Appendix

Content Generator API

This section describes the Document Element classes that represent an Application Programming Interface (API) for implementing a Content Generator. Each subsection describes the class for each Document Element Type along with an example implementation and helpful information for the use of each.

Table Document Element

Code Sample

```
// create table with 4 rows and 4 columns
TableDocumentElement tableElement = (TableDocumentElement) this.Factory.NewTable("table", 4, 4);
// or this.Factory.NewTable();
// or this.Factory.NewTable("table");
// merge first cell to the right
tableElement.MergeCells(0, 0, MergeDirection.Right);
// merge third cell in the second row to the left, twice
tableElement.MergeCells(1, 2, MergeDirection.Left);
tableElement.MergeCells(1, 2, 3);
// merge last cell in second row with cell below, twice
tableElement.MergeCells(1, 3, MergeDirection.Down);
tableElement.MergeCells(1, 3, 2);
// unmerge first cell
tableElement.UnmergeCells(0, 0);
// delete last row deletion
tableElement.RemoveRow(tableElement.RowCount - 1);
// add new row at the end
tableElement.AddRow();
// add new row at the beginning
tableElement.AddRow(0);
// add new column at the end
tableElement.AddColumn();
// add new column before the second
tableElement.AddColumn(1);
// remove third column
tableElement.RemoveColumn(2);
targetElement.AddChild(tableElement);
```

Additional Information

- If table element Name is not explicitly passed in the `NewTable()` constructor, then first appropriate schema element will be used.

- Factory method `NewTable()` takes as parameters `rowCount` and `columnCount` values. If these parameters are omitted, then table will be created with 2 rows and columns (default value).
- In methods like `AddRow()` , `AddColumn()` , `RemoveRow()` and etc., Index parameter can be omitted. In this case method will be applied to the last row or column. Table must have at least one row and column.
- In method `MergeCells()` parameter `mergeDirection` can take one of the enumeration values: (`MergeDirection.Up,MergeDirection.Right, MergeDirection.Down, MergeDirection.Left`) or integer value (0, 1, 2, 3), numbers correspond to the enumeration.

List Document Element

Code Sample

```
ListDocumentElement listElement = (ListDocumentElement) this.Factory.NewList("list1", 3);
// or this.Factory.NewList();
// or this.Factory.NewList("list1");
// set list style
listElement.ListType = ListTypes.Numeric;
// create new list item at the end
listElement.AddItem();
// create new list item at the beginning
listElement.AddItem(0);
// remove second item
listElement.RemoveItem(1);
targetElement.AddChild(listElement);
```

Additional Information

- If list element Name isn't passed in `NewList()` constructor, then first appropriate schema element will be used.
- Factory method `NewList()` takes as a second parameter `itemCount` value. If this parameter is omitted and there is no information about default children count in the schema definition, then list will be empty (there is no default value).
- In methods `AddItem()` , `RemoveItem()` parameter Index can be omitted. In this case method will be applied to the last list Item.
- Property `ListType` can take one of the following enumeration values: `ListTypes.Bullet` , `ListTypes.Numeric` , `ListTypes.Alpha` .

Text Document Element

Code Sample

```
// create text element with «Hello World!» content
TextDocumentElement textElement = (TextDocumentElement) this.Factory.NewText("ptxt",
"Initial Text!\n");
// or this.Factory.NewText();
// or this.Factory.NewText("ptxt");
// replace all existing strings with a new one
textElement.Text = "Hello, World!\n";
// add new string
textElement.AddString("How are you, World?\n");
// add new string with «italic» style
textElement.AddString("Bye, World!\n", "bold");
// set style for second string
textElement.Strings[1].SetStyle("italic bold under strike");
// set style for substring
textElement.Strings[2].SetStyle("sub", 3, 7);
// set link on element of another document (root element in this case)
textElement.Strings[4].SetLink("1D93C56E1EBA4F6A9B68D348444C9C17",
"1D93C56E1EBA4F6A9B68D348444C9C17");
// remove link
textElement.Strings[4].RemoveLink();
targetElement.AddChild(textElement);
```

Additional Information

- Factory method `NewText()` has an optional second parameter `initialText` value.
- Text Document Elements contain `StringDocumentElement` s as its children. Separate `StringDocumentElement` s can be obtained with indexer property `Strings` .

```
StringDocumentElement stringElement = textElement.Strings[1];
```

- `StringCount` property identifies the number of `StringDocumentElement` s.

- Property `Text` returns concatenated content of all strings of the `TextDocumentElement` . If property used as a setter, then all current strings will be replaced by the given string.
- Method `SetStyle()` of `StringDocumentElement` takes text Style string as a parameter. It contains desired styles, which are delimited by the 'space' character. Additionally the `SetStyle()` method can take `startRange` and `endRange` parameters, in this case string may be split on substrings and total number of strings in the `TextDocumentElement` will be increased.

```
// after this command, the number of strings in the text Element will increase to two
textElement.Strings[2].SetStyle("sub", 3, 7);
```

- Method `SetLink()` takes `targetBlockId` (id of existing Technical Document) and `targetElementId` (`id` of any element in this Document). Currently will work properly for root document element and its children.

Image Document Element

Code Sample

```
ImageDocumentElement imageElement = (ImageDocumentElement) this.Factory.NewElement("graphic");
// get tp_Image item from Innovator
Item imageItem = this.Factory.InnovatorInstance.newItem("tp_Image", "get");
imageItem.setID("70CA6215D95C4658ACA68BFDB73E4DEE");
imageItem = imageItem.apply();
// set reference on image item
imageElement.SetImage(imageItem);
targetElement.AddChild(imageElement);
```

Additional Information

- Currently there is no explicit constructor for `ImageDocumentElement` in `SchemaElementFactory.` It should be created with base element constructor (`Factory.NewElement`).
- By default, `ImageDocumentElement` s are empty. Method `SetImage()` can be used to set the image Item reference and takes an `Item` as a parameter. Currently `ImageDocumentElement` s can have references only to `tp_Image` Items.

Item Document Element

Code Sample

```
ItemDocumentElement itemElement = (ItemDocumentElement) this.Factory.NewElement("aras-part");
// get item from Innovator
Item itemItem = this.Factory.InnovatorInstance.newItem("Part", "get");
itemItem.setID("CD2B48A0C3C042898AC00254930FC8A1");
itemItem = itemItem.apply();
// set item reference
itemElement.SetItem(itemItem);
targetElement.AddChild(itemElement);
```

Additional Information

- Currently there is no explicit constructor for `ItemDocumentElement` in `SchemaElementFactory.` It should be created with base element constructor (`Factory.NewElement`).

- By default, `ItemDocumentElement` s are empty. Method `SetItem()` can be used to set the Item reference and takes an `Item` as a parameter. Currently `ItemDocumentElement` s can have refences only to `tp_Item` Polysource Items.

Item Property Document Element

> **Important**
>
> Item Property Document Elements are also referred to as *Mapped* Property Document Elements.

Code Sample

```
Item partItem = bomItem.getPropertyItem("related_id");
ItemDocumentElement itemElement = (ItemDocumentElement)this.Factory.NewElement("PartItemRow");
// set item reference
itemElement.SetItem(partItem);
ItemPropertyDocumentElement nameText = (ItemPropertyDocumentElement) this.Factory.NewItemProperty("NamePr
…
itemElement.AddChild(nameText);
```

Additional Information

- This example adds an ItemPropertyDocumentElement based on the following Schema Definition:

```
<xs:element name="NameProperty">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemProperty">
<xs:attribute name="property" type="xs:string" fixed="name"/>
<xs:attribute name="mode" type="aras:itemPropertyModeEnum" fixed="write"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

- Note that in the example above, the property attribute is fixed for the ‘name’ Property and the mode is fixed to ‘write’. These are optional in the schema although it’s useful to define default values.
- The example above includes the name of the Property in the `NewItemProperty` method. This is required even though it was defaulted to a value in the schema.

Configuration Examples

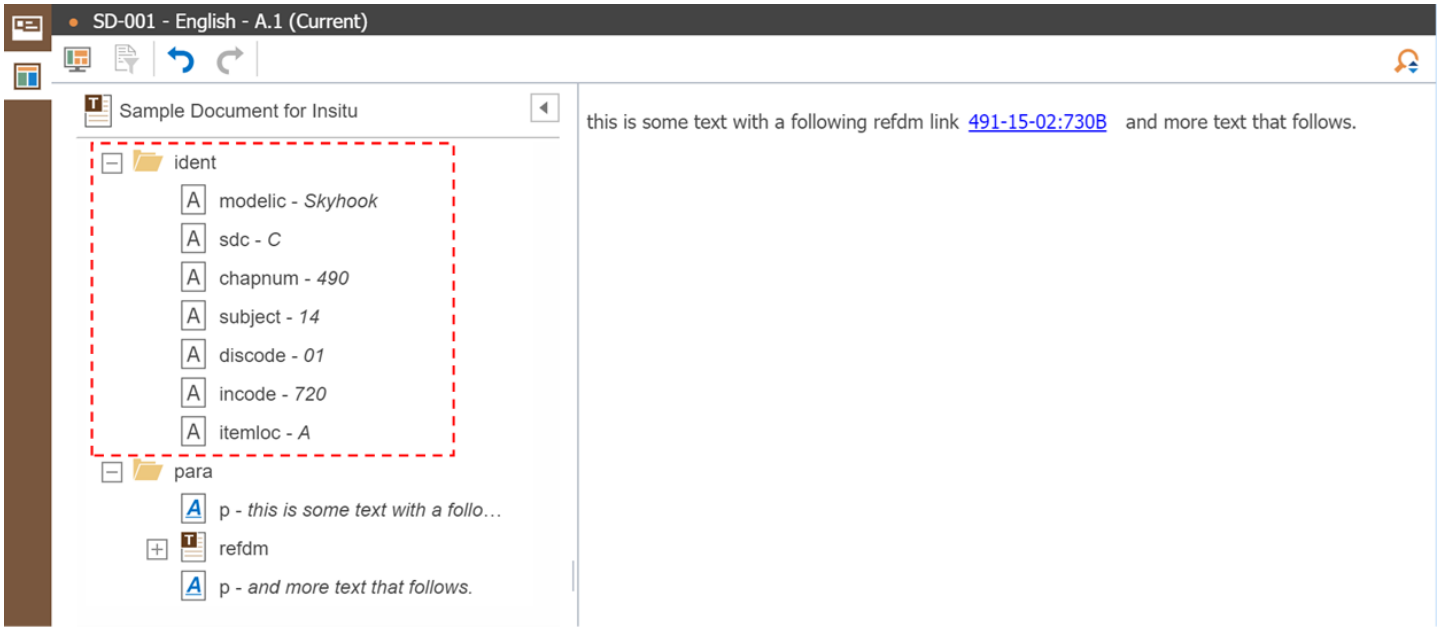
Overview

This section provides examples that describe how to configure a Tech Doc-Enabled Item to achieve a specified purpose. Each sub-section includes an Objective that identifies the purpose of the example.

Example – Auto-creating hidden elements from Tech Doc Item Properties

Objective

Show how Document Elements can be automatically created, using values from the active Technical Document instance which then are included, but not displayed in the editor. This example shows how an Administrator can configure the system to generate S1000D *ident*-like information at the beginning of a document by using S1000D Data Module (DM) parameters stored as Properties on the Technical Document. In this case, the DM can use the same 'Ident' information to identify the purpose of the Technical Document to authors who need to search for existing documents. This example uses XML Elements from an S1000D specification, but it can be used for any Properties added to the Tech Doc-Enabled ItemType.



Solution

Use a Content Generator for a container 'ident' Document Element that will extract parameter values from the associated Technical Document and add them to the selected node. Use CSS to target nodes within the container and hide the rendering of the corresponding content.

Note: regarding the creation of Content Generator Methods, the first commented line in the code is required - `//MethodTemplateName=CSharp:Aras.TDF.ContentGenerator(Strict);` It is used to identify the proper method template.

Content Generator

```
//MethodTemplateName=CSharp:Aras.TDF.ContentGenerator(Strict);
// a Document Element object representing the Document Element node that was
// placed in the document and associated with this Content Generator is passed
// into this Method call as 'targetElement'.  The cast here is actually not
// necessary since it's the base class type for Document Elements.  Container
// nodes will be instantiated as 'DocumentSchemaElement' objects.  This is used
// for the 'ident' node.
DocumentSchemaElement targetItem = targetElement as DocumentSchemaElement;
// The Innovator instance is needed so that we can query Properties from the
// Technical Document Item associated with this ident node
Innovator inn = this.Factory.InnovatorInstance;
// The object 'executionContext' is passed into this Method and contains the
// id of the current document.  Use it to query for the Tech Doc Properties
Item doc = inn.getItemById( "tp_Block", executionContext.DocumentId );
if(doc == null || doc.isError())
return;
// The targetElement will be null if the case above failed.  In this case, the
// input targetItem is not the Document Element Type expected
if (targetItem != null) {
// Content Generators can be executed multiple times for a node by the user
// This call will clear all children of this target node in case this has
// already been called.
targetItem.ClearChilds();
// This should be empty
if (!targetItem.IsEmpty)
{
// Use the doc Item to set properties for non-formatted text nodes
// Note the call to the constructTextNode method below
targetItem.AddChild(constructTextNode("modelic", doc.getProperty("modelic")));
targetItem.AddChild(constructTextNode("sdc", doc.getProperty("sdc")));
targetItem.AddChild(constructTextNode("chapnum", doc.getProperty("chapnum")));
targetItem.AddChild(constructTextNode("subject", doc.getProperty("subject")));
targetItem.AddChild(constructTextNode("discode", doc.getProperty("discode")));
targetItem.AddChild(constructTextNode("incode", doc.getProperty("incode")));
targetItem.AddChild(constructTextNode("itemloc", doc.getProperty("itemloc")));
}
}
} // note the extra closing brace
// This method will construct a node used for non-formatted text in a Technical
// Document.  Note that nodes of this type are different than formatted text
// nodes.  In this case, the logic within this method is the only way to create
// nodes of this type
DocumentTextNode constructTextNode(string name, string value)
{
XmlNode xmlNode = this.Factory.ConstructElementOrigin(name);
xmlNode.InnerText = value;
DocumentTextNode txtNode = new DocumentTextNode(this.Factory, xmlNode);
return(txtNode);
// Note the absence of a closing node here
```

CSS

```
.ident .modelic, .ident .sdc, .ident .chapnum, .ident .subject, .ident .discode, .ident .incode, .ident .
{
display: none;
}
```

All node names in a schema are used as class names in the generated HTML. As such, you can use the class designator – for example: '.ident' – to refer to the corresponding element in a Technical Document. In this case, the child nodes ('modelic', 'sdc', 'chapnum', etc.) are all children of the 'ident' element. So, the example CSS rule - `.ident .modelic` – targets any node with a class of 'modelic' that is a descendent of an 'ident' node. For all these types, the CSS ' `display: none` ' will cause the browser to hide the contents of the node. In this case, the text. Also, if a 'modelic' (or any of the others) were not included as a descendent of an 'ident' node, they *would* display.

Schema

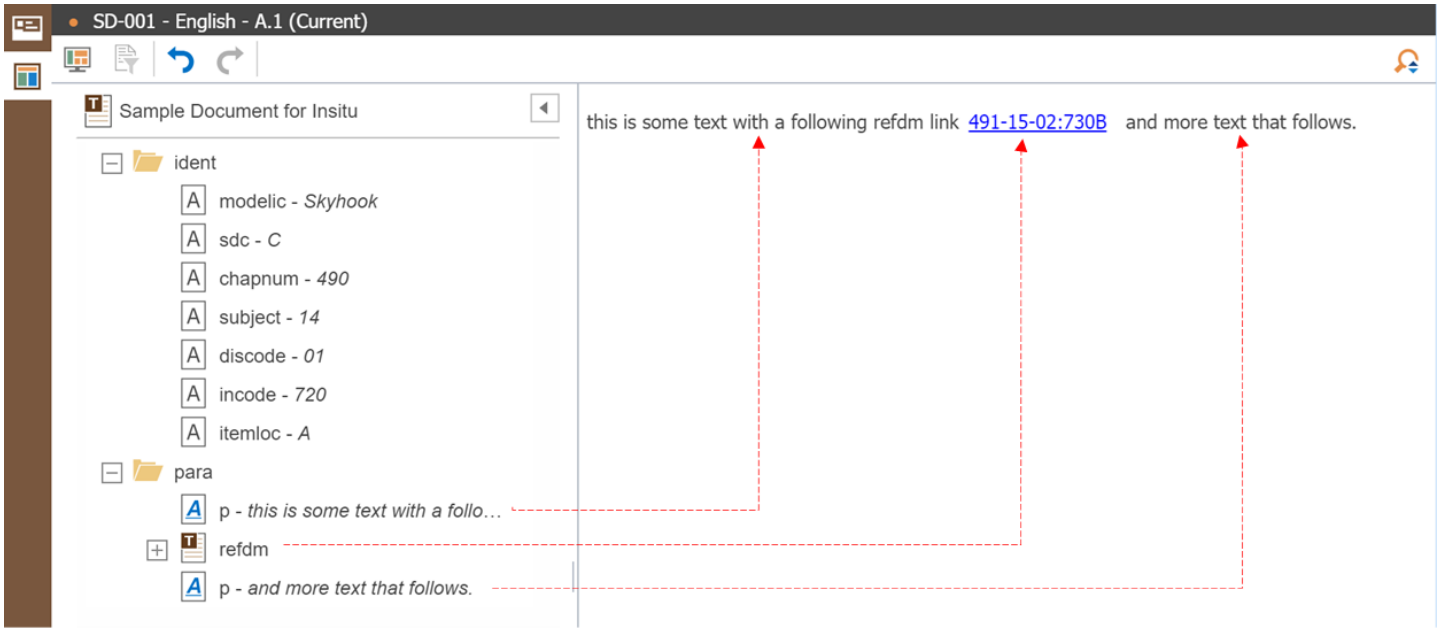
```
<xs:element name="ident">
<xs:complexType>
<xs:sequence>
<xs:element ref="modelic"/>
<xs:element ref="sdc"/>
<xs:element ref="chapnum"/>
<xs:element ref="subject"/>
<xs:element ref="discode"/>
<xs:element ref="incode"/>
<xs:element ref="itemloc"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="modelic"><xs:complexType mixed="true"/></xs:element>
<xs:element name="sdc"><xs:complexType mixed="true"/></xs:element>
<xs:element name="chapnum"><xs:complexType mixed="true"/></xs:element>
<xs:element name="subject"><xs:complexType mixed="true"/></xs:element>
<xs:element name="discode"><xs:complexType mixed="true"/></xs:element>
<xs:element name="incode"><xs:complexType mixed="true"/></xs:element>
<xs:element name="itemloc"><xs:complexType mixed="true"/></xs:element>
```

The 'ident' node is a container. Each child element uses references (for example `ref="modelic"`) to identify the corresponding node. Note that these nodes do not extend the `aras:text` type and instead are declared as a complex type with ' `mixed=true` '. These are treated as non-formatted text nodes in Tech Docs.

Example – Auto-creating inline link and text to external references:

Objective

Show how inline text content and a ‘ `refdm` ’ element, that is used to reference a separate Data Module (DM), can be authored within a Technical Document. S1000D content typically contains paragraphs of text with embedded nodes. In XML-speak, that is referred as ‘*mixed content*’. The Technical Document Framework does not support this behavior and nodes must either contain only child nodes or text; but not both. In addition, although links in Tech Docs can either refer to content within the same document or content within other Technical Documents, there is no out-of-the-box mechanism to automatically generate link text based on the content that is being referenced.



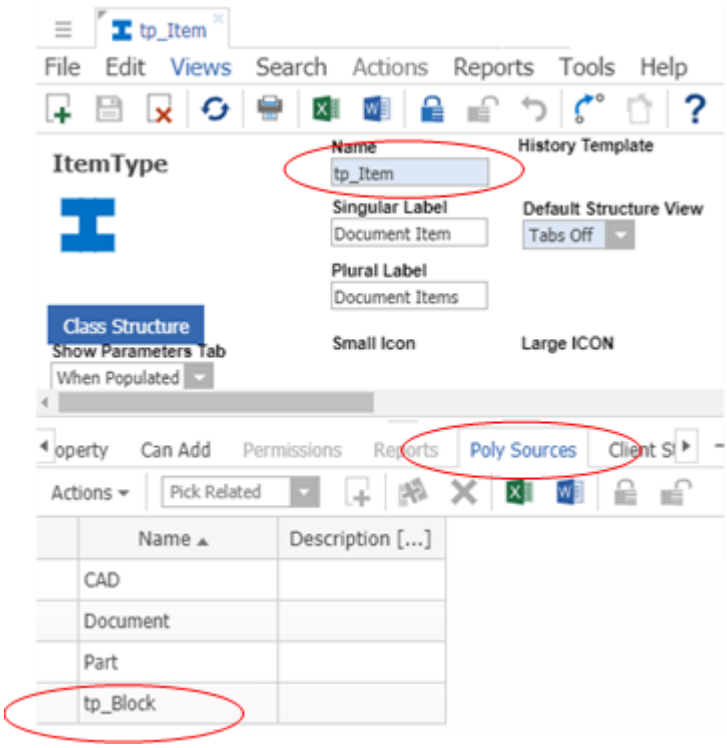
Solution

Use an ‘Item Document Element’ to show how a Document Element within a Technical Document can be used to reference another Technical Document Item in Innovator. Item Document Elements can be configured to always reference a particular ItemType. In this case, use the Technical Document ItemType (‘tp_Block’). Use a Content Generator to query content from the referenced Item to automatically generate link text and use the Content Generator API to format the generated link text as a link to the referenced document. This example also necessitated the creation of a separate (and non-S1000D compliant) node – ‘ `p` ’ that will be used for inline text content. Note that the intent here is to use this node solely for inline text content. Use CSS to target `p` and `refdm` nodes within the container and force them to render as inline elements when rendered in HTML and PDF.

Schema

```
<xs:element name="para">
<xs:complexType>
<xs:choice maxOccurs="unbounded">
<xs:element ref="p" minOccurs="0" maxOccurs="unbounded" />
<xs:element ref="refdm" minOccurs="0" maxOccurs="unbounded" />
</xs:choice>
</xs:complexType>
</xs:element>
<xs:element name="p">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:text"/>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="refdm">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemType">
<xs:choice maxOccurs="1">
<xs:element ref="p" minOccurs="1" maxOccurs="1"/>
</xs:choice>
<xs:attribute name="typeId" type="xs:string" fixed="ED9F61ADA4334D7D94361F426C081DB5" />
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

In this schema example, we are treating an S1000D `para` node as a container with child elements of type ‘ `p` ’ and ‘ `refdm` ’. In this case, `p` elements can be interleaved with `refdm` elements without any restrictions on cardinality or order. The `refdm` node is configured here to extend the `aras:itemType` element. When this is done, Document Elements of this type can reference any ItemType specified in the `tp_Item` PolyItem. For this example, I added the Technical Document ItemType – `tp_block` – to this PolyItem. For example:



Also, in the schema definition, the addition of the `typeId` attribute for the `refdm` element will identify the Item ID for the ItemType to restrict references to. In this case, I added the ItemType ID for the `tp_Block` ItemType (`ED9F61ADA4334D7D94361F426C081DB5`): thus, targeting references to only a Technical Document. *This is important because we make assumptions about the referenced Item on the Content Generator logic described below*.

CSS

```
.para .p, .para .refdm
{
display: inline-block;
padding-right: .5em;
}
```

These CSS rules target `p` and `refdm` classes only when they are descendants of a `para` element. The styling will render the content of these nodes as *inline* (instead of the default *block*). In addition, it adds some right padding to adequately space each `p` and `refdm` content.

Content Generator

```
//MethodTemplateName=CSharp:Aras.TDF.ContentGenerator(Strict);
// A Document Element object representing the Document Element node that was
// placed in the document and associated with this Content Generator is passed
// into this Method call as 'targetElement'.  The cast here is necessary since
// the base class type for Document Elements is 'DocumentSchemaElement' However,
// this Method uses an Item Document Element since the refdm node is configured
// in the schema as an extension to 'aras:item'.  When this Content Generator is
// called, the target Element will include a reference to a Technical Document
// that the user has chosen after they placed the refdm node in the document.
ItemDocumentElement targetItem = targetElement as ItemDocumentElement;
// The targetElement will be null if the case above failed.  In this case, the
// input targetItem is not the Document Element Type expected
if (targetItem != null) {
// Content Generators can be executed multiple times for a node by the user
// This call will clear all children of this target node in case this has
// already been called.
targetItem.ClearChilds();
// This should be empty
if (!targetItem.IsEmpty)
{
// The target Item, being of type ItemDocumentElement, will have a
// method (GetItemProperty) to retrieve a Property value of the referenced
// Item.  In this case, we want to build the link text based on
// added Properties to the Technical Document ItemType (e.g., chapnum,
// subject, etc.).  This code builds that link text
string linkText = targetItem.GetItemProperty("chapnum", " ") + "-";
linkText += targetItem.GetItemProperty("subject", " ") + "-";
linkText += targetItem.GetItemProperty("discode", " ") + ":";
linkText += targetItem.GetItemProperty("incode", " ");
linkText += targetItem.GetItemProperty("itemloc", " ");
// This line will construct a child 'p' node that will be used for the
// link text.
TextDocumentElement pElement = (TextDocumentElement) this.Factory.NewText("p", linkText);
// This line will format the text content of the p node as a link
// and have it reference the same Technical Document Item the user has
// selected when them placed the refdm node.  The first argument is the
// id of the Technical Document Item, the second is the id of the node
// within the document.  Since the schema locked the referenced ItemType
// to a Technical Document, we know what we can create a link to it. The
// root node in a Technical Document has the same id as the document Item
pElement.Strings[0].SetLink(targetItem.ItemId,targetItem.ItemId);
// Add the p node as a child of the given refdm node
targetItem.AddChild(pElement);
}
}
```

Example – Using Attributes to apply styling

Objective

Provide a mechanism to allow authors to size images in a technical document. By default, images are placed left justified and sized to fit the maximum space within their container. This is typically the full width of the space minus any margins.

Solution

Use attribute values together with CSS selectors to target and apply a set of style rules for a selected image. This technique can be applied to any Document Element and any style configuration, including positioning. This example demonstrates two techniques. The first example will show how to use an attribute to identify a select rule. The second uses attribute values to set specific styles typed by the author. The first option is probably the most useful as it is easy for the end user. The second option can provide for greater flexibility (and unwanted consequences)

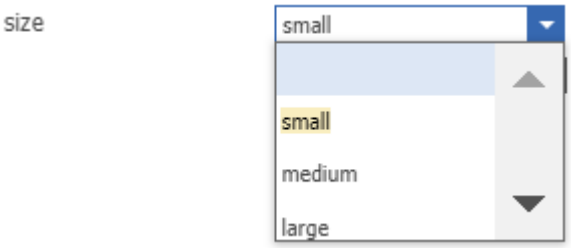
Option A

In this solution an attribute is added to the Image Document Element – `Graphic` . The attribute name is ‘ `size’` and there will be three allowed choices: ‘ `small` ’, ‘ `medium` ’, and ‘ `large` ’. Here is the related schema:

Schema

```
<xs:element name="Graphic">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:imageType">
<xs:attribute name="size" type="ImageSizeAttType"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:simpleType name="ImageSizeAttType">
<xs:restriction base="xs:string">
<xs:enumeration value="small"/>
<xs:enumeration value="medium"/>
<xs:enumeration value="large"/>
</xs:restriction>
</xs:simpleType>
```

In this sample, there is an added ‘ `size` ’ attribute to the `Graphic` Document Element and, using a restriction, specified the allowed values of ‘ `small` ’, ‘ `medium` ’, and ‘ `large` ’. The Attribute editor will display these values in a picklist:



CSS:

```
.Graphic[size='small'] img
{
max-width: 100px;
}
.Graphic[size='medium'] img
{
max-width: 300px;
}
.Graphic[size='large'] img
{
max-width: 600px;
}
```

These style rules target the `img` tag using the parent ‘ `Graphic` ’ container with a ‘ `size` ’ attribute of either ‘small’, ‘medium’, or ‘large’. Of course, other values can be added as well. In addition, other styles can be applied. The example is really showing how the author can use attribute values to assign styling.

Option B

In this alternative example, an option is provided to allow the user to set a specific value for size. Here is the related schema:

Schema

```
<xs:element name="Graphic">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:imageType">
<xs:attribute name="style" type="xs:string"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

In this sample, there is an added ‘ `style` ’ attribute to the Graphic Document Element. Note the name ‘style’ is important here. The Attribute editor will display the attribute using a simple string editor:

In this example, we use the value of the attribute to identify a variable – ‘ `width` ’ in this case and its value. This is applied to the width style rule in the following CSS:

CSS

```
.Graphic
{
width: var(width);
}
```

Of course, other style rules can be added as well. This example shows an approach that adds flexibility but requires the user to know more about what they’re typing – Use with caution.

Example – Populating PDF header content from attribute data

Objective

Automate the creation, placement, and content for a PDF page header using information from the technical document. In this example, the user wants to use information stored in the attributes of a particular Document Element in the PDF Header of a published document.

Solution

Use CSS rules and functions that apply to page-based media to create the PDF header and add static and custom content. Note that this example uses Attributes for the source for some of the content, but another element content can be used as well. For example, the text in a *Title* Document Element.

In this example, there is an added ‘ `chapnum` ’ and ‘ `modelic` ’ attribute to a Document Element named ‘ `ident` ’. These values can be populated by the Attribute Editor or added by a Content Generator like:

Content Generator

```
// Use Property values from the Technical Documents Item for attribute values
targetItem.SetAttribute("modelic", doc.getProperty("modelic"));
targetItem.SetAttribute("chapnum", doc.getProperty("chapnum"));
```

CSS

```
@page
{
size: 8.5in × 11.0in;
margin: 40pt 0 40pt 0;
padding-bottom: 1em;
padding-top: 1em;
@top-right
{
content: "Test " string(modelic) ", " string(chapnum);
color: white;
font-family: Tahoma, Geneva, sans-serif;
background-color: #8C0827;
padding-right: 3em;
}
}
.ident
{
string-set: modelic attr(modelic), chapnum attr(chapnum);
}
```

The Style applied to the PDF output uses `@page` and the margin identifier `@top-right` to position the content for the header. In this case, the content is derived from two string sets: ‘ `modelic` ’ and ‘ `chapnum` ’; which are populated from the attributes of the same name in the ‘. `ident` ’ class DOM elements using the `attr()` function. Note that only one `string-set` rule can be used in a group. In this case, we set the value of both string set variables separated by a comma.

PDF

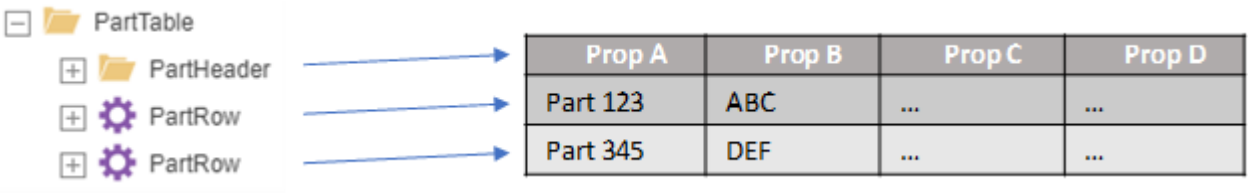
Here is the header portion of a generated PDF document. Note that the Header text is as follows:

Static Text – ‘**Test**’, followed by the value of the `modelic` attribute (in this case ‘**Skyhook**’), followed by a comma and space – ‘**,** ‘, followed by the value of the `chapnum` attribute (in this case ‘**490**’).

Example – Creating a Table, in which each row points to individual Parts

Objective

Provide a means of auto-generating Table Rows where each Row Object is an Item Document Element, ‘pointing to’ Part Items and the cells of the Row contain individual Part Item Property data.



Solution

For this solution, a typical Table Document Element can’t be used because it derives from an `aras:table` element. The same is true for `aras:row` and `aras:cell` elements. Also, the first row of the table – the header row - needs to be auto-created with a cell for each column. Finally, the ‘Row’ Document Elements need to be ‘ `ItemDocumentElement` s since each added row should be associated with a selected Item (Part in this example).

Schema

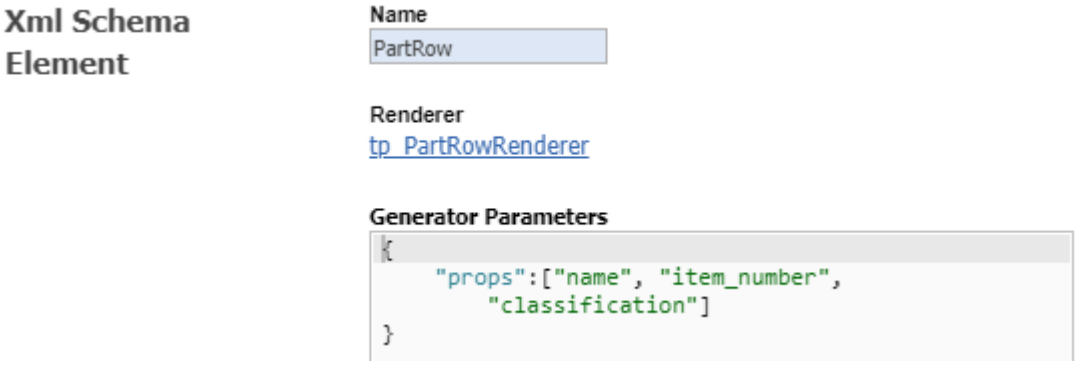
For this structure, the top-level node – `PartTable` – is the container with a *required* first child of `PartHeader` , and any number of `PartRow` s. Each `PartRow` has any number of `PartCell` s; which contain a single Text Document Element. Note the use of default `style` attributes in this schema. They use ‘display: table/table-row/table-cell’ values, which when used within `<div>` style attributes treat the `PartTable` like a standard HTML table, `PartHeader/Row` s like table rows, and `PartCell` s like table cells, etc.

```
<xs:element name="PartTable">
<xs:complexType>
<xs:sequence>
<xs:element ref="PartHeader" minOccurs="1" maxOccurs="1"/>
<xs:element ref="PartRow" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="style" default="display:table"/>
</xs:complexType>
</xs:element>
<xs:element name="PartHeader">
<xs:complexType>
<xs:sequence>
<xs:element ref="PartCell" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="style" default="display:table-row" />
</xs:complexType>
</xs:element>
<xs:element name="PartRow">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemType">
<xs:choice maxOccurs="unbounded">
<xs:element ref="PartCell" minOccurs="0" maxOccurs="unbounded"/>
</xs:choice>
<xs:attribute name="style" default="display:table-row"/>
<xs:attribute name="typeId" type="xs:string" fixed="4F1AC04A2B484F3ABA4E20DB63808A88"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="PartCell">
<xs:complexType>
<xs:sequence>
<xs:element ref="Text" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
<xs:attribute name="style" default="display:table-cell"/>
</xs:complexType>
</xs:element>
```

Content Generator

There needs to be a Content Generator for the `PartHeader` Document Element, which will be called immediately after the `PartTable` Document element is placed. This Method will add the `PartHeader` , `PartCell` s, and `Text` for all columns required for the table. Similarly, a Content Generator needs to be created for the `PartRow` Document Elements; which will point to some selected Part (given that they extend `aras:itemType` and the ‘ `typeId` ’ is fixed to the ItemType Id of Part. In Innovator 11.0 SP15, the ability to pass JSON-formatted string content to the Content Generator Method was added. We can use this to pass in the names of Columns and Properties to

add for the PartHeader and PartRow Content Generators, respectively. For the `PartHeader` Document Element, the following JSON is used for this example:



Note the 'Generator Parameters' form element. It uses a JSON-configured editor so that JSON elements can be color-coded as shown. Opening the XML Schema Element exposes this form. In this case, a single array element is used – `props` – with a list of all Properties names to use for the Part Row Contents. The number of array elements determines the number of columns. A similar construct is used for the `PartHeader` Content Generator; but using Column Names as the array elements.

```
//MethodTemplateName=CSharp:Aras.TDF.ContentGenerator(Strict);
// The Target Items should be a PartRow Document Element
ItemDocumentElement targetItem = targetElement as ItemDocumentElement;
if (targetItem != null) {
targetItem.ClearChilds();
// if referenced item was set, then
if (!targetItem.IsEmpty)
{
// The configured parameters should contain an array labeled 'props'
// that contains an array of string values to indicate the properties
// to use for the associated referenced Item
if (!this.GenerationParameters.ContainsKey("props"))
{
throw new InvalidOperationException("'props' generation parameter was not passed.");
}
// Create cells for each property name configured as parameters
System.Collections.ArrayList props = (System.Collections.ArrayList) this.GenerationParameters["props"];
foreach(String propName in props)
{
DocumentSchemaElement cellElement = (DocumentSchemaElement) this.Factory.NewElement("PartCell");
cellElement.AddChild(this.Factory.NewText("Text", targetItem.GetItemProperty(propName, " ")));
// Add as a child
targetItem.AddChild(cellElement);
}
}
```

Renderer

The following Editor renderer will display the part number for the node text in the Tree View. This is helpful in identifying the referenced Part in the Tree. Note that this example is using data in the content for the part number, so it assumes that the part number (in this case) is stored in the second column. Note that Render Methods are described in Section Document Element Configuration.

```
return
{
constructor: function (args)
{},
// Set the name of the tree node based on the referenced Part
GetTreeName: function(/*WrappedObject*/renderObject)
{
// Get the of the referenced Part
// return the content of the second column - Part Item Number
if(renderObject.origin.childNodes.length > 1)
return renderObject.origin.childNodes[1].text;
else
return "Part";
}
};
```

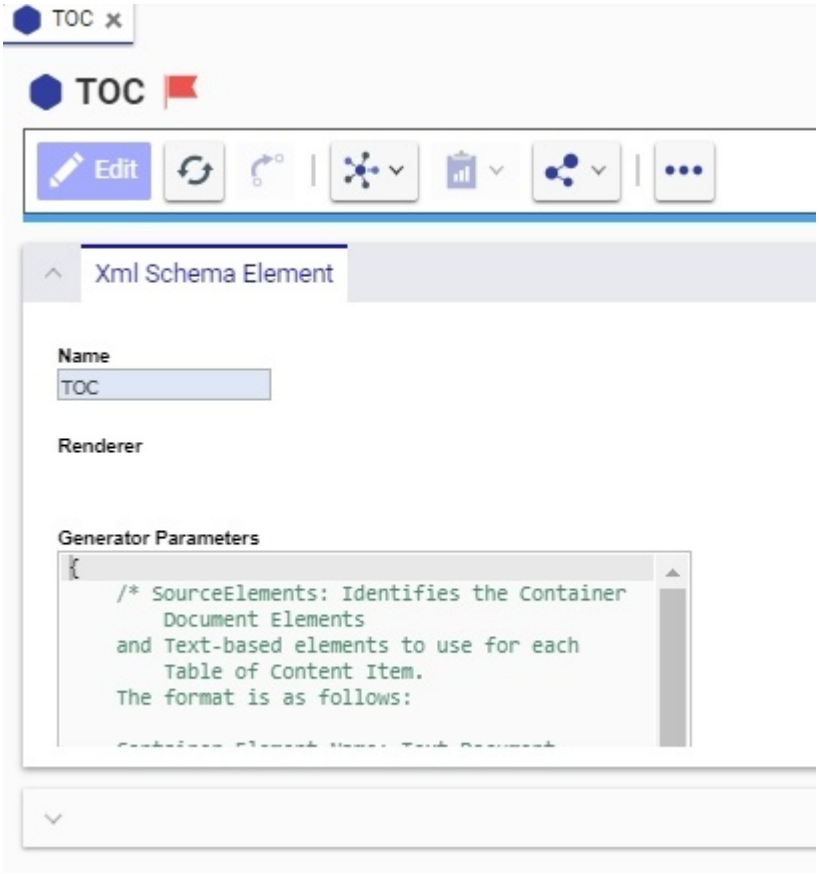
CSS

The CSS targets each Document Element as follows. Note the use of ‘ `border-collapse` ’ for the `PartTable` class.

```
.PartCell {
border: 1px solid black;
padding-left: 1em;
padding-right: 1em;
padding-top: 0em;
padding-bottom: 0em;
}
.PartHeader .PartCell
{
background: grey;
text-align: center;
}
.PartHeader .PartCell .Text
{
font-weight: bold;
font-size: 1.2em;
color: white;
}
.PartCell .Text
{
padding: 0em;
}
.PartTable {
border-collapse: collapse;
}
```

Configuring a Table of Contents

The Content Generator enables you to create and modify the Table of Contents for a document using the TOC XML schema element and the `tp_TOCContentGenerator` method. The TOC XML Schema element contains the parameters used to create a Table of Contents as shown in the following screenshot.



You can specify the:

- Chapter and Section headings.
- Maximum depth of headings that appear in the Table of Contents.
- Name of the document element to create for each TOC item.

The **SourceElements** JSON Object Parameter identifies the Container Document Elements and Text-based elements to use for each Table of Content Item. The format is as follows:

Container Element Name: Text Document Element Name

For example, the following searches for ‘Chapter’ and ‘Section’ Document Elements and will use the content of the first ‘Title’ Document Element within each for the TOC text.

```
"SourceElements": {
"Chapter": "Title",
"Section": "Title"
},
```

The **MaxLevel** Parameter specifies the maximum depth to use for the Table of Contents. Valid values are equal to or greater than 1. The following example sets the depth to 3; meaning that the TOC Generator will search to a depth level of 3 for child and grandchild Chapter and/or Section Documents for the content of the TOC.

```
"MaxLevel": 3,
```

The **TOCItemName** Parameter specifies the Name of the Document Element to create for each TOC Item generated. The following example will create a 'TOC-Item' Document Element for each TOC Item it creates:

```
"TOCItemName": "TOC-Item"
```

The values you specify should be passed to the `tp_TOCGenerator` method in JSON format. The **Generator Parameters** property accesses the specified values. It's critical that JSON formatting be accurate when setting these properties. The following example show the proper configuration for the example settings discussed above:

```
{
"SourceElements": {
"Chapter": "Title",
"Section": "Title"
},
"MaxLevel": 3,
"TOCItemName": "TOC-Item"
}
```

Important

In the example above, note the use of the outer '{'s and the use of '{'s for the **SourceElements** JSON Object. Also note the use of ','s for separating values and the use of ":" for separating name and value pairs.

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.

Managing Languages

The Aras Technical Documentation solution gives you the option of specifying the language used for creating the documents. By default, the language is set to English.

Important

If the default language is changed, external links will disappear from existing Technical Documents. The language should only be changed if there are no Technical Documents in the database.

To edit the language used for creating Technical Documents:

1. Confirm that Language and Locale preferences are set up in the database.

For more information on setting up Language and Locale preferences, see *Aras Innovator - Configuring Internationalization*.

2. Log into Aras Innovator as an administrator.
3. Go to **Administration\Variables\Search** in the TOC and open the **tp_DefaultLanguage** item for editing.
4. Change the **Value** to the desired language.

Important

The Value must match the name of the corresponding Language item.

Attributes Validation Support

This appendix describes the data types most used in Technical Documentation.

String Data Types

The following table describes the available string data types. All listed types are processed on the client as “string” types. They are only validated against defined restrictions.

Type Name	Description	Schema validation	Attribute dialog validation
ENTITIES		Yes	No
ENTITY		Yes	No
ID	A string that represents the ID attribute in XML (only used with schema attributes)	Yes	No
IDREF	A string that represents the IDREF attribute in XML (only used with schema attributes)	Yes	No
IDREFS		Yes	No
language	A string that contains a valid language id	Yes	No
Name	A string that contains a valid XML name	Yes	No
NCName		Yes	No
NMTOKEN	A string that represents the NMTOKEN attribute in XML (only used with schema attributes)	Yes	No
NMTOKENS		No	No
normalizedString	A string that does not contain line feeds, carriage returns, or tabs	No	No
QName		Yes	No
string	A string	Yes	Yes
token	A string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces	No	No
anyURI		Yes	Yes
base64Binary		Yes	No
hexBinary		Yes	No
NOTATION		Yes	No

Date and Time Data Types

Type Name	Description	Schema validation	Attribute dialog validation
date	Defines a date value	Yes	Yes. Works like dateTime.
dateTime	Defines a date and time value	Yes	Yes
duration	Defines a time interval	Yes	No
gDay	Defines a part of a date - the day (DD)	Yes	No
gMonth	Defines a part of a date - the month (MM)	Yes	No
gMonthDay	Defines a part of a date - the month and day (MM-DD)	Yes	No

gYear	Defines a part of a date - the year (YYYY)	Yes	No
gYearMonth	Defines a part of a date - the year and month (YYYY-MM)	Yes	No
time	Defines a time value	Yes	No

Numeric Data Types

1. Numeric types are processed as a ‘long’ on the client and not validated against type boundaries.
2. There is a limitation within the .Net processing logic for ‘default/fixed’ values validation. If the `min/maxExclusive` restriction is specified and it has a value that exceeds the type of boundary by one (‘1’), then system will show a validation error. For example: **byte** has a range of values -128 .. 127. If the `maxExclusive` restriction is given a value of ‘128’ then a validation error will be shown.
3. The Javascript `parseInt/Float()` method is used to process strings for client-side validation within the Attributes Dialog. Strings that contain invalid characters for a numeric value will not show as invalid (warning icon).
4. Restriction ' `totalDigits` ' is not processed for integer, long, int, short, and byte types.

Type Name	Description	Schema validation	Attribute dialog validation
byte	A signed 8-bit integer	Yes [2]	Yes [1]
unsignedByte	An unsigned 8-bit integer	Yes [2]	No
short	A signed 16-bit integer	Yes [2]	Yes [1,3]
unsignedShort	An unsigned 16-bit integer	Yes [2]	No
decimal	A decimal value	Yes [2]	Yes [3]
int	A signed 32-bit integer	Yes [2]	Yes [1,3]
unsignedInt	An unsigned 32-bit integer	Yes [2]	No
integer	An integer value	Yes [2]	Yes [1,3]
long	A signed 64-bit integer	Yes [2]	Yes [3]
unsignedLong	An unsigned 64-bit integer	Yes [2]	No
negativeInteger	An integer containing only negative values (..,-2,-1)	Yes [2]	No
nonNegativeInteger	An integer containing only non-negative values (0,1,2,..)	Yes [2]	No
nonPositiveInteger	An integer containing only non-positive values (..,-2,-1,0)	Yes [2]	No

positiveInteger	An integer containing only positive values (1,2,..)	Yes [2]	No

Type Name	Description	Schema validation	Attribute dialog validation
double		Yes	Yes [3]
float		Yes	Yes [3]

Miscellaneous Data Types

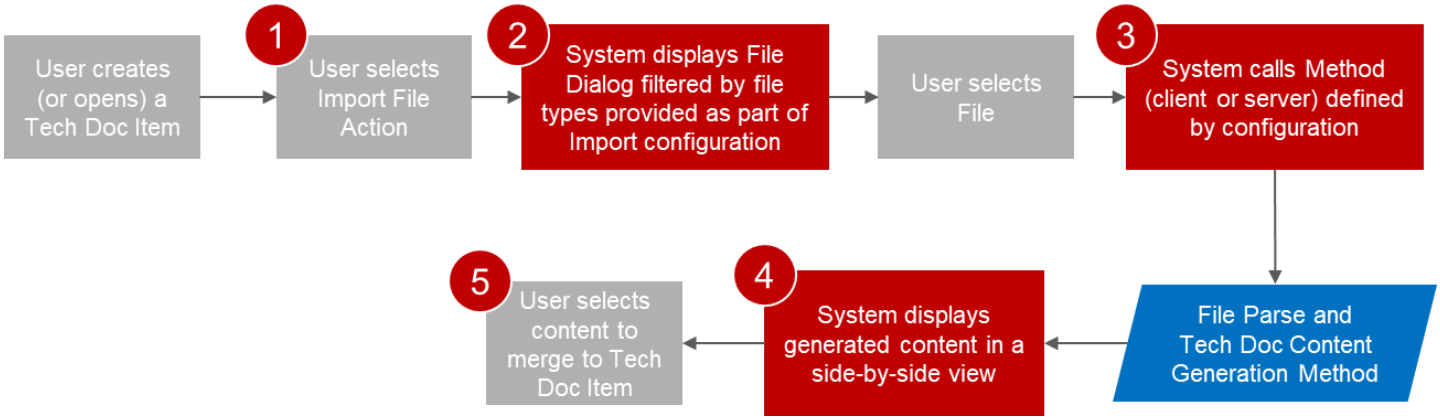
[1] Value can be represented as ‘0’-‘1’ or ‘true’-‘false’ pairs in the XML Schema. However, since the editor field on the client is represented as a checkbox we support only ‘true’-‘false’ pair.

Type Name	Description	Schema validation	Attribute dialog validation
boolean		Yes	Yes [1]

File Import API

Overview

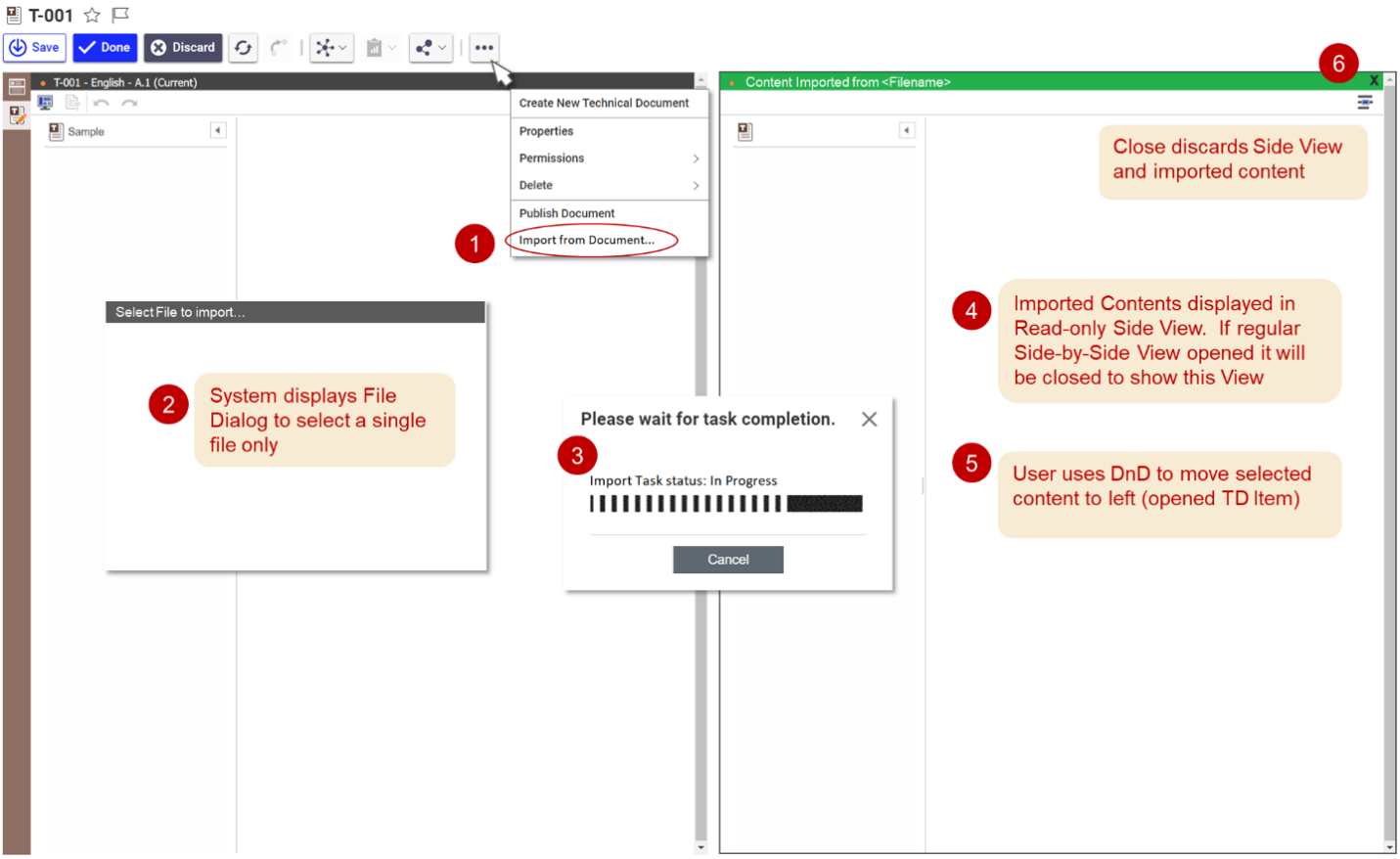
The File Import API provides a framework for creating technical document content programmatically by extracting data from a selected file. The process works as follows:



1. Using an existing or a new Technical Document, the User initiates the File Import operation by selecting an Action. Note that this Action is only enabled when the Technical Document is in Edit mode.
2. The system prompts the user with a File Dialog, filtered based on configured Document Types (extensions)
3. After the User chooses a single file, the system executes a custom Method that parses the data from the selected file and generates Document Element content

4. The system displays the generated content in a side-by-side view (right side) where the content is read-only
5. The user can copy/merge all generated content with a single operation or can selectively drag and drop individual Document Elements from the view to the opened Technical Document
6. Closing the right-side view discards the generated content

A similar description is provided in the following diagram:



The custom Method (see Define the Import Method in this Appendix) is responsible for reading the content from the file and use the Content Generator API (see Content Generator API) to programmatically generate Document Element content based on the parsed file data. Thus, custom implementations need only provide the logic to read the specific file format content and use the associated Content Generator Classes/Methods to create document content.

Creating a File Importer

The following example shows how to set up File Import for MS Word (.docx) files.

Important

Before creating a File Import Method it's important to understand the Document Schema being used for the generated content. Generated Document Elements use a specified Document Type. System Administrators should understand the structure and limitations of the configured schema before implementing the logic to generate content for that schema.

Initializing the Methods for the Import File Action

System Administrators provide the following as part of File Import Configuration:

1. Supported Document type(s). This is specified using file name extensions. The system will use this information to filter the files displayed in the File Dialog.
2. Identify whether the system should provide the custom Method with file content in ASCII text format or as binary Base64 format. The latter is required for non-ASCII files.
3. Identify whether the system should store the selected file in the vault. Imported files can first be uploaded and stored in the vault. End-user customizations are responsible for managing these File Items.
4. Identify the Method that will be executed to parse the file data and return the generated Document Elements

Update Method cui_tdf_ivicb_fileimport_init

Method `cui_tdf_ivicb_fileimport_init` is used to initialize the Configurable UI (CUI) action for File Import. Within this Method is the JavaScript software to initialize the `ImportManager` object. The `configs` array parameter for the `ImportManager` identifies the configuration for each supported File Type. It needs to be populated. As an example, the following sample code initializes the Import Manager with two file types: JSON, and MS Word.

```
if (!importManager) {
Promise.resolve(window.TDF || import('../jsBundles/tdf.es.js')).then(() => {
importManager = additionalData.importManager =
new window.TDF.import.managers.FileImportManager({
aras: window.aras,
configs: [{
extensions: ['.json'],
processing: {
methods: ['fi_tdf_importcontent_json'],
},
},
{
extensions: ['.docx'],
processing: {
methods: ['td_WordImporter'],
},
}
]
});
```

The `configs` property must be defined to include one or more JSON objects with the following properties:

- `extensions` : array of strings for all related file extensions. Each string identifies the file extension starting with a '.'
- `processing` : Identifies one or Method names that will be called in succession after the User has chosen a File for import
- `reading:` Optional (see below)
- `contextData:` Optional (see below)

Update Method cui_tdf_ivicb_fileimport_click

Method `cui_tdf_ivicb_fileimport_click` is called when the Import File Action is executed. Within this Method is the JavaScript software that extends the `ImportManager` configuration with data used for calling the configured Methods. The `reading` parameter for the `ImportManager` identifies how the configured Method should be called. For example, the following sample code configures the Import Manager to read the selected File data, provide it as Base64 content to each configured Method, and *not* save the file to the vault.

```
if (isEditorActive) {
const { tdfSettings } = tabController.shareData;
const xmlSchema = window.aras.getItemProperty(window.item, 'xml_schema');
const { importManager } = additionalData;
importManager
?.import(null, {
reading: {
readContent: true,
outputFormat: 'base64',
saveFile: false,
},
contextData: {
xmlSchema: xmlSchema,
contentBuilder: tdfSettings.contentBuilderMethod,
},
})
.then((result) => {
if (result) {
const editorWindow =
tabController.views['editor_container'].domNode
.contentWindow;
editorWindow.openImportPanel(result);
}
});
```

The `reading` property must be defined to include one or more JSON objects with the following properties:

- `readContent` : If ‘true’, the file contents are read from the selected file and provided as input to the configured Method(s).
- `outputFormat` : Either ‘text’ or ‘base64’(default). This determines the format of the input data provided for the configured Method(s). `saveFile:` if ‘true’, the selected file is saved in the Vault. If ‘false’, it won’t.
- `contextData:` Contains configurable options for the schema and Content Builder Methods. In this example, the `xmlSchema` is retrieved from the opened Technical Document and the `contentBuilder` uses the default. It is best to leave these values as provided in the sample.

Important

It is possible to define both of these parameters when modifying the Method cui_tdf_ivicb_fileimport_init (see above). In this case, separate reading and contextData can be defined for each supported file type.

Define the Import Method

The following example shows a server-side Method used to read data from a MS Word (docx) file and generate Document Elements. Note that the details related to parsing/extracting content from the input file are not included. The code is used to describe how to access the input data and how to use the Schema Factory and Content Generator API to create Document Elements and return the results in the expected format. Much of the code can be copied and used as is.

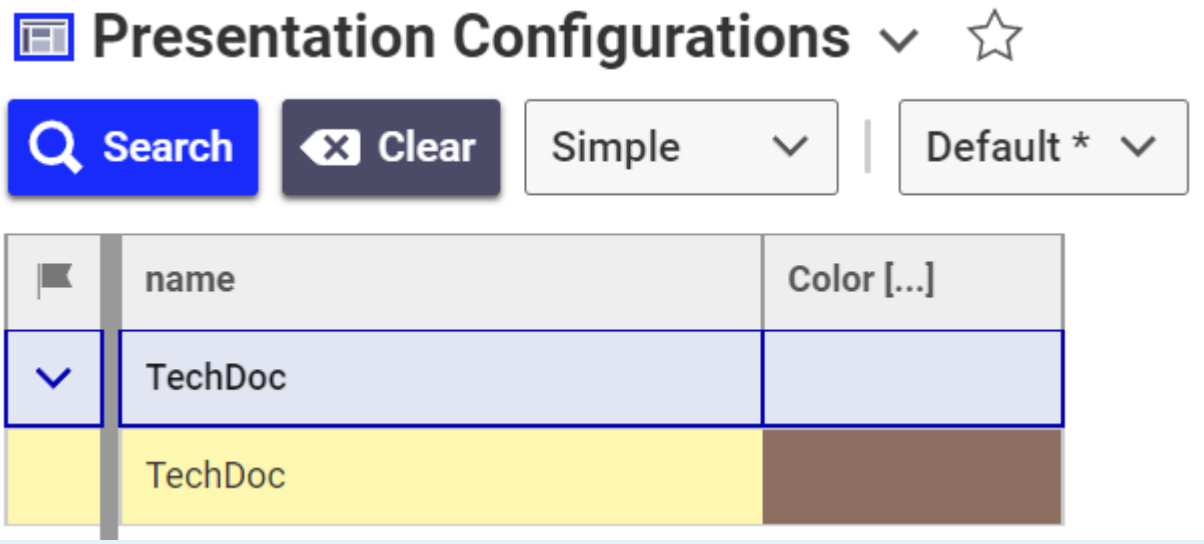
```
return Execute(this, RequestState);
}
internal static Item Execute(Item sourceItem, Aras.Server.Core.IContextState requestState)
{
Innovator innovator = sourceItem.getInnovator();
string languageCode = innovator.getI18NSessionContext().GetDefaultLanguageCode();
var executionContextData = Aras.TDF.Base.ComplexObjectJsonDeserializer.Deserialize(sourceItem.getProperty
var contextData = executionContextData["context"] as Dictionary<string, object>;
string schemaId = contextData["xmlSchema"] as string;
string contentBuilder = contextData["contentBuilder"] as string ?? "tp_DocumentGet";
var importData = executionContextData["input"] as Dictionary<string, object>;
var fileContent = Convert.FromBase64String(importData["content"] as string);
var memStr = new MemoryStream(fileContent);
Aras.TDF.Base.SchemaElementFactory factory = CreateFactory(sourceItem, innovator, schemaId, contentBuilde
factory.CustomContentEnabled = true;
factory.DefaultLanguageCode = languageCode;
using (WordprocessingDocument wordprocessingDocument = WordprocessingDocument.Open(memStr, true))
{
mainDocPart = wordprocessingDocument.MainDocumentPart;
Body body = mainDocPart.Document.Body;
var rootElement = factory.NewBlock() as Aras.TDF.Base.DocumentSchemaElement;
…
}
var importedData = factory.ExportElementToXml(rootElement);
return innovator.newResult(importedData);
}
public static Aras.TDF.Base.SchemaElementFactory CreateFactory(Item thisItem, Innovator innovator, string
{
if (thisItem == null)
{
throw new ArgumentException("Processing error for File Import");
}
var executionContext = new Aras.TDF.Base.SchemaElementExecutionContext(thisItem.getID(), thisItem.getType
return new Aras.TDF.Base.SchemaElementFactory(innovator, schemaId, executionContext);
```

1. The context of the Method (`this`) is the Document Item opened or created as part of the Use Case to initiate the File Import Action (see above). `RequestState` is not used or needed for this implementation. Note the missing initial curly brace – it is based on the Method template chosen.
2. Input for this Method is passed as a JSON-encoded string. This line de-serializes the data into Objects for ‘ `input` ’ and ‘ `context` ’.

3. `context` data is a series of name/value pairs that were defined in the client Action Method (see above). In this case, it includes the XML Schema (DocumentType) and the Content Builder. For the purposes of File Import, this data is only required when instantiating the Schema Factory (#6 below) used for creating Document Elements.
4. `input` data describes the file being imported. This includes the file name without the path (‘ `name` ’), file extension string (‘ `extension` ’), file size (‘ `size` ’), MIME Type (‘ `mimeType` ’), and file content (‘ `content` ’).
5. Because the file data was passed as a Base64 stream (see above), it needs to be converted back to the format as it was defined in the file.
6. The embedded method – `CreateFactory()` – shows the code necessary for creating the Schema Factory used to create Document Elements. This code can be used as is.
7. The Factory object is used as a mechanism to create individual Document Elements. The root element must be a `Block` as shown on this line.
8. The Factory object is used to convert the entire hierarchy of Document Element Objects into the XML representation needed for the expected return of this Method. **Note: The XML for the Technical Document content should be generated using the Schema Factor only to ensure compatibility with new and future versions of the Technical Document Editor.**
9. The generated XML is included in the returned result. This is used for display in the right-side view within the Document Editor.

Enable the Menu Item

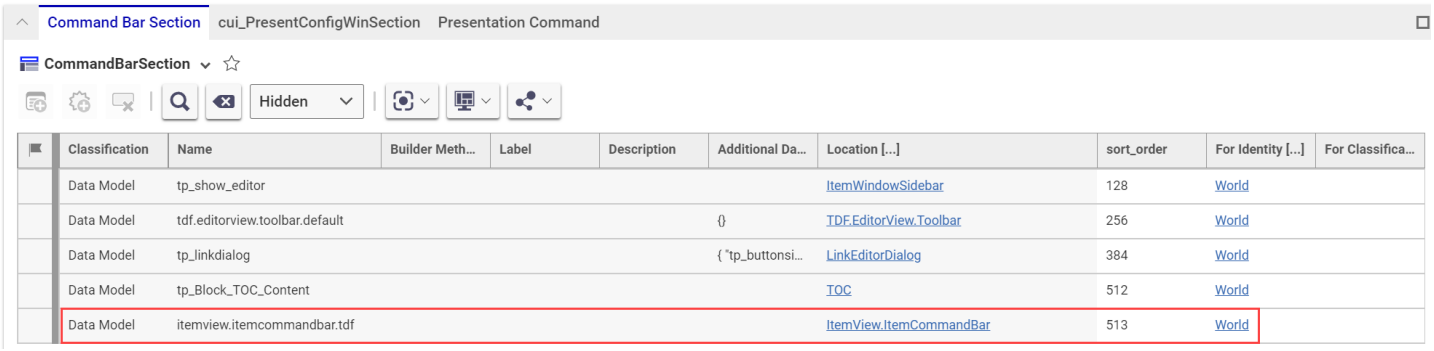
The **Import from Document…** Menu Item needs to be enabled using the Configurable User Interface configuration. The menus and toolbar buttons for the Technical Document Editor are configured within the **TechDoc** Presentation Configuration.



Important

The Presentation Configuration ItemType is not added to the Innovator Table of Contents (TOC) by default. To add it, so that a search for the **TechDoc** Presentation Configuration can be opened for Edit, use the TOC Editor as described in the Configurable User Interface Administrator Guide.

Edit the **TechDoc** Presentation Item to add the **itemview.itemcommandbar.tdf** Command Bar Section.



Select the **Add CommandBarSection** button, search for and add the **itemview.itemcommandbar.tdf** Command Bar as shown, setting the Sort Order and For Identity as required.

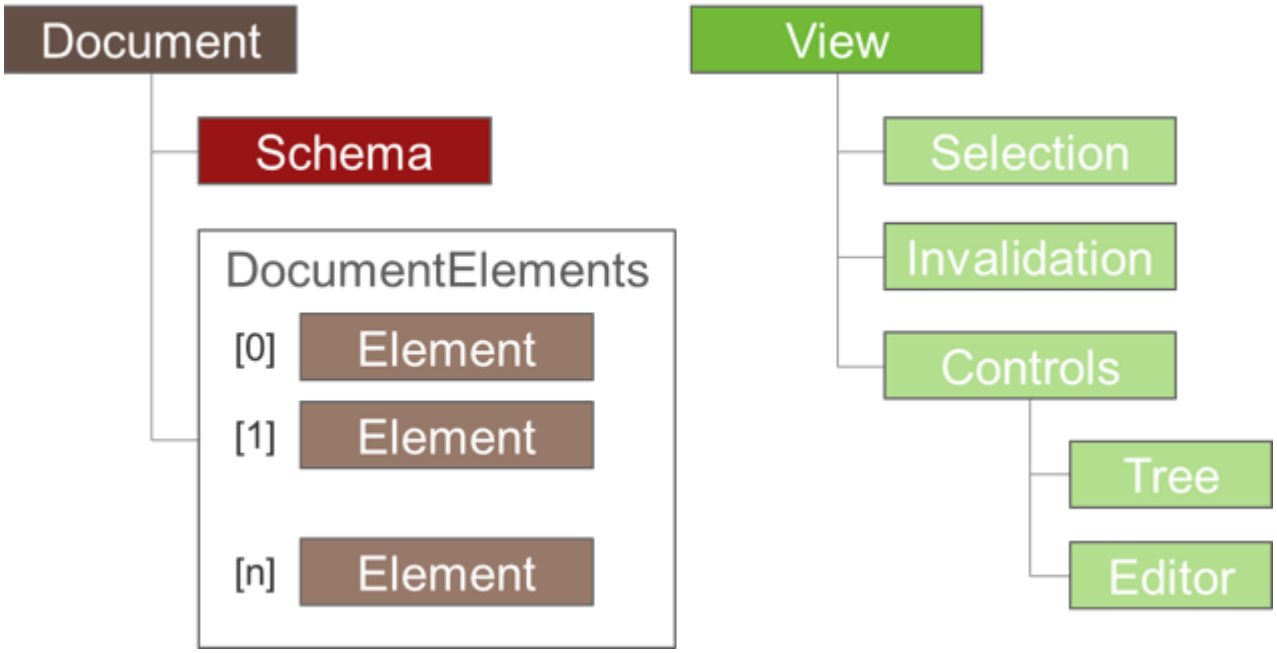
Client API

Overview

The Technical Document Editor is implemented using JavaScript classes/methods that provide the ability to, for example, add new Document Elements, update styling, link content to selected Items, manipulate content, etc. The Technical Documentation Client Application Programming Interface (Client API) exposes the same underlying functionality in a simplified interface so that it can be used to add custom features within the Editor. This API is made available in user-developed Methods attached to new toolbar items, menus, or key processing events to customize and extend the operation of the Editor. Toolbars, menus, and key events are configured using the Configurable User Interface (CUI) (see the Configurable User Interface Administrator Guide for more information on CUI and how to use this framework to custom the User Interface within Aras Innovator).

Scope

The Client API consists of 3 main classes: Document, Schema, and View.



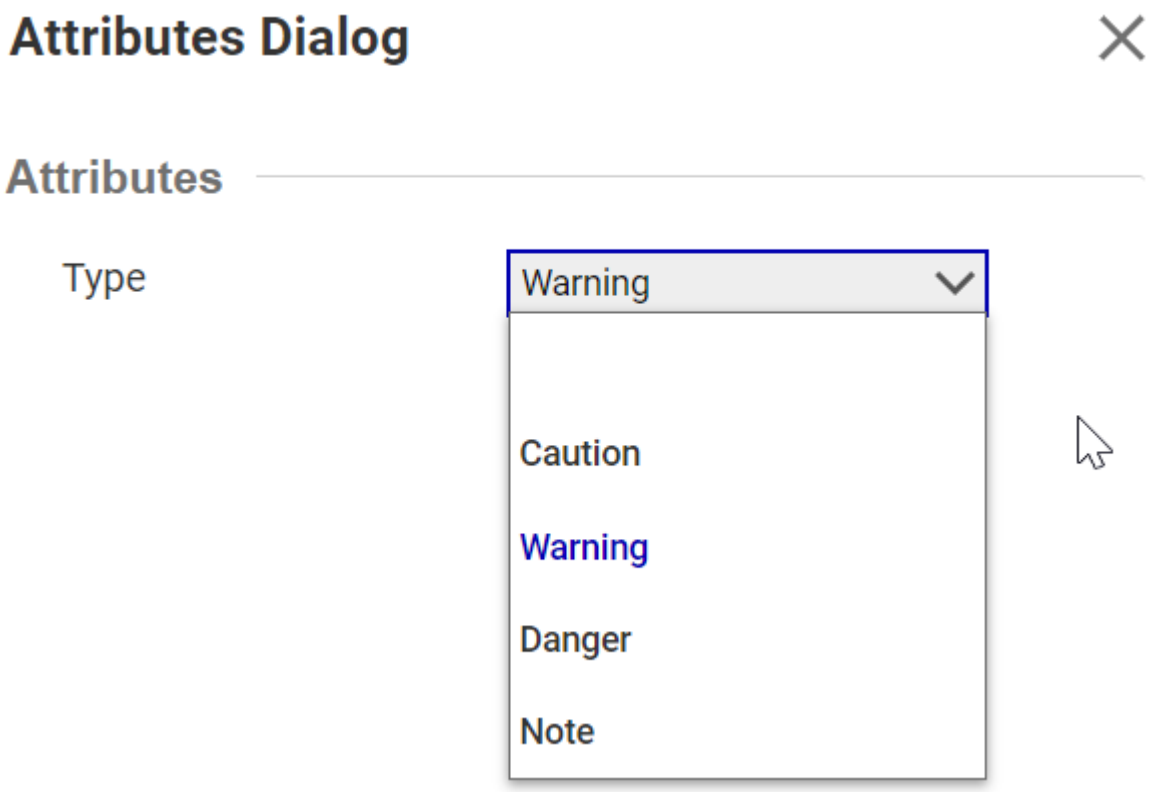
The **Document** class provides access to all the existing content within an open Technical Document, provides the ability create new Document Elements, and remove (or move) existing Document Elements. The **Schema** class provides the ability to query the associated Document Schema to check or validate content operations against the user-defined XML Schema. The **View** class provides access to selected Elements. When used along with custom logic implemented with JavaScript Methods, the breadth of customization options is extensive.

Customization Examples

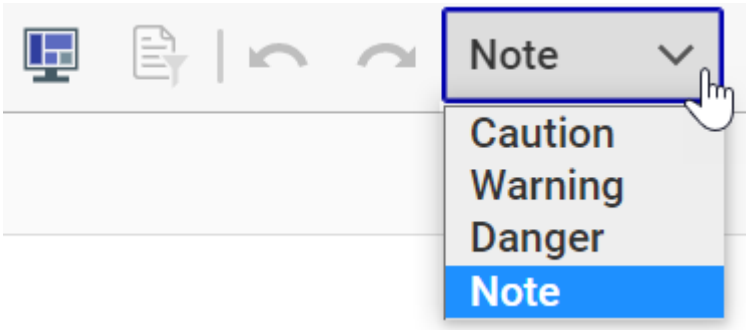
To get a sense of the possibilities of using the Client API, this section provides some samples that illustrate the ways in which the API can be leveraged for specific Use Cases.

Setting Document Element Attributes

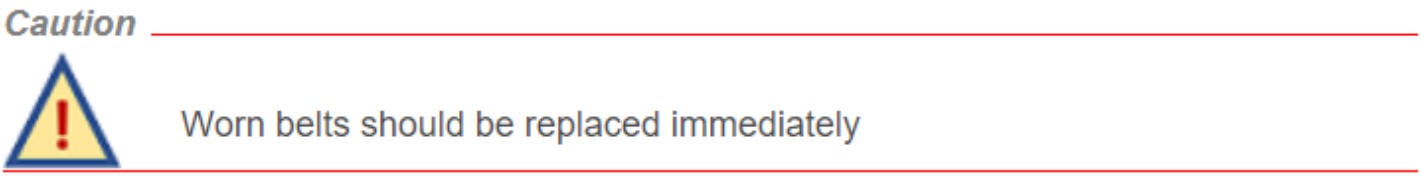
Document Elements can have one or more Attributes that can be used to characterize them, or otherwise provide additional semantic information. Attributes can also be used to distinguish styling (see Section Example – Using Attributes to apply styling). In the following example, a *Note* Document Element has 4 Attributes; the values of which help identify its *Type* and are used to isolate unique Styling so that each Note Type can be rendered appropriately. These enumerations are displayed in the Attributes dialog as valid selections for this Attribute:



This example adds a dropdown list control to the Technical Document Editor toolbar that is enabled when the user selects a Note Document Element. This can be used in lieu of the Attributes Dialog as a more convenient mechanism for settings these values.



Note Type Selection Dropdown A sample of the rendered Note Document Element types are as follows:



‘Caution’ Type example

‘Note’ Type Example
Schema

```
<xs:element name="Note">
<xs:complexType>
<xs:sequence>
<xs:element ref="Text" minOccurs="1" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="type" type="NoteAttType" default="Note"/>
</xs:complexType>
</xs:element>
<xs:simpleType name="NoteAttType">
<xs:restriction base="xs:string">
<xs:enumeration value="Caution"/>
<xs:enumeration value="Warning"/>
<xs:enumeration value="Danger"/>
<xs:enumeration value="Note"/>
</xs:restriction>
</xs:simpleType>
```

CSS

```
…
.Note[type='Note']    { border-top: 1px solid blue; border-bottom: 1px solid blue; }
.Note[type='Caution'] { border-top: 1px solid red; border-bottom: 1px solid red;}
…
```

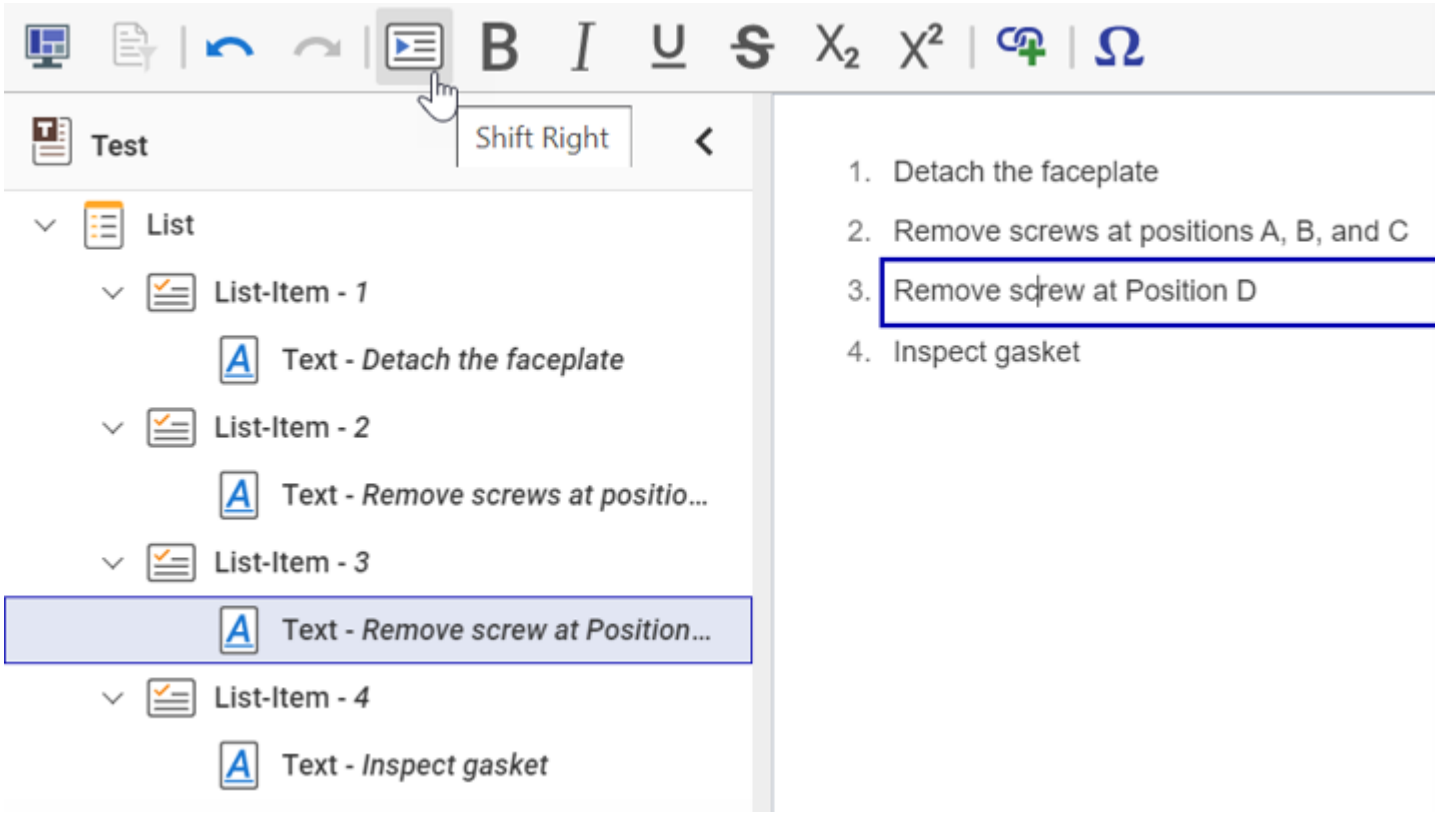
The Method logic to make the change is as follows:

```
const { document, view } = options.contextData;
const selectedElements = view.selection.get();
// Set the 'type' Attribute for the selected 'Note' Document Element
// based on the chosen value from the dropdown box
selectedElements[0]?.setAttribute('type', target.value);
```

In general, the selected Element is identified, and the Attribute is set based on the user's choice. Thus, with two lines of JavaScript, and some CSS, a convenient and data-driven mechanism is provided to the Technical Author to automatically characterize and render Note information in a Technical Document.

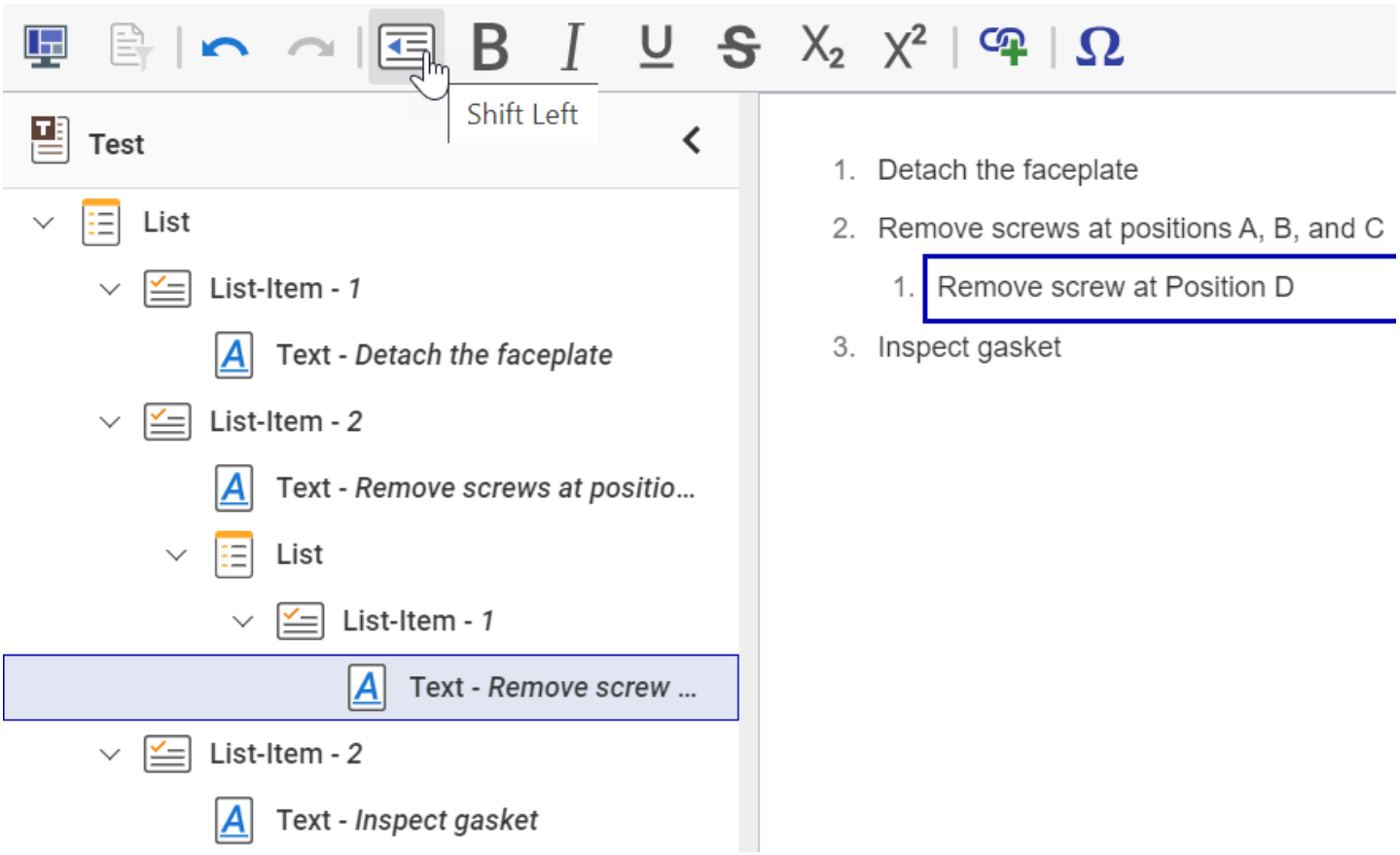
Manipulating Document Elements

When authoring Technical Document content, there may be a set of common manual operations that would be more convenient to automate to increase productivity and provide an increased level of consistency. The following Use Case shows an example of *List Item Indentation* whereby users can increase (or decrease) indentation by inserting (or removing) *List* Items to provide a sub-*List* for selected Text.



Shift Right Custom Function

When using List Document Elements (see List Document Element), indentation is a matter of creating ‘sub-lists’: Lists that are children of a List Item. For example, in the previous image there is a single List with 4 List Items; each with a single Text Document Element. To indent a List Item, a new child List needs to be added with content added as it’s List Items and so on.



Shift Left Custom Function

The previous two images also show a customized **Shift Right** and **Shift Left** button that has been added to the Editor toolbar. Toolbar buttons can be configured to be displayed and/or enabled based on the state of the Document (whether it is opened for Edit) and/or depending on what the user has selected. In this example, when the user selects any Document Element that is a List Item or a child of a List Item either the **Shift Left** or **Shift Right** Button is displayed. The logic to enable this capability (not shown):

1. Checks to see if the selected Document Element is either a List Item or is a direct descendant of one *and* that the List Item is not the first in its List
2. If so, the entire List Item Document Element is removed from its parent List
3. A new List is created and appended to the previous List Item
4. The selected List Item is added to the newly created List

The Client API provides the ability to perform each of these steps (and more). Specifically for this example, it uses the **View** class to ensure that a single Document Item is selected. It uses the **Document** class to create a new List Document Element, and it uses the selected List **Element** (explained later) to remove the selected List Item and reposition it in a new List.

Client API Definition

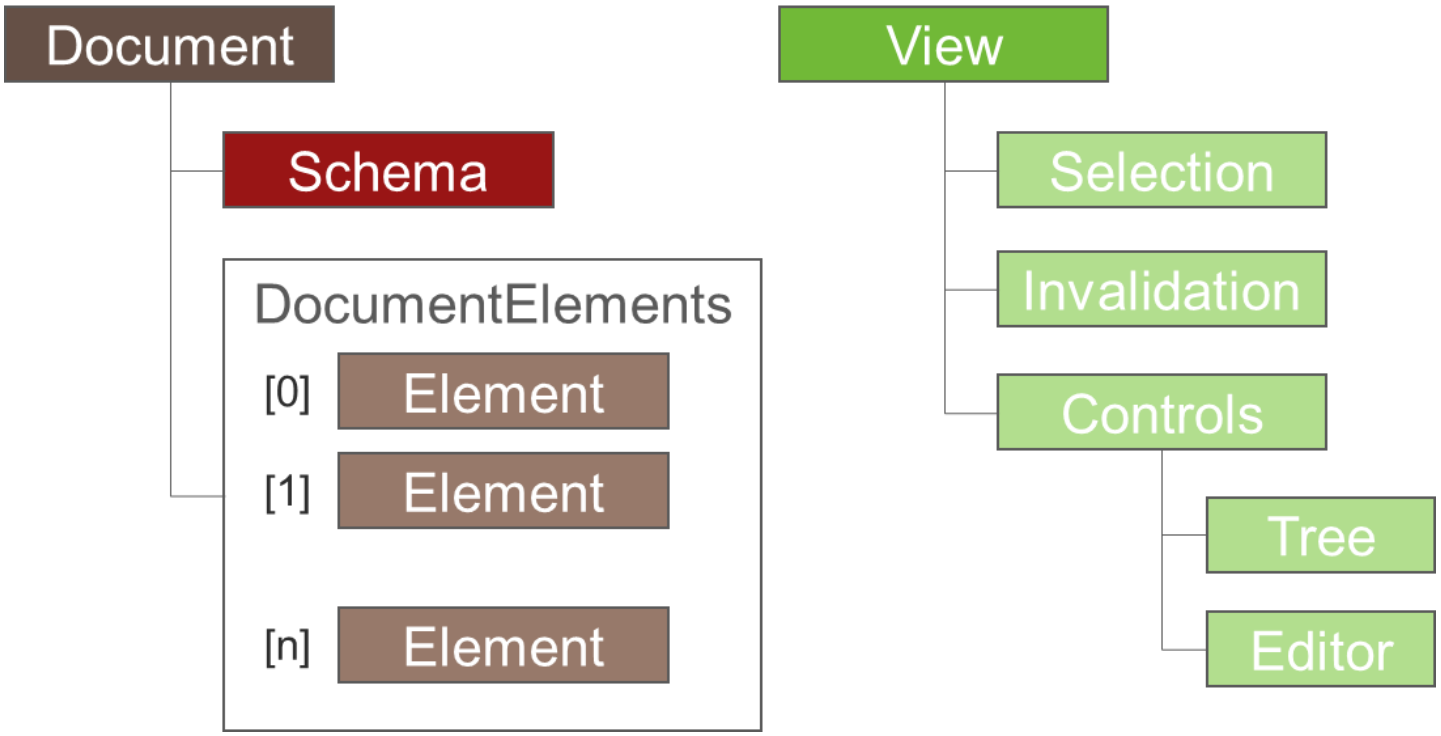
Overview

This section provides a detailed description of the Technical Documentation Client API. The Client API consists of several JavaScript classes; a definition of the available properties and methods of each are included.

> **Important**
>
> This API description will serve as the authoritative source for classes and related functionality for the Technical Documentation Client API. The classes/methods/properties included here are guaranteed to be available unless officially deprecated through an explicit notice. Any use of JavaScript functionality for Technical Document Editor customization not described in this section will be at the risk of the implementor since that capability may change without notice.

Each Method used with a CUI Item is passed several input parameters. This is standard design for Methods used with CUI. One of them – `options` – contains a `contextData` object that is used to access the Technical Document **Document** and **View** objects. **Document** and **View** are core objects defining the Client API (see Figure 17 [MANUAL ACTION REQUIRED: unresolved cross-reference]). For example, the following code fragment will create local `document` and `view` variables to be used to access the **Document** and **View** respectively:

```
const { document, view } = options.contextData;
```



Document / View / Schema Classes

Document

The **Document** class is the access point for document content as well as information about the Technical Document Item in which the document content is contained.

Document Class Properties

Name	Write	Description
`item`	F	An XML Node representing the Technical Document Item (e.g., `tp_Block`). This object can be used to access Properties or related Items. For example: `docName = aras.getItemProperty(document.item, 'name');`
`schema`	F	Object handle to the **Schema** class defined later in this section
`language`	F	String representing the language used for the current document. For example: ‘ `en` ’
`editable`	F	Boolean that identifies whether the current Technical Document is opened for editing. This is useful when enabling CUI controls that modify Document content. For example: `if (document.editable) { // logic to update document…`
`elements`	F	Handle to **DocumentElements** class (defined later in this section) that provides access to, and creation of, Document Elements. For example: `const newList = document.elements.create('List');` or `const firstElem = document.elements.get(1);`

Document Class Methods

Name	Input	Description
`createElement`		Used to create a Document Element. Document Elements are created based on the Element name provided in the Document Schema (e.g., ‘List’, ‘ItemInfo’, ‘Note’, etc.). The returned **Element** object can then be used/added. For example: `document.createElement(‘ItemInfo’, {item, partItem});` or `document.createElement(‘Note’);`
		Important An Element is not added to a document when it is created. Rather it must be explicitly added as a child of some existing Element.
`createElement`	name	Document Element Name
`createElement`	parameters	(Optional) **Element** Object Properties. The Properties of the object used are based on the type of Document Element being created. Creating instances of **Element** types are described later in the document
`clear`		Removes all content from the document. Use with caution!

DocumentElements

The **DocumentElements** class is a container for Document Elements. Note that this class manages Document Element content as a flat list. However, children of specific Document Elements can be accessed using the **ElementChildren** class defined later.

DocumentElements Class Properties

Name	Write	Description
`root`	F	Handle to the top-most **Element** in a Technical Document. Note that all Technical Documents have a single, root Element – `block` – that contains all the content. Note also that root is the same Element as returned by `get(0)` – see below.
`count`	F	Number of Document Elements, including the root block, in the document.

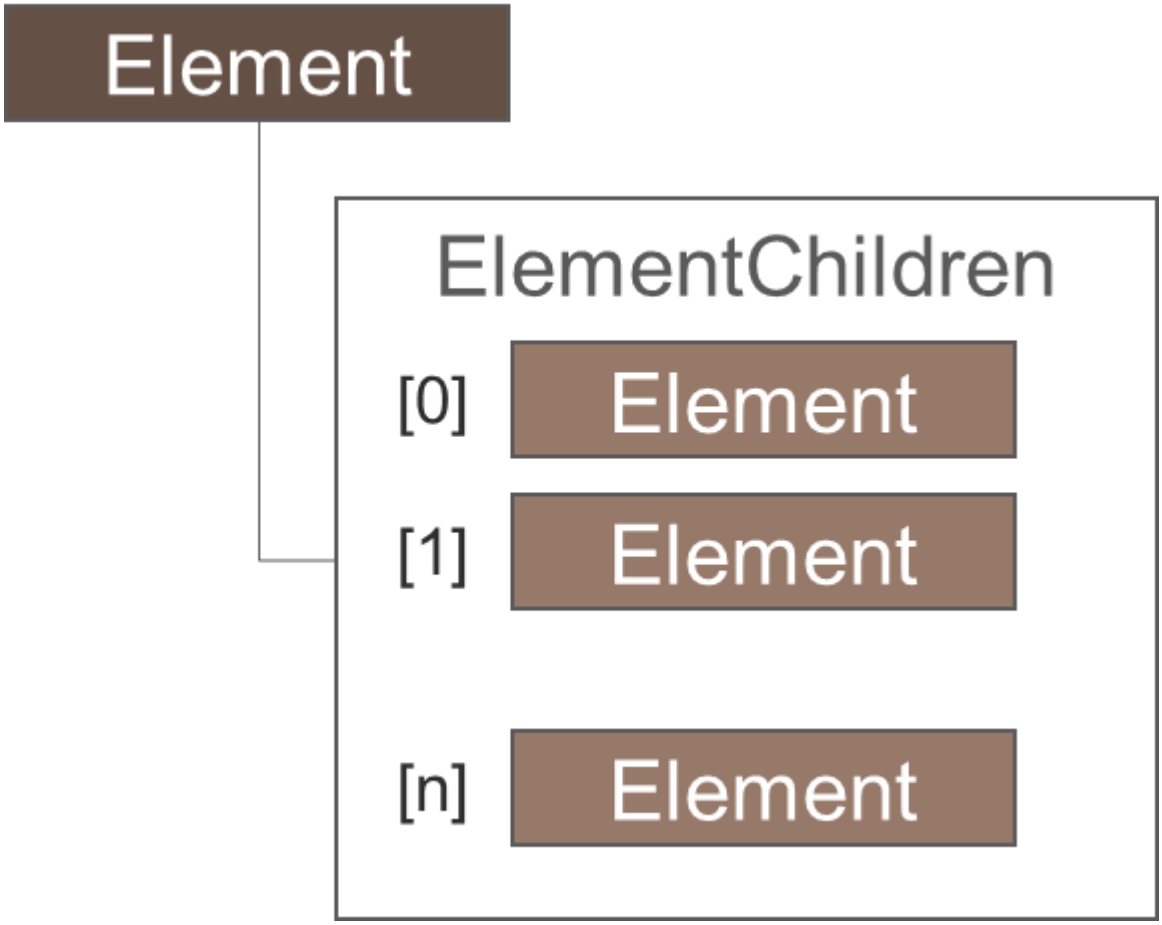
DocumentElements Class Methods

Name	Input	Description
`get`		Used to retrieve a Document Element. Document Elements can be accessed based on an index – a sequential number starting from 0 and progressing in the order the Elements are displayed in the Tree View, and id – internal id value for each Element, or uid – a unique 32 character identifier that persists with the Element. For example: `const blockElem = document.elements.get(0); const fourthElement = document.elements.get(4, ‘index’); const listElement = document.elements.get(’24DKR82LFMN5927DJ4MNSHR7693KR821’, ‘uid’);`

`get`	index	Positive integer starting from 0 or a 32 character string based on criteria. Note that the first (‘0’) Element is the root block, which contains all Document **Element**s.
`get`	criteria	(Optional) – Either ‘index’ (default), ‘id’, or ‘uid’. If provided the given index parameter is interpreted accordingly.
`index`	Returns the index number of the given Element. You can check an Element’s position within the document by calling this method. For example: `let contentPosition = document.elements.index(someElement);`	
`index`	element	Element to check
`create`	Used to create an Element. Document Elements or each type (see Document Element types in Standard XML Schema) can be created using this method. The optional `parameters` are used to include type-specific data. Note that an Element is not included in a document until it is explicitly added as a child of some existing Element, including the root block. For example: `let newTable = document.elements.create('Table',{rowCount: 5, columnCount: 3}); let itemProperty = document.elements.create('ItemProperty', {property: 'name', mode: 'write'}); let text = document.elements.create('Text', {text: 'some text'});`	
`create`	name	Name of the Document Element as defined in the Document schema (e.g., ‘List’, ‘ItemInfo’, ‘Text’, etc.).
`create`	parameters	(Optional) Object containing name:value pairs used to construct an Element. The Document Element type determines the valid values. See details for each Element Type below.
`toArray`		Returns a newly generated array of all Elements in the document. Use with caution with large documents.

Element

The **Element** class is the base class for all Element Types: List, Table, Item, etc.



Element Classes

Element Class Properties

Name	Write	Description
`id`	F	Internal integer ID assigned sequentially as document content is read/created. Note that an Element's *index* is different from its *ID*.
`uid`	F	Unique, alphanumeric 32-char identifier that persists with the element content.
`name`	F	Element name as assigned in the Document Schema
`parent`	F	Handle to an elements direct parent Element
`document`	F	Handle to the **Document** object
`dynamic`	F	Boolean. True, if the Element is associated with a Content Generator that is dynamic
`external`	F	Boolean. True if the Element is a block that is referenced within the opened Technical Document. Element children of an external block will have an external value of *False* even though the content is included in a referenced document.

`editable`	F	Boolean. True if the Element can be modified
`children`	F	Handle to **ElementChildren** class. See below
`descendants`	F	Generated array of a full recursive/deep hierarchy of all content under the Element. For example: `// Get the full set of Document Elements const select = document.elements.root.descendants; // Restrict to a subset of only Text Elements allTxt = select.filter((element) => element.is('Text'));`

Element Class Methods

Name	Input	Description
`contains`		Returns True if the given Element is a descendant of the Element on which this method is called. A descendant is any Element child, grandchild, etc that exists *under* an Element.
`contains`	element	Element to check.
`getAttribute`		Returns the Attribute value with the given name. Null if the Attribute does not exist.
`getAttribute`	name	Attribute name to check
`setAttribute`		Sets the Attribute with the given name to the given value. It is possible to set an Attribute for an Element that is not defined in the Document Schema and this Attribute/Value will persist with the document.
`setAttribute`	name	Attribute name
`setAttribute`	value	Attribute value
`is`		Returns True is the Element *is* the given Type. For example: `if (selectedElement.is('Note')) {`
`is`	type	Type name, as defined in the Document Schema, to check
`clone`		Returns a deep copy of the Element
`remove`		Removes the Element from the document To remove the Element only from is parent, use `DocumentElements.remove()` or `ElementChildren.remove()` .

ElementChildren

The **ElementChildren** class is a container for Documents Elements that are direct children of an associated Element. Similar to an array, the Element children are managed as a sequence representing the order in which content is displayed in the document.

ElementChildren Class Properties

Name	Write	Description
`count`	F	Number of direct children. Returns -1 if the associated Element does not contain any child Elements

ElementChildren Class Methods

Name	Input	Description
`get`		Returns the Element at the given index.
`get`	index	Index value (starting from 0) of the Element to retrieve. Similar to an array, the index is the ordered position of the Element.
`index`		Returns the index number (starting from 0) of the given Element. Returns -1 if the given Element is not a child.
`index`	element	Element to check.
`insert`		Inserts the given element at the optional given position. For example: `targetList.children.insert(listItem, 2);` Document Element are added to a document only by inserting/appending to either a Document or an Element.
`insert`	element	Element to add.
`insert`	position	(Optional) Position (starting at 0) to insert the given Element. If the given value is greater than or equal to the `count` , the new Element will be appended. If the given value is less than the count, the given Element will be inserted at that location, effectively shifting all other children accordingly.
`append`		Inserts the given element at the end of the list. For example: `firstCellRow2.children.append(cellText);` Document Element are added to a document only by inserting/appending to either a Document or an Element.
`append`	element	Element to add.
`clear`		Removes all child Elements.
`toArray`		Returns a newly generated array of all child Elements. Use with caution with large /deep Element hierarchies.
`remove`		Removes the given Element from its parentTo remove the Element completely, use `Element.remove()` .
`remove`	element	

TextElement

TextElement extends the **Element** class by providing content specifically for Text Document Elements. Text Document Elements are composed of one or more string fragments (`<aras:emph>`) that are used to specify distinct formatting. For example, the following string: This is a sentence with **bold**, *italics*, and underlined content. …would be decomposed into the following string fragments:

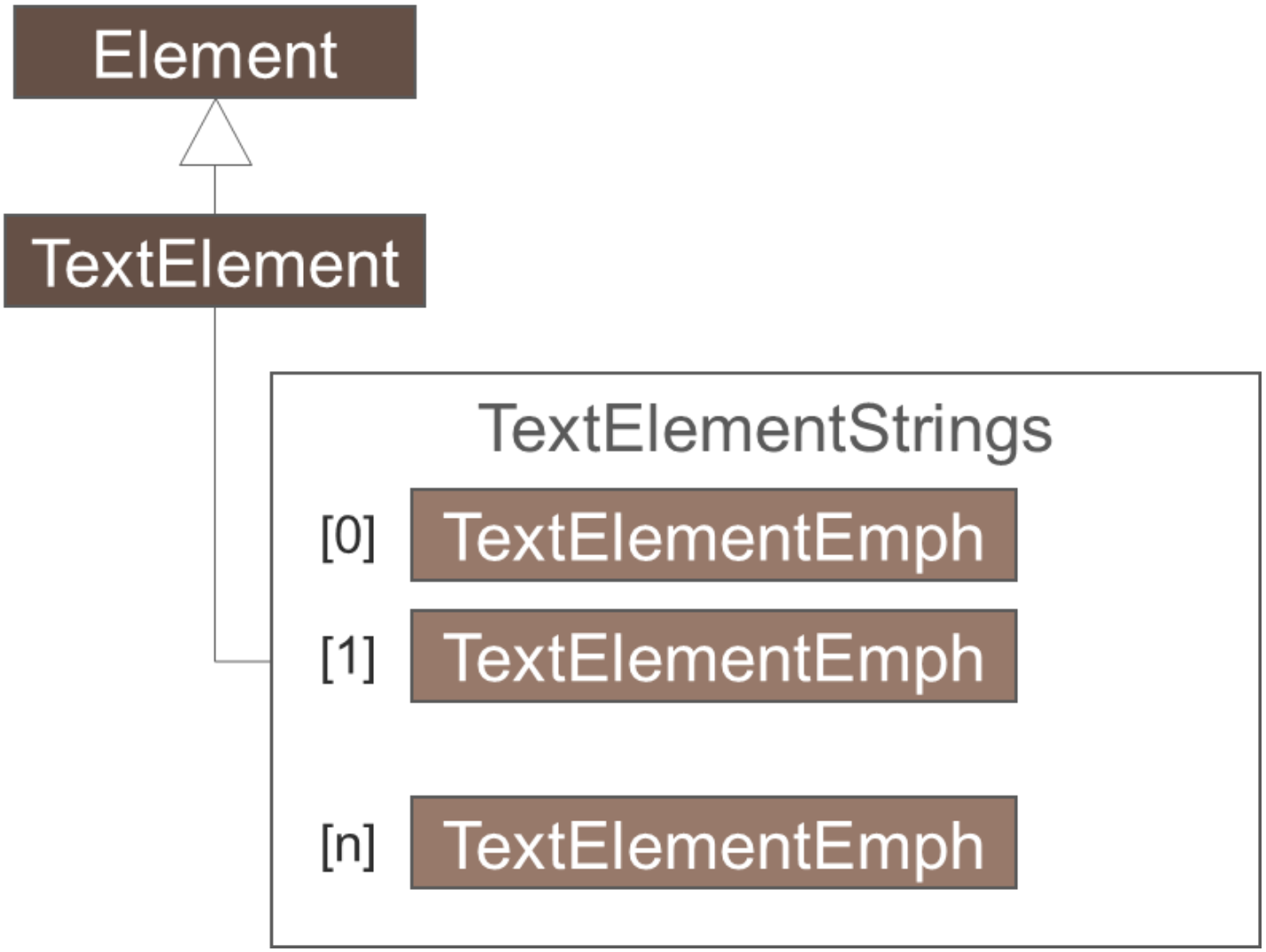
```
<Text>
<aras:emph>This is a sentence with </aras:emph>
<aras:emph>bold="true">bold</aras:emph>
<aras:emph>, </aras:emph>
<aras:emph italic="true">italics</aras:emph>
<aras:emph>, and </aras:emph>
<aras:emph under="true">underlined </aras:emph>
<aras:emph>content.</aras:emph>
</Text>
```

The terms ‘string fragment’ and ‘emph’ are used interchangeably in this section.

Important

Technical Documentation does not store interleaved (mixed) XML/text. Instead, it uses individual XML elements (< emph >) to differentiate styling or other distinct metadata

TextElement objects manage the text content as individual objects that can be created, retrieved, and/or updated using the Client API.



Text Element Classes

The JavaScript to create the sample sentence above using the Client API:

```
let formattedText = document.createElement('Text');
formattedText.strings.add({text: 'This is a sentence with '});
formattedText.strings.add({text: 'bold', style: {bold: true}});
formattedText.strings.add({text: ', '});
formattedText.strings.add({text: 'italics', style: {italic: true}});
…
document.elements.root.children.insert(formattedText);
```

TextElements are created using either the **Document**.createElement() or **DocumentElements**.create() methods. For **TextElement** objects, the second parameter is optional

and can include the text Property as specified in the **TextElementEmph** class. For example, the following code sample demonstrates an alternative way of creating a new Text Document Element by including the text:

```
// Create a new Text Element with a single String Fragment
const text = document.elements.create('Text', {text: 'test text'});
```

> **Important**
>
> Non-formatted Text Elements (see Text Elements in Document Elements Types) do not consist of String fragments and only contain a single text property. Elements can be set using the code sample above.

Like all dynamically created **Element**s, they must be added to the children list of some existing Element to be included in the document.

The remainder of this section includes a description of the **TextElement**, **TextElementStrings**, and **TextElementEmph** classes.

The **TextElement** class represents a formatted Text Document Element with all its *emph* content.

TextElement Class Properties

Name	Write	Description
`text`	T	Generated String that concatenates the text (without formatting) from all the constituent string fragments. This Property is writable. Setting it will overwrite all text content with a single, non-formatted string fragment.
`strings`	F	Handle to the **TextElementStrings** class. See below.

The **TextElementStrings** class is a container for the formatted string fragments – *emph*s.

TextElementStrings Class Properties

Name	Write	Description
`count`	F	Number of string fragments

TextElementStrings Class Methods

Name	Input	Description
`get`	Returns the string fragment at the given index.	
`get`	index	Index value (starting from 0) of the string fragment to retrieve. Similar to an array, the index is the ordered position of the Element.
`add`	Appends the given String or String fragment object to the set.	

`add`	element	String (text) or **TextElementEmph** object
`remove`		Removes the string fragment provided or at the given position. For example: `formattedText.strings.remove(2); // remove 3rd fragment` or `let thirdEmph = formattedText.strings.get(2); formattedText.strings.remove(thirdEmph);`
`remove`	value	Either an index of a **TextElementEmph** to remove.

The **TextElementEmph** class manages each formatted string fragment – *emph*.

TextElementStrings Class Properties

Name	Write	Description
`text`	T	Text content of the string fragment
`link`	T	String URL if this string fragment is a link
`style`	T	Object containing name:value pairs for styling information. The following style properties relate to the standard set provided in the Editor:boldtalicundersupersubstrikeBy default, these are false. Setting any combination of these style parameters as True will result in the corresponding style being applied to the text in the string fragment. For example: `formattedText.strings.add({text: '…', style: {bold: true}});` It is possible to set any style property that's defined in CSS using this parameter and the results will persist with the document content. For example: `formattedText.strings.get(0).style = {fontcolor: 'red', fontsize: '24px'};` Overriding Style settings as defined in the Document Type (see Section CSS Styles) is discouraged given the design intent of Technical Documentation to separate content from the way it's presented

TextElementStrings Class Methods

Name	Input	Description
`divide`		Splits the string fragment at the given text position (0 – beginning of string fragment) and returns a newly generated string fragment containing the remainder of the text. The new string fragment is added automatically to the TextElement. For example: `let splitEmph = formattedText.strings.get(0).divide(7); splitEmph.style = {under: true};` Null is returned if the given position is not within the bounds of the string fragment.
`divide`	position	Character position (starting from 0) of the string fragment at which to split the string.
`clone`		Returns a copy of String fragment object.
`remove`		Removes the string fragment

TableElement

TableElement extends the **Element** class by providing content specifically for Table Document Elements and its related Table Row, Column, and Cell classes. The row and column content of a Table Document Element is managed by the Client API to ensure that all tables are 'rectangular'.

That is, that all rows have the same number of columns, and all columns have the same number of rows. Thus, Cells should not be added/removed independent of their respective rows and columns.

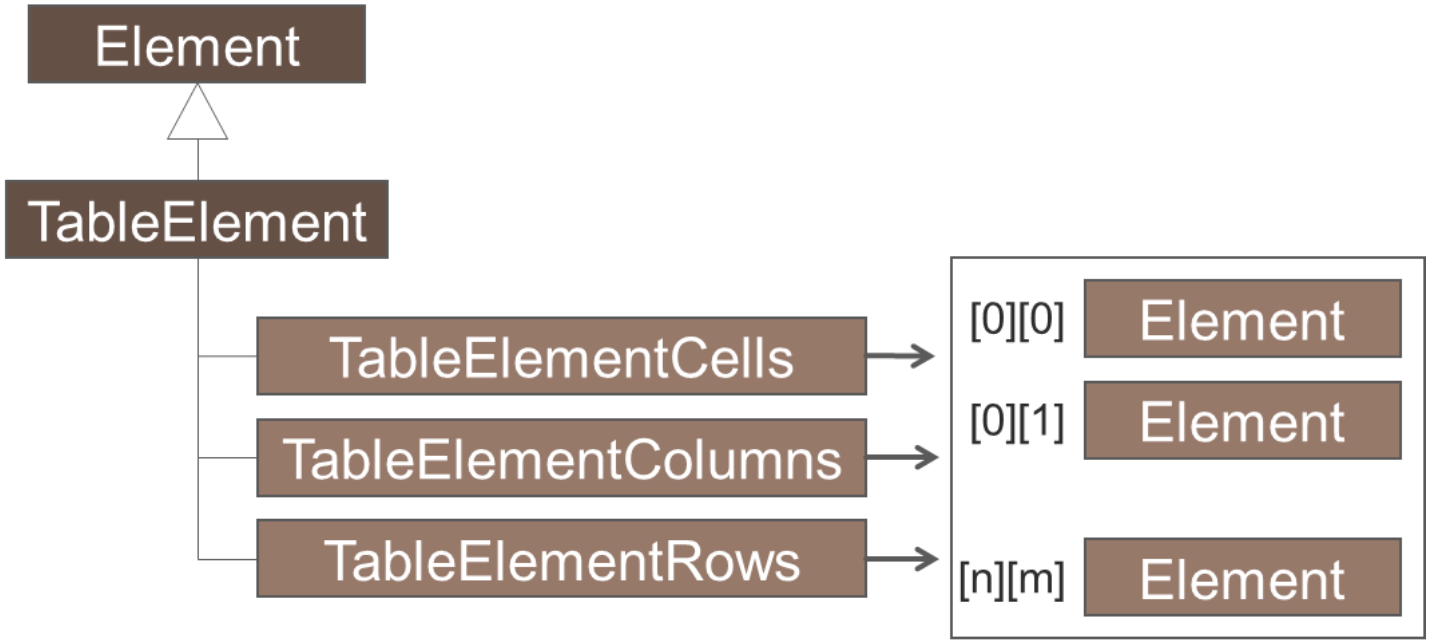


Table Element Classes All Table Document Elements must be constructed with a given row and column count > 0. After, new rows and/or columns can be added and removed using the API. The content hierarchy of a Table Document Element is as defined in the schema. Table Document Elements (`aras:table`) contain Row Document Elements (`aras:tr`), which contain Cell Document Elements (`aras:td`) (see Section Table Document Element). Thus, when a Table Element is constructed, it's children should consist entirely of Row Document Elements. Row Document Elements should consist of Cell Document Elements. The content of a Table is stored as children of Cell Document Elements. The Client API will construct/remove/update the appropriate Row and Cell Document Elements based on the associated Table function applied. The following code sample shows a newly created table with row additions, added content, and cell merging.

```
// Create a new 3X4 Table
const table = document.elements.create('Table', {rowCount: 3, columnCount: 4});
// Add it to the beginning of the document
document.elements.root.children.insert(table);
// Get the first cell in the second row
const firstCellRow2 = table.cells.get(1,0);
// Insert a new Text Element
const cellText = document.createElement('Text', {text: 'Cell text'});
firstCellRow2.children.append(cellText);
// Add a new second row
table.rows.add(1);
// Adjust the column width percentages
table.columns.widths = ['20', '20', '20', '40'];
// Merge the cell with added Text with its right neighbor
table.cells.merge(2, 0, 'right');
```

The resulting table:

Cell text			

Client API-generated Table

The remainder of this section includes a description of the **TableElement**, **TableElementColumns**, **TableElementRows,** and **TableElementCells** classes. The **TableElement** class represents a Table Document Element and is the top-most reference object for updating and accessing all content within a Table. All actions on a table are provided by one of the row, column, or cell classes. Each are accessible through the corresponding Property.

TableElement Class Properties

Name	Write	Description
`rows`	F	Handle to **TableElementRows** class (see below).
`columns`	F	Handle to **TableElementColumns** class (see below).
`cells`	F	Handle to **TableElementCells** class (see below).

TableElements are created using either the **Document**.createElement() or **DocumentElements**.create() methods. For **TableElement** objects, the second parameter to these

methods must be included and identify the number of columns and rows using ‘ `columnCount’` and ‘ `rowCount’` respectively. For example, the following code sample demonstrates the only valid way of creating a new Table:

```
// Create a new 3X4 Table
const table = document.elements.create('Table', {rowCount: 3, columnCount: 4});
```

Like all dynamically created **Element**s, they must be added to the children list of some existing Element to be included in the document.

TableElementRow

The **TableElementRows** class is an accessor for table Row Elements and provides the ability to add/remove rows. Note that the Row Elements are types of **Element**s and thus extend the same functionality as an **Element** object. For example, the children of a Row Element are the set of Cell Document Elements.

> **Important**
>
> Cell Document Elements should not be added to a Row Element, because doing so would make the table non-rectangular. Instead, use the **TableElementColumns** class to add an entire column – with Cells added automatically to each Row.

TableElementRows Class Properties

Name	Write	Description
`count`	F	Number of Rows

TableElementRows Class Methods

Name	Input	Description
`get`		Returns the Row Element at the given index.
`get`	index	Index value (starting from 0) of the row to retrieve. Similar to an array, the index is the ordered position of the Element.
`add`		Inserts a new Row into the Table. Each Row will have the appropriate number of Cell Elements based on the current number of Columns. The Row at the given index will be shifted if not appending to the end and all rows will be shifted accordingly.
`add`	index	Position of the row (starting from 0) of the new Row.
`remove`		Removes the Row at the given position. For example: `Table.rows.remove(2); // remove 3rd row`
`remove`	row	Either an index or a Row Element to remove.

TableElementColumns

The **TableElementColumns** class provides the ability to add and remove Columns. The Client API will automatically add or remove the corresponding Cell Elements (and their content) respectively.

TableElementColumns Class Properties

Name	Write	Description
`count`	F	Number of columns
`widths`	T	String array containing the percentage widths of all columns. This can be modified by setting with a string array with the size equaling the number columns. The number of strings in the array should equal the column count. For example: `// Adjust the column width percentages table.columns.widths = ['20', '20', '20', '40'];`

TableElementColumns Class Methods

Name	Input	Description
`add`		Inserts a new Column into the Table. Each Column will have the appropriate number of Cell Elements based on the current number of Rows. If not appending to the end, the Column at the given index will be shifted accordingly.
`add`	index	Position of the Column (starting from 0) of the new Column.
`remove`		Removes the Column at the given position. For example: `Table.columns.remove(2); // remove 3rd column`
`remove`	index	Position of the Column (starting from 0) to remove.

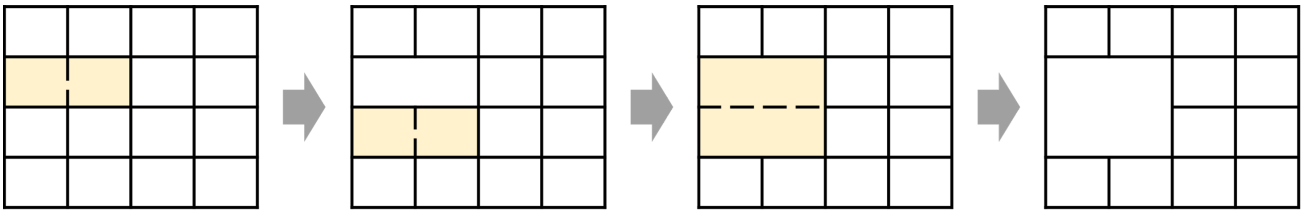
TableElementCells

The **TableElementCells** class is an accessor for Table Cell Elements and provides the ability to retrieve and merge/unmerge Table Cells. Note that the Cell Elements are types of **Element**s and thus extend the same functionality as an **Element** object. For example, the children of a Cell Element are the set of allowed Document Elements as defined by the schema.

TableElementCells Class Methods

Name	Input	Description
`get`		Returns the cell at the given index. The given row and cell numbers must be >=0 and less than the count of rows/cell respectively.
`get`	row	Index value (starting from 0) of the Cell row
`get`	column	Index value (starting from 0) of the Cell Column
`merge`		Merges the cell at the given coordinates based on the given *direction*. Cells can be merged with other cells of similar spans. This mimics the functionality within the Technical Document Editor. For example, to merge 4 cells (2 rows and 2 columns) the cells in each

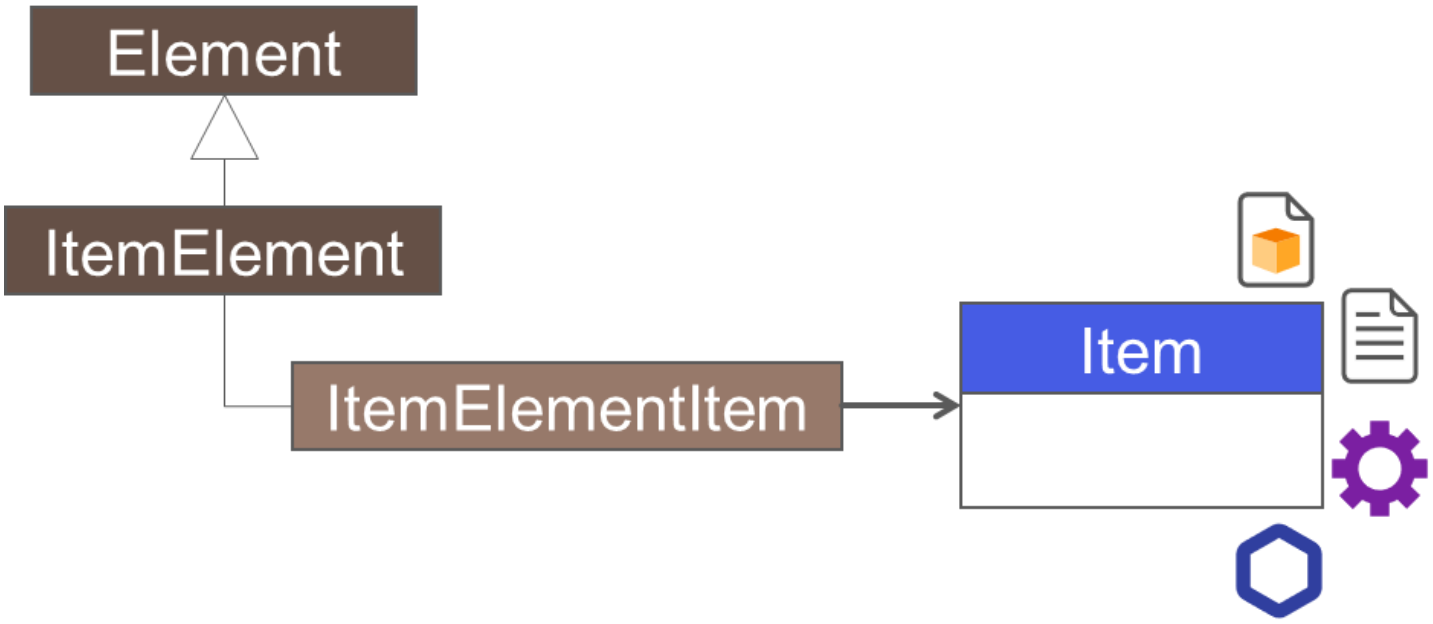
row or column must be first merged before these merged cells are merged with the other column or row. Thus, 3 merge operations are required:



`merge`	row	Index value (starting from 0) of the Cell row
`merge`	column	Index value (starting from 0) of the Cell column
`merge`	direction	String representing the direction of the cell to merge with. Must be one or ‘ `left` ’, ‘ `right` ’, ‘ `top` ’, ‘ `bot` ’.
`unmerge`	Un-Merges the cell at the given coordinates. Any cell within a cluster of merged cells can be given as a coordinate. All merged cells will be unmerged with this operation.	
`unmerge`	row	Index value (starting from 0) of the Cell row
	column	Index value (starting from 0) of the Cell column

ItemElement

The **ItemElement** class extends **Element** with capability for setting and retrieving Item data. **ItemElement** objects represent Item Document Elements (see Item Elements), which are containers with a direct link to a referenced Item. This link is maintained as a Relationship between the encompassing Document Item and the referenced Item. Item Document Elements typically have a configured Content Generator that, upon creation, executes custom logic to automatically populate the child content; presumably with data from the referenced Item and/or its related content and so on. The Client API extends the ability of a Content Generator to update existing Item Document Elements.



ItemElement Classes

The following code sample shows the process of creating an **ItemElement** and setting one of the referenced Item's Properties:

```
// Get the reference Item to use
const partItem = window.aras.getItemByKeyedName('Part', 'P-Tmp1')
// Create an Item Document Element and add to the beginning of the document
const itemElement = document.createElement('ItemInfo', {item: partItem});
document.elements.root.children.insert(itemElement);
// Set the referenced Part's Make/Buy Property – changes set on Save
itemElement.item.setProperty('make_buy', 'Buy');
```

ItemElement extends **Element** and contains a handle to the **ItemElementItem** class, which is a wrapper for the referenced Item and provides accessors/mutators for Property data on the referenced Item. This section includes a description of these classes

ItemElement Class Properties

Name	Write	Description
`empty`	F	True if there's no related Item set
`item`	F	Handle to the **ItemElementItem** object (see below).

ItemElementItem

ItemElementItem Class Properties

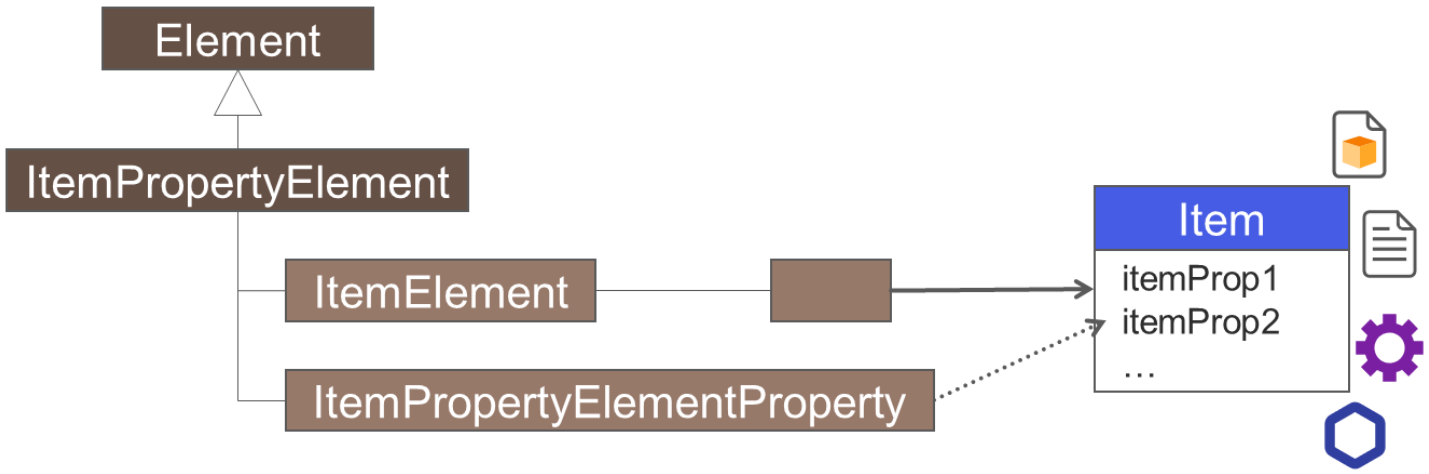
Name	Write	Description
`node`	T	Referenced Item Object. The object handle is equivalent to that which is returned by `Innovator.getItemBy*()` methods. This can be changed by the Client API by setting this node. Changing the node Item changes the reference to that Item. If setting, it's necessary to ensure that the Item used is of an ItemType included in the `tp_Item` Polysource.
`id`	F	ID of the referenced Item
`type`	F	ItemType name of the reference Item. For example, 'Part', 'Document'.
`editable`	F	True if changes to the referenced Item are allowed.
`modified`	F	True, if any Property was set prior to being saved.

ItemElementItem Class Methods

Name	Input	Description
`getProperty`		Returns the value of the referenced Item Property with the given name
`getProperty`	name	Property name
`setProperty`		Sets the value of the referenced Item Property with the given name to the given value. Setting a Property on a referenced Item is not saved to that Item until the Technical Document is saved. Authors will see a pencil icon, signifying that the Property of the referenced Item has not been updated, next to the Item Document Element in the Tree View. Upon saving the pencil icon will be hidden.
`setProperty`	name	Property name
`setProperty`	value	Property name
`dropChanges`		Removes pending Property changes (established by the `setProperty` () method).

ItemPropertyElement

The **ItemPropertyElement** class extends **Element** with capability for setting and retrieving referenced Item Property data. **ItemPropertyElement** objects represent Mapped Property Document Elements (see Mapped Properties), which are unformatted Text Document Items that *link* to a specific Property on an Item referenced by some Item Document Element in its parent lineage. This link is loose in that there's no established Relationship generated. Property mappings are by configured by Property name only. Changing the associated Item Document Element will cause changes to all descendant Mapped Properties regardless of whether the referenced Items were the same ItemType.



ItemPropertyElement Classes

The **ItemPropertyElement** is an accessor to both the **ItemPropertyElementProperty** Element – that represents the actual mapped Property in the document – and the **ItemElement** - that is a handle to the associated Item Document Element. Mapped Properties are either read-only, in which case the ItemPropertyElement provides a view of the actual Property data, or they are writable. Writable Mapped Properties provide a *proxy editing mechanism* using the Technical Document Editor. The Client API caches updates to writable Mapped Properties until the Technical Document is saved and the referenced Items are updated/synched.

The following code sample shows the process of creating an **ItemPropertyElement**:

```
// Create a Mapped Property set to refer to the ‘tmp’ Property
const cellMappedProp = document.createElement('ItemProperty');
cellMappedProp.property.name = 'tmp';
// Add it to the document
firstCellRow2.children.append(cellMappedProp);
```

When rendered in the document, the generated Mapped Property will display the Property value of whatever Item Element precedes it in the element ancestry lineage if one exists. Otherwise, the content will be empty.

ItemPropertyElement extends **Element** and contains a handle to the **ItemPropertyElementProperty** class, which is a wrapper for the referenced Mapped Property and

provides accessors/mutators for Property data on the referenced Item. This section includes a description of these classes

ItemPropertyElement Class Properties

Name	Write	Description
`property`	F	Handle to **ItemPropertyElementProperty** (see below)
`sourceElement`	F	Handle to the **ItemElement** object

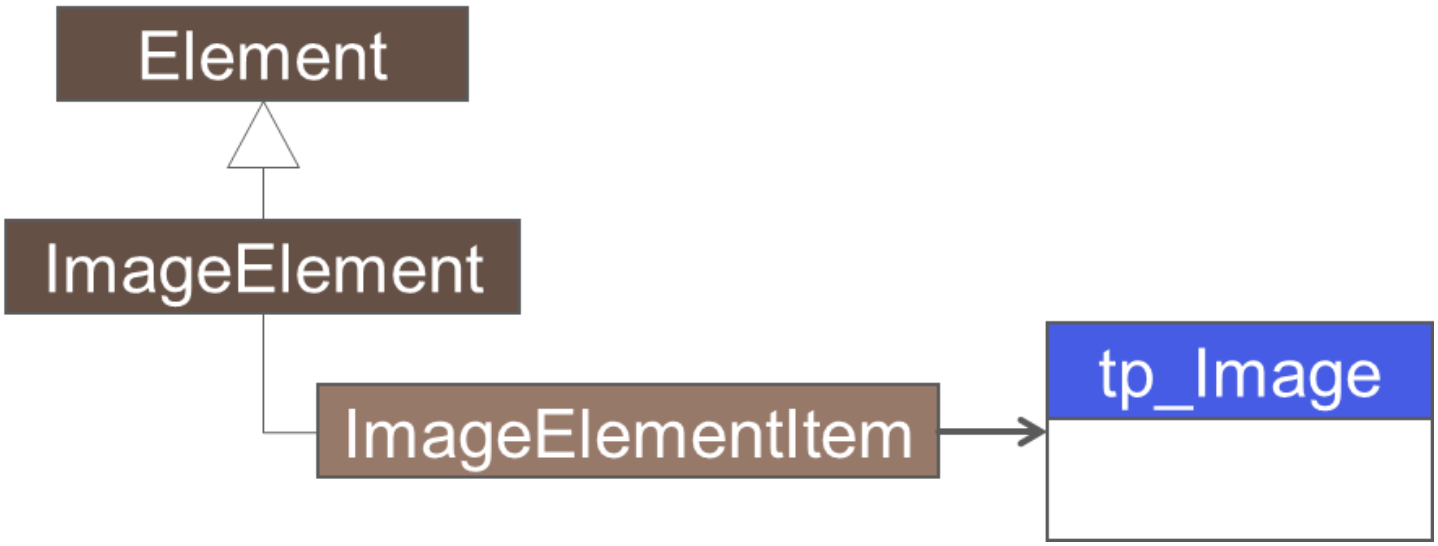
ItemPropertyElementProperty

ItemPropertyElementProperty Class Properties

Name	Write	Description
`name`	T	Mapped Property name
`mode`	T	Mode – either 'read' or 'write'
`localValue`	F	Current value of the Mapped Property
`modified`	F	True, if the Property was set prior to being saved.

ImageElement

The **ImageElement** class extends **Element** with capability for setting and retrieving referenced image Items (tp_Image). The 2D Graphic data must come from a `tp_Image` Item.



ImageElement Classes

The following code sample shows the process of creating an **ImageElement**:

```
// Get the tp_Image Item to use for the image data
const imgItem = window.aras.getItemByKeyedName('tp_Image', 'G-00016');
// Create the ‘Graphic’ Image Element and set the tp_Image
const imageElement = document.createElement('Graphic');
imageElement.image.node = imgItem;
// Add it to the document
document.elements.root.children.insert(imageElement);
```

This section includes a description of these classes.

ImageElement Class Properties

Name	Write	Description
`empty`	F	True if there’s no associated Image Item attached. Image Elements without an image will display with a ‘placeholder’ image in the viewer.
`image`	F	Handle to the **ImageElementItem** object (see below)

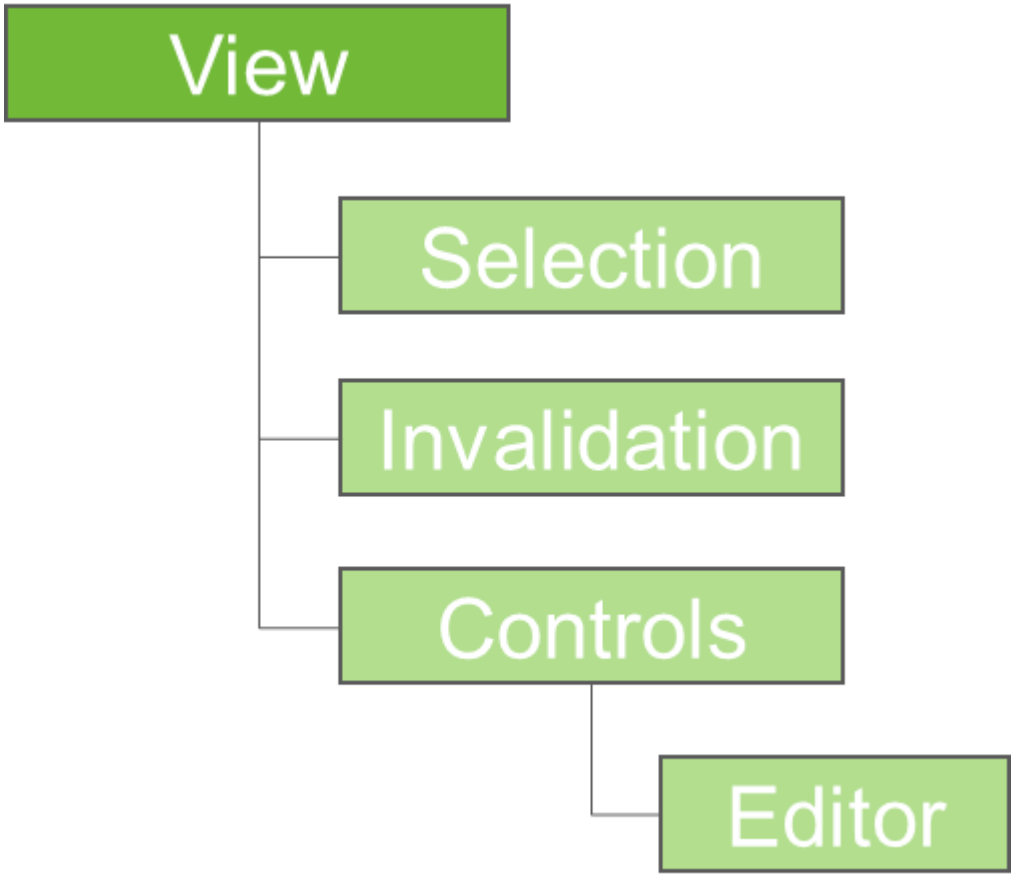
ImageElementItem

ImageElementItem Class Properties

Name	Write	Description
`node`	T	Item (of type tp_Image) for setting only. Setting this property on an existing Image Element will cause the Image Element to update with the new image data.This property is for setting only. To get the associated `tp_Image` Item, use the `id` Property, and retrieve using IOM.
`id`	F	ID of the associated `tp_Image` Item
`src`	F	URL to vault file used for the image

View

View provides access to the Technical Document Editor selection, scroll, and invalidation capability. This section provides a description of each of these functions and how they can be applied.



View Classes

View is an accessor to each of the Classes that provide for the functions mentioned above. Its only content (for the Client API) are handles to the Selection, Invalidation, and Editor Controls defined in this Section.

View Class Properties

Name	Write	Description
`selection`	F	Handle to the **Selection** object (see below)
`invalidation`	F	Handle to the **Invalidation** object (see below)
`controls`	F	Handle to the **Controls** object (see below)

Selection

The **Selection** class provides the ability to get a list of selected **Element**s or set this list. This is useful for identifying the Element (or Elements) to apply a custom function to or direct the user to an

Element (or Elements) by programmatically selecting them. The following code sample shows the process of selecting a specific Text Element:

```
// Get the full set of Document Elements
const select = document.elements.root.descendants;
// Restrict to a subset of only Text Elements
const selectText = select.filter((element) => element.is('Text'));
if (selectText.length >= 1) {
const lastTextElement = selectText[selectText.length-1];
view.selection.set(lastTextElement);
}
```

Selection Class Methods

Name	Input	Description
`set`		Sets the **Element**s selected in the viewer.It is possible to select multiple non-contiguous Elements. However, editing functions typically can only be applied within the editor to those that are contiguous.
`set`	Element(s)	Either a single **Element** or **Element** array
`get`		Returns an array of selected Elements. This array is empty if there's no current selections.

Invalidation

The **Invalidation** class provides the ability to suspend the updates made visible in the editor so that multiple functions can be applied atomically. Within the Client API, when an Element is added, removed, or modified the UI is simultaneously updated. This may be time-consuming and/or result in unnecessary synchronization operations if there are many updates. In addition, the undo buffer is updated with each of these individual function calls. The Invalidation class provides a `suspend` and `resume` method to be used to wrap multiple operations so that they are applied together. The following code sample will create a table like this one:

Cell text			

Sample Generated Table

```
// Temporarily suspend Editor UI operations
view.invalidation.suspend();
// Create a new 3X4 Table
const table = document.elements.create('Table', {rowCount: 3, columnCount: 4});
// Add it to the beginning of the document
document.elements.root.children.insert(table);
// Get the first cell in the second row
const firstCellRow2 = table.cells.get(1,0);
// Insert a new Text Element
const cellText = document.createElement('Text', {text: 'Cell text'});
firstCellRow2.children.append(cellText);
// Add a new second row
table.rows.add(1);
// Merge the cell with added Text with its right neighbor
table.cells.merge(2, 0, 'right');
// Resume to apply changes
view.invalidation.resume();
```

The suspend and resume method calls must be applied in pairs, or the results are undefined. In this example, there are several operations that would cause an update to the UI. These include the insertion of the newly created Table, the insertion of the newly created Text, the additional Row, and the Merge of two Cells. By wrapping these operations within suspend/resume calls, the entire set of updates are treated as one.

Invalidation Class Methods

Name	Input	Description
`suspend`		Suspends updates to the viewer.This must be followed by a call to `resume` before the CUI Method returns.
`resume`		Resumes updates to the viewer.This must be preceded by a call to `suspend` .

Editor Controls

The **Controls** class provides access to the **Editor** class for use in programmatically scrolling the View. This is useful when directing the user to a newly selected Element. The following code sample shows the process of selecting and scrolling to a specific Text Element:

```
// Get the full set of Document Elements
const select = document.elements.root.descendants;
// Restrict to a subset of only Text Elements
const selectText = select.filter((element) => element.is('Text'));
if (selectText.length >= 1) {
const lastTextElement = selectText[selectText.length-1];
// Make sure the selected Element is in view
view.controls.editor.scrollTo(lastTextElement);
view.selection.set(lastTextElement);
}
```

Note that the **Editor** class is accessed through a single Property of the **Controls** class as shown above.

Editor Class Methods

Name	Input	Description
`scrollTo`		Scrolls the view to the given **Element** in the viewer.The given Element must exist in the document. That is, it must be contained in the children list of some Element.
`scrollTo`	Element	Element within the document to scroll to

Schema

The **Schema** class provides the ability to access Document Schema information and validate operations on content within a document against the limitations and restrictions specified by the Document Schema. Schemas define restrictions with respect to the type of Document Elements that can be used, allowed Attributes, and cardinality. The following code sample shows how the schema can be used to check for the addition of an **Element**.

```
const { document, view } = options.contextData;
const schema = document.schema;
…
const lastTextElement = selectText[selectText.length-1];
// Check to ensure that a new Text Element can be added to the document
const allowed = schema.checkElementAction('append', lastTextElement, 'Text');
if(allowed)
{
// Code to append new content…
}
```

The logic in the sample retrieves the last Text Element in the document and checks the schema as to whether appending another Text Element is allowed. Actions of '*append*' and '*insert*' are permitted. Programmatically validating operations to a document in this manner ensures that the structure of a technical document conforms with the rules designated by the associated schema.

Schema Class Properties

Name	Write	Description
`item`	F	Handle to the Document Type corresponding to the opened Technical Document.
`elements`	F	Object with Name/Object pairs for each Document Element defined in the document schema. Each object in `elements` contains information about the XML schema element defined in the schema using the `declaration` property. The data within `declaration` is defined by functionality outside the Client API and is thus not directly controlled by it. In addition, the actual properties and population of property data is dependent on the content in the XML Schema. However, the property – `declaration.elementAttributes` is used with the Technical Document editor's Attribute dialog and is useful in other Use Cases associated with Technical Documentation. It is explained below for these purposes. As such the content in `elements.declaration.elementAttributes` is the only valid functionality to be used within the Client API for this Property. See example below.

The `elements` parameter is dynamically generated based on the XML Elements defined in the XML Schema (see XML Schema). Data for each XML Element is accessed using the Element's name as an index. For example, in the sample below to access data about the XML Element 'Note' use `elements['Note']` . The most useful part of this information is the attributes list, which contain the permitted values as defined in the schema if they are defined there. Element attributes provide metadata, which can be used to characterize or classify element instances. The following example shows how you can process the schema example in Setting Document Element Attributes to extract the defined/permitted Attribute values for the Note XML Element:

```
// Retrieve the Note Document Element Schema Item
const noteElem = schema.elements['Note'];
// Get the Attributes definition for this Element
const attrs = noteElem.declaration.elementAttributes;
// Search for the 'type' attribute to get the defined/allowed values
for(const attr of attrs)
{
if(attr.Name === 'type')
{
var restrictions = attr.Restriction !== null ? attr.Restriction : [];
// Use the attribute value setting from the schema
resultSettings.options = restrictions.map(function(restVal) {
return {
label: restVal.Value,
value: restVal.Value
};
});
}
}
```

Each XML attribute object in `elementAttributes` has a `Name` and a `Restriction` Property array if restrictions were defined in the XML schema. The `Restriction` Property contains Type/Value pairs.

Schema Class Methods

Name	Input	Description
`checkElementAction`		Returns True is the specified action on the given Element with the new/target Element is allowed. False otherwise.The given Element must exist in the document. That is, it must be contained in the children list of some Element.
`checkElementAction`	action	Either ‘ `insert` ’ or ‘ `append` ’. Designates the intended action to check.
`checkElementAction`	source	Element that the specified Action is to be performed on. Since the accepted actions only apply to the addition of content, they are checked relative to an existing Element.
`checkElementAction`	target	Type name (String) representing the Document Type to check to see if an append or insertion if allowed.
`getElementNames`		Returns a String array of all the Document Element Names in the schema