Document Type Configuration

Overview

A Technical Document Configuration defines the XML Schema used for a Technical Document, style configuration using Cascading Stylesheets, Methods used for Content Generation and Editor rendering customizations, and is maintained by a Document Type (`tp_XmlSchema`). This section provides a description of each of these areas.

Document Schema

The content of all Technical Documents is defined as XML and the Technical Document Editor generates this XML as authors add Document Elements. The types of Document Elements, along with the number of each allowed (cardinality) is defined by an XML Schema.

Important

XML and XML Schema are ubiquitous for defining digital content and the rules for this information respectively. This document assumes the reader's familiarity with both.

Important

This document will use the terms 'XML Element' and 'Document Element' interchangeably. However, the term 'Document Element' has additional meaning in the way they are handled specifically by the system. Document Element 'Types' are defined in Section Document Element Types.

Standard XML Schema

The Technical Documentation Application includes a default schema that defines a document structure called **Standard**. It is possible to customize this schema as well as develop additional custom schemas to meet business requirements.

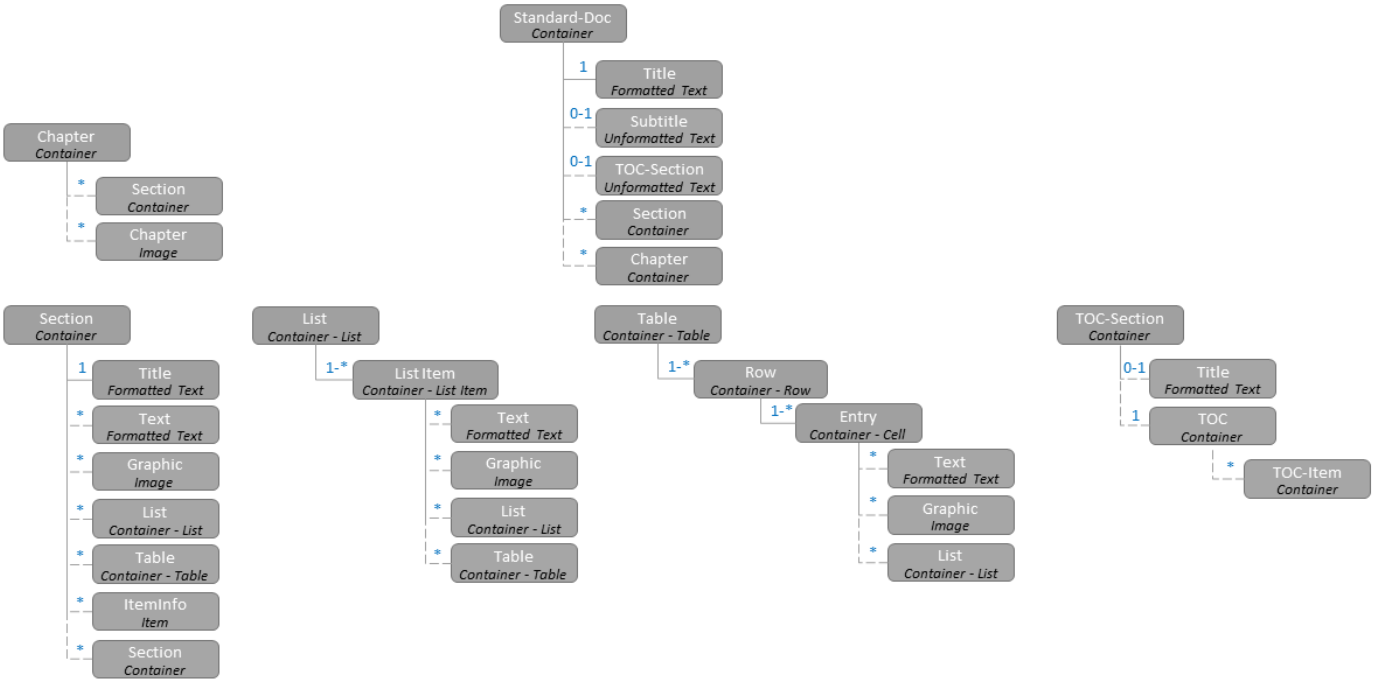
Important

For more information on how to use the Technical Documentation Editor, see Aras Technical Documentation – User Guide for reference.

Important

Administrators should use the *Standard* Document Type for reference only. Modifications / Customizations to a Document Type is best applied to a new of copy of the Standard Document Type.

The **Standard** schema implements the following structure:



1. *Standard* Document Type

XML Schema

To understand how XML schema is used, it is helpful to discuss a portion of the schema provided with the ‘Standard’ Document Type, which is included by default.

```
<xs:element name="Standard-Doc">
<xs:complexType>
<xs:sequence>
<xs:element ref="Title"/>
<xs:element ref="Subtitle" minOccurs="0" maxOccurs="1"/>
<xs:element ref="TOC-Section" minOccurs="0" maxOccurs="1"/>
<xs:choice>
<xs:element ref="Section" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="Chapter" minOccurs="0" maxOccurs="unbounded"/>
</xs:choice>
</xs:sequence>
</xs:complexType>
</xs:element>
```

The top-most schema element with name '**Standard-Doc'** is essentially a *container* for the content defined within it. For Document Types, content within another schema element is either defined as a *sequence* or a *choice*. In the XML Schema definition, a sequence is an ordered list, and a choice is unordered.

Important

There are other restrictions for using either < xs:sequence > or < xs:choice > that are not covered in this document.

Each schema element for a complex type is defined within either a `sequence` or `choice` and can have specific cardinality defined, which identifies the minimum and maximum number of XML Elements allowed. Minimum and Maximum counts are defined by the `minOccurs` and `maxOccurs` attributes respectively. Undefined `min` / `maxOccurs` attributes default to the value ' `1` '. In this case, and XML Element with name 'Standard-Doc' can have, as children, the following XML Elements:

- A single, required '**Title**' XML Element, followed by
- An optional '**Subtitle**' XML Element, followed by
- An optional '**TOC-Section**' XML Element (see Section 0), followed by either
- 0 or more '**Section**' XML Elements or
- 0 or more '**Chapter**' XML Elements

Each of the XML Elements in this example is defined in another portion of the XML Schema. XML Schema 'structures' defined in this manner thus define the structure of the corresponding content in a Technical Document. The following example shows what content using this schema might look like.

Note that the content for a '**Chapter**' XML Element was not described above but is included in the Standard Document Type.

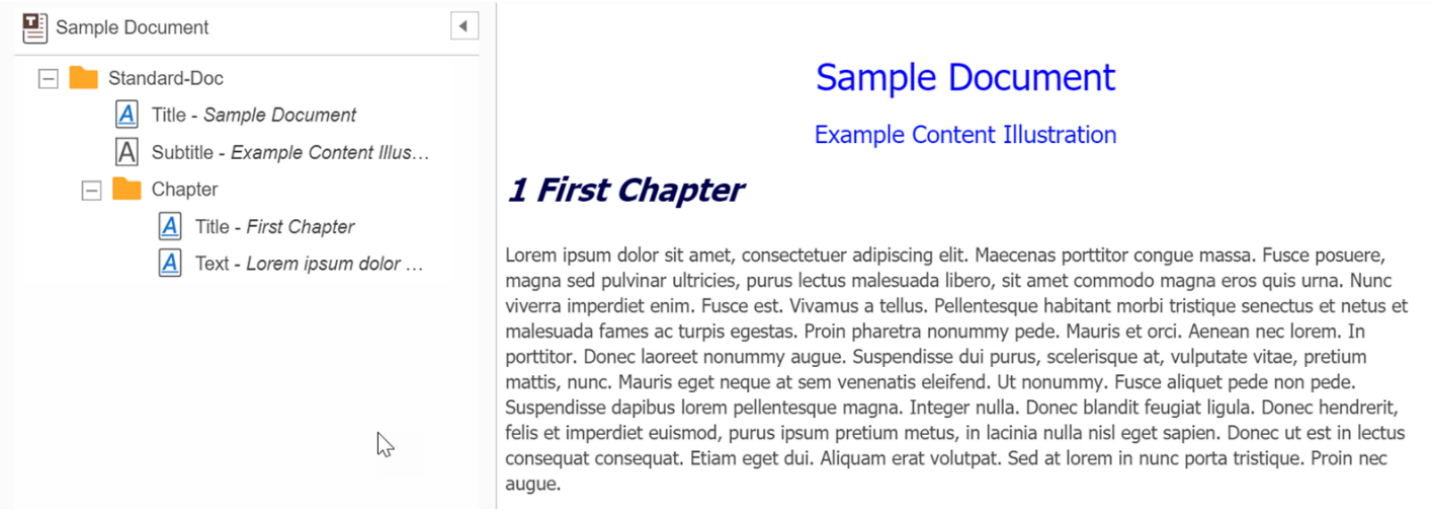


Figure: Sample Content from Standard Document Type

Document Element Types

XML Elements defined within the XML Schema for a Document Type can be configured to extend XML Elements defined by the system. In this way these Document Elements are processed differently and provide additional functionality within the Technical Document Editor. This section describes each of the Document Element Types.

Container Elements

The simplest Document Element is identified as a 'Container Element' because it is used specifically to *contain* other Document Elements and thus has no other special processing by the Editor. 'Standard-Doc' is an example of this type. 'Section', 'Chapter', and 'Graphic-Block' are also examples of Container Elements defined in the 'Standard' Document Type. Note that many of the other Document Types defined in this section are also used to contain other content, but they have additional qualities and are thus processed differently by the Editor.

Item Elements

Item Document Elements are used to associate content within a Technical Document with a separate Item (Business Object). Typically, Item Document Elements also have a Content Generator (Section

4) defined so that the Item Document Element's content can be automatically generated based on information extracted from the associated Item. This type of association is critical to a Digital Thread and helps maintain a link between technical documentation and the business objects it describes. Once a Document Element is created, the system will prompt the author to select the referenced Item and a relationship will be created between that selected Item and the Technical Document Item in which the Item Document Element was placed. To illustrate this process, the following example of an Item Document Element is described:

```
<xs:element name="PartData">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemType">
<xs:choice maxOccurs="1">
<xs:element ref="Table" minOccurs="0" maxOccurs="1"/>
</xs:choice>
<xs:attribute name="typeId" type="xs:string" fixed="<Part ItemType Id>"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

This example describes a '**PartData**' Item Document Element. Item Document Elements extend the `aras:itemType` XML Element. This example defines a single child Document Element – **Table**. In addition, the XML Attribute `typeId` is used to fix the associated/referenced ItemType to a **Part** ItemType.

Important

Defining the typeId is optional. If it is not defined the ItemType of the referenced Item can be any of the PolySources defined within tp_Item. The value of the fixed attribute must be the ItemType ID of one of the PolySources configured.

When placing an Item Document Element, the author has a choice to either choose an existing Item or create a new one.

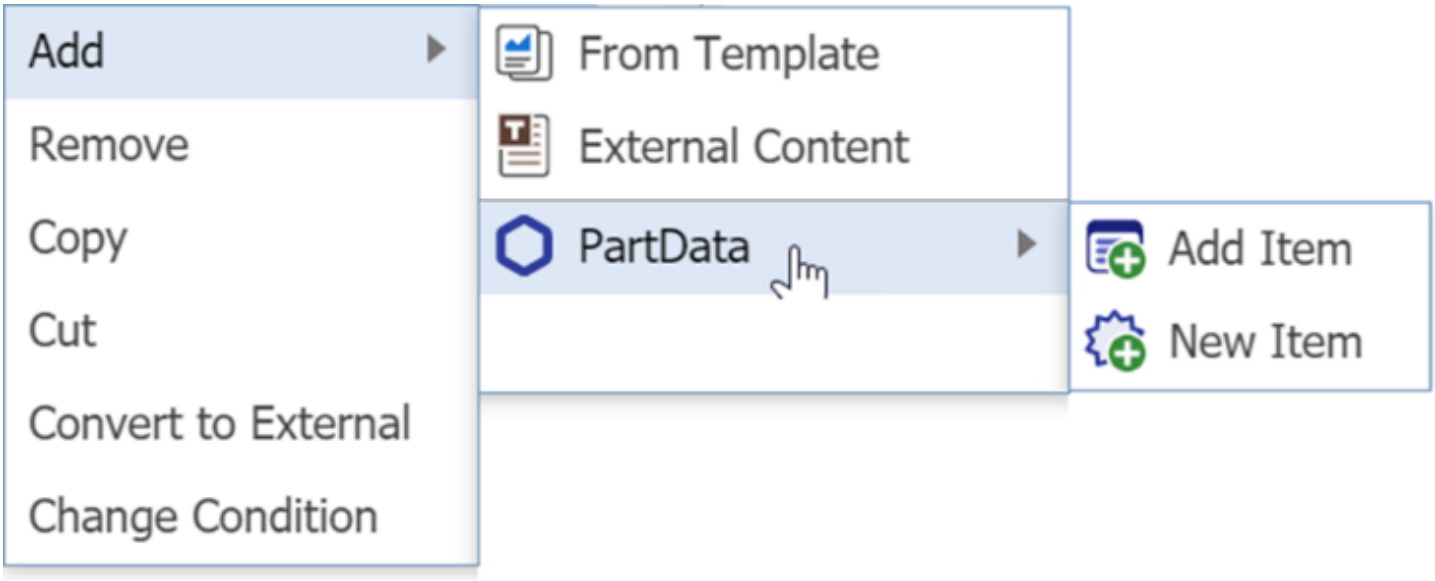


Figure: Adding an Item Document Element

If the author chooses '**Add Item**', the system will display a search dialog to choose a specific Item. If the `typeId` is used in the schema, only ItemTypes corresponding to the ItemType id specified will be displayed. Otherwise, the user can select the specific ItemType or search all.

If the author chooses '**New Item**', the system will display the Form of the ItemType so that the Property values can be specified before the Item is created. If the `typeId` is not specified in the schema, the author will first need to select the specific ItemType to be created. Authors can change the referenced Item for existing Item Document Elements with either a different, existing Item or based on a newly created Item. The functionality is like 'Add' and 'New' as defined above respectively.

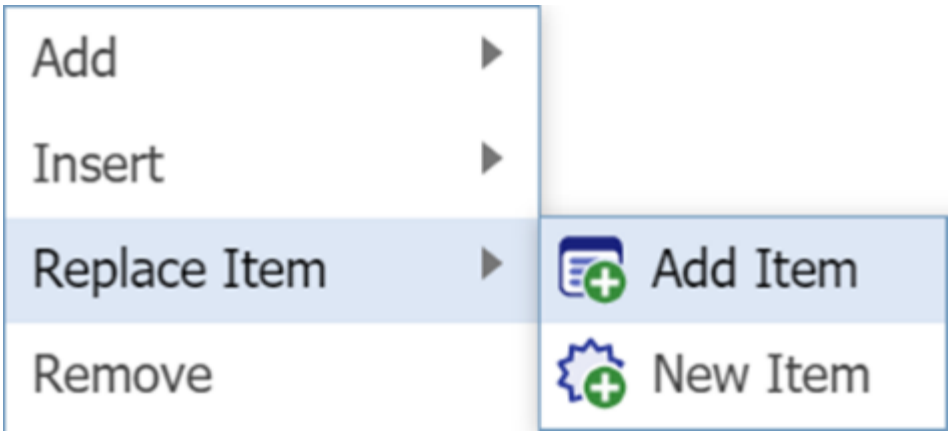


Figure: Replacing a Referenced Item

Text Elements

Text Document Elements are used for textual data. Text Document Elements can either be formatted or non-formatted. Formatted Text Document Elements can have text content within them formatted with a limited set of options. Unformatted Text Document Element do not have formatting options.

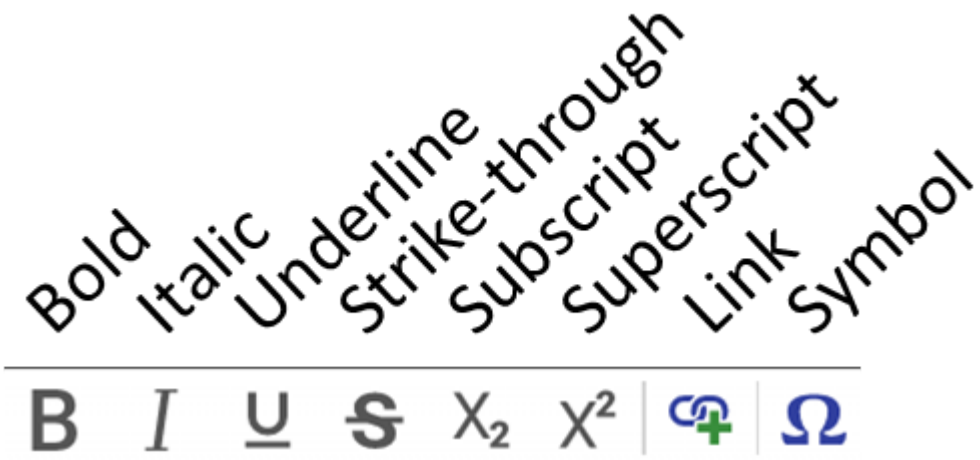


Figure: Formatted Text Options

Important

The intent for Technical Documentation is to have the format of content be managed by CSS Styles as much as practical. This establishes consistency across all technical documents that are based on the same Document Type.

The following example shows two types of Text Document Elements.

```
<xs:element name="Subtitle">
<xs:complexType mixed="true"/>
</xs:element>
<xs:element name="Text">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:text"/>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

In this example the '**Subtitle**' Document Element is unformatted, and the '**Text**' Document Element is formatted. Unformatted Text Document Elements are defined as a complex type with the `mixed` attribute set to ' `true` '. Formatted Document Elements extend the `aras:text` XML Element.

Important

The Technical Document Editor will attempt to place a Text Document Element whenever the user has selected some Container Document Element (Container, Cell, List Item, etc.) and types a character. At this point, the system will search for any child Document Element of type Text allowed by the schema, insert a new instance of that Text Document Element, apply the characters typed by the user, and place the cursor at the end for further text input.

Image Elements

Image Document Elements are used to place 2D images in a Technical Document. All Image Document Elements use the `tp_Image` ItemType to contain the 2D image file and are thus independently managed.
When a user places an Image Document Element, the system will display a Search Dialog for Items of type `tp_Image` (Graphic). Like Item Document Elements, an author can either select an existing Graphic Item or create a new Item. In the latter case, the system will present a Form for the author to enter necessary Properties before the new Graphic Item is created. Referenced Graphic Items will have a Relationship Item generated between the Graphic Item and the Technical Document Item in which the Image Document Element was placed. The following example shows an Image Document Element schema definition:

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

In this example, the '**Graphic**' Image Document Element is created. Image Document Elements must extend the `aras:ImageType` XML Element.

Important

Authors can also insert new Graphic Items by pasting an Image from the clipboard. If a Document is selected in the Technical Document Editor that *allows* a following sibling Image Document Element of any name defined in the schema, the system will

automatically create a Graphic Item using the Item Reference Configuration (see Item Reference Configuration) to automatically populate required Graphic Item Properties and upload the image in the clipboard as File Item to the vault.

Mapped Properties

Mapped Property Document Elements are a type of Unformatted Text Document Element that are used to reference a Property of an Item referenced by an ancestor Item Document Element. These are also referred to as *Item* Property Document Elements. When authors place instances of a Mapped Property Document Element, the system will automatically apply the content of the associated Property of the referenced Item. In the following example, a '**ShippingInformation**' Item Document Element contains '**LineItem**' Container Document Elements, which contain a single '**Text**' (Formatted Text Document Element) and a '**ShipProperty**' Mapped Document Element:

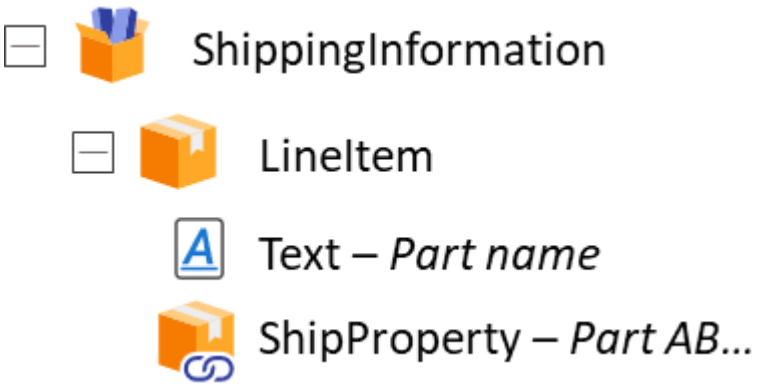


Figure: Mapped Property Example

The corresponding XML Schema would be specified as follows:

```
<xs:element name="ShippingInformation">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:itemType">
<xs:choice maxOccurs="1">
<xs:element ref="LineItem" minOccurs="1" maxOccurs="unbounded"/>
</xs:choice>
<xs:attribute name="typeId" type="xs:string" fixed="<ItemType Id>"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="LineItem">
<xs:complexType>
<xs:sequence>
<xs:element ref="Text" minOccurs="1" maxOccurs="1"/>
<xs:element ref="ShipProperty" minOccurs="1" maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="ShipProperty">
<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="read"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

The example illustrates a possible scenario in which Shipping Information is maintained within a separate Item (Business Object). In this case, there may be several Properties related to the shipping information managed. Using Item Document Elements with Mapped Property Document Elements in this manner provide a convenient mechanism to automatically populate content in a technical document when these Document Elements are placed manually and update the content dynamically when the referenced Item is changed.

See Item Document Element in the Appendix.

The '**ShippingInformation**' XML Element extends `aras:itemType` , is *fixed* to refer to an ItemType (presumably containing Shipping Information), and contains a single '**LineItem**' Container Document Element. The '**LineItem**' Container Document Element contains a single Text Document Element - '**Text**' - and a single Mapped Property Document Element – '**ShipProperty**' – which extends `aras:itemProperty` . The '**ShipProperty**' Mapped Property Document Element is configured with two, optional Attributes: '**property**' and '**mode**'.

Important

The ‘property’ Attribute specifies the name of the Property to reference. The association between a Mapped Property Document Element and an Item Property is by the Property’s name. This association is loose, in that any ItemType with the same Property name will result in a mapping. Since referenced Items are associated via a PolySource, mapping by name allows the reference to ‘float’ to different ItemTypes.

Important

The ‘mode’ Attribute specifies whether the reference Item Property is editable. A Mode value of ‘read’ specifies that the Property is read-only. A Mode value of ‘write’ specifies that the Property of the referenced Item can be updated based on changes in the technical document.

If the ‘**property**’ Attribute is configured, it can specify the default Property name to map to (using the ‘ `default` ’ attribute) or fix the value using the ‘ `fixed` ’ attribute. If the ‘property’ Attribute is not configured, authors can manually specify the Property to map using the Attributes Dialog:

Figure: Attributes Dialog for Mapped Property Document Elements

If the ‘**mode**’ Attribute is configured, it can specify the default Property mode (using the ‘ `default` ’ attribute) or *fix* the value to either ‘read’ or ‘write’ using the ‘ `fixed` ’ attribute. If the ‘property’ Attribute is not configured, authors can manually specify the Property to map using the Attributes Dialog. Fixed values for these of these two Attributes will disable the ability to change them in the Attributes Dialog.

Mapped Property Document Elements are edited using a pop-up Form in the Technical Document Editor. To edit a value, select the Document Element in the Editor or Tree View and/or double-click to

display the Property Editor Form. Document Elements with a grey, dotted border can be edited. Document Elements with a red, solid border are read-only. For example:

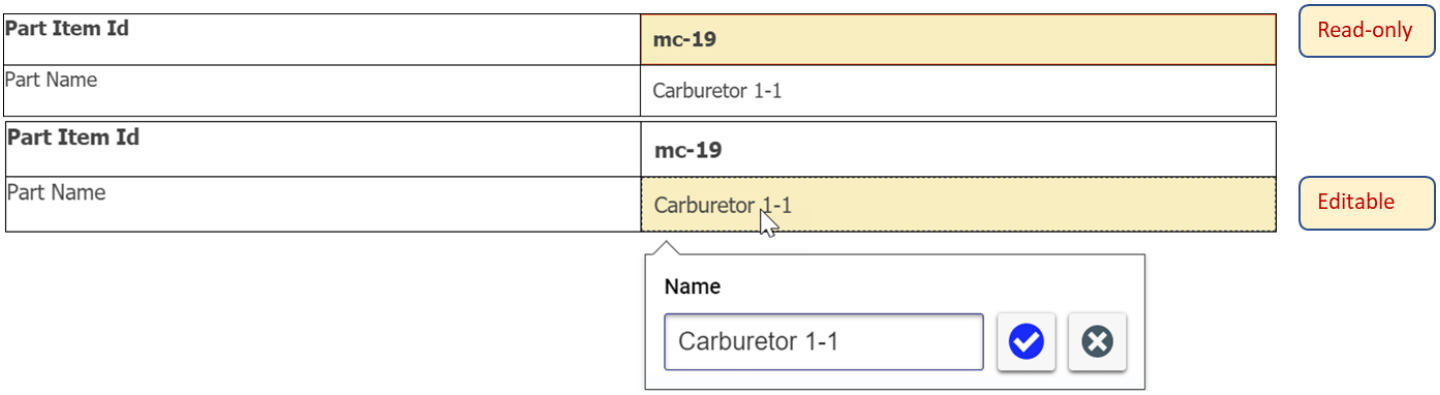


Figure: Mapped Property Editor Form

Values can only be changed using this Form. Selecting the ‘Check’ button will update the Mapped Property Document Element in the technical Document only.

Authors will see a pencil icon, signifying that the Property of the referenced Item has not been updated, next to the Document Element in the Tree View. The referenced Item will be updated when the technical document is saved, in which case the pencil icon will be hidden. Selecting the ‘X’ button will cancel the update without any changes to the technical document.

Important

Selecting the pencil icon () in the Tree View will prompt the author with a Dialog to discard the updated Property. In this case selecting ‘OK’ will revert the modified Property(ies) to their previous value.

The following Property Value Types are supported:

- String
- Text
- Numeric (Integer, Decimal, etc.)
- Date
- Boolean
- Item
- List

- Image (Display only - Editor not available)

The Edit Field in the Property Edit Form is based on the Property Type. For Text and Numeric Fields, an attempt to set a value that does not match the type will result in an Error being displayed in a Dialog.

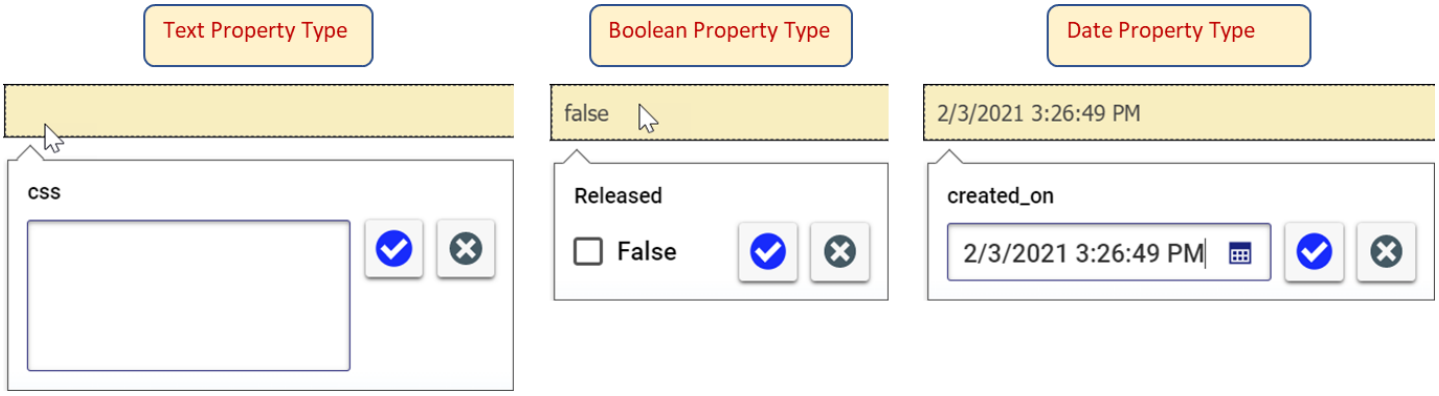


Figure: Mapped Property Edit Forms

Modifying a versionable Item vis-à-vis Mapped Property Document Elements will result in the referenced Item to be versioned automatically. Since Technical Documents reference Items via fixed Relationships, the Item Document Element will show as stale since it will refer to the previous generation of the Item.

Table Document Element

Table Document Elements are used to group content in a tabular format. Document Elements that are configured as Tables have additional functionality provided in the Technical Document Editor to provide the ability to add columns, rows, merge cells, etc. Thus Tables, Rows, and Cell Document Element are a type of Container Document Element, but with extended capability.

> **Important**
>
> It is possible to display content in a Technical Document as a Table without extending the Table and related Row or Cell XML Element Types using CSS Flex Box for example, but these Elements will not have the additional Table functions.

Configuring a Document Element as a Table requires corresponding Document Elements for Rows and Cells. Table Document Elements can be nested within other Table Document Elements. The following is an example of a Table Configuration in XML Schema:

```
<xs:element name="Table">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:table"/>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="Row" substitutionGroup="aras:tablerow">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:tr"/>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="Entry" substitutionGroup="aras:tablecell">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:td">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="Text" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="List"/>
<xs:element ref="Graphic"/>
<xs:element ref="Table"/>
</xs:choice>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

Table Document Elements extend the `aras:table` XML Element. Table Document Elements must contain Row Document Elements as their only child element. Likewise, Row Document Elements must contain Cell Document Elements as their only child element. Cell Document Elements can contain other types of content, including other Tables as specified above. Row Document Elements extend the `aras:tr` XML Element. Cell Document Elements extend the `aras:td` XML Element.

Important

The XML Schema file Aras.xsd contains the core XML Schema for Technical Documentation. The schema file defines aras:tablerow and aras:tablecell, shown as substitution groups in the example above.

Important

The Technical Document Editor will attempt to place a Text Document Element whenever the user has selected some Container Document Element (Container, Cell, List Item, etc.) and types a character. At this point, the system will search for any child Document Element of type Text allowed by the schema, insert a new instance of that Text Document Element, apply the characters typed by the user, and place the cursor at the end for further text input.

Important

There can only be one Table, Row, and Cell Document Element defined in a Document Type.

List Document Element

List Document Elements are used to group content in a sequential list format. Document Elements that are configured as Lists have additional functionality provided in the Technical Document Editor to provide the ability to set the individual List Items as bulleted, numeric, or alpha. Thus Lists, and List Item Document Elements are a type of Container Document Element, but with extended capability.

Important

It is possible to display content in a Technical Document as a List without extending the List and related List Item XML Element Types using CSS display='list-item' for example, but these Elements will not have the core functionality to set the bullet, numeric or alpha capability.

Configuring a Document Element as a List requires corresponding Document Elements for List Items. There can be multiple List Document Elements defined and it is possible to nest Lists within Lists. The following is an example of a List Configuration in XML Schema:

```
<xs:element name="List">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:list">
<xs:choice maxOccurs="unbounded">
<xs:element ref="List-Item" minOccurs="1" maxOccurs="unbounded"/>
</xs:choice>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="List-Item">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:listitemType">
<xs:choice maxOccurs="unbounded">
<xs:element ref="Text" />
<xs:element ref="Graphic"/>
<xs:element ref="List"/>
</xs:choice>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

List Document Elements extend the `aras:list` XML Element. List Document Elements must contain List Item Document Elements as their only child element. List Item Document Elements can contain other types of content, including other Lists as specified above. List Item Document Element extend the `aras:listitemType` XML Element.

Important

The default List Document Element specifies at least one child List Item Document Element, as shown in the example above. When there is a single child Document Element that is required, the system will automatically place the child Document Element. For this List example, when the user places a List, the system will automatically insert the first List Item Document Element.

Important

The Technical Document Editor will attempt to place a Text Document Element whenever the user has selected some Container Document Element (Container, Cell, List Item, etc.) and types a character. At this point, the system will search for any child Document Element of type Text allowed by the schema, insert a new instance of that Text Document Element, apply the characters typed by the user, and place the cursor at the end for further text input.

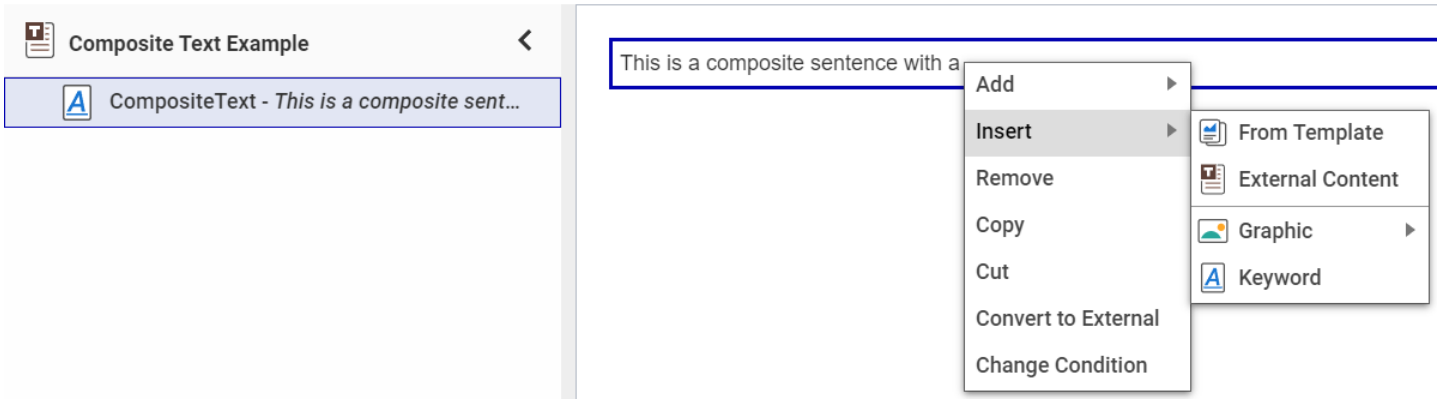
Embedded Document Element

Embedded Document Elements are any Text, Image, Item, or Item Property Document Element that is configured as a direct child of another Text Document Element. The ability to *embed* Document Elements in this manner allows sentences to include other types of content interleaved within the text. The following example show a configuration for a ‘CompositeText’ Text Document, which can contain a ‘Keyword’ Text Document Element or a ‘Graphic’ Image Document Element.

```
<xs:element name="CompositeText">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:text">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="Keyword"/>
<xs:element ref="Graphic"/>
</xs:choice>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="Keyword">
<xs:complexType>
<xs:complexContent>
<xs:extension base="aras:text"/>
</xs:complexContent>
</xs:complexType>
</xs:element>
```

In the example schema above the ‘CompositeText’ Text Document Element is configured to optionally include either a ‘Keyword’ Document Element (which is another Text Document Element) or a ‘Graphic’ (which is explained in other sections of this document). Embedding graphics/images allows text element to contain images within the text. This is useful for symbols that are stored as images for example.

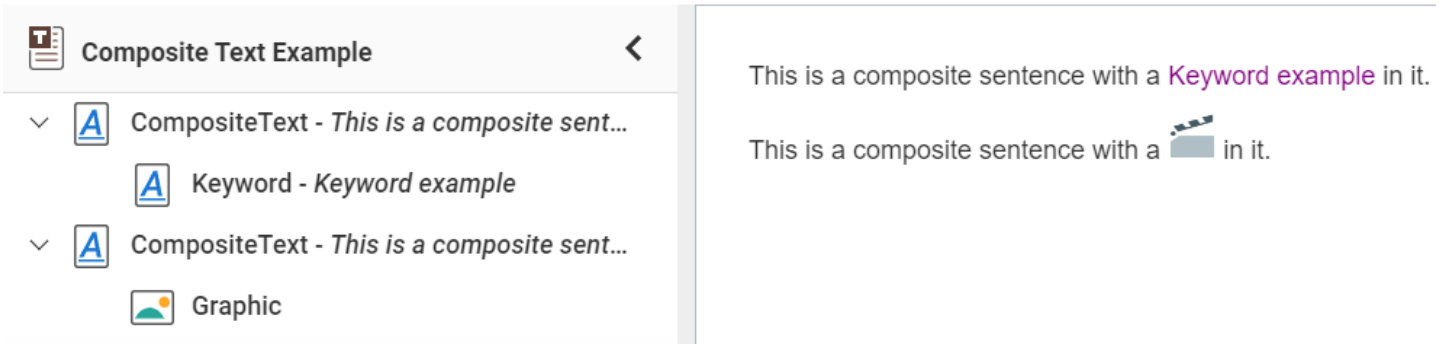
When configuring one type of Text that can be included within another the embedded text can be distinguished semantically and/or visually. When Text Document Elements are configured with Embedded Elements an **Insert** menu option is added, which will display the configured child types. Using the schema sample above would result in the following Insert options:



Inserting Embedded Elements

The following example shows the result of a Keyword and a Graphic being embedded in separate

composite sentences. Note that the Keyword is styled to be rendered in purple.



Schema Validation

The Document Type Form uses a Text Editor to display and edit the XML Schema definition. This Editor has been configured specifically for XML Schema to highlight XML keywords and show validation errors/warnings with indicators of such on the corresponding line in the User Interface. The XML Schema is validated as the user edits the content and when the user Saves the Document Type Item. If there are any warnings or errors, the user is prompted with a message. It is not possible to save a Document Type Item if there are schema errors. This is to prevent the inadvertent use of the schema if it is not valid.

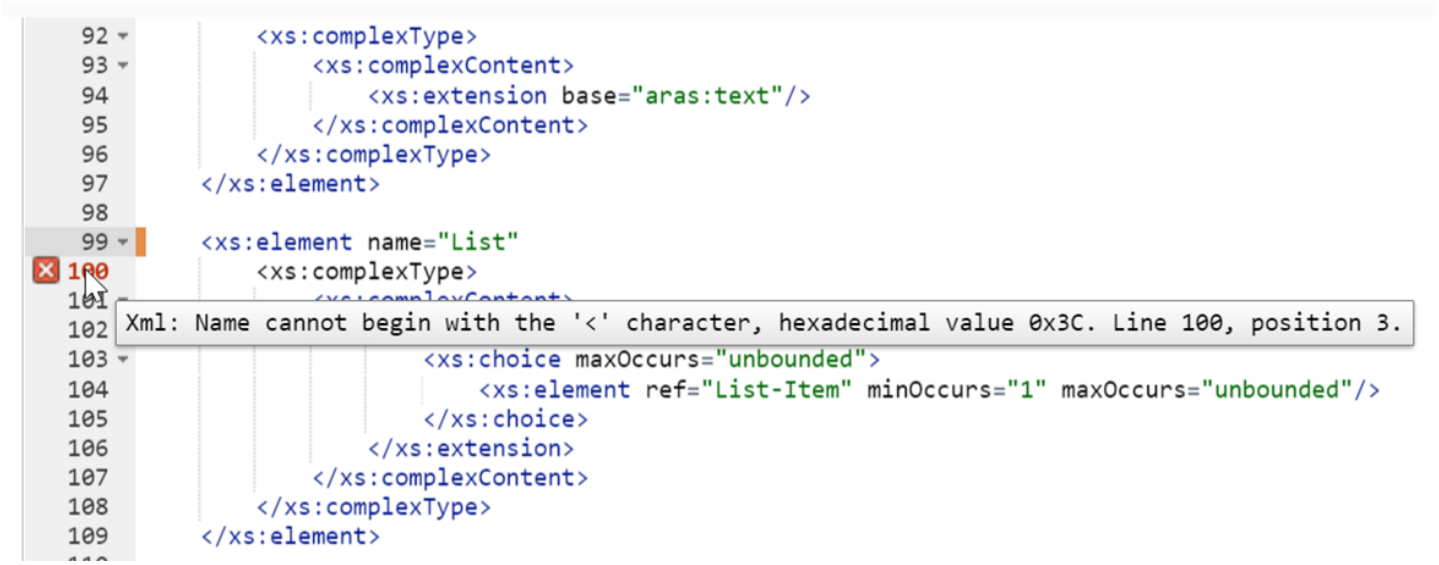


Figure: XML Schema Validation Warning

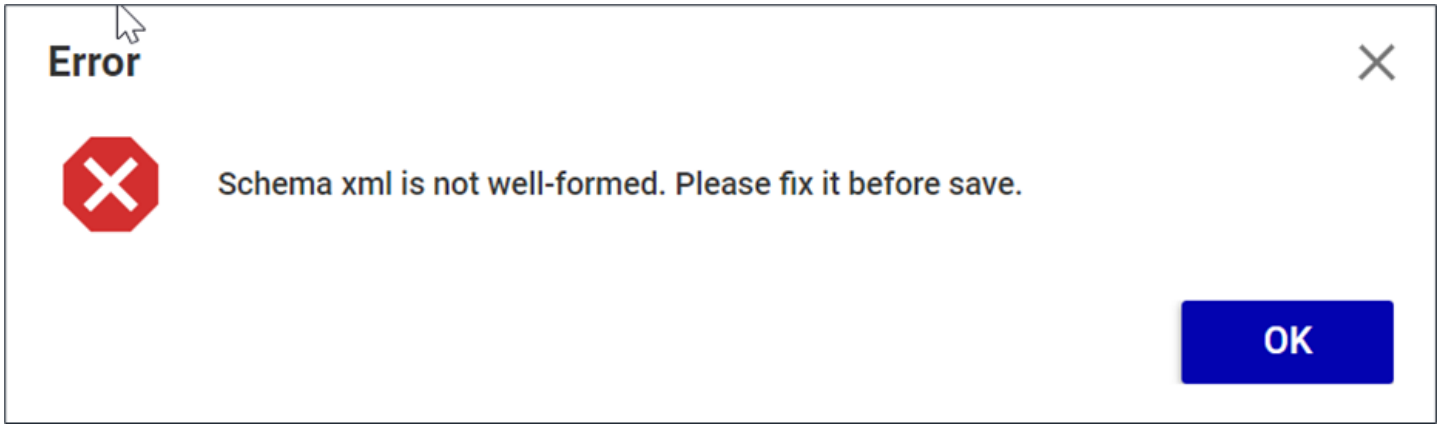


Figure: XML Schema Validation Error Dialog

Document Element Configuration

For each Document Element, the Technical Document Framework provides the ability to configure Content Generators , Methods to prepopulate and validate Properties on referenced Items , custom Tree Node Renders, and the default action for specific Classifications of Technical Document Enabled Items. These configuration options extend the default behavior of Document Element Types.

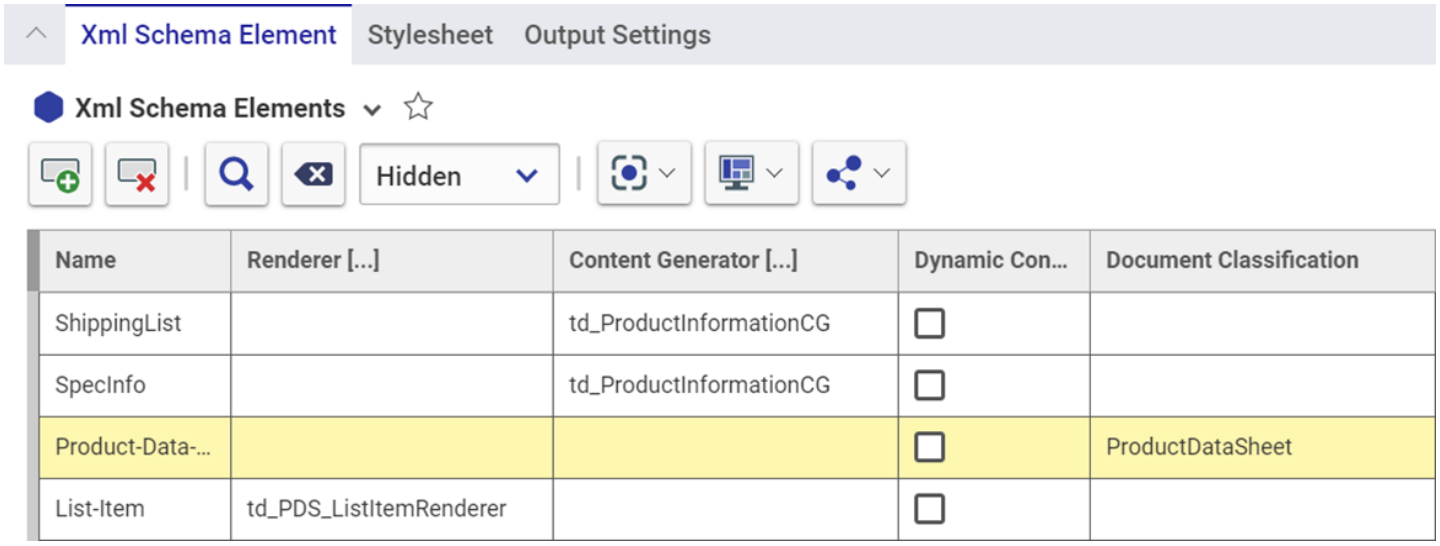


Figure: Document Element Configuration User Interface

Document Element configuration is specified using XML Schema Element Items. References to these Items are added in the Document Type Form. For each XML Schema Element Item, the associated Document Element is identified by **Name**. It is important that the spelling of the name

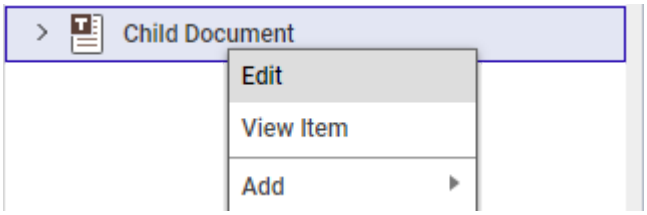
match what is defined for the XML Element in the XML Schema. The **Renderer** identifies the JavaScript Method that is used as an alternative to the default logic for populating the Tree Node Icon and Text. The **Content Generator** identifies the C# Server-side Method that is called whenever an instance of the corresponding Document Element is placed in a Technical Document or when it is refreshed. The **Dynamic Content** checkbox determines how often a configured Content Generator is executed. When *False*, the Content Generator is only executed when the Document Element is first created or when it is manually refreshed in the Technical Document Editor. When *True*, the Content Generator is additionally executed whenever the Technical Document is opened or saved.
Document Classification associates the Document Element with a Classification for the Technical Document Enabled Item. When the Technical Document Enabled Item is first created, with a specified Classification, the system will automatically add the Document Element as the root. In this case, only a single root element is allowed for this technical document.

Editing Referenced Technical Documents

Technical Documents can reference (and thus be composed of an aggregation of) the content of other Technical Documents so that documentation is managed by individual document *modules*. By default, all referenced document content cannot be edited within the context of a Technical Document that references it. Changing this content would thus require opening the associated Technical Document Item and editing the content from there. However, it is possible to enable referenced content editing in one of two modes: Implicit or Explicit.

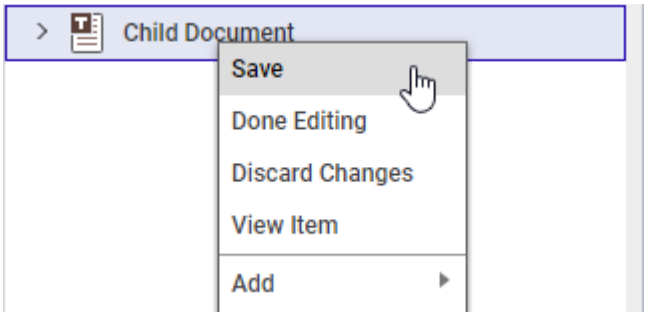
> **Important**
>
> See Enabling the Ability to Edit Child Documents for more information on how to configure external edit mode.

Implicit Edit Mode allows authors to edit referenced content as if the referenced content was part of the opened document without any need to lock this content prior. In this mode, when the parent (or opened) document is saved, the referenced document content is locked, updated, unlocked in a continuous series of steps for any referenced document content that was changed.
Explicit Edit Mode will also allow authors to edit referenced content but will require that the associated Technical Document Item is first locked for Edit. This can be done directly in the Technical Document Editor or separately as is part of normal Lock/Unlock operations in Aras Innovator.



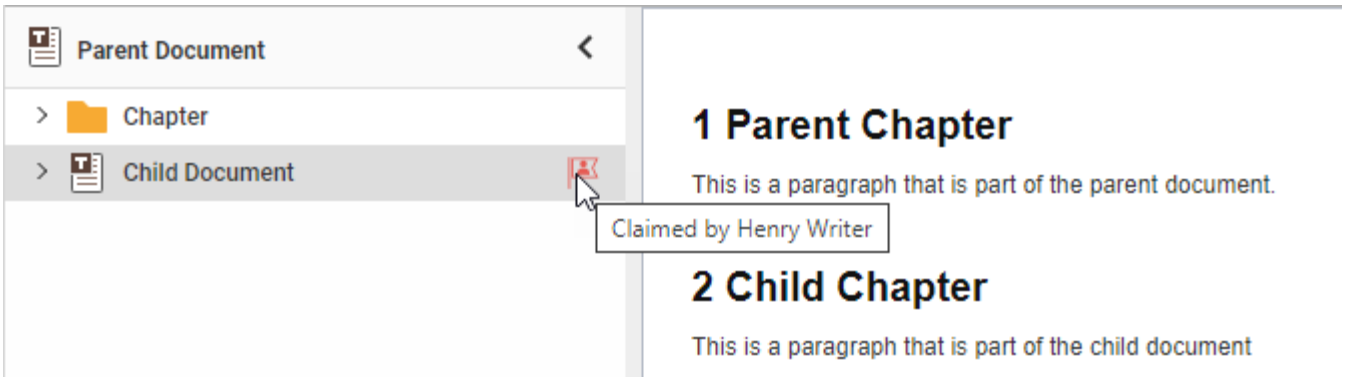
Example Edit Command in Context Popup Menu

Explicit Edit Mode also requires that referenced Technical Document Items be unlocked when the author has completed their updates. This can be done within the Technical Document Editor or separately as part of the normal Item Lock/Unlock operations in Aras Innovator.



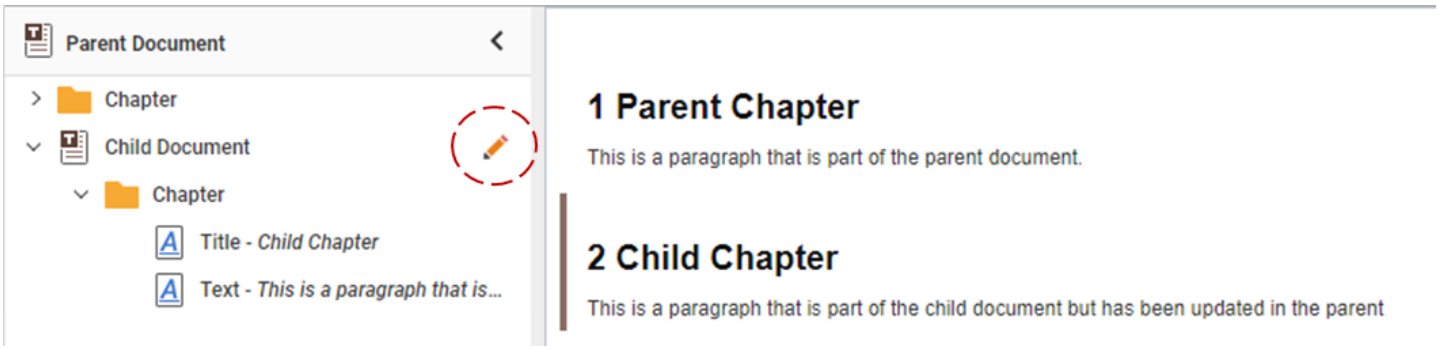
Example Save, Done Editing, Discard Changes Command in Context Popup Menu

In Explicit Mode, referenced and locked Technical Document Items are saved when the parent document is saved. Authors can also save referenced document content directly using the context popup menu. When a referenced document is claimed or locked by a separate User, the Technical Document Editor will display an icon for the node associated with the referenced document in the Structure Tree.



Example of Locked Referenced Content

Hovering the mouse over the icon will display the name of the User who has the Technical Document claimed or locked. Any referenced document content that has been changed and not saved will also cause the display of an icon in the Structure Tree as an indication that the content has not been saved.



Example of Unsaved Referenced Content

CSS Styles

The format and placement of Document Element content in a technical document is largely controlled by the CSS settings configured with the associated Document Type.

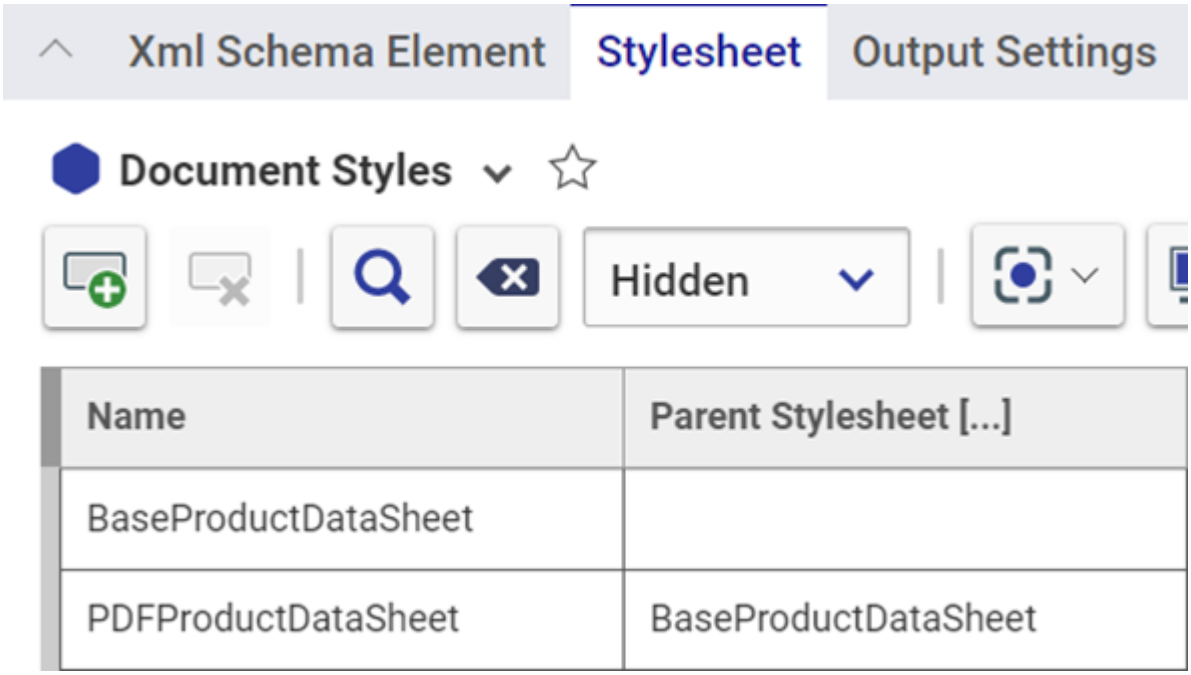


Figure: Cascading Stylesheet Items There can be multiple Stylesheet Items configured and each one can extend (cascade) from others through the **Parent Stylesheet** selection. When the content a Technical Document is rendered in the Editor or in the HTML published format, each Document Element in the document is converted to a `<div>` HTML element with a class set equal to the Document Element name. For example: `<div class="Text">…</div>` .

Important

Table Document Elements are converted to <table>, <tr>, and <td> elements. List Document Element are converted to <ul>, <ol>, and <li> elements. Image Document Elements use <img> elements.

CSS Example

The example Technical Document:

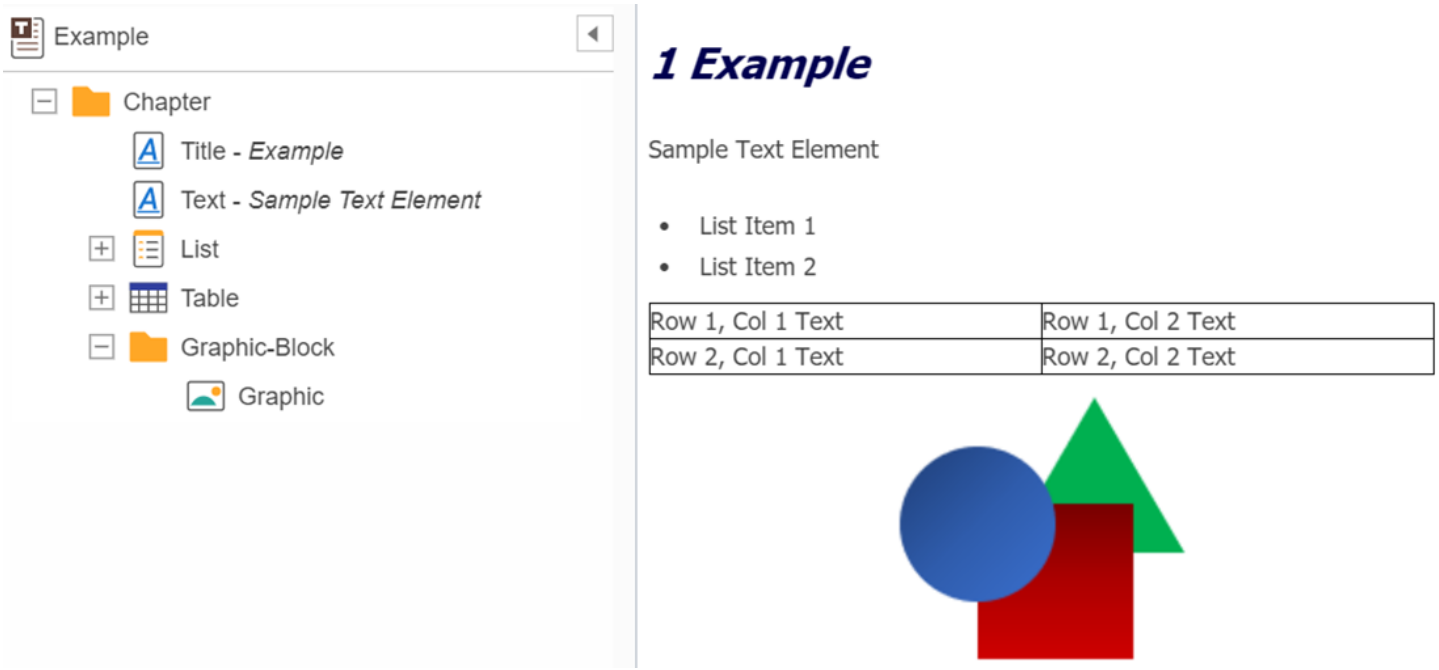


Figure: Sample Technical Document … will result in HTML like:

```
<body>
<div documentId="…">
<div class="block ArasXmlElement" blockId="…" by-reference="internal">
<div class="Chapter …">
<div class="Title … ArasTextElement">
<span class="emph … ArasStringElement">Example</span>
</div>
<div class="Text … ArasTextElement">
<span class="emph … ArasStringElement">Sample Text Element</span>
</div>
<ul class="List … ArasListElement" type="bullet">
<li class="List-Item … ArasListItemElement bulletListItem">
<div class="Text … ArasTextElement">
<span class="emph … ArasStringElement">List Item 1</span>
</div>
</li>
<li>…</li>
</ul>
<table class="Table … ArasTableElement" ColWidth="50|50">
<colgroup><col width="50%" /><col width="50%" /></colgroup>
<tr class="Row … ArasTableRowElement">
<td class="Entry … ArasTableCellElement" valign="top" align="left">
<div class="Text … ArasTextElement">
<span class="emph … ArasStringElement">Row 1, Col 1 Text</span>
</div>
</td>
<td>…</td>
</tr>
<tr>...</tr>
</tr>
</table>
<div class="Graphic-Block …">
<img class="Graphic … ArasImageElement" src=”…” ... />
</div>
</div>
</div>
</div>
</body>
```

Figure: Sample Generated HTML

The Example above contains instances of common Document Elements (i.e., Text, Table, List, Image). The XML content (not shown) is like the HTML. The top-most HTML element is a `<div>` that represents the entire document and contains an attribute for the Technical Document Item Id. The second level is a single `<div>` used to represent the content of the document. It is assigned a class of ‘block’.

The Document Elements placed into the Document start at the third level of `<div>` s. Notice that class value contains the name of the Document Element as assigned in the Document schema. It also contains class names assigned based on the XML Elements that are extended in the Document

Type schema. Also notice that Formatted Text Document Elements (‘Text’ in this example) have embedded `<emph>` tags around the text content. These are used to assign the formatted style (Bold, Italic, Underline, etc.) assigned in the Technical Document Editor.

Defining a Cascading Stylesheet

CSS content is defined using the CSS Editor, accessed from the Style Items added to the Document Type:

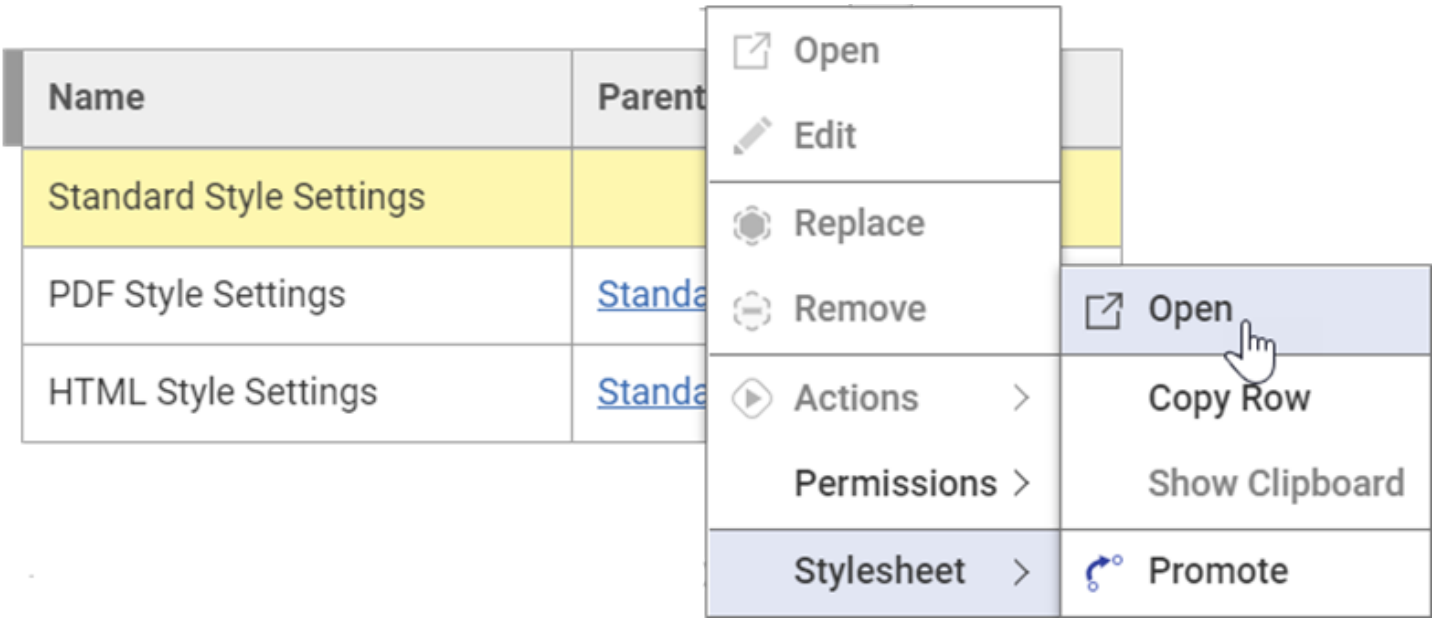


Figure: Opening a Stylesheet Item

Figure: Stylesheet User Interface

The CSS Form Editor is configured to display CSS content with syntax highlighting to match the elements of the content. Note that CSS selectors using the class name is the preferred method to identify the generated HTML elements.

> **Important**
>
> This document assumes the reader's familiarity with CSS and the methods used to isolate specific elements within HTML.

The editor form element can be extended/resized by dragging the handle at the lower right corner.

CSS Formatting Examples

This section contains examples for setting CSS attributes for various types of Document Elements. It is meant to provide a guide for customizing the layout and style of the reader's content.

Identifying a class of Document Element

HTML Elements can be identified in a CSS Selector by the standard element name (e.g., div, p, li, etc.), using an element's ID, or by a class name. The class name selector is the approach used for Technical Documents. The class selector uses a dot '.' followed by the name of the class. As described in this document, the name of the Document Element is stored within the class attribute so it can be used as a mechanism to target the application of CSS parameters. The following selector is used to declare properties for the entire HTML body:

```
body {
text-align: left;
font-family: "Tahoma", Geneva, sans-serif
}
```

The following selector is used to declare properties for the Document Element with name 'Text':

```
.Text {
color: #505050;
}
```

In the first example, the HTML Element name is used. The second example uses the class style selector is used (not the preceding '.') to select *all* Document Elements with the name 'Text'. The following selector is used to declare properties for the Document Element with name 'Text' that are within a Table:

```
.Table .Text {
color: #FF0000;
}
```

In this example above the first part of the Selector identifies the elements with a class of 'Table', followed by a space, followed by the Selector for elements of class 'Text'. In this case, any Text Document that is a descendant of a Table Document Element will use the Property Declaration.

```
.Row:nth-child(1) {
background-color: #ccc;
text-align: left;
font-weight: bold;
}
```

In this example above, any content that is a descendant of the first instance of a Row Document Element will use the Property Declaration. Such a format is used to modify the style for text in the

first row of a table. In this last example, alternating rows in a table will have different background colors:

```
.Row:nth-child(even) {
background-color: #ddd;
}
.Row:nth-child(odd) {
background-color: #eee;
}
```

Centering Images

Centering images within the Editor, HTML output, and PDF output can be configured with CSS like:

```
.Graphic img {
display: block;
margin-left: auto;
margin-right: auto;
}
img.Graphic {
display: block;
margin-left: auto;
margin-right: auto;
}
```

This example defines a Graphic Document Element, named ‘Graphic’. All image content uses the `img` HTML Element for displaying the image. However, the Technical Document Editor uses a format like:

```
<div class=”Graphic>
<img …/>
</div>
```

The HTML output format used for HTML and PDF however uses a format like:

```
<img class=”Graphic”…/>
```

Therefore, both declaration formats are required to center an image in the Editor and in visual output formats. In the top declaration, the img element are targeted as children of other elements with a class = ‘Graphic’. The bottom declaration targets `img` elements with a class of ‘Graphic’.

Using Counters

Technical Documents rely on CSS Counters for automatic numbering used for Chapter, Section, Figures, and Lists. CSS Counters use Properties that are implicitly defined in the CSS content to manage counter values. Accessor functions within the CSS language can then be used to increment and reference Counter Property Values. The following example show the process of declaring, incrementing, and accessing Counter Property values:

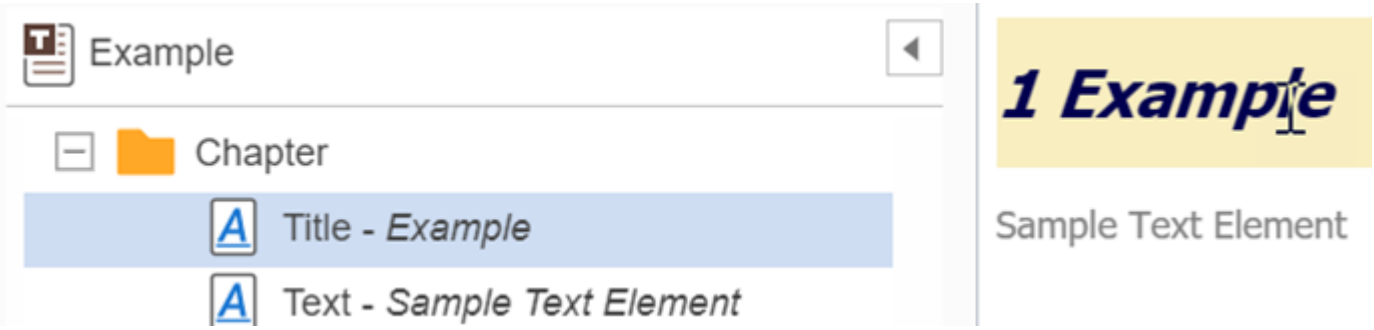


Figure: Chapter Numbering Example

```
body {
counter-reset: sectionNum; /* Counter used for Chapter / Section numbering */
}
.Chapter {
counter-increment: sectionNum;
}
.Chapter > .Title::before {
content: counter(sectionNum) ' ';
}
```

In this example a Counter Property – ‘sectionNum’ is declared (using ‘counter-reset’) within the body declaration. It is incremented (using ‘counter-increment) within the Chapter class declaration and used (via the counter() function) with the ‘before’ pseudo-element for the ‘Title’ Document Element that is an immediate child of a ‘Chapter’ Document Element.

Counter Properties must be declared before they are incremented or used/accessed. In this case, the CSS declaration for body ensures that the Counter is declared before any ‘Chapter’ elements are encountered. ^[1] Each time the HTML process encounters elements with a class = ‘Chapter’ it will increment the value of the counter (counters start at ‘0’ for numerical counters). Once an element with a class name of ‘Text’ is encountered, that is a direct child of a ‘Chapter’ element, it will precede the content for the ‘Text’ Document element with the value of the Counter followed by a single space.

Important

Using CSS Counters is an advanced topic. Readers should consult with CSS Reference guides for a full description for how to apply this functionality.

---

1. CSS is processed top down. Thus, the first <body> element will be processed before and descendant elements.

Adding Attributes to customize Document Element Style

In this example, an Attribute (named 'size') is added to an Image Document Element (named 'Graphic) in the Document Type 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>
```

The values for the 'size' Attribute are restricted to the values: 'small', 'medium', and 'large', as defined by the 'ImageSizeAttType' Simple Type defined in the XMl Schema. The values can be set for the size Attribute for any 'Graphic' Document Element using the Attributes Dialog in the Technical Document Editor:

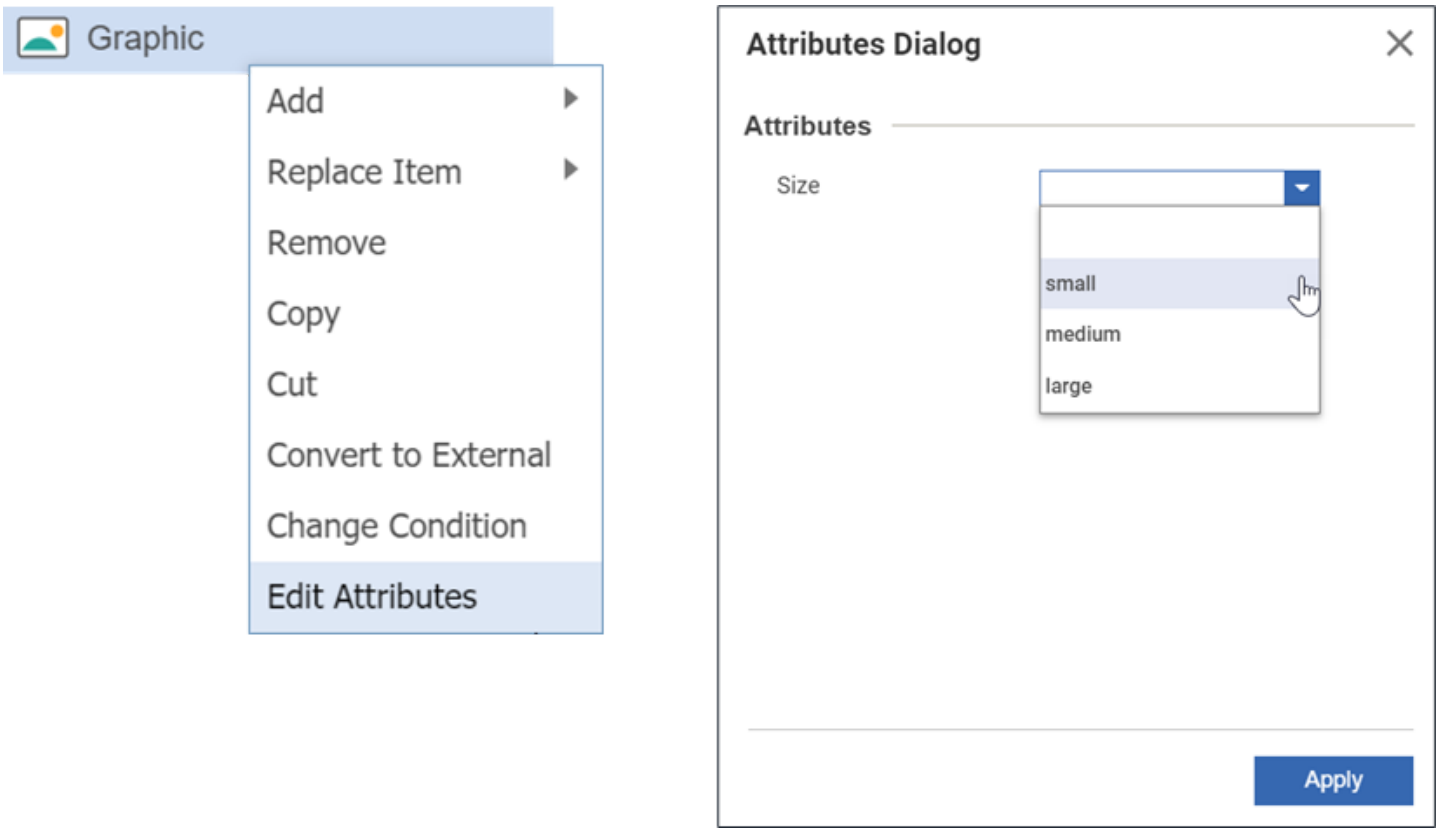


Figure: Image Size Attribute Example

CSS selectors can then be defined to isolate ‘Graphic’ elements with size Attributes with specific values as follows:

```
img[size='small'].Graphic, .Graphic[size='small'] img
{
max-width: 200px;
}
img[size='medium'].Graphic, .Graphic[size='medium'] img
{
max-width: 400px;
}
img[size='large'].Graphic, .Graphic[size='large'] img
{
max-width: 600px;
}
```

In this example the CSS Attribute Selector is defined for each of the Attribute values to set the maximum image width for each. In this manner, when the author selects certain values, the size of

the image is updated accordingly. There is no default value specified for the ‘size’ Attribute so empty values will default to the default size of the image.

Important

HTML content for images is generated differently for the Editor than for the HTML and PDF output.

Publishing Technical Documents

Aras Technical Documentation solution provides the capability to export the contents of a Technical Document as XML, HTML, or PDF. This part of the solution is available to Aras’s subscribers. Publishing also requires that the Aras Conversion Server and Aras Agent Service be configured properly.

Required Installations

To publish Technical Documents, the following needs to be installed:

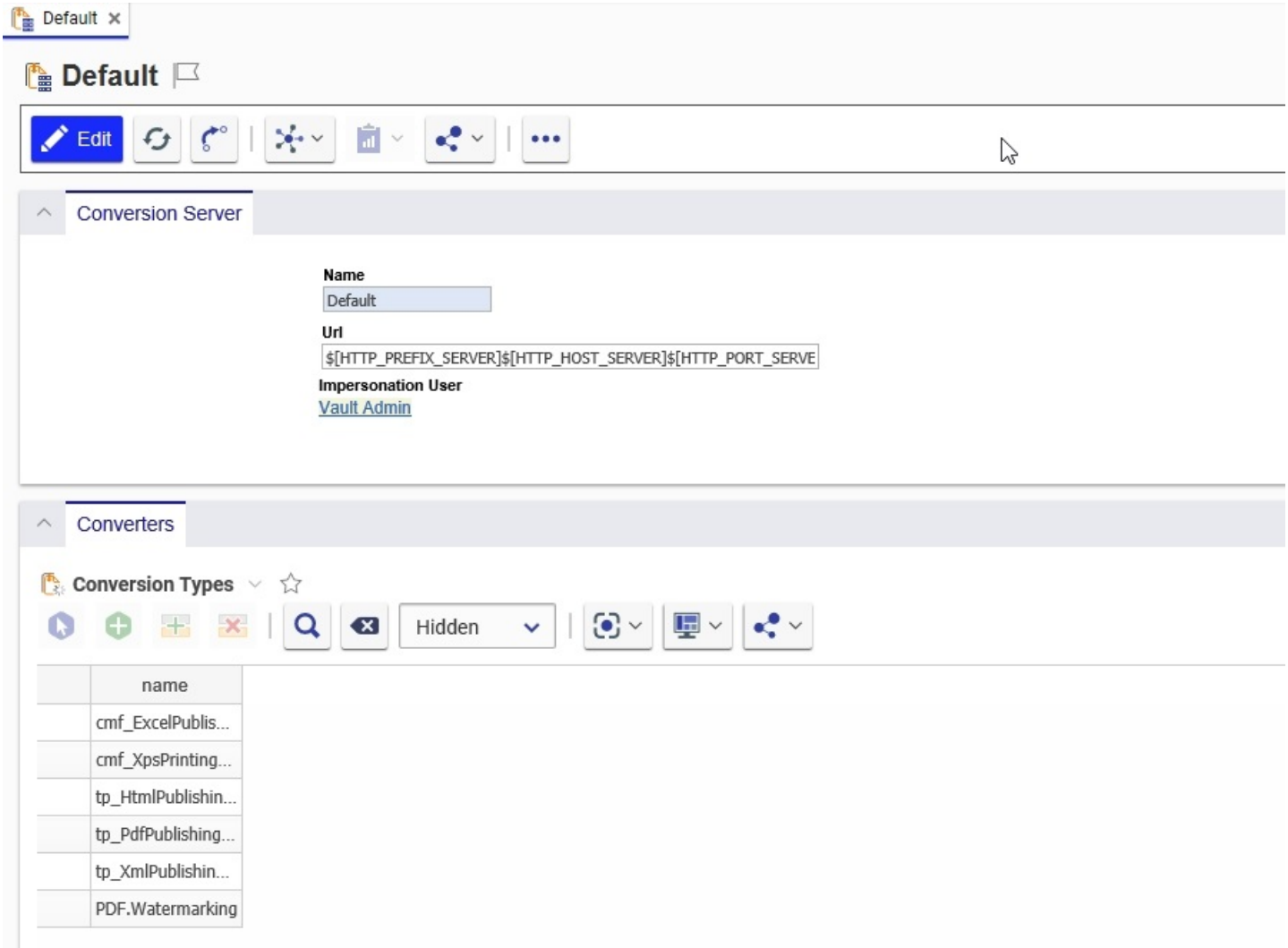
- Aras Conversion Server (See Aras Innovator - Conversion Server Setup Guide)
- Aras Agent Service (See Section Restarting the IIS Server After Configuration of the Aras Innovator – Installation Guide)
- Aras Publishing Service (See Aras Innovator – Publishing Service Setup Guide)
- Aras HTML to PDF Converter (See Aras Innovator – HTML to PDF Converter Setup Guide)

Publishing Technical Documents requires two Feature Licenses – Aras.PublishingService and Aras.HTMLtoPDFConverter.

Managing Conversion Rules

By default, the conversion rules required for Aras Technical Documentation Publishing are added to the **Default** Conversion Server. If your Innovator setup requires the use of a different Conversion Server, the following Conversion Rules should be added to the **Converters** tab on the appropriate **Conversion Server** item in the database:

- tp_HtmlPublishingConverter
- tp_PdfPublishingConverter
- tp_XmlPublishingConverter



Output Settings

The Output Settings map a stylesheet, configured in the **Stylesheet** tab (Section CSS Styles), to a selected Output Type: *Editor*, *HTML*, or *PDF*. Although it may be preferable to have document content to appear the same in all three, there are specific CSS settings that only apply to paged content (e.g, PDF) and Administrators may choose to have certain content styled differently in the Editor.

Important

There are default style settings that are applied for each Output Type. Setting the style for each will apply the configured stylesheet in addition to the default settings.

To create an Output Type, Edit the associate Document Type and add a row in the Output Setting tab:

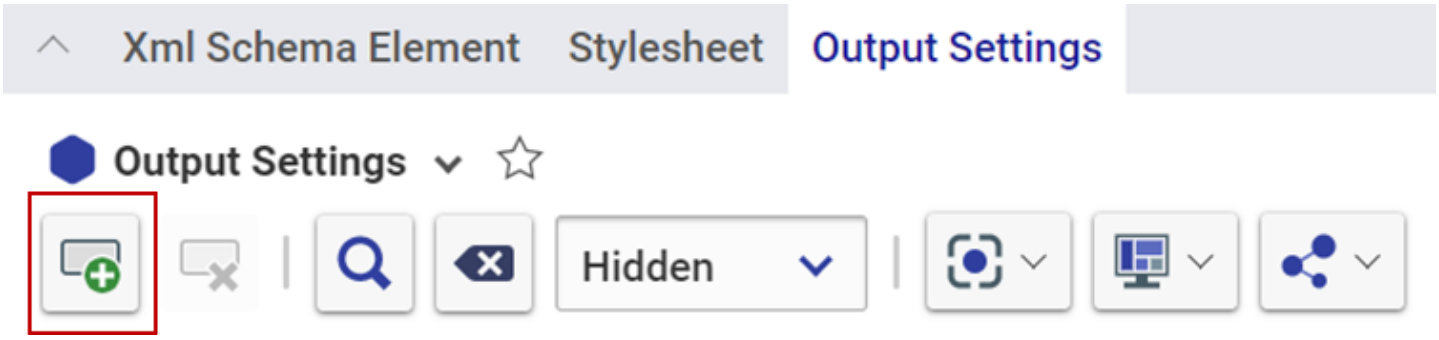


Figure: Create Output Setting Select the Output Type:

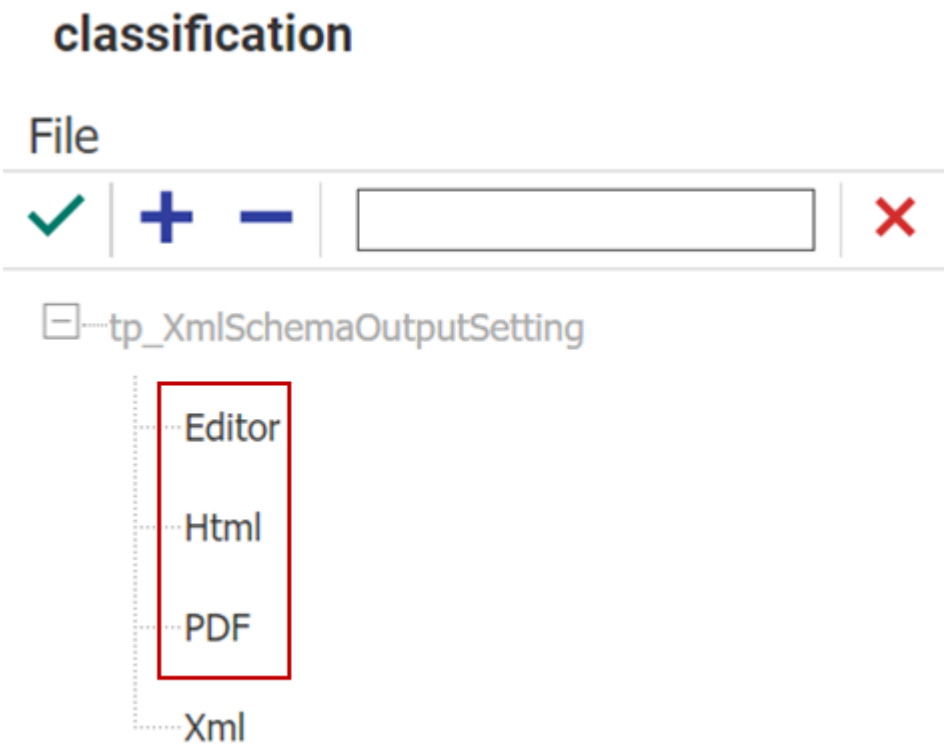


Figure: Output Type

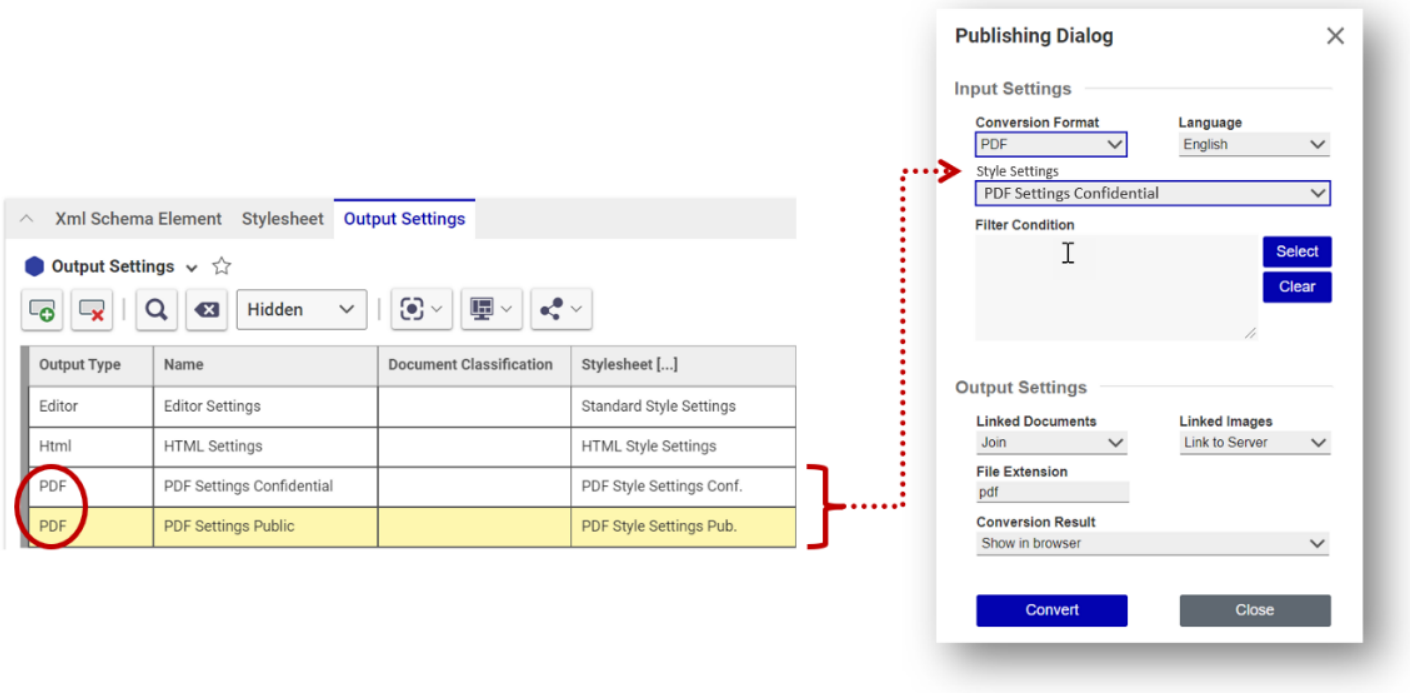
Important

'Xml' is not a valid Output Type for the Output Settings. XML content is always displayed as ASCII content with no configurable styling.

Specify a unique Name for the Output Type. Optionally, a Classification for the associated Tech Doc Enabled ItemType (e.g., tp_Block) can be selected. In this case, the selected Stylesheet will only apply for Documents that have this Classification specified. Finally, choose the Stylesheet created in CSS Styles.

Style Settings

For HTML and PDF output formats, and when multiple Output Settings are configured, the user has the ability to select one of the Output Style Settings when publishing. This capability allows a user to publish the same Technical Document with different styles/configuration based on a choice made in the Publishing Dialog.



Style Selection for Published Content

The **Style Settings** form field will be enabled, and the drop-down list populated, whenever there has been more than one Output Setting configured for the selected **Conversion Format**. In these cases, the initial/default selection is based on the order of the Output Settings configured for the Document

Type. Selecting the drop-down list will expose all other Output Settings. Selecting a specific Style Setting will result in the selected Output Setting to be applied during the publish operation. If there is only one, it will be selected by default and the Style Settings drop-down will be disabled.

Key Handling

Key handling in the Document Editor refers to how the editor responds to key events (e.g., Enter, Backspace, Delete, etc.). Although an author would seldom notice how key event handling is executed, how these events are processed are important in a structured document editor. This is because how key events are processed can significantly improve the user experience and provide more efficient authoring.

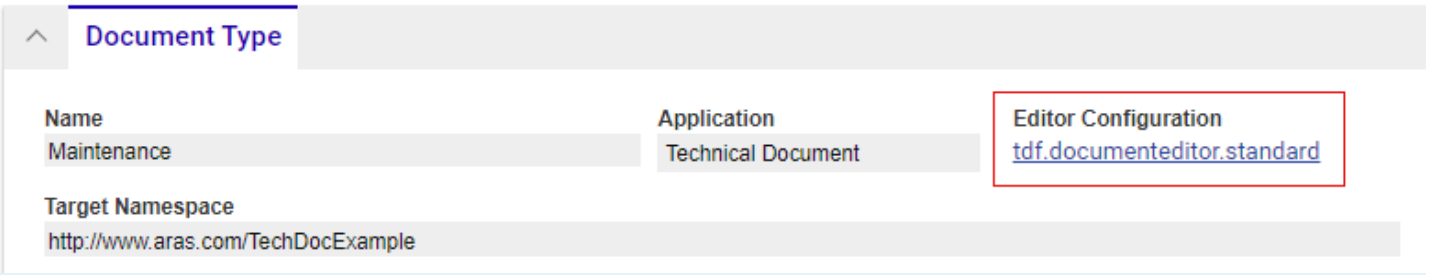
It is possible to enable extended key-handling logic, or create custom key-handlers, using CUI. This section describes the process for enabling extended key-handling logic to improve the authoring user experience by forcing the editor to behave more like common text editing or word-processing applications.

Enabling Extended Key-Handlers

A Document Type can be configured to reference OOTB key-handling logic that alters the standard functionality of the Document Editor for Technical Documents that reference that specific Document Type. This will affect how the Delete, Backspace, and Enter key functions execute for Text and List / List Item Document Elements.

For Text Elements, selecting the Enter Key while editing text within a sentence will automatically create and append a separate Text Document Element and place the cursor at the beginning of the line. Any text that existed after the cursor will be removed from the previous Text Document Element and pasted in the new Text Document Element. Similarly, when a Backspace Key is selected while the cursor is at the start of a Text Element will automatically cut the text from the current Document Element, paste it at the end of the previous Text Document Element, and remove the current Text Document Element. This operation is only done if there exists a preceding Text Document Element. The Delete Key executes in a similar way but only when the cursor is at the end of a Text Document Element and there exists a following, peer Text Document Element. Thus, these actions execute in a manner similar to how paragraphs are automatically split or merged in a typical text editor. For Text Document Elements that are the last child of a List Item Document Element, selecting the Enter Key will result in the creation of a new List Item Element, with a Text Document Element child and place

the cursor in the next Text Document Element. Any text that was after the previous cursor location will be cut and added to the new Text Document Element. Thus, these actions execute in a manner similar to how items in a list are created in a typical text editor. These additional key handling features need to be configured for each Document Type as they are not added by default. To do so, add the ‘tdf.documenteditor.standard’ Presentation Configuration to the Editor Configuration property of the Document Type.



Important

The key handling logic in the ‘tdf.documenteditor.standard’ Presentation Configuration will process Text Document Elements and List Item Document Elements only.

Important

When adding the Presentation Configuration, be sure to select the one with the name ‘tdf.documenteditor.standard’. There may be a Presentation Configuration instance with the name ‘tdf.editorconfiguration.standard [14.0.29 obsolete]’. This uses a version of the TDF Client API that is no longer supported with versions of Aras Innovator 40 and above.