Field Suite Instructional Guide | 10-11-2012

Prepared by: Tim Braga

VERSION 1.0

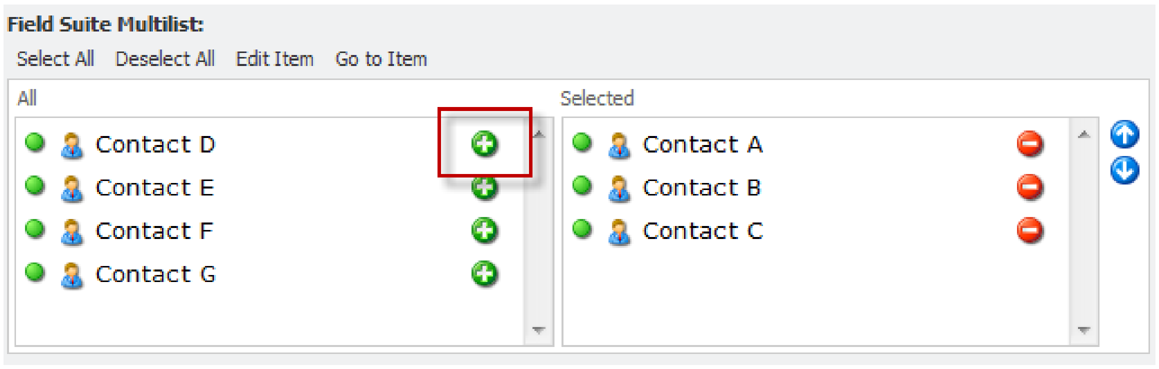
Contents

Contents 2
Basic Usage 3
  Adding and Removing Selected Items 3
  Tooltips 3
  Edit Item Modal 4
  Go to Item 4
  Field Gutter 5
Understanding the Configuration File 6
  ControlSources 6
  Auto-Publishing 7
  Field Gutter 7
  Images Field 8
  Field Placeholder 8
Customizations 9
  Creating a Field Gutter Item 9
  Creating a Field Placeholder 10

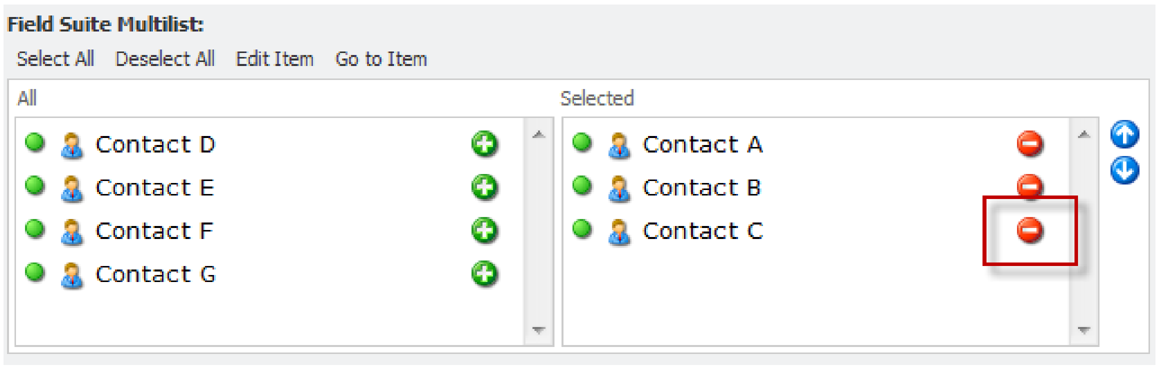
Basic Usage

Adding and Removing Selected Items

To add an item from the selected side of the field (Treelist, Multilist), click the green addition button to push that item over to the right side.

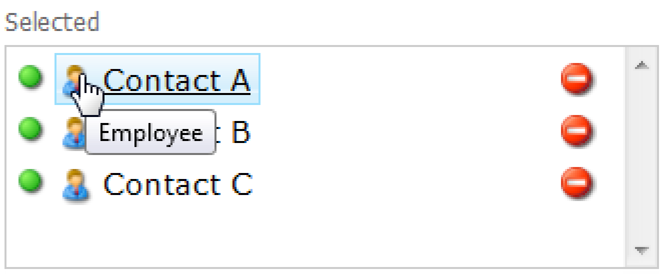


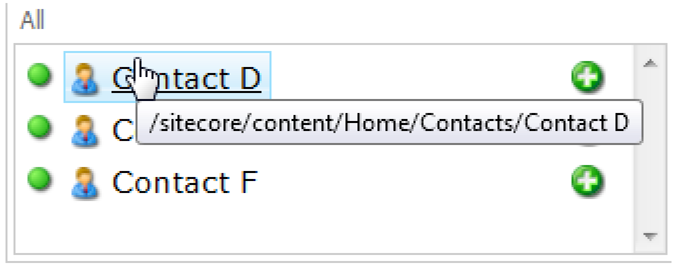
Similarly, to remove an item from the selected side, click on the red circle as shown below. This will move the item over to the left side.



Tooltips

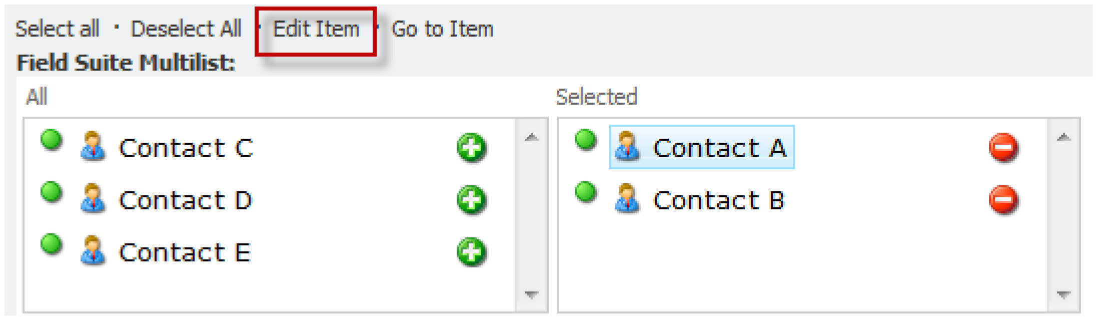
Often we need to know where a referenced item lives within the content tree and from which template it is derived. The Field Suite allows you to mouse over (hover over) the name of a referenced item to retrieve its item path or to mouse over the template icon to see its template name.





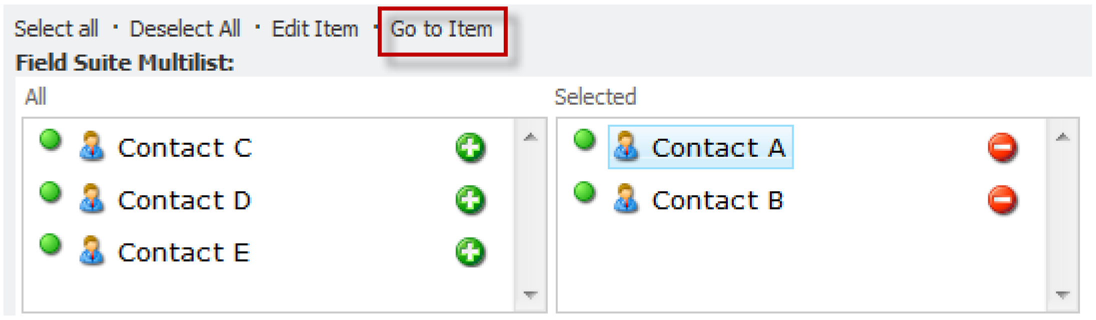
Edit Item Modal

We typically make changes to items that are being referenced in one of the current items fields. The Edit Form attempts to limit the number of times you need to leave the current item you are authoring, creating a streamlined approach. To launch the Edit Form we want to single click a referenced item (thus putting it in a highlighted state) and click on the Edit Item button above the field. With the Edit Form modal, we are able to make field changes to the referenced item, save and close the modal and continue authoring the current item. This removes the challenge of finding the item in the content tree by leaving the current item you are authoring and opening up the referenced item. As you can see below, we have the Contact A item selected and by clicking the Edit Item button, the modal is launched.



Go to Item

The Edit Form allows for editing of the field values -- but what if we needed to change more, possibly an action in the ribbon or right click the item in the content tree? For this we need to navigate directly to the item. We know by using our mouse overs that we can find the item path, but we now have a better approach. Single click the item you wish to navigate to and click the Go to Item button. This sends a request to Sitecore to change the current item to the item you have selected.



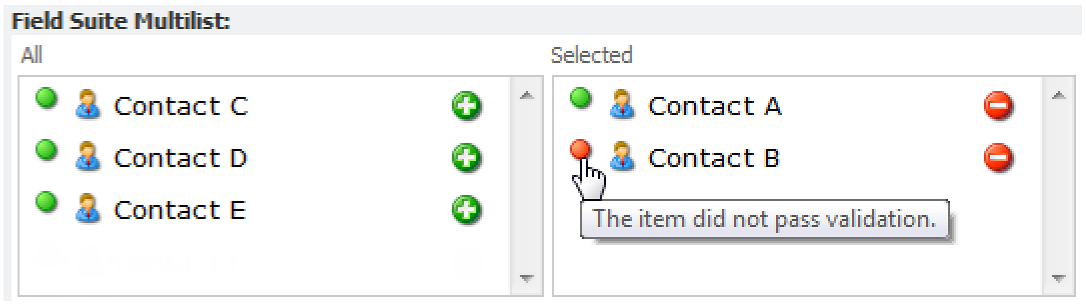
Field Gutter

You can think of the Field Gutter as a piece of real estate for each item in a field that allows for additional information to be given to the content authors represented by HTML. I will speak later in this document on how to customize the field gutter. This section is to explain its purpose and how the Item Comparer's implementation is used and should be understood.

The Published Item Comparer is a shared source module that compares an item in the authoring environment to the same item in the production (Web) environment. It does this by running through a list of validation points as defined in the ItemComparer.config file located under App_Config/Include directory.

The Field Suite implements the Item Comparer functionality within the Field Gutter, but there is a difference. The Published Item Comparer module runs through all the validation points in the configuration file while the Field Gutter's implementation only runs the ItemValidator. The reason for only running the ItemValidator is there is a concern about running all the validators against every item in a field. It is not intensive for the actual Published Item Comparer module to run through all of the validations for one item, but when you start expanding the amount of items exponentially it can degrade performance. For the Field Suite, the ItemValidator checks the following: If the item exists; If the paths are the same; and if the field values match.

With the screenshot on the next page, we can quickly and visually see that Contact A has passed validation and matches that of the item in the production environment. This indicates to us that the item is published. Contact B is showing a red circle, letting us know that there is a discrepancy between the two databases. When we click the red circle it opens the Item Comparer application letting us know why the item has failed validation.

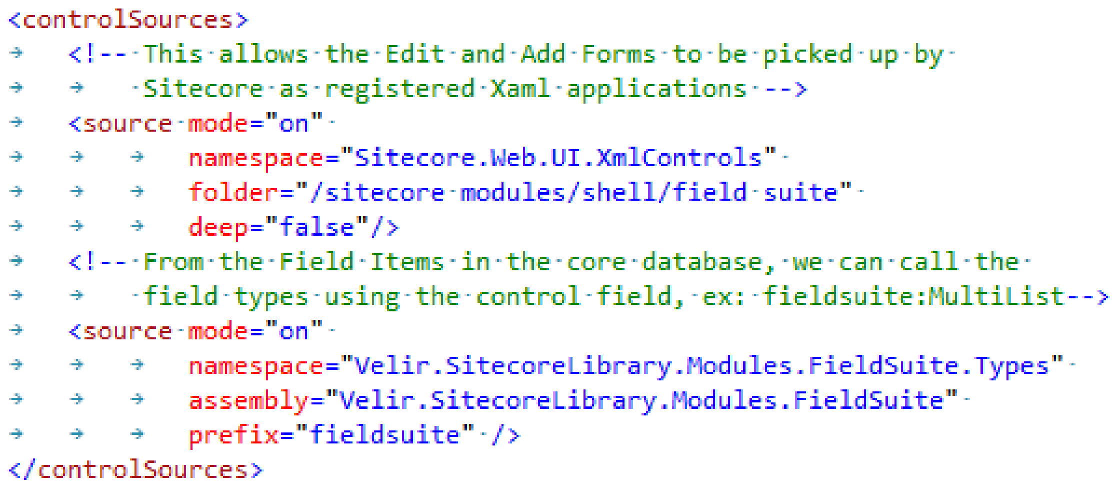


Understanding the Configuration File

ControlSources

ControlSources allow you to define controls within Sitecore. This could be a XAML application or a field type. The advantage of using control sources is two pronged. Usually when we create XAML files we add them to the layouts directory. This is because the layouts directory by default is a Sitecore defined control source for XAML files, meaning that when Sitecore starts up, it examines this directory looking for XAML files. Within our site configuration file or our module configuration file we can list our module directory (/sitecore modues/shell/field suite/) as a control source. This is shown in the upper half of the screenshot below.

The second advantage is how ControlSources allows us to define a group of field types in the same namespace. To associate a custom field to a class file, you usually create a field type in the core database and define the field type by setting the Assembly field as "Velir.SitecoreLibrary.Modules.FieldSuite" and the Class field as "Velir.SitecoreLibrary.Modules.FieldSuite.Types.Multilist." With control sources you are able to define a list of field types within the same namespace. For example, with ControlSources we can define the same field by leaving the Assembly and Class fields blank and setting the Control field as "fieldsuite:Multilist." The fieldsuite is a prefix we defined in the configuration as shown below that maps back to the namespace defined. The second half, "Multilist," is the name of the class within that namespace. We can do this by adding to the control sources as shown in the bottom half of the screenshot on the next page.



```
<controlSources>
    <!-- This allows the Edit and Add Forms to be picked up by
        Sitecore as registered Xaml applications -->
    <source mode="on"
            namespace="Sitecore.Web.UI.XmlControls"
            folder="/sitecore modules/shell/field suite"
            deep="false"/>
    <!-- From the Field Items in the core database, we can call the
        field types using the control field, ex: fieldsuite:MultiList-->
    <source mode="on"
            namespace="Velir.SitecoreLibrary.Modules.FieldSuite.Types"
            assembly="Velir.SitecoreLibrary.Modules.FieldSuite"
            prefix="fieldsuite" />
</controlSources>
```

Auto-Publishing

The Auto-Publishing configuration removes Sitecore's default functionality for adding items to the publishing queue and replaces it by patching with the Velir.SitecoreLibrary equivalent.

There is also a setting called AutoPublishFieldValues. This setting controls whether we add referenced items to the publishing queue or use Sitecore's standard out of the box functionality. We are able to toggle this functionality on/off by using the values "1" or "0."

We may need the target this functionality to specific templates due to business constraints or concerns about process. By using the AutoPublishedFieldValues Templates key, we can pipe delimiter a list of templates to apply this functionality. An empty string here will apply to all templates.

Field Gutter

The Field Gutter configuration consists of a single processor and one, many or no gutter items.

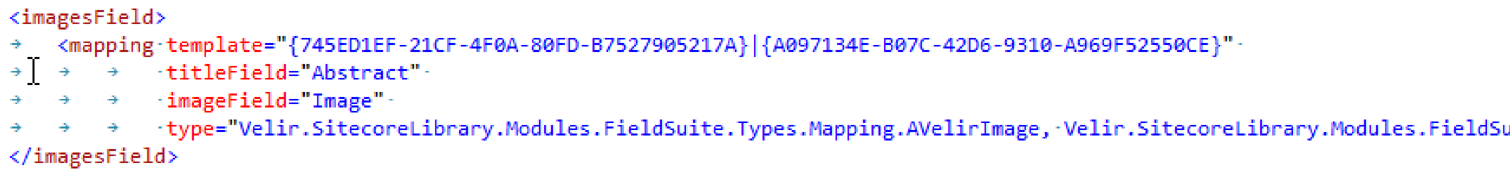
Processor: Runs the core functionality for translating and making requests to the gutter items. This core functionality for the field gutter is decoupled, following with Sitecore's practice to allow implementation of a custom processor.

Gutter Item: A gutter item is responsible for taking arguments and returning HTML to the processor. For example, the ItemComparerGutterItem performs actions against the passed arguments and returns HTML indicating whether or not the item has passed validation. There can be multiple gutter items listed, and they can be sorted, commented and customized. To implement a custom gutter item, see the section later in the document about Customizations.

Images Field

The Images Field is unique in that it needs to be configured before it can be used against any content items. This is because while rendering each item, it needs to know which field to target for that item’s image. An employee template may have the image field as “Contact Image” and an article template may use the field name “Image”. For each template to work with the Images field, we need to define it in the configuration file as shown below in the screenshot.

*The titleField is an optional attribute that will default to the Display Name if left empty.
*The Template field can be a pipe delimitered value to hold multiple templates ids.



```
<imagesField>
	<mapping template="{745ED1EF-21CF-4F0A-80FD-B7527905217A}|{A097134E-B07C-42D6-9310-A969F52550CE}" 
			 titleField="Abstract" 
			 imageField="Image" 
			 type="Velir.SitecoreLibrary.Modules.FieldSuite.Types.Mapping.AVelirImage, Velir.SitecoreLibrary.Modules.FieldSu
</imagesField>
```

Field Placeholder

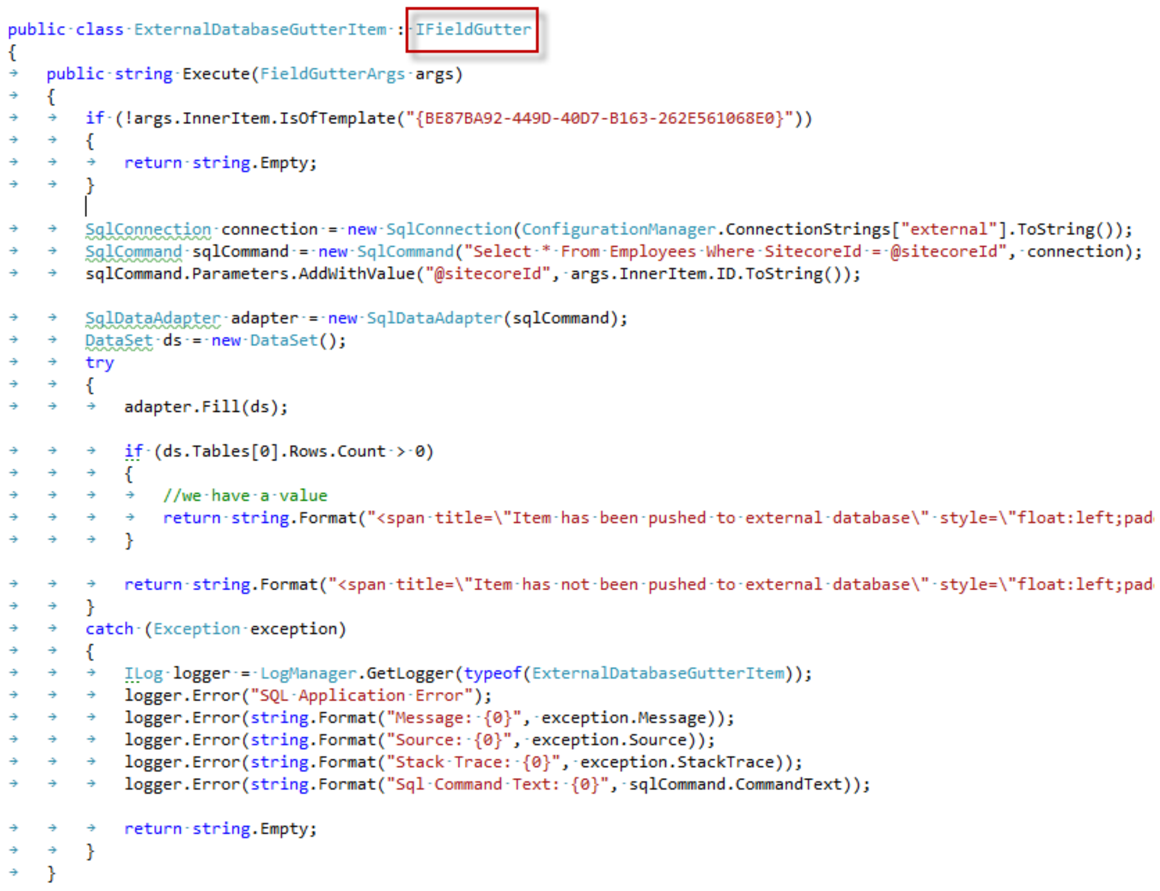
The purpose of the Field Placeholder is to tokenize the click events of the Field Buttons, allowing us to swap out a token with other text. For example, with the FieldID placeholder we are looking for $Target in the Click field of a Field Button item and replacing that value with the appropriate Field ID during rendering.

Customizations

Creating a Field Gutter Item

The Field Gutter was built on a decoupled platform. You are able to swap out the processor, add and remove gutter items and adjust attributes such as Max Count. To create a new gutter item we will need to create a class that implements the IFieldGutter interface. Let's take a moment and walk through the steps.

1. In your projects class library, create a class that implements the IFieldGutter interface.
2. Create logic within the Execute method
   - The Execute method is passed arguments from the processor. In the example below, we only want to have this gutter item run against items of a certain template.
   - Run a simple query to a database and depending on the value of the query, return HTML showing the status of the item.

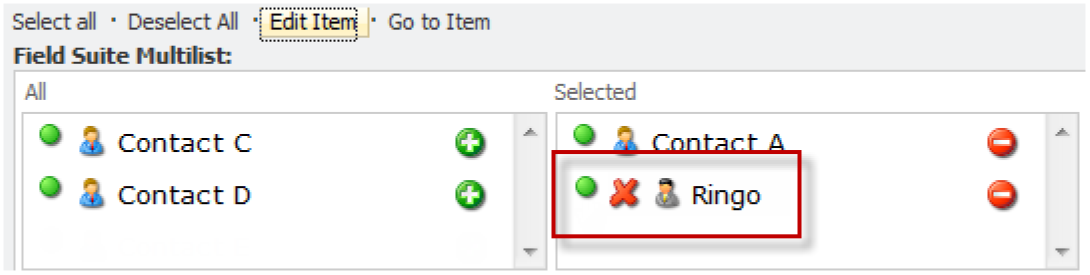


```
public class ExternalDatabaseGutterItem : IFieldGutter
{
    public string Execute(FieldGutterArgs args)
    {
        if (!args.InnerItem.IsOfTemplate("{BE87BA92-449D-40D7-B163-262E561068E0}"))
        {
            return string.Empty;
        }

        SqlConnection connection = new SqlConnection(ConfigurationManager.ConnectionStrings["external"].ToString());
        SqlCommand sqlCommand = new SqlCommand("Select * From Employees Where SitecoreId = @sitecoreId", connection);
        sqlCommand.Parameters.AddWithValue("@sitecoreId", args.InnerItem.ID.ToString());

        SqlDataAdapter adapter = new SqlDataAdapter(sqlCommand);
        DataSet ds = new DataSet();
        try
        {
            adapter.Fill(ds);

            if (ds.Tables[0].Rows.Count > 0)
            {
                //we have a value
                return string.Format("<span title=\"Item has been pushed to external database\" style=\"float:left;pad
            }

            return string.Format("<span title=\"Item has not been pushed to external database\" style=\"float:left;pad
        }
        catch (Exception exception)
        {
            ILog logger = LogManager.GetLogger(typeof(ExternalDatabaseGutterItem));
            logger.Error("SQL Application Error");
            logger.Error(string.Format("Message: {0}", exception.Message));
            logger.Error(string.Format("Source: {0}", exception.Source));
            logger.Error(string.Format("Stack Trace: {0}", exception.StackTrace));
            logger.Error(string.Format("Sql Command Text: {0}", sqlCommand.CommandText));

            return string.Empty;
        }
    }
```

3. Build the project/solution.
4. Add the appropriate gutter item xml in the configuration file. You are able to sort the gutter items by placing them before or after each other.

```
<gutterItem
type="Sandbox65.Library.FieldSuite.FieldGutter.ExternalDatabaseGutterItem,
Sandbox65.Library" />
```

5. Output example:
   - The green circle is from the Item Comparer gutter item
   - The red X is from our example External Database gutter item



Creating a Field Placeholder

Under some field types in the core database, there is a list of menu buttons. Each menu button has a Click field that sets the OnClick event for that button when rendered in the content editor. This could be used in editing the field, such as Select All and Deselect All. Sitecore only supports one token out of the box, $Target, which is hardcoded in the class file for rendering the content editor. You might have seen this: contentmultilist:selectall(id=$Target). When the content editor is rendering a field menu button, it will take this string and replace $Target with the ID of the field. For the Field Suite we needed a way to pass additional tokens such as source ($Source). To do this I needed to modify the class that renders the content editor. From there I built the functionality to be decoupled and configurable.

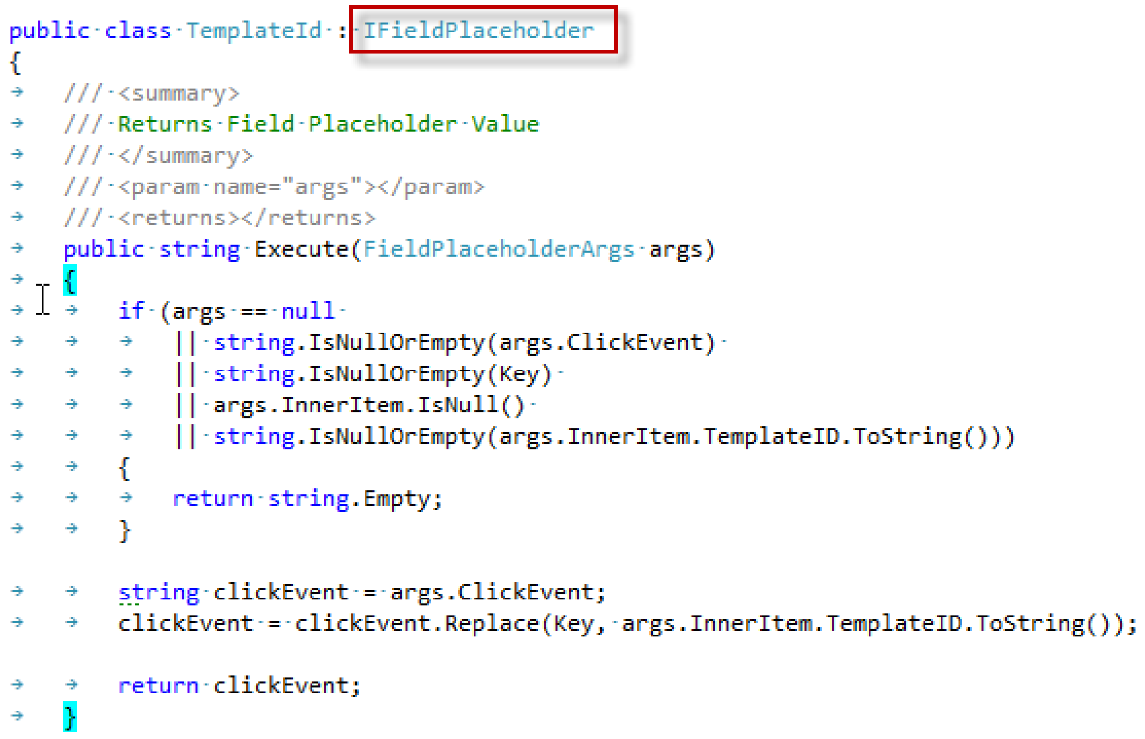
Let's take a moment and walk through the steps of creating a new field placeholder.

1. In your projects class library, create a class that implements the IFieldPlaceholder interface.
2. Create logic within the Execute method
   - In the example below we want to have this field placeholder swap out the token $TemplateID for the template id of the current item we are on in the tree.

3. Add the xml for the new field placeholder to the configuration file.

```
<placeholderItem key="$TemplateID"
type="Sandbox65.Library.FieldSuite.FieldPlaceholder.TemplateId,
Sandbox65.Library" />
```

4. We have a menu item’s click field set as: Velir.Fields.EditItem('$Target');
   - When this is rendered it will be outputted as Velir.Fields.EditItem('FIELD1210278909');



```
public class TemplateId : IFieldPlaceholder
{
	/// <summary>
	/// Returns Field Placeholder Value
	/// </summary>
	/// <param name="args"></param>
	/// <returns></returns>
	public string Execute(FieldPlaceholderArgs args)
	{
		if (args == null 
			|| string.IsNullOrEmpty(args.ClickEvent) 
			|| string.IsNullOrEmpty(Key) 
			|| args.InnerItem.IsNull() 
			|| string.IsNullOrEmpty(args.InnerItem.TemplateID.ToString()))
		{
			return string.Empty;
		}

		string clickEvent = args.ClickEvent;
		clickEvent = clickEvent.Replace(Key, args.InnerItem.TemplateID.ToString());

		return clickEvent;
	}
```