Content Generation

Overview

The placement of Document Elements within a technical document can trigger the execution of a ‘Content Generator’. Content Generators are server-side Methods (C#) that can utilize a software API for creating Document Elements programmatically. Content Generators can apply whatever custom logic is necessary, including information about the document that the Document Elements will be inserted into, to automate the generation of technical document data.

Content Generators are typically configured to execute on the placement of an Item Document Element (See Document Element Types) to extract information about the selected Item and use that to populate content in the generated Document Elements.

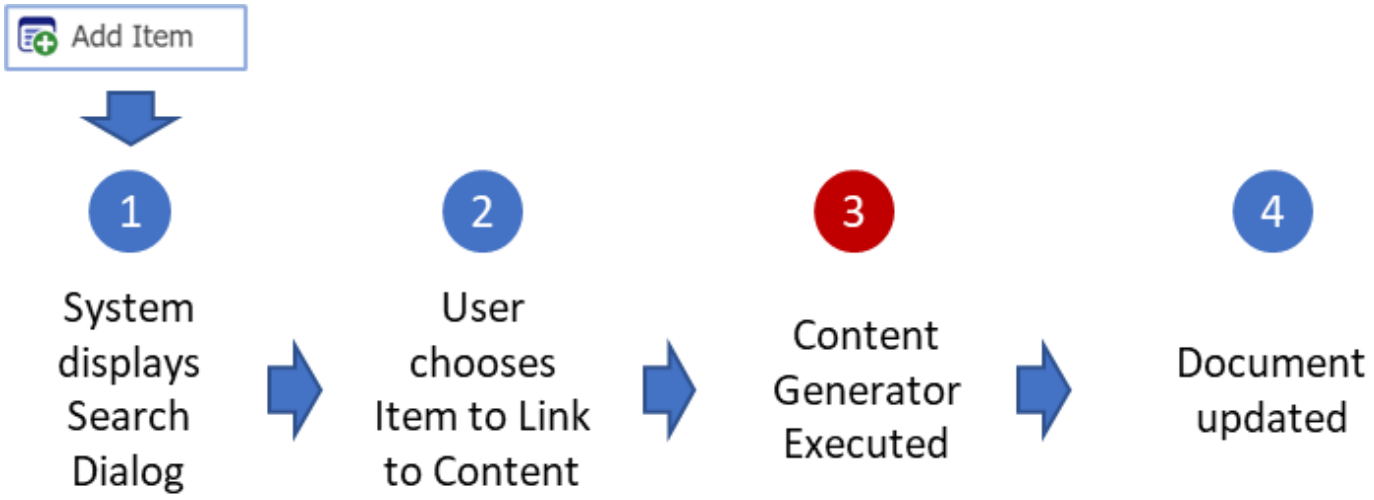
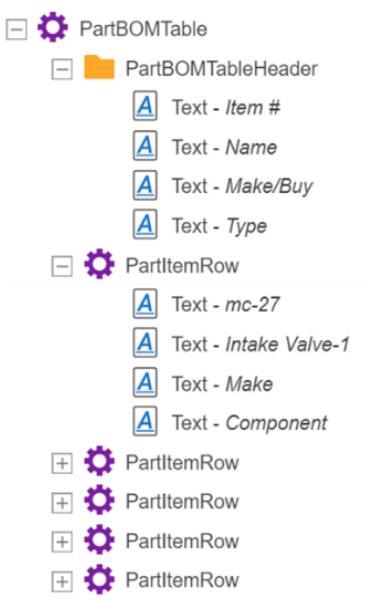


Figure: Content Generator Execution

The execution of Content Generators can be parameterized to provide additional flexibility/configurability to Technical Document Administrators. This section will describe the process of configuring a Content Generator.

Content Generator Example

This section provides an example Content Generator that constructs a table consisting of a row for each child BOM Part of a selected Assembly. The output will be similar to the following:



Item #	Name	Make/Buy	Type
mc-27	Intake Valve-1	Make	Component
mc-28	Intake Lifter 2 Bottom-1	Make	Component
mc-29	Intake Lifter 1 Bottom-1	Make	Component
mc-30	Intake Camshaft-1	Make	Component
mc-31	Intake Lifter 2 Top-1	Make	Component
mc-32	Intake Lifter 1 Top-1	Make	Component
mc-33	Exhaust Valve-1	Make	Component
mc-34	Exhaust Lifter 2 Top-1	Make	Component
mc-35	Exhaust Lifter 2 Bottom-1	Make	Component
mc-36	Exhaust Camshaft-1	Make	Component

Figure: Content Generator Output Example

The top-most Item Document Element, with name ‘ `PartBOMTable` ’ contains two child Document Elements. The first is ‘ `PartBOMTableHeader` ’; which is a Container Document Element for Text Document Elements that will provide the top-most header row. The second is another Item Document Element, with name ‘ `PartItemRow` ’ that also contains Text Document Elements with values for selected Part Properties. The Document Type XML Schema is as follows:

```
<xs:element name="PartBOMTable">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemType">
<xs:sequence maxOccurs="unbounded">
<xs:element ref="PartBOMTableHeader" minOccurs="1" maxOccurs="1"/>
<xs:element ref="PartItemRow" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="typeId" type="xs:string" fixed="Part ItemType ID"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="PartBOMTableHeader">
<xs:complexType>
<xs:sequence>
<xs:element ref="Text" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="PartItemRow">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemType">
<xs:choice maxOccurs="unbounded">
<xs:element ref="Text" minOccurs="0" maxOccurs="unbounded"/>
</xs:choice>
<xs:attribute name="typeId" type="xs:string" fixed="Part ItemType ID"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

For more information on how to construct Document Type schema see Standard XML Schema. The source code for the Content Generator that produces this output is as follows:

```
//MethodTemplateName=CSharp:Aras.TDF.ContentGenerator(Strict);
ItemDocumentElement targetItem = targetElement as ItemDocumentElement;
if (targetItem != null) {
targetItem.ClearChilds();
String AML = @"<AML>"
+"<Item type='Part BOM' action='get'>"
+"  <source_id>" + targetItem.ItemId + "</source_id>"
+"  <related_id>"
+"    <Item type='Part' action='get'/>"
+"  </related_id>"
+"</Item>"
+"</AML>";
DocumentSchemaElement headerElement = (DocumentSchemaElement) this.Factory.NewElement("PartBOMTableHeader
TextDocumentElement hd1 = (TextDocumentElement) this.Factory.NewText("Text", "Item #");
TextDocumentElement hd2 = (TextDocumentElement) this.Factory.NewText("Text", "Name");
TextDocumentElement hd3 = (TextDocumentElement) this.Factory.NewText("Text", "Make/Buy");
TextDocumentElement hd4 = (TextDocumentElement) this.Factory.NewText("Text", "Type");
headerElement.AddChild(hd1);
headerElement.AddChild(hd2);
headerElement.AddChild(hd3);
headerElement.AddChild(hd4);
targetElement.AddChild(headerElement);
Item bomItems = this.Factory.InnovatorInstance.applyAML(AML);
int numItems = bomItems.getItemCount();
// Loop through each Catalog Entry and fill the remaining table rows
for(int i=0; i < numItems; i++)
{
Item bomItem = bomItems.getItemByIndex(i);
Item partItem = bomItem.getPropertyItem("related_id");
ItemDocumentElement itemElement = (ItemDocumentElement) this.Factory.NewElement("PartItemRow");
// set item reference
itemElement.SetItem(partItem);
TextDocumentElement numText = (TextDocumentElement) this.Factory.NewText("Text", partItem.getProperty("it
TextDocumentElement nameText = (TextDocumentElement) this.Factory.NewText("Text", partItem.getProperty("n
TextDocumentElement mbText = (TextDocumentElement) this.Factory.NewText("Text", partItem.getProperty("mak
TextDocumentElement typeText = (TextDocumentElement) this.Factory.NewText("Text", partItem.getProperty("c
itemElement.AddChild(numText);
itemElement.AddChild(nameText);
itemElement.AddChild(mbText);
itemElement.AddChild(typeText);
targetElement.AddChild(itemElement);
} // for() loop over catalog entries
} // if (targetItem != null)
```

1. The first line of a Content Generator Method must identify the Method Template to use as a comment. In this case, it should be specified as it is in this example –

```
//MethodTemplateName=CSharp:Aras.TDF.ContentGenerator(Strict);
```

2. The Content Generator in this example is configured with an Item Document Element – **PartBOMTable**. The parameter ‘ `targetElement` ’ is passed to the Content Generator

Method as a more generic `DocumentElement` object. This line casts the object to the expected `ItemDocumentElement` type. The target Element represents the instance of the Document Element that the Content Generator is executed against.

3. The check for null is necessary to assure that the cast on the previous line was successful. Trying to cast to a Document Element Type class that is not represented by the type of Document Element will cause an error and the variable will be assigned `null` .
4. Content Generators can be executed multiple times, either because they are configured as dynamic or because the author explicitly reruns (refreshes) the Document Element. In these cases, the Content Generator Method is executed again on the associated Document Element. Clearing all previously generated content is therefore necessary. Note that the *grammar* of the function is wrong. ‘ `ClearChilds()` ’ will remove all child content from the document.
5. The Document Element for the first child element for **PartBOMTable** is created using the `Factory` Object. The `DocumentSchemaElement` class is used for Container Document Elements. The name of the Document Element (‘**PartBOMTableHeader**’) is passed to the constructor.
6. For this example, the **PartBOMTableHeader** Document Element contains Text Document Elements as its children. The next 4 lines create a new `TextDocumentElement` for the ‘labels’ of the Part BOM Table. The `TextDocumentElement` constructor is passed the name of the Text Document Element (in this case – ‘Text’) and the text used for the content.
7. Each Text Document Element is added to the parent **PartBOMTableHeader**
8. The **PartBOMTableHeader** is added to the target Element.
9. An AML Query is executed to extract the Part Items for the Item linked to the target Element.
10. A loop to extract each Part Item.
11. Each Part will be linked to a newly generated Item Document Element with the name **PartItemRow**.
12. The Item object returned from the execution of the AML is used to set the link for the **PartItemRow** Item Document Element. This will create a separate link for each row and likewise, a separate Relationship from the Document to the Part Item.
13. Like the header, new Text Document Elements are added with text extracted from the selected Part properties.
14. These Item Document Elements are added to the target Element.

In summary, all Content Generator Methods should:

1. Set the appropriate Method Template
2. Create a variable that is cast to the appropriate type

3. Clear the content, if necessary
4. Modify the target Element and/or add child content as appropriate

Content Generator Configuration

The Content Generator Method is specified in the Document Type (see Section 2). Content Generators can be configured as ‘dynamic’ or ‘static’. *Static*, the default, will result in the Content Generator executing when the Document Element is added to the document or when the author explicitly refreshes the Document Element using the context menu for the associated Document Element. *Dynamic* is set if the check box is selected in the Document Type. Dynamic Content Generators are executed when the Document Element is created or refreshed, but also when the Document is opened or saved. Depending on the complexity of the Content Generator, setting the Dynamic flag can affect performance.

Content Generator Parameters

Content Generator Methods can be passed parameters (or any type of input data) defined as a JSON-formatted string. Such an approach can be used to configure the execution of the Content Generator. The JSON is specified on the XML Schema Element for the specific Document Element.

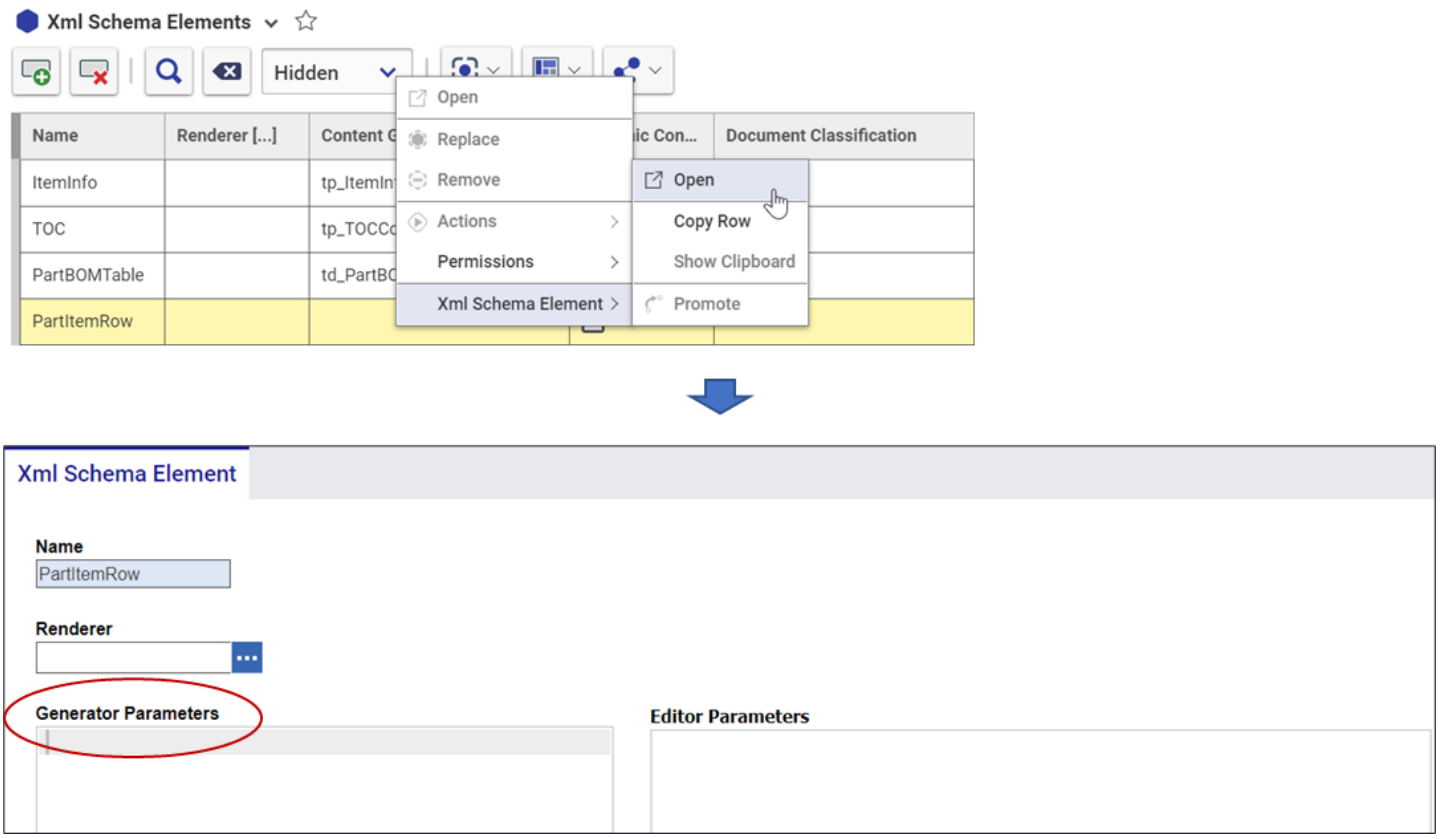


Figure: XML Schema Element Form - Content Generator Parameters

The **Generator Parameters** input form is configured for displaying JSON content. Syntactical errors will be displayed if the text content does not conform to JSON formatting. It is important that the content be properly formatted JSON data, or it will not be converted to content that can be accessed in the C# Content Generator Method.

Important

JavaScript Object Notation (JSON) is used to serialize JavaScript Object data. It is used in the Technical Document Framework to enable complex configuration data for parameterizing Methods. A description of JSON, including how JSON can be represented via Collection classes in .Net, is outside the scope of this document.

Content Generator Example – Name / Value Pair

The following example shows how to configure and parse a single name/value pair for a Content Generator. In this example, a parameter ‘ `type` ’ can have one of two values: ‘ `product` ’ or ‘ `shipping` ’. The JSON formatted string is stored in the UI as follows:

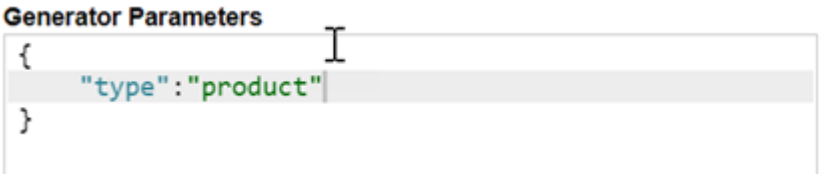


Figure: Content Generator Parameter Example - Name / Value Pair
The text is for this must be:

```
{
"type":"product"
}
```

Or

```
{
"type":"shipping"
}
```

Note the open and closing brackets (‘{‘ and ‘}’). The Content Generator Method can retrieve this information using the following example:

```
// The configured parameters should contain a single object with a
// type property with values of either 'product', or 'shipping' like:
//{"type":"product"}
if (!this.GenerationParameters.ContainsKey("type"))
{
throw new InvalidOperationException("'type' generation parameter was not configured.");
}
string type = (string)this.GenerationParameters["type"];
```

1. Uses the public object `GenerationParameters` to check for the existence of the `type` variable. If it does not exist, it throws an `InvalidOperationException` with an appropriate message. If parameters are required, a similar mechanism should be used to warn the Technical Document Administrator that the Content Generator is not configured properly.
2. The value for the `type` parameter is extracted as a string to be used in the subsequent logic of the Content Generator. Casting is required.

Content Generator Example – Lists

The following example shows how to configure and parse a list of name/value pairs for a Content Generator. In this example, a parameter ‘ `list` ’ can have multiple objects; each consisting of a name and value. The JSON formatted string for this is as follows:

```
{
"list":[{"name":"Prop 1", "value":"Prop 1 Value"},
{"name":"Prop 2", "value":"Prop 2 Value"},
{"name":"Prop 3", "value":"Prop 5 Value"}]
}
```

Note the open and closing brackets (‘{‘ and ‘}’) and the use of brackets for each list item. Variable names and values are specified as quoted strings. The Content Generator Method can retrieve this information using the following example:

```
// The configured parameters should contain an array labeled 'list'
// that contains objects with name/value properties, like:
//{
//  "list":[{"name":"Prop 1", "value":"Prop 1 Value"},
//          {"name":"Prop 2", "value":"Prop 2 Value"},
//          {"name":"Prop 3", "value":"Prop 5 Value"}]
//}
if (!this.GenerationParameters.ContainsKey("list"))
{
throw new InvalidOperationException("'list' generation parameter was not configured.");
}
// Create List Items based on given Name/Value pairs
System.Collections.Generic.List<object> list = (System.Collections.Generic.List<object>) this.GenerationP
foreach(System.Collections.Generic.Dictionary<string, object> listItem in list)
{
…
string name = (string)listItem["name"];
string value = (string)listItem["value"];
…
```

1. Uses the public object `GenerationParameters` to check for the existence of the `list` variable. If it does not exist, it throws an `InvalidOperationException` with an appropriate message. If parameters are required, a similar mechanism should be used to warn the Technical Document Administrator that the Content Generator is not configured properly.
2. The value for the `list` parameter is extracted as a `Generic.List` to be used in the subsequent logic of the Content Generator. Casting is required.
3. Loop over the contents of the List. Note the use of a `Generic.Dictionary` in this example.
4. The name and value are extracted using array notation.

Content Generator API

The Content Generator API provides access to the same core classes that the system uses to generate technical document content. It consists of a class hierarchy with extensions for each of the main Document Element Types (See Appendix - Content Generator API).