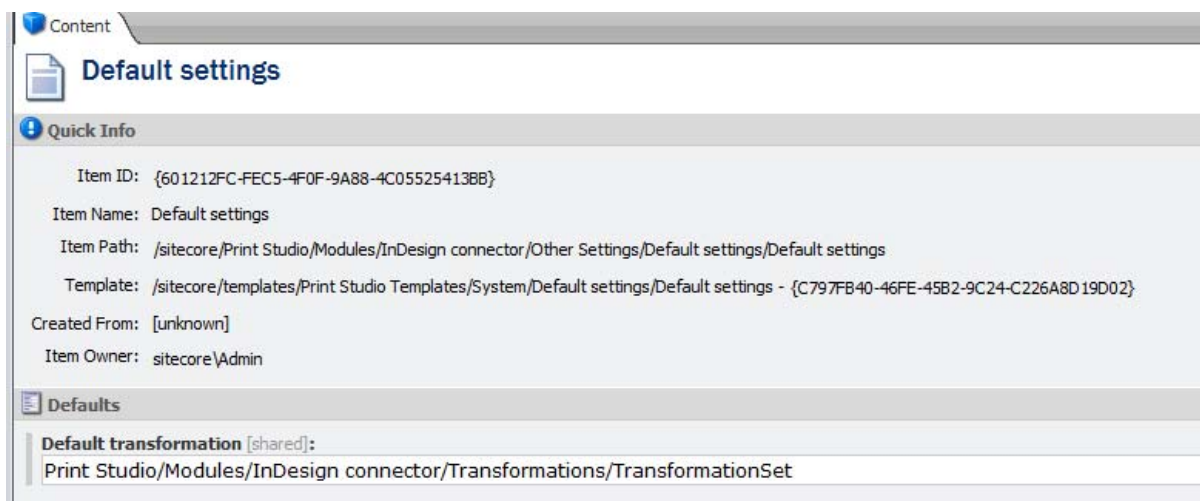
Transformations

To transform HTML (web) to XML (InDesign) when publishing to InDesign or to transform XML (InDesign) to HTML (web) when saving content from InDesign, a transformation can be used.

3.1 Creating transformations

To transform HTML content from Rich Text fields to engine XML, transformation items can be created (please refer to the administrator manual). To transform the content from HTML to XML (when publishing) or from XML to HTML (when saving) by using additional logic, the transformation item can use a “scriptobject” as well.

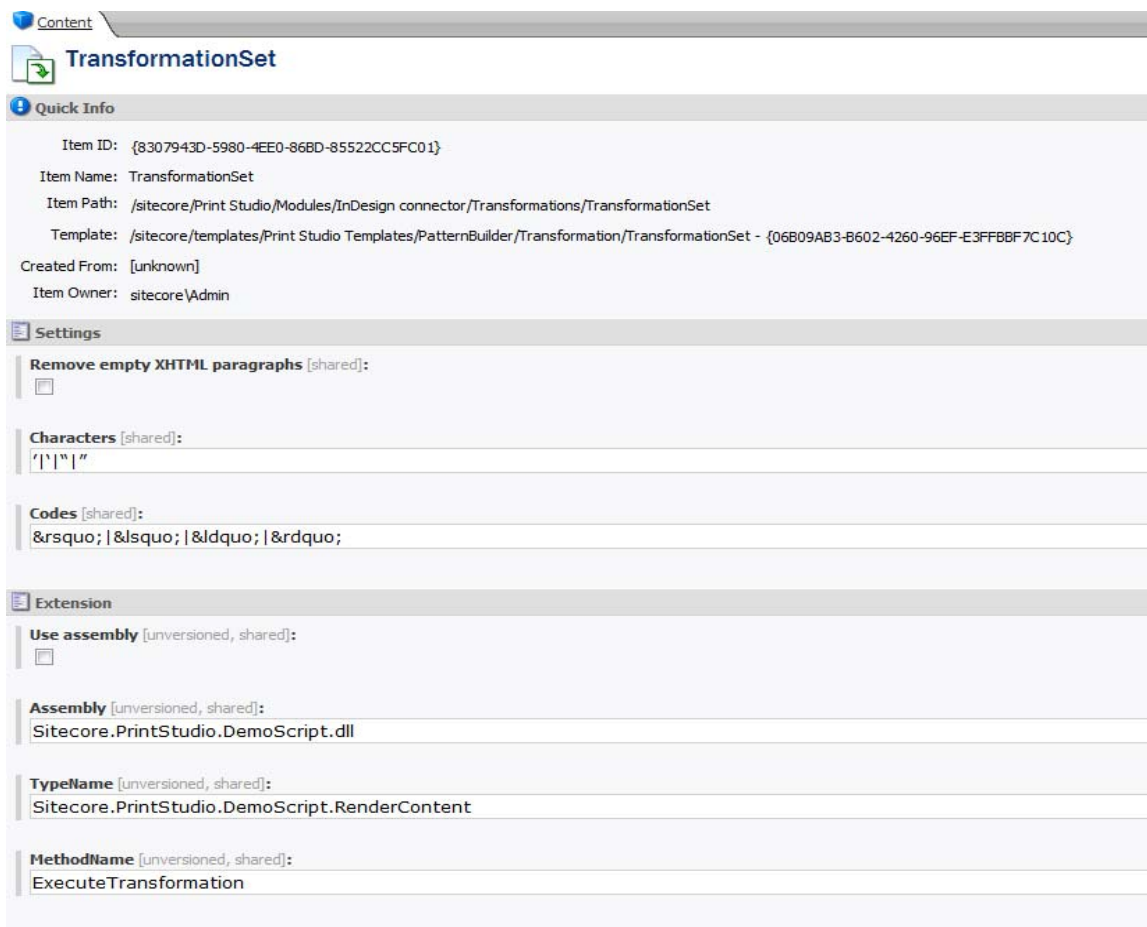
The default transformation item used to transform the content is set in the “Default settings” item (path: /sitecore/Print Studio/Modules/InDesign connector/Other Settings/Default settings/Default settings), “Default transformation” as shown below:



The screenshot displays the 'Default settings' item in the Sitecore Content Manager. The interface is divided into several sections:

- Content**: The top navigation bar.
- Default settings**: The main title of the item.
- Quick Info**: A section containing the following details:
 - Item ID: {601212FC-FEC5-4F0F-9A88-4C05525413BB}
 - Item Name: Default settings
 - Item Path: /sitecore/Print Studio/Modules/InDesign connector/Other Settings/Default settings/Default settings
 - Template: /sitecore/templates/Print Studio Templates/System/Default settings/Default settings - {C797FB40-46FE-45B2-9C24-C226A8D19D02}
 - Created From: [unknown]
 - Item Owner: sitecore\Admin
- Defaults**: A section containing the following information:
 - Default transformation [shared]:** Print Studio/Modules/InDesign connector/Transformations/TransformationSet

The “Default transformation” field points to a transformation item.



The screenshot displays the configuration for a **TransformationSet** item in the Sitecore Content Editor. It is organized into several sections:

- Quick Info:**
 - Item ID: {8307943D-5980-4EE0-86BD-85522CC5FC01}
 - Item Name: TransformationSet
 - Item Path: /sitecore/Print Studio/Modules/InDesign connector/Transformations/TransformationSet
 - Template: /sitecore/templates/Print Studio Templates/PatternBuilder/Transformation/TransformationSet - {06B09AB3-B602-4260-96EF-E3FFBBF7C10C}
 - Created From: [unknown]
 - Item Owner: sitecore\Admin
- Settings:**
 - Remove empty XHTML paragraphs [shared]:**
 - Characters [shared]:** | | | | "
 - Codes [shared]:** ’ | ‘ | “ | ”
- Extension:**
 - Use assembly [unversioned, shared]:**
 - Assembly [unversioned, shared]:** Sitecore.PrintStudio.DemoScript.dll
 - TypeName [unversioned, shared]:** Sitecore.PrintStudio.DemoScript.RenderContent
 - MethodName [unversioned, shared]:** ExecuteTransformation

To use external logic for the transformation, check the “Use assembly” checkbox and fill in the correct values for the “Assembly”, “TypeName” and “MethodName” fields.

The following parameters are passed:

Key	Type	Description
dictionary["ItemID"]	Guid	The transformation item ID
dictionary["InputString"]	string	The input data (when publishing this is the value from the Rich Text field, when saving this is the XML structured value from InDesign)
dictionary["SourceFormat"]	string	Whether the source is XHTML (publishing) or XML (saving)
dictionary["TargetFormat"]	string	Whether the target format is XHTML (saving) or XML (publishing)

The result of the called method is returned and used to either publish or save.