Oracle® Agile Product Lifecycle Management for Process Extensible Columns Guide

Feature Pack 4.3

E79162-01

May 2017

ORACLE®

Copyrights and Trademarks

Agile Product Lifecycle Management for Process

services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface

Audience

This guide is intended for client programmers involved with integrating Oracle Agile Product Lifecycle Management for Process. Information about using Oracle Agile PLM for Process resides in application-specific user guides. Information about administering Oracle Agile PLM for Process resides in the *Oracle Agile Product Lifecycle Management for Process Administrator User Guide*.

Variability of Installations

Descriptions and illustrations of the Agile PLM for Process user interface included in this manual may not match your installation. The user interface of Agile PLM for Process applications and the features included can vary greatly depending on such variables as:

- Which applications your organization has purchased and installed
- Configuration settings that may turn features off or on
- Customization specific to your organization
- Security settings as they apply to the system and your user account

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Software Availability

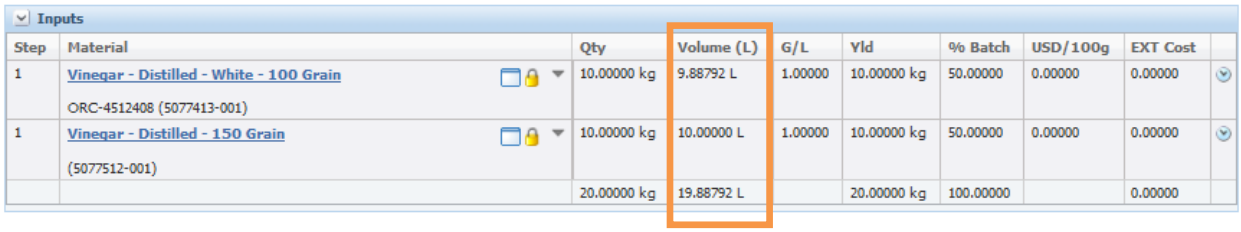
Oracle Software Delivery Cloud (OSDC) provides the latest copy of the core software. Note the core software does not include all patches and hot fixes. Access OSDC at:

http://edelivery.oracle.com

Chapter 1—Overview

Extensible Columns are extension points that allow clients to create their own custom columns and add them to various user interface grids in the application. These columns can be used to display existing data elements or calculate new data values.

> For example, a customer might want to calculate the volume associated with each input item on a formulation spec, and display that volume in the UI. This column would take the quantity value and use the density on the material to calculate and display the volume.



Inputs

Step	Material	Qty	Volume (L)	G/L	Yld	% Batch	USD/100g	EXT Cost
1	Vinegar - Distilled - White - 100 Grain ORC-4512408 (5077413-001)	10.00000 kg	9.88792 L	1.00000	10.00000 kg	50.00000	0.00000	0.00000
1	Vinegar - Distilled - 150 Grain (5077512-001)	10.00000 kg	10.00000 L	1.00000	10.00000 kg	50.00000	0.00000	0.00000
		20.00000 kg	19.88792 L		20.00000 kg	100.00000		0.00000

Extensible Columns are created through custom plugin classes and configuration changes. Column plugins for the Formulation spec grids are configured differently than the plugins on other grids, and include the ability to display a footer, perform custom calculation methods before or after the main calculations occur, and store the calculation results.

This document details the process of creating, configuring, and deploying Extensible Columns to extend the PLM for Process application.

Chapter 2—Availability

Extensible Columns are available in the locations specified in the table below, each implemented by an ExtensibleColumnPlugin class, and enabled by a FeatureConfig entry. Formulation input and output extensible columns are configured differently, in the CustomFormulationExtensions.xml file.

Note that the main grids each support up to 5 extensible columns and plugins. However, only the first plugin will show in the print output. By default, the column will be added in the last position in the printed section. You can customize this view by adjusting the print XSL templates. See the print extensibility guide for more information around how to adjust print XSL templates.

Table 1 - Extensible Column Locations

Location	Feature Configuration	Plugin Name
Cross References Grids (in GSM, SCRM and PQM)	GSM.CrossReferences.XCol.Enabled GSM.CrossReferences.XCol[2-5].Enabled SCRM.CrossReferences.XCol.Enabled SCRM.CrossReferences.XCol[2-5].Enabled PQM.CrossReferences.XCol.Enabled PQM.CrossReferences.XCol[2-5].Enabled	CrossReferencesPlugin CrossReferencesPlugin[2-5]
Material Spec - Sourcing Approvals grid	GSM.Material.Suppliers.XCol.Enabled GSM.Material.Suppliers.XCol[2-5].Enabled	GSMMaterialSuppliersXColPlugin GSMMaterialSuppliersXColPlugin[2-5]
Menu Item Spec - Menu Item Build Items grid	GSM.Menu.ItemBuild.XCol.Enabled GSM.Menu.ItemBuild.XCol[2-5].Enabled	GSMMenuItemBuildXColPlugin GSMMenuItemBuildXColPlugin[2-5]
Packaging Spec - Printed Packaging grid	GSM.Packaging.PrintedPackaging.XCol.Enabled GSM.Packaging.PrintedPackaging.XCol[2-5].Enabled	GSMPackagingPrintedPackagingXColPlugin GSMPackagingPrintedPackagingXColPlugin[2-5]
Packaging Spec - Sub-component grid	GSM.Packaging.SubComponent.XCol.Enabled GSM.Packaging.SubComponent.XCol[2-5].Enabled	GSMPackagingSubComponentXColPlugin GSMPackagingSubComponentXColPlugin[2-5]
Packaging Spec - Sourcing Approvals grid	GSM.Packaging.Suppliers.XCol.Enabled GSM.Packaging.Suppliers.XCol[2-5].Enabled	GSMPackagingSuppliersXColPlugin GSMPackagingSuppliersXColPlugin[2-5]
Printed Packaging Spec - Parent Packaging grid	GSM.PrintedPackaging.ParentPackaging.XCol.Enabled GSM.PrintedPackaging.ParentPackaging.XCol[2-5].Enabled	GSMPrintedPackagingPackagingXColPlugin GSMPrintedPackagingPackagingXColPlugin[2-5]
Printed Packaging Spec - Sourcing Approvals grid	GSM.PrintedPackaging.Suppliers.XCol.Enabled GSM.PrintedPackaging.Suppliers.XCol[2-5].Enabled	GSMPrintedPackagingSuppliersXColPlugin GSMPrintedPackagingSuppliersXColPlugin[2-5]
Equipment Spec - Sourcing Approvals grid	GSM.Equipment.Supplier.XCol.Enabled GSM.Equipment.Supplier.XCol[2-5].Enabled	GSMEquipmentSuppliersXColPlugin GSMEquipmentSuppliersXColPlugin[2-5]
Product Spec - Sourcing Approvals grid	GSM.Product.Suppliers.XCol.Enabled GSM.Product.Suppliers.XCol[2-5].Enabled	GSMProductSuppliersXColPlugin GSMProductSuppliersXColPlugin[2-5]
Trade Spec - Alternates Packaging grid	GSM.Trade.AlternatesPackaging.XCol.Enabled GSM.Trade.AlternatesPackaging.XCol[2-5].Enabled	GSMTradeAlternatesPackagingXColPlugin GSMTradeAlternatesPackagingXColPlugin[2-5]
Trade Spec - Lower Level Items grid	GSM.Trade.Child.XCol.Enabled GSM.Trade.Child.XCol[2-5].Enabled	GSMTradeChildXColPlugin GSMTradeChildXColPlugin[2-5]
Trade Spec - Material Spec grid	GSM.Trade.Material.XCol.Enabled GSM.Trade.Material.XCol[2-5].Enabled	GSMTradeMaterialXColPlugin GSMTradeMaterialXColPlugin[2-5]

Location	Feature Configuration	Plugin Name
Trade Spec - Packaging grid	GSM.Trade.Packaging.XCol.Enabled GSM.Trade.Packaging.XCol[2-5].Enabled	GSMTradePackagingXColPlugin GSMTradePackagingXColPlugin[2-5]
Trade Spec - Parent Trade Specs grid	GSM.Trade.Parent.XCol.Enabled GSM.Trade.Parent.XCol[2-5].Enabled	GSMTradeParentXColPlugin GSMTradeParentXColPlugin[2-5]
Trade Spec - Sourcing Approvals grid	GSM.Trade.Suppliers.XCol.Enabled GSM.Trade.Suppliers.XCol[2-5].Enabled	GSMTradeSuppliersXColPlugin GSMTradeSuppliersXColPlugin[2-5]
Facilities - Spec-Related Sourcing Approvals grid	SCRM.Facility.SAC.XCol.Enabled SCRM.Facility.SAC.XCol[2-5].Enabled	SCRMFacilitySACXColPlugin SCRMFacilitySACXColPlugin[2-5]
	FORMULATION COLUMNS	
Formulation Spec - Formulation Tab Inputs	* enabled in CustomFormulationExtensions.xml	* plugin declared in CustomFormulationExtensions.xml
Formulation Spec - Formulation Tab Outputs	* enabled in CustomFormulationExtensions.xml	* plugin name declared in CustomFormulationExtensions.xml
Formulation Spec - Process Tab Material Inputs	* enabled in CustomFormulationExtensions.xml	* plugin name declared in CustomFormulationExtensions.xml
Formulation Spec - Process Tab Packaging Inputs	* enabled in CustomFormulationExtensions.xml	* plugin name declared in CustomFormulationExtensions.xml
Formulation Spec - Process Tab Outputs	* enabled in CustomFormulationExtensions.xml	* plugin name declared in CustomFormulationExtensions.xml

Chapter 3—Extensible Column Plugins

Configuration

Extensible Column plugins are configured in the `CustomPluginExtensions.xml` file located in the `config/Extensions` folder.

To configure a plugin for use, there must be a `Plugin` tag with attributes `name` and `FactoryURL`, under the appropriate plugin type tag. The value of the `name` attribute identifies the specific extension point which will call out to the plugin defined here. The value of the `FactoryURL` attribute is an `ObjectLoaderURL` for the factory to be called to create an instance of the plugin. When specifying the `FactoryURL` attribute, you must also add an extra attribute to turn the existing plugin inheritance off: `ignoreInheritFromPluginName="true"`.

For example:

```
<ExtensibleColumnPlugins configChildKey="name">
       <Plugin name="FormulationInputVolumeColumnPlugin"
ignoreInheritFromPluginName="true"
FactoryURL="Class:FormulationExtensionsSample.Inputs.InputVolumeExtensibleColumnPluginFactory,
FormulationExtensionsSample"/>
```

Plugins can inherit their settings from other plugins, rather than specify the same `FactoryURL` data, by using the `inheritFromPluginName` attribute and specifying the name of the plugin they should inherit from.

For example:

```
<ExtensibleColumnPlugins configChildKey="name">
   <Plugin name="FormulationInputVolumeColumnPlugin"
FactoryURL="Class:FormulationExtensionsSample.Inputs.InputVolumeExtensibleColumnPluginFactory,
FormulationExtensionsSample"/>


   <Plugin name="FormulationOutputVolumeColumnPlugin"
inheritFromPluginName="FormulationInputVolumeColumnPlugin"/>
```

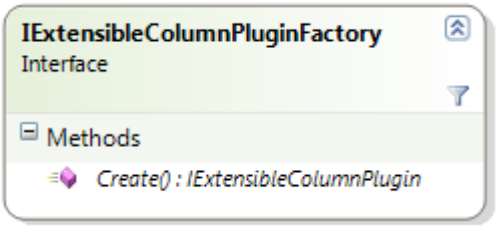
Extensible Column Plugin Classes

Extensible Column Plugin classes consist of a Header property for displaying column header information and a list of Cell values for displaying individual cell entries in the grid.

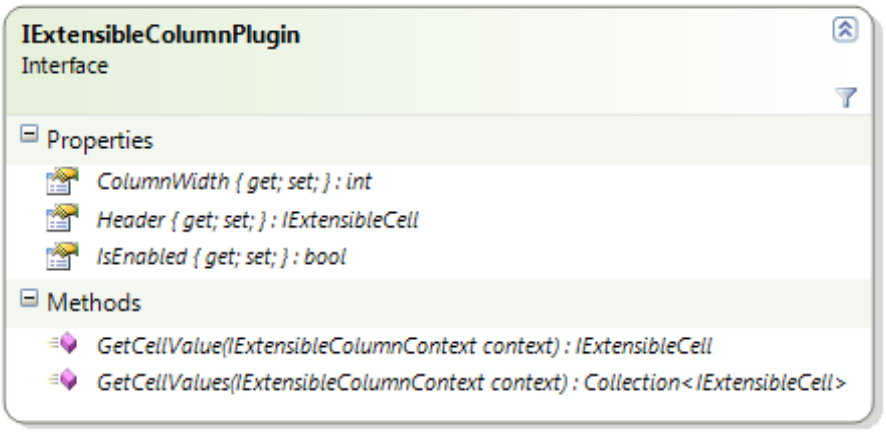
Class structure

```
Namespace: Xeno.Prodika.PluginExtensions.Plugins
Assembly: PluginExtensions.dll
```

An Extensible Column plugin factory class is required to create an Extensible Column plugin. This factory class is referenced in the configuration file. The factory must implement the `IExtensibleColumnPluginFactory` interface:



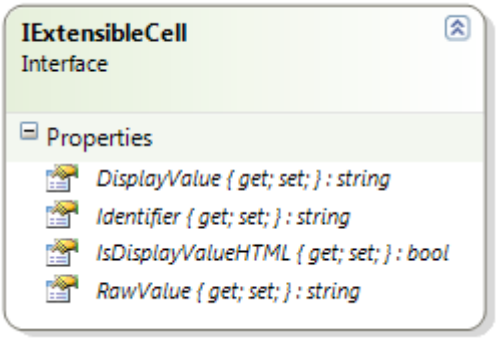
Each Extensible Column plugin must implement the `IExtensibleColumnPlugin` interface, which has the following properties and methods:



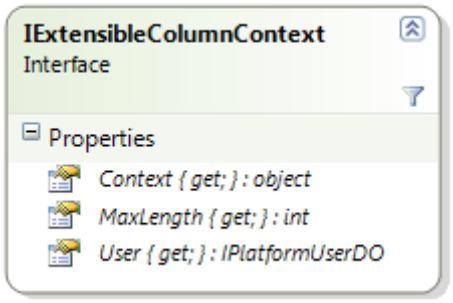
- **Header** – returns the column header data for use in the grid, using an IExtensibleCell
- **GetCellValues** – returns a list of IExtensibleCell values representing the individual cells within the column.
- **GetCellValue** – returns a specific IExtensibleCell value representing a specific cell within the column; this method is used for Printing.

These methods and properties each return an **IExtensibleCell** which contains the following Properties:

- **DisplayValue** for UI display
- **IsDisplayValueHTML** indicator which should return true if the DisplayValue contains HTML formatting
- **RawValue** the non-formatted value
- **Identifier** the unique identifier (PKID) of the row item corresponding to this cell



The ExtensibleColumnPlugin's GetCellValues and GetCellValue methods takes an IExtensibleColumnContext:



The context holds a reference to the list of ViewModels used to populate the data grids, via the **Context** property. Each grid in the user interface uses different view models.

Chapter 4—Extensible Formulation Column Plugins

The GSM Formulation Spec's Bill of Material sections use the Extensible Column Plugins described above, but have an additional configuration layer, specified in the CustomFormulationExtensions.xml file. The additional configuration allows for more fine-grained control of extensible column behavior, and adds the capability of additional custom calculation logic to be added to calculation process.

Column Visibility

Multiple columns can be defined for each of the following formulation grids and may be included in the formulation print results.

Formulation UI Grids	Formulation Printing
Formulation Tab Inputs	Main print content
Formulation Tab Outputs	Included Formulation Steps content
Process Tab Material Inputs	
Process Tab Packaging Inputs	
Process Tab Outputs	

Each column can be easily configured to be visible for individual or multiple calculation paths

- Fixed Input Percent
- Fixed Input Percent Range
- Input Percent
- Input Percent Range
- Input Quantity
- Input Quantity Range
- Input Yield
- Input Yield Range
- Any custom calculation paths

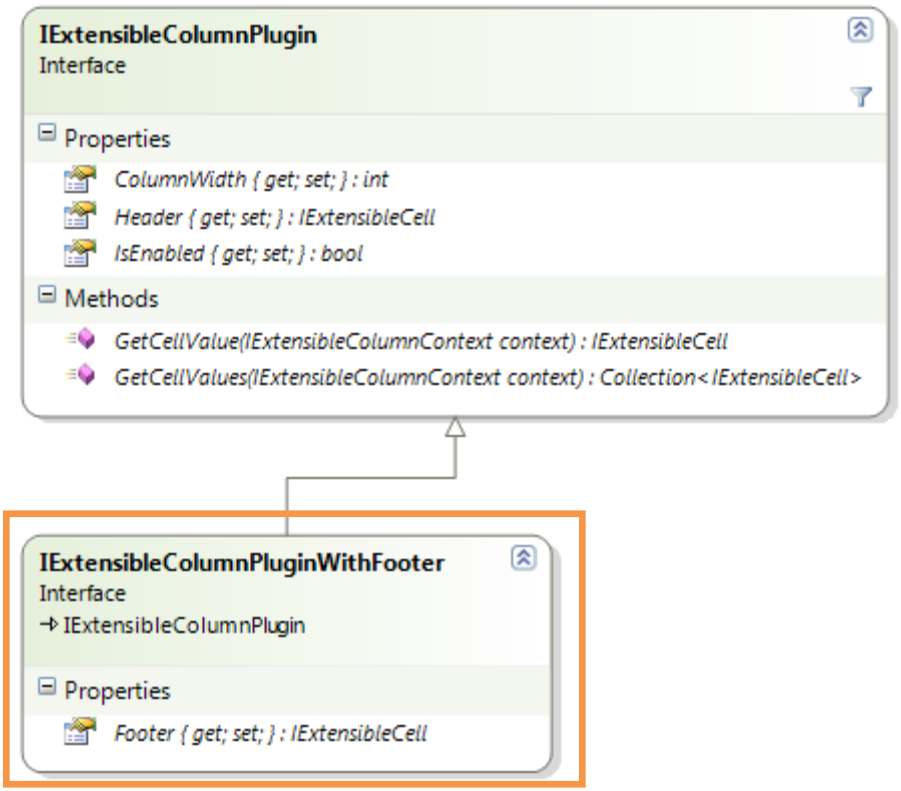
Column visibility rules can also be closely controlled through a separate Validate Plugin class, if needed. If a valid plugin name is specified in the VisibilityHandler attribute (and is configured in the CustomPluginExtensions.xml file), it will be loaded and evaluated; if it returns true, the column will be visible.

See the Extensible Formulation Column Configuration section for more details.

Column Footers

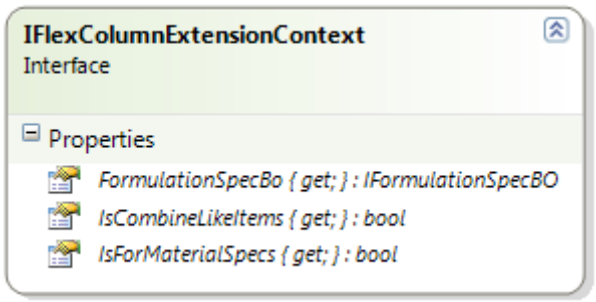
The formulation grids in the user interface contain footers, where data from the column can be rolled up. To allow for footer data for column plugins, an additional interface is provided that extends the IExtensibleColumnPlugin interface. This interface provides a Footer property, which returns an IExtensibleCell, just like the Header property.

IExtensibleColumnPlugin and IExtensibleColumnPluginWithFooter interfaces



Context

The context passed into an Extensible Formulation Column plugin's Context property is an IFlexColumnExtensionsContext, providing access to the current Formulation Spec business object, an IsCombineLikeItems indicator, and an indicator, IsForMaterialSpecs, to tell the plugin if the current GetCells method should be returning Material specs or Packaging/Printed Packaging specs.



Extensible Formulation Column Configuration

Configuration of Extensible Formulation column plugins is specified in the CustomFormulationExtensions.xml file, located in the config\Extensions directory, in the following sections:

- FormulationInputFlexColumns
- FormulationOutputFlexColumns

These sections define the columns being added to the Formulation Input and Formulation Output Items. To add a custom column, insert a FormulationColumnExtension xml element to the FormulationInputFlexColumns node for the Inputs grids and to the FormulationOutputFlexColumns node for the Outputs grids.

Each FormulationColumnExtension xml element contains the following possible attributes:

a. **ID** – the unique configuration name assigned to the column
b. **CalculationPaths** – the calculation path(s) where the new column can be added. This should be a comma separated list for multiple paths or "*" for all paths.
    i. FixedInputPercentRange
    ii. InputPercentRange
    iii. InputPercent
    iv. InputQuantityRange
    v. InputQuantity
    vi. FixedInputPercent
    vii. InputYieldRange
    viii. InputYield

> For added flexibility, clients wishing to configure different column plugins for different business lines can create duplicate Calculation Paths using different names, and then reference these Calculation Paths in the configuration and in the formulation Settings control.
>
> For example, a Baked Goods business line could show a moisture column in the inputs grid and a % sweetener extended attribute column in the outputs, while a Beverages business line would show a volume column in inputs and a Brix extended attribute column in the outputs grid.
>
> The Calculation Paths are defined in the **gsmStepCalcTypeLookupItems** table. Simply create a duplicate entry of the desired calculation path, changing the PKID and the StepCalcPathUserKey values. Then add a related entry to the **gsmStepCalcTypeLookupItemsML** table, and restart IIS. Then use the StepCalcPathUserKey for the configuration entry.

c. **UseIn** –the control where the new column can be displayed. This should be a comma separated list for multiple Use In values. Use "*" for all UseIn values.
    i. FormulationTab
    ii. FormulationTabOutputs
    iii. ProcessTab
    iv. ProcessTabOutputs
    v. ProcessTabPackaging
    vi. PrintMain – primary print content
    vii. PrintSteps – included formulation steps content

d. **HeaderTranslationId** – the column header translation used for display, or blank.

This translation will need to be added to the commonXLAExtensionCacheItem table.

To add new translations, add a new entry into the `commonXLAExtensionCacheItem` table, where the `fkParent` value is the pkid of the `'FORMULATION_COLUMN_EXTENSIONS'` entry in the `commonXLAExtensionCache` table. To add a "Volume (L)" column header, for example, add lblVolumeLiters as the HeaderTranslationId value, and use the script below:

```
insert into commonXLAExtensionCacheItem values ('1059'+UPPER(NEWID()),
'1058c29d416f-2d37-4f5c-b387-174f724d9cd8', 0, 'lblVolumeLiters', 'Volume (L)');
```

Leaving this attribute as a blank (HeaderTranslationId="") will allow the specified Extensible Column Plugin's Header property (which returns an ExtensibleCellValue) to be used as the column header. **This allows for some more flexibility, such as adding images for the column header.**

e. **ColumnPosition** – the position within the control where the column will appear. This must be a number between 4 and 18 on the inputs control and 2 and 11 on the outputs control.

Note: There are hidden columns that could cause the position to be different than expected. If using the same column plugin in multiple grids, such as the Process Tab and the Formulation Tab, the column positions may differ. For instance, the following configuration will have the Preferred UOM column to display after the Qty column

```
        <FormulationColumnExtension ID="PreferredUOMInputQty"
CalculationPaths="InputQuantity" UseIn="FormulationTab,PrintMain" HeaderTranslationId=""
ColumnPosition="4"  VisibilityHandler="ShowPreferredUOMInputQuantity"
EditabilityHandler="false" ExtensibleColumnPluginName="PreferredUOMInputQuantity" />

        <FormulationColumnExtension ID="PreferredUOMInputQty_ProcessTab"
CalculationPaths="InputQuantity" UseIn="ProcessTab" HeaderTranslationId=""
ColumnPosition="5" VisibilityHandler="ShowPreferredUOMInputQuantity"
EditabilityHandler="false" ExtensibleColumnPluginName="PreferredUOMInputQuantity" />
```

You must experiment to find the right location.

f. **VisibilityHandler** – the rules to be applied when deciding if the new column will be visible.
   i. True
   ii. False
   iii. <name of Validate Plugin to call for more customized logic>

The Custom Validate Plugin class would be the plugin name of a Validate Plugin written and configured in the CustomPluginExtensions.xml file.

An example plugin used to control visibility is the class PreferredUOMInputYieldConversionValidatePlugin, which acts to hide the related PreferredUOMInputYield conversion column if all inputs are already listed in the user's

preferred UOM. See the FormulationExtensionsSample reference project and the Inputs\PreferredUOMValues folder.

Note that controlling Visibility conditionally using Validate plugins only works if you have one column plugin at a time. If using multiple column plugins together, and wanting to hide a column, consider making the plugin return no data as an alternative.

g. **ExtensibleColumnPluginName** – the location of the business logic for the new column. This should be the name of the plugin configured in the CustomPluginExtensions.xml.

Below is an example for inputs and outputs from the CustomFormulationExtensions.xml file.

```
<FormulationInputFlexColumns configChildKey="ID">
  <FormulationColumnExtension ID="volume" IsCore="false"
    CalculationPaths="InputQuantity"
    UseIn="FormulationTab"
    HeaderTranslationId="lblVolumeLiters"
    ColumnPosition="4"
    VisibilityHandler="true"
    ExtensibleColumnPluginName="FormulationInputVolumeColumnPlugin" />
</FormulationInputFlexColumns>
<FormulationOutputFlexColumns configChildKey="ID">
  <FormulationColumnExtension ID="outputvolume" IsCore="false"
    CalculationPaths="*"
    UseIn="*"
    HeaderTranslationId="lblVolumeLiters"
    ColumnPosition="4"
    VisibilityHandler="true"
    ExtensibleColumnPluginName="FormulationOutputVolumeColumnPlugin" />
</FormulationOutputFlexColumns>
```

Calculation Methods & Added Data Storage

Clients wishing to add custom calculation methods to support the extensible formulation columns can implement the **Calculation Methods** extension. This extension allows clients to write added formulation calculations that are triggered before or after the main formulation calculations. The Calculation Methods can then leverage new data storage elements on the Formulation Inputs and Outputs to store the calculation results; the Extensible Formulation column plugins could then pull data from these storage elements and display them. This allows the data to be used for reporting.

For example, to display the Volume of formulation inputs, a Calculation Method would be created which calculated the Volume of the input based on the density, and then saves that resulting volume into the storage element for each formulation input. An extensible formulation column plugin would then retrieve the data stored and display it in the UI.

Available Formulation Input & Output Data Storage

A new object property, **InputCustomerExtendedData** which implements IFormulationInputCustomerData, is available on an IFormulationInput. This object, stored in the `FormulationInputCustomerData` table, provides the 5 Date fields, 15 double fields, 5 integer fields, and 25 string fields. Clients can store anything they wish here.

```
namespace Xeno.Data.GSM
{
    public interface IFormulationInputCustomerData : IXUniqueObject, ISaveableDataObject
    {
        DateTime Date1 { get; set; }
        DateTime Date2 { get; set; }
        DateTime Date3 { get; set; }
        DateTime Date4 { get; set; }
        DateTime Date5 { get; set; }
        double Float1 { get; set; }
        double Float2 { get; set; }
        double Float3 { get; set; }
        double Float4 { get; set; }
        double Float5 { get; set; }
        double Float6 { get; set; }
        double Float7 { get; set; }
        double Float8 { get; set; }
        double Float9 { get; set; }
        double Float10 { get; set; }
        double Float11 { get; set; }
        double Float12 { get; set; }
        double Float13 { get; set; }
        double Float14 { get; set; }
        double Float15 { get; set; }
        int Number1 { get; set; }
        int Number2 { get; set; }
        int Number3 { get; set; }
        int Number4 { get; set; }
        int Number5 { get; set; }
        string String1 { get; set; }
        string String2 { get; set; }
        string String3 { get; set; }
        string String4 { get; set; }
        string String5 { get; set; }
        string String6 { get; set; }
        string String7 { get; set; }
        string String8 { get; set; }
        string String9 { get; set; }
        string String10 { get; set; }
        string String11 { get; set; }
        string String12 { get; set; }
        string String13 { get; set; }
        string String14 { get; set; }
        string String15 { get; set; }
        string String16 { get; set; }
        string String17 { get; set; }
        string String18 { get; set; }
        string String19 { get; set; }
        string String20 { get; set; }
        string String21 { get; set; }
        string String22 { get; set; }
        string String23 { get; set; }
        string String24 { get; set; }
        string String25 { get; set; }
    }
}
```

To store the calculated volume, for example, one could store the numeric volume in the Float1 field, and the Unit of Measure PKID value in the String1 field.

A similar object **OutputCustomerExtendedData**, which implements IFormulationOutputCustomerData, is available on an IFormulationOutput. This object, stored in the `FormulationOutputCustomerData` table, has the same properties as those listed above for the InputCustomerExtendedData.

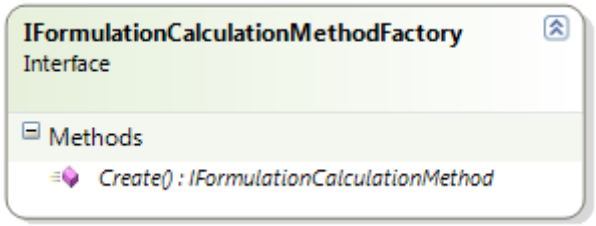
A reference implementation, **InputFloatWithOptionalUOMColumnPlugin** (along with the InputNumericWithOptionalUOMColumnPluginFactory which would be declared in the CustomPluginExtensions config), uses this concept to display numeric values stored in the InputCustomerExtendedData property. This allows the Calculation Methods to do the business logic and persistence work, while the column plugin just displays the calculated stored data.

> The InputNumericWithOptionalUOMColumnPluginFactory takes parameters in the config which drive which column/property (e.g., Float1 for value, String1 for UOM) to pull from.

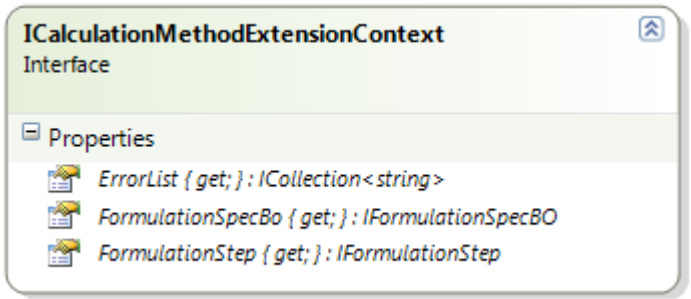
Calculation Methods Class Structure

`Namespace: Xeno.Prodika.GSMLib.Formulation.Extension`
`Assembly: GSMLib.dll`

A Calculation Method factory class is required to create a CalculationMethod. This factory class is referenced in the configuration file. The factory must implement the `IFormulationCalculationMethodFactory` interface:



Each CalculationMethod class must implement the `IFormulationCalculationMethod` interface, which has a Calculate method. The Calculate method gets passed an ICalculationMethodExtensionContext, which contains a reference to the current formulation spec, the current formulation step, and an error list:



Calculation Methods Configuration

Configuration of Calculation Methods is specified in the CustomFormulationExtensions.xml file, located in the config\Extensions directory, in the following sections:

- **PreCalculationMethods** – called before the core calculation. Use caution when using the precalculationmethods since the core values haven't been calculated.
- **PostCalculationMethods** – called after the core calculation

To add a custom Calculation Method, insert a CalculationMethodExtension XML element to the PreCalculationMethods or PostCalculationMethods node.

Each CalculationMethodExtension XML element contains the following possible attributes:

a. **ID** – the unique configuration name assigned to the calculation method
b. **Order** – the order in which the calculations will be executed.
c. **CalculationPaths** – the calculation path(s) where the new column can be added. This should be a comma separated list for multiple paths. Use "*" for all paths.
    i. FixedInputPercentRange
    ii. InputPercentRange
    iii. InputPercent
    iv. InputQuantityRange
    v. InputQuantity
    vi. FixedInputPercent
    vii. InputYieldRange
    viii. InputYield
d. **EnableHandler** – Determines if the calculation method should be called.
    i. True
    ii. False
    iii. <name of Validate Plugin to call for more customized logic>

    Note: The custom class would be the plugin name of a Validate Plugin written and configured in the CustomPluginExtensions.xml file.
e. **FactoryURL** – the object URL for the calculation method factory class.

Below is an example for post calculation methods from the CustomFormulationExtensions.xml file.

```
<PostCalculationMethods configChildKey="ID">
    <CalculationMethodExtension ID="InputAsVolume" Order="1" IsCore="false" CalculationPaths="*"
    EnableHandler="true"
    FactoryURL="Class:FormulationExtensionsSample.Calculations.InputAsVolumeCalculationMethodFactory,
    FormulationExtensionsSample" />
</PostCalculationMethods>
```

Example Implementations

Formulation Input as Volume Column Extension

The following example implementation will use a CalculationMethod to calculate a formulation input's Volume, and store that calculation result in the InputCustomerExtendedData property, which will get saved when the formulation spec is saved. A Formulation Column Extension plugin is then used to retrieve the data from that property and display it.

First, a Calculation Method Factory is needed to construct a Calculation Method. The following one simply creates a new calculation method class.

```
    public class InputAsVolumeCalculationMethodFactory : IFormulationCalculationMethodFactory
    {
        public IFormulationCalculationMethod Create()
        {
            return new InputAsVolumeCalculationMethod();
        }
    }
```

The InputAsVolumeCalculationMethod will calculate the volume (in Liters) for the given Step's inputs (for inputs that are Material specs or formulation outputs from another step) and the Step's outputs.

- The volume is calculated using the Density of the Input and the Output, and then is stored in the Float1 field.
- The Liters UOM pkid is stored in the String1 field.
- The String2 field is used as an indicator for displaying that the input or output spec has an invalid Density field.
- This will be used by the ExtensibleColumn plugin to display a warning icon if there is a value in the String2 field.
- If an error occurs converting the quantity to a Volume, a user facing message is added to the ErrorList of the calculationMethodContext, which will show up in the UI.

```
    public class InputAsVolumeCalculationMethod : IFormulationCalculationMethod
    {
        #region Implementation of IFormulationCalculationMethod

        public bool Calculate(ICalculationMethodExtensionContext calculationMethodContext)
        {

            var volumeUOM = UOMService.LitersUOM;

            foreach (IFormulationInput formulationInput in calculationMethodContext.FormulationStep.Inputs
.Where(x => x.MaterialType == EnumUniversalDataObjectType.IngredientSpecification.Value || x.MaterialType
== EnumUniversalDataObjectType.FormulationOutputDO.Value))
            {
                CalculateInput(calculationMethodContext, volumeUOM, formulationInput);
            }

            foreach (IFormulationOutput formulationOutput in calculationMethodContext.FormulationStep.Outp
uts)
            {
                CalculateOutput(calculationMethodContext, volumeUOM, formulationOutput);
            }

            return calculationMethodContext.ErrorList.Count == 0;
```

```
        }

        private static void CalculateOutput(ICalculationMethodExtensionContext calculationMethodContext, U
OM volumeUOM,
                                            IFormulationOutput formulationOutput)
        {
            var density = formulationOutput.ApproximateYield.GetMergedFinalDensity();
            if (density.IsThisValid)
            {
                try
                {
                    double volume = density.Convert(formulationOutput.ApproximateYield.ApproximateYield.Qu
antity,
                                                    formulationOutput.ApproximateYield.ApproximateYield.UO
M,
                                                    volumeUOM);

                    formulationOutput.OutputCustomerData.Float1 = volume;
                    formulationOutput.OutputCustomerData.String1 = volumeUOM.PKID;
                    formulationOutput.OutputCustomerData.String2 = null;
                }
                catch (Exception)
                {
                    calculationMethodContext.ErrorList.Add(
                        "Error in InputAsVolumeCalculationMethod -  calculating volume for output " +
                        formulationOutput.Material.SpecSummary.SpecNumber);
                }
            }
            else
            {
                formulationOutput.OutputCustomerData.String2 = "Invalid density for Output " +
                                                               formulationOutput.Material.SpecSummary.Spec
Number;
            }
        }

        private static void CalculateInput(ICalculationMethodExtensionContext calculationMethodContext, UO
M volumeUOM,
                                           IFormulationInput formulationInput)
        {
            if (formulationInput.IsMaterialReferenceAvailable()
                && formulationInput.MaterialType != EnumUniversalDataObjectType.PackagingSpecification.Val
ue
                && formulationInput.MaterialType != EnumUniversalDataObjectType.FinishedPackagingSpecifica
tion.Value)
            {
                var density = formulationInput.MergedAttributes.Density;
                if (density.IsThisValid)
                {
                    try
                    {
                        double volume = density.Convert(formulationInput.InputQuantity.Quantity,
                                                        formulationInput.InputQuantity.UOM,
                                                        volumeUOM);

                        formulationInput.InputCustomerData.Float1 = volume;
                        formulationInput.InputCustomerData.String1 = volumeUOM.PKID;
                        formulationInput.InputCustomerData.String2 = null;
                    }
                    catch (Exception)
                    {
                        calculationMethodContext.ErrorList.Add(
                            "Error in InputAsVolumeCalculationMethod -  calculating volume for Input " +
                            formulationInput.Material.SpecSummary.SpecNumber);
                    }
                }
                else
                {
                    formulationInput.InputCustomerData.String2 = "Invalid density for input " +
```

```
                                                                formulationInput.Material.SpecSummary.Spe
cNumber;
                }
            }
        }

        #endregion

        private IUOMService UOMService
        {
            get { return AppPlatformHelper.ServiceManager.GetServiceByType<IUOMService>(); }
        }
    }
```

To include this CalculationMethod (for all calculation paths), the following configuration is used:

```
<PostCalculationMethods configChildKey="ID">
   <CalculationMethodExtension ID="InputAsVolume"
       Order="1" IsCore="false"
       CalculationPaths="*"
       EnableHandler="true"
       FactoryURL="Class:FormulationExtensionsSample.Calculations.InputAsVolumeCalculationMeth
odFactory,FormulationExtensionsSample" />
</PostCalculationMethods>
```

An Extensible Column plugin can then retrieve the values stored in the InputCustomerData object and display the values. To display the sum of the cells in the footer, a running total must be calculated in the plugin when iterating over the cells and assigned to the Footer cell.

When using the Combine Like Items feature, implementing the column plugins for formulation inputs can get complex, as the items to combine must be aggregated into one cell.

Related Examples

The reference implementation class **InputFloatWithOptionalUOMColumnPlugin** (and especially its base class InputFloatWithOptionalUOMColumnPluginBase) demonstrate the required logic to handle the CombineLikeItems logic, retrieving data from the InputCustomerData (although in a reusable way that makes the code slightly more complex), using a warning icon in the UI, and more.

Other reference implementations demonstrate how a plugin can be written without using the Data Storage that a calculation method would use. The OutputVolumeNoStorageExtensibleColumnPlugin and the OutputVolumeExtensibleColumnPlugin show this difference.

Preferred UOM Quantity/Yield Column Extensions

Reference implementation column plugin classes show the user formulation BOM items in the user's preferred UOM for the Input Quantity and Input Yield, as well as for the Output Quantity and Output Yield.

An abstract base class, PreferredUOMInputConversionColumnPluginBase, implements most of the business logic, delegating the retrieval of the actual input's value to convert(input quantity or input yield) to the subclasses via the abstract method GetQuantityForInput

```
protected abstract IReadOnlyMeasurement GetQuantityForInput(IFormulationInput input);
```

The base class PreferredUOMInputConversionColumnPluginBase retrieves the user's preferred Unit of Measure so that the formulation input value (Qty or Yld) will be displayed in that UOM. It calls a helper class, FormulationColumnHelper, which retrieves the preference from the current user. This same helper class returns the desired inputs to parse through (ignoring output based inputs) in the GetCellValues method.

Each input is then converted to the target UOM using the UOMService – the result is then added to the collection of IExtensibleCell items that are returned by the GetCellValues method. Two versions of each result are added – one is a formatted value within an HTML tab, which is used to display in the UI, and then a "raw" value that is used for printing. These values are added as a SimpleExtensibleCell, with a true parameter passed to the constructor's isDisplayValueHTML argument, indicating that the display value result has HTML formatting. Each converted quantity is added to a running total, which is then used for the Footer cell values.

```
    public abstract class PreferredUOMInputConversionColumnPluginBase : IExtensibleColumnPluginWithFooter
    {
        #region Implementation of IExtensibleColumnPlugin

        protected UOM TargetUOM { get; set; }

        protected PreferredUOMInputConversionColumnPluginBase()
        {
            // get the user's preferred UOM to convert to
            TargetUOM = FormulationColumnHelper.GetUserPreferredUOM();

        }

        /// <summary>
        /// Used to retrieve the desired quantity for converstion
        /// </summary>
        /// <param name="input">The formulation input business object IFormulationInput containing the qua
ntity to return</param>
        /// <returns>the desired quantity to convert as an IReadOnlyMeasurement</returns>
        protected abstract IReadOnlyMeasurement GetQuantityForInput(IFormulationInput input);

        public Collection<IExtensibleCell> GetCellValues(IExtensibleColumnContext context)
        {
            var cells = new Collection<IExtensibleCell>();
            var columnContext = (IFlexColumnExtensionContext)context.Context;

            double total = 0.0;
            // get the relevant inputs to convert the quantities for - this excludes Output-
based inputs (eg., step 1 output used as an input in step 2)
            var inputList = FormulationColumnHelper.GetFormulationInputsByType(columnContext);

            foreach (IFormulationInput formulationInput in inputList)
            {
                try
                {
                    // child class returns the relevant quantity to convert
                    var inputQty = GetQuantityForInput(formulationInput);
                    if (inputQty != null && inputQty.IsValid)
                    {

                        double newValue = AppHelper.UOMService.Convert(inputQty.UOM, TargetUOM, inputQty.Q
uantity);
```

```
                        string rawValue = String.Format("{0:0.00000} {1}", newValue, TargetUOM.Abbreviatio
n);
                        string displayValueHTML = String.Format("<span style='white-
space: nowrap;'>{0}</span>", rawValue);
                        // use the input pkid as the identifier for the cell, so that it can be linked to
the actual input row
                        cells.Add(new SimpleExtensibleCell(formulationInput.PKID, displayValueHTML, rawVal
ue, true));
                        // add value to the total variable, which will be used at the end for the Footer c
ell
                        total += newValue;
                    }
                }
                catch (Exception)
                {
                    cells.Add(new SimpleExtensibleCell(formulationInput.PKID, "error", "error", false));
                }


            }
            // set the footer cell as the sum of the input cells from above.
            string totalRawValue = String.Format("{0:0.00000} {1}", total, TargetUOM.Abbreviation);
            string totalRawValueHTML = String.Format("<span style='white-
space: nowrap;'>{0}</span>", totalRawValue);
            Footer = new SimpleExtensibleCell("SimpleMeasurementFooterCell", totalRawValueHTML, totalRawVa
lue, true);


            return cells;
        }
```

The subclasses, therefore, are quite simple:

- They create the Header cell as “Yld (<target UOM>)” or “Qty (<target UOM>)”, though typically this would be implemented using translations.
- The GetQuantityForInput abstract method returns the formulation Input’s InputYield value or InputQuantity value.

```
public class PreferredUOMInputYieldConversionColumnPlugin : PreferredUOMInputConversionColumnPluginBase
    {
        public PreferredUOMInputYieldConversionColumnPlugin(): base()
        {
            // sets the header cell to display the user's preferred UOM. Here we have hardcoded "Yld" but
this can be made a translation instead
            var headertitle = String.Format("Yld ({0})", TargetUOM.Abbreviation);
            Header = new SimpleExtensibleCell("SimpleMeasurementHeaderCell", headertitle, headertitle, fal
se);
        }


        protected override IReadOnlyMeasurement GetQuantityForInput(IFormulationInput input)
        {
            return input.InputYield;
        }
    }
```

```
    public class PreferredUOMInputQuantityConversionColumnPlugin : PreferredUOMInputConversionColumnPlugin
Base
    {
        public PreferredUOMInputQuantityConversionColumnPlugin() : base()
```

```
        {
            // sets the header cell to display the user's preferred UOM. Here we have hardcoded "Qty" but
this can be made a translation instead
            var headertitle = String.Format("Qty ({0})", TargetUOM.Abbreviation);
            Header = new SimpleExtensibleCell("SimpleMeasurementHeaderCell", headertitle, headertitle, fal
se);
        }

        protected override IReadOnlyMeasurement GetQuantityForInput(IFormulationInput input)
        {
            return input.InputQuantity;
        }
    }
```

See the SetupForPreferredUOMColumn.txt in the source code for configuration instructions.

A related set of plugins are included for Outputs. Note that the Outputs grid in the Formulation Step view (Process Tab) *does display* the internal Outputs quantity/yield values, whereas the Formulation View does not. Therefore the configuration for outputs is more complex, as the column plugin class must know which location is being displayed, to determine whether or not to show these values. See the configuration examples PreferredUOMOutputQuantity and PreferredUOMOutputQuantity_ProcessTab, which call the same plaguing, but pass a different argument value as an indicator.

```
 <Plugin name="PreferredUOMOutputQuantity" FactoryURL="Class:FormulationExtensionsSample.Outpu
ts.PreferredUOMValues.PreferredUOMOutputQuantityConversionColumnPluginFactory,FormulationExten
sionsSample" />

 <Plugin name="PreferredUOMOutputQuantity_ProcessTab" FactoryURL="Class:FormulationExtensionsS
ample.Outputs.PreferredUOMValues.PreferredUOMOutputQuantityConversionColumnPluginFactory,Formu
lationExtensionsSample$true" />
```

When creating custom column plugins, be sure to deploy the custom dll(s) into web\gsm\bin and RemotingContainer\bin

Chapter 5—Performance Implications

Be aware that adding complex and inefficient column plugins, or a large number of column plugins, may impact performance of the application. Use the Performance Instrumentation utility to get a performance profile of your column extensions.

Appendix A—Custom Formulation Extensions

Sample CustomFormulationExtensions.xml

```
<CustomFormulationExtensions>
        <FormulationInputFlexColumns configChildKey="ID">
                <FormulationColumnExtension ID="volume" IsCore="false" CalculationPaths="InputQuantity"
                UseIn="*" HeaderTranslationId="lblVolumeLiters" ColumnPosition="4" VisibilityHandler="true"
                ExtensibleColumnPluginName="FormulationInputVolumeColumnPlugin" />

        </FormulationInputFlexColumns>
        <FormulationOutputFlexColumns configChildKey="ID">
                <FormulationColumnExtension ID="outputvolume" IsCore="false"
                CalculationPaths="InputQuantity" UseIn="FormulationTab,PrintMain"
                HeaderTranslationId="lblVolumeLiters" ColumnPosition="4"
                VisibilityHandler="DefaultTruePlugin"
                ExtensibleColumnPluginName="FormulationOutputVolumeColumnPlugin" />

                <FormulationColumnExtension ID="outputEA_TestNumeric" IsCore="false" CalculationPaths="*"
                UseIn="*" HeaderTranslationId="lblTestNumeric" ColumnPosition="5" VisibilityHandler="true"
                ExtensibleColumnPluginName="FormulationOutputVolumeColumnPlugin_TestNumeric" />

                <FormulationColumnExtension ID="FormulationOutputTotalFat" IsCore="false"
                CalculationPaths="*" UseIn="*" HeaderTranslationId="lblTestFat" ColumnPosition="6"
                VisibilityHandler="true"
                ExtensibleColumnPluginName="FormulationOutputTotalFat" />
        </FormulationOutputFlexColumns>
        <PreCalculationMethods configChildKey="ID">
        </PreCalculationMethods>
        <PostCalculationMethods configChildKey="ID">
                <CalculationMethodExtension ID="InputAsVolume" Order="1" IsCore="false"
                CalculationPaths="*" EnableHandler="true"
                FactoryURL="Class:FormulationExtensionsSample.Calculations.InputAsVolumeCalculationMetho
                dFactory,FormulationExtensionsSample" />
        </PostCalculationMethods>
</CustomFormulationExtensions>
```

Appendix B—Custom Plugin Extensions

Sample CustomPluginExtensions.xml

```
<ExtensibleColumnPlugins configChildKey="name">
        <Plugin name="FormulationInputVolumeColumnPlugin"
        FactoryURL="Class:FormulationExtensionsSample.Inputs.InputNumericWithOptionalUOMColumnPluginFa
        ctory,FormulationExtensionsSample$NameValuePair:ValueFloat_FieldName=Float1&amp;UomPKIDString
        _FieldName=String1&amp;BaseUOMIsoCode=LT&amp;ErrorIndicatorString_FieldName=String2"/>

        <Plugin name="FormulationOutputVolumeColumnPlugin2"
        FactoryURL="Class:FormulationExtensionsSample.Outputs.OutputVolumeExtensibleColumnPluginFactory,
        FormulationExtensionsSample"/>

        <Plugin name="FormulationOutputVolumeColumnPlugin"
        FactoryURL="Class:FormulationExtensionsSample.Outputs.OutputNumericWithOptionalUOMColumnPlugi
        nFactory,FormulationExtensionsSample$NameValuePair:ValueFloat_FieldName=Float1&amp;UomPKIDStr
        ing_FieldName=String1&amp;BaseUOMIsoCode=LT&amp;ErrorIndicatorString_FieldName=String2"
        FilterResolverFactoryUrl="Class:Xeno.Prodika.PluginExtensions.Plugins.DefaultPlugins.DefaultEmptyXColFi
        lterResolver, PluginExtensions">
        </Plugin>
        <Plugin name="FormulationOutputVolumeColumnPlugin_TestNumeric"
        FactoryURL="Class:FormulationExtensionsSample.Outputs.OutputNumericWithOptionalUOMColumnPlugi
        nFactory,FormulationExtensionsSample$NameValuePair:ExtendedAttributeID=Test_Numeric1&amp;Base
        UOMIsoCode=KG&amp;ErrorIndicatorString_FieldName=String4"
        FilterResolverFactoryUrl="Class:Xeno.Prodika.PluginExtensions.Plugins.DefaultPlugins.DefaultEmptyXColFi
        lterResolver, PluginExtensions">
        </Plugin>
        <Plugin name="FormulationOutputTotalFat"
        FactoryURL="Class:FormulationExtensionsSample.Outputs.OutputNumericWithOptionalUOMColumnPlugi
        nFactory,FormulationExtensionsSample$NameValuePair:NutrientInFoodsID=FAT&amp;BaseUOMIsoCode
        =GR&amp;ErrorIndicatorString_FieldName=String4"
        FilterResolverFactoryUrl="Class:Xeno.Prodika.PluginExtensions.Plugins.DefaultPlugins.DefaultEmptyXColFi
        lterResolver, PluginExtensions">
        </Plugin>
</ExtensibleColumnPlugins>
```