Arke.Sitecore.MetaTags Module

Arke Systems, www.arkesystems.com

Version History

Version	Notes	By
1.0	Initial release	Andy Uzick
1.1	Added support for property tags. Added “custom tags” and removed “dynamic tags”. Added pipeline processors to support all tag types. Improved debug logging. Expanded the documentation.	

Background

Recently, a client was concerned about the S.E.O. ramifications of having identical pages at more than one URL on their Sitecore site (for example, with and without "/en" in the path). Having pages like this (or worse, internal links formed differently) can decrease search ranking. If the same page is at two URLs, and there are links to both, the "link juice" is diluted across those pages.

The best solution for this is to emit a “canonical” meta tag with a common URL. Unfortunately, the client did not have a common base layout, so the only way to do this site-wide was via a pipeline processor.

This kind of thing has come up often enough that I was motivated to create a MetaTag management module. Different chunks of code need to create meta tags, and it’s not always convenient or even possible to limit the timing of meta tag generation to the layout.

At some point, almost every solution needs a way to manage and emit meta tags. Sometimes we need code to manage meta tags, sometimes content managers, sometimes it would be best done with configuration. As often as not, diverse parts of the solution use different techniques for injecting meta tag into the page, which leads to redundant code and/or hardcoded tags lost in a sea of markup.

Overview

Conceptually, there are two parts to managing tags in this module. Tags must be *defined*, and *added* to pages. In most cases, the act of defining the tag (either in the Content Editor or in config) also adds it to pages. However, it is also possible to define optional tags, which content authors can selectively add to their pages, and to create pipeline processors, which emit zero or more tags based on any custom logic.

Most tags can be defined entirely in configuration, without coding. This includes both *static tags*, an also *method tags* and *property tags*, which derive their values at runtime by leveraging existing methods and properties.

Static, Custom, Method and Property tags can be defined and used both in the Content Editor and in the pipeline config.

Coders can define custom tags. These tags also derive their values at runtime, but require more logic than simply leveraging an existing method or property. These custom tags can be leveraged globally in either the Content Editor or in config, or selectively in specific pages.

Coders can also create pipeline processors, which can generate tags at runtime. These processors can selectively add zero or more tags depending on context, using your custom logic.

Tags can be added four ways:

1. By adding processors to the InjectMetaTags pipeline.
2. In the content editor, by adding tag definitions to a GlobalTags folder (these tags appear site-wide).
3. In the content editor, by selecting pre-defined tags in a field that can be added to any template (these tags appear on individual pages)
4. In code, at any point in the page lifecycle. This allows code in layouts, sublayouts and renderings to add meta tags "ad hoc".

On the web, there are two known subspecies of meta tag markup, those that use a "property" attribute and those that use "name". Both are still effectively name/value pairs. The "name" style tag seems to be more common, but notable sets of commonly-used tags (like OpenGraph tags) use the "property" structure. This module allows both styles.

Architecture

The module uses a collection of tag description objects (not markup) that persist through the page lifecycle. Tags are added to this collection in various ways, and are flushed to the head section of the page after pre-render. To flush the tags to the page, each is formed into a custom `MetaTag` control.

No modifications to the solution's code or markup are required; all of the artifacts and event hooks are created using pipeline processors added to the `httpRequestBegin` and `insertRenderings` pipelines.

A template is provided, which can be added to the base templates of any site template. This gives the content author an opportunity to include pre-defined tags in any given page. These pre-defined tags are managed in the Modules section of the content tree to create tags that are either optional, or included globally in the site.

The process for gathering, forming and injecting tags is itself managed with a custom pipeline. Developers can add tags by inserting processors into this pipeline. They can also change the behavior of the process by replacing existing processors.

Defining Tags

There are several types of tags that can be defined:

1. **Static tags:** Tags where both the name and value are static, such a "robots noindex".
2. **Custom tags:** Tags where either the name, value or both are generated by custom logic.

3. **Method tags:** Tags where the name is static, and the value is derived from an existing static method.
4. **Property tags:** Tags where the name is static, and the value is derived from an existing static property.

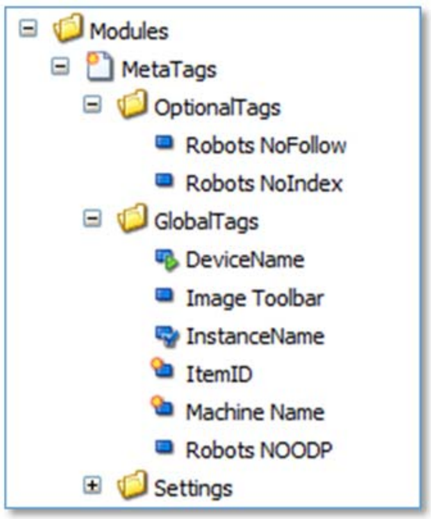
Static tags, Method tags and Property tags can all be defined entirely in configuration … no coding is required. Custom tags are defined in code, and then used in the solution through configuration.

Note that pipeline processors can add tags, using custom logic, using any of these pre-defined tag types.

Tags can be defined either in the Content Editor, or the InjectMetaTags pipeline.

Defining tags in the Content Editor

Tags are defined in the Content Editor by creating Tag Items in the MetaTags Module section.



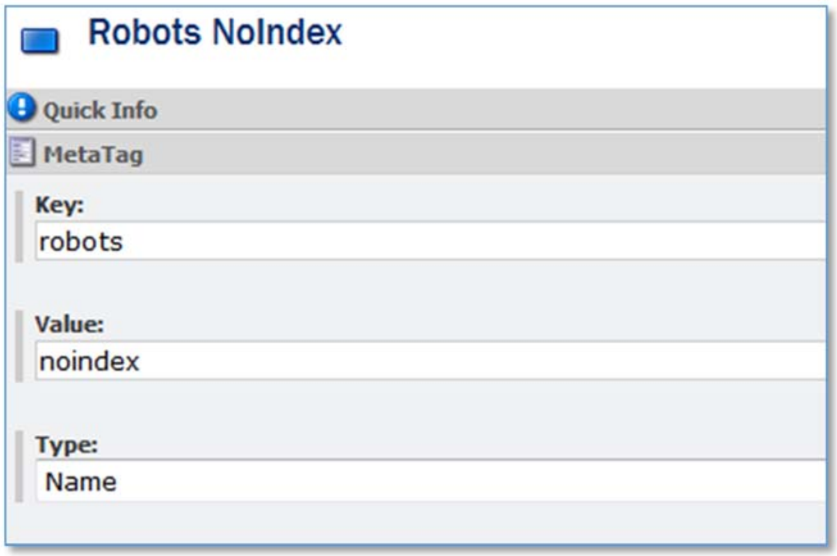
There are two places where tags are defined in the Content Editor

- Items created in the GlobalTags folder define tags that will be inserted on every page of the site.
- Items created in the OptionalTags folder are available for content authors to include selectively on individual pages.

There are four templates for creating new tag definitions: *StaticMetaTag*, *PropertyMetaTag*, *MethodMetaTag* and *CustomMetaTag*.

Defining a StaticMetaTag in the Content Editor

Static tags are simply name/value pairs for common tags. These tags are created by adding a content item using the `StaticMetaTag` template. For example, there is a pre-defined tag for “`Robots NoIndex`”:



Robots NoIndex

Quick Info

MetaTag

Key: robots

Value: noindex

Type: Name

- **Key**: The value for the "name" or "property" attribute (depending on the "Type" setting).
- **Value**: The value of the "content" attribute.
- **Type**: Select "Name" for a meta tag that expresses the key as a "name" attribute, or "Property: for a meta tag that expresses the key as a "property" attribute.

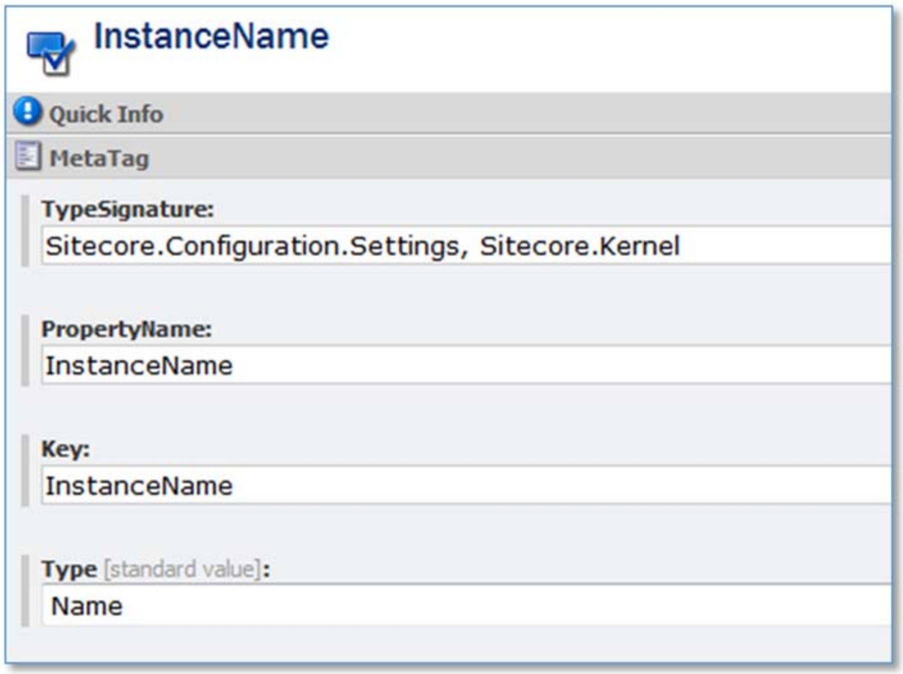
The example above generates this tag:

```
<meta name="robots" content="noindex" />
```

To define a Static tag in the pipeline, see Defining a StaticMetaTag in the InjectMetaTags pipeline.

Defining a PropertyMetaTag in the Content Editor

Property tags have static names, but their value is derived at runtime from the value of any static property, without coding. This might be a property in your solution, or a property in the Sitecore namespace. These tags are created by adding a content item using the `PropertyMetaTag` template. For example, there is a pre-defined tag for "`InstanceName`":


InstanceName

Quick Info

MetaTag

TypeSignature:
Sitecore.Configuration.Settings, Sitecore.Kernel

PropertyName:
InstanceName

Key:
InstanceName

Type [standard value]:
Name

- **TypeSignature**: The class name and assembly where the property is located.
- **PropertyName**: The property in that class that returns a string value to be used in the tag.
- **Key**: The value for the “name” or “property” attribute (depending on the “Type” setting).
- **Type**: Select “Name” for a meta tag that expresses the key as a “name” attribute, or “Property: for a meta tag that expresses the key as a “property” attribute.

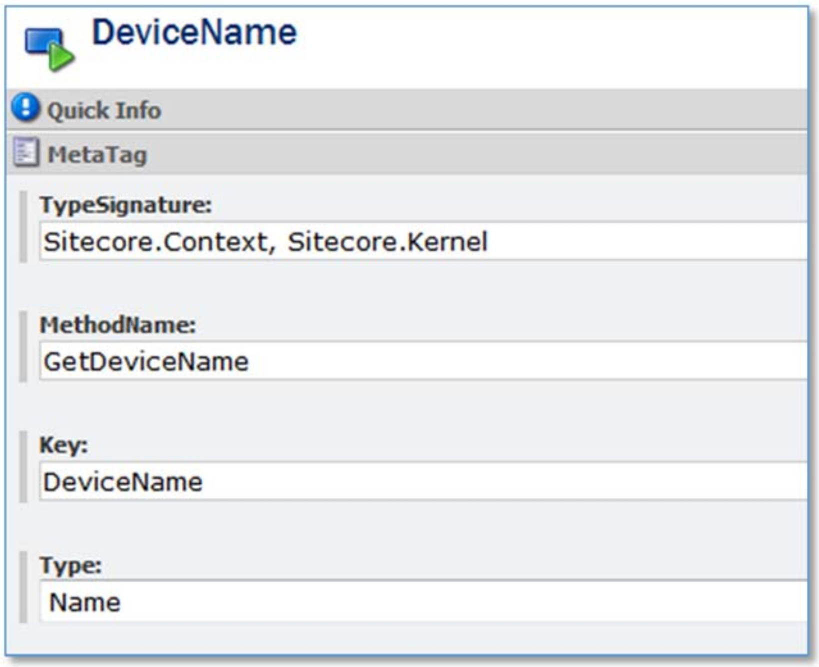
The example above generates this tag:

```
<meta name="InstanceName" content="MYMACHINE-MYSITE"/>
```

To define a Property tag in the pipeline, see Defining a PropertyMetaTag in the InjectMetaTags pipeline.

Defining a MethodMetaTag in the Content Editor

Method tags have static names, but their value is derived at runtime from the value of any static method, without coding. This might be a method in your solution, or a method in the Sitecore namespace. These tags are created by adding a content item using the `MethodMetaTag` template. For example, there is a pre-defined tag for “`DeviceName`”:



DeviceName

Quick Info

MetaTag

TypeSignature:
Sitecore.Context, Sitecore.Kernel

MethodName:
GetDeviceName

Key:
DeviceName

Type:
Name

- **TypeSignature**: The class name and assembly where the property is located.
- **PropertyName**: The property in that class that returns a string value to be used in the tag.
- **Key**: The value for the “name” or “property” attribute (depending on the “Type” setting).
- **Type**: Select “Name” for a meta tag that expresses the key as a “name” attribute, or “Property: for a meta tag that expresses the key as a “property” attribute.

The example above generates this tag:

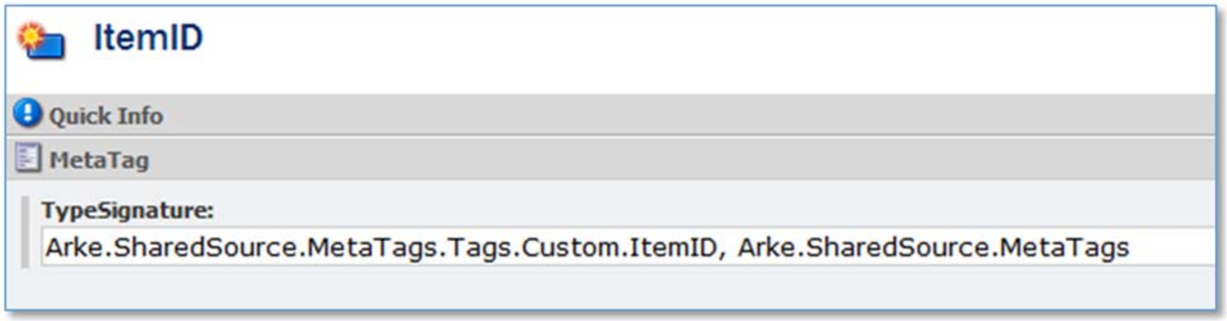
- `<meta name="DeviceName" content="Default"/>`

To define a Method tag in the pipeline, see Defining a MethodMetaTag in the InjectMetaTags pipeline.

Defining a CustomMetaTag in the Content Editor

In a Custom tag, both the name and value are defined in custom code. This custom code is implemented by creating a class that inherits from `Arke.SharedSource.MetaTags.Tags.BaseTag`. For more information, see Coding Custom meta tags.

Custom tags are used in content editor by creating an item using the `CustomMetaTag` template:



ItemID

Quick Info

MetaTag

TypeSignature:
Arke.SharedSource.MetaTags.Tags.Custom.ItemID, Arke.SharedSource.MetaTags

- **TypeSignature**: The class name and assembly of the custom class.

The example above generates this tag:

```
<meta name="ItemID" content="{110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9}"/>
```

Defining tags in the InjectMetaTags pipeline

Tags are created in the InjectMetaTags pipeline by adding processors to the config. These processors are added after the “checks” processors, and before the “flush” processor. Tags added in the pipeline are added to all pages in the site.

Static tags can be added, where the name and value are statically defined in the processor definition.

Fully custom processors are created using a class that interits from `IInjectMetaTagsPipelineProcessor`. These custom processors can add one or more tags (or none at all), based on your logic.

However, many runtime-generated tags can be added entirely in configuration, without coding (or that leverage existing code):

- Method tags
- Property tags
- Custom tags

```
<Arke.MetaTags.InjectMetaTags>
  <-- These standard processors check to see if the pipeline
      should proceed -->
  <processor type="Arke...CheckContextItem, Arke...MetaTags" />
  <processor type="Arke...CheckHeader, Arke...MetaTags" />

  <-- A Static tag -->
  <processor type="Arke...StaticTag, Arke...MetaTags">
    <TagName>copyright</TagName>
    <TagValue>&amp;copy; MySite.com</TagValue>
    <TagType>name</TagType>
  </processor>

  <-- A Method tag -->
  <processor type="Arke...MethodTag, Arke...MetaTags">
    <TypeSignature>Sitecore.Context, Sitecore.Kernel</TypeSignature>
    <MethodName>GetSiteName</MethodName>
    <TagName>sc_site</TagName>
    <TagType>name</TagType>
  </processor>

  <-- A Property tag -->
  <processor type="Arke...PropertyTag, Arke...MetaTags">
    <TypeSignature>Sitecore.Context, Sitecore.Kernel</TypeSignature>
    <PropertyName>RequestID</PropertyName>
    <TagName>RequestID</TagName>
    <TagType>name</TagType>
  </processor>
  <-- A Custom tag -->
  <processor type="Arke...CustomTag, Arke...MetaTags">
    <TypeSignature>Arke...TemplateKey, Arke...MetaTags</TypeSignature>
  </processor>

  <-- A custom processor -->
  <processor type="Arke...SearchScopes, Arke...MetaTags" />

  <-- These processors gather the tags that are
      Defined in the content editor -->
  <processor type="Arke...GlobalTags, Arke...MetaTags" />
  <processor type="Arke...ItemTags, Arke...MetaTags" />

  <-- This processor flushes all the tags to the page -->
  <processor type="Arke...FlushMetaTags, Arke...MetaTags" />
</Arke.MetaTags.InjectMetaTags>
```

There are several types of processors available.

Defining a StaticMetaTag in the InjectMetaTags pipeline

Static tags are simply name/value pairs for common tags. These tags are created using a `Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.StaticTag` pipeline processor

```
<processor type="
    Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.StaticTag,
    Arke.SharedSource.MetaTags">
  <TagName>copyright</TagName>
  <TagValue>&amp;copy; MySite.com</TagValue>
  <TagType>name</TagType>
</processor>
```

- **TagName**: The value for the "name" or "property" attribute (depending on the "Type" setting).
- **TagValue**: The value of the "content" attribute.
- **TagType**: Use "Name" for a meta tag that expresses the key as a "name" attribute, or "Property: for a meta tag that expresses the key as a "property" attribute.

The example above generates this tag:

```
<meta name="copyright" content="&copy; MySite.com" />
```

To define a Static tag in the Content Editor, see Defining a StaticMetaTag in the Content Editor.

Defining a PropertyMetaTag in the InjectMetaTags pipeline

Property tags have static names, but their value is derived at runtime from the value of any static property, without coding. This might be a property in your solution, or a property in the Sitecore namespace. These tags are created created using a

`Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.PropertyTag` pipeline processor.

```
<processor type="
    Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.PropertyTag,
    Arke.SharedSource.MetaTags">
  <TypeSignature>Sitecore.Context, Sitecore.Kernel</TypeSignature>
  <PropertyName>RequestID</PropertyName>
  <TagName>RequestID</TagName>
  <TagType>name</TagType>
</processor>
```

- **TypeSignature**: The class name and assembly where the property is located.
- **PropertyName**: The property in that class that returns a string value to be used in the tag.
- **Key**: The value for the “name” or “property” attribute (depending on the “TagType”).
- **Type**: Use “Name” for a meta tag that expresses the key as a “name” attribute, or “Property: for a meta tag that expresses the key as a “property” attribute.

The example above generates this tag:

```
<meta name="RequestID" content="{a5e26cf8-b33b-4b26-9f24-
8201c39ec7fa}"/>
```

To define a Property tag in the Content Editor, see Defining a PropertyMetaTag in the Content Editor.

Defining a MethodMetaTag in the InjectMetaTags pipeline

Method tags have static names, but their value is derived at runtime from the value of any static method, without coding. This might be a method in your solution, or a method in the Sitecore namespace. These tags are created using a `Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.MethodTag` pipeline processor.

```
<processor type="
    Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.MethodTag,
    Arke.SharedSource.MetaTags">
  <TypeSignature>Sitecore.Context, Sitecore.Kernel</TypeSignature>
  <MethodName>GetSiteName</MethodName>
  <TagName>sc_site</TagName>
  <TagType>name</TagType>
</processor>
```

- **TypeSignature**: The class name and assembly where the property is located.
- **MethodName**: The method in that class that returns a string value to be used in the tag.
- **Key**: The value for the “name” or “property” attribute (depending on the “TagType”).
- **Type**: Use “Name” for a meta tag that expresses the key as a “name” attribute, or “Property: for a meta tag that expresses the key as a “property” attribute.

The example above generates this tag:

- `<meta name="sc_site" content="website"/>`

To define a Method tag in the Content Editor, see Defining a MethodMetaTag in the Content Editor.

Defining a CustomMetaTag in the InjectMetaTags pipeline

In a Custom tag, both the name and value are defined in custom code. This custom code is implemented by creating a class that inherits from `Arke.SharedSource.MetaTags.Tags.BaseTag`. For more information, see Coding Custom meta tags.

Custom tags are created using a `Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.CustomTag` pipeline processor:

```
<processor type="
    Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.CustomTag,
    Arke.SharedSource.MetaTags">
  <TypeSignature>
    Arke.SharedSource.MetaTags.Tags.Custom.TemplateKey,
    Arke.SharedSource.MetaTags
  </TypeSignature>
</processor>
```

- **TypeSignature**: The class name and assembly of the custom class.

The example above generates this tag:

<meta name="TemplateKey" content="sample item"/>

To define a Custom tag in the Content Editor, see Defining a CustomMetaTag in the InjectMetaTags pipeline.

Defining a Custom Processor in the InjectMetaTags pipeline

While other tag types add exactly one tag, custom processors are “wide open” and can inject any number of tags at runtime. Custom tags are implemented in code, and are wired to the InjectMetaTags pipeline just like any other pipeline processor:

```
<processor
  type="Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.SearchScopes,
  Arke.SharedSource.MetaTags" />
```

Custom processors can be defined in the InjectMetaTags pipeline, but not in the content editor.

For more information, see Coding InjectMetaTags pipeline processors.

Adding Tags

Adding tags in the InjectMetaTags pipeline

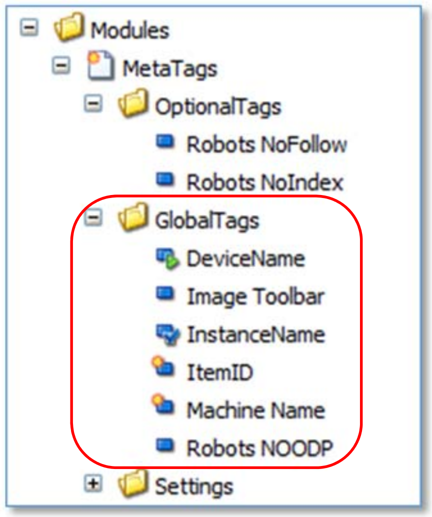
All tags defined on the InjectMetaTags pipeline are added to every page of the site. Tags are not added to pages in the shell site.

Tags defined in the pipeline using the StaticTag, MethodTag or CustomTag processors each add exactly one tag to the page. If the method or property returns null, no tag is added.

Custom processors can add zero or more tags (or none at all) based on your custom logic.

Adding global tags in the content editor

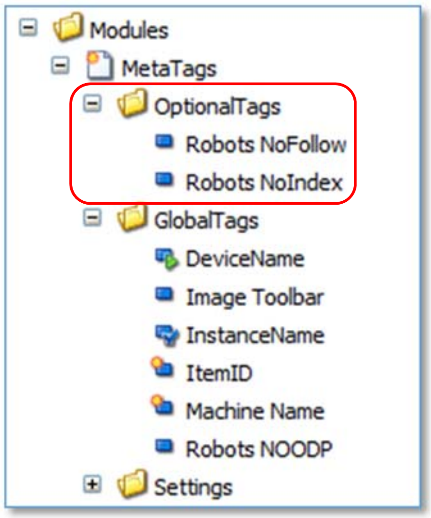
Tags defined in the GlobalTags folder of the MetaTag module are added to every page of the site.



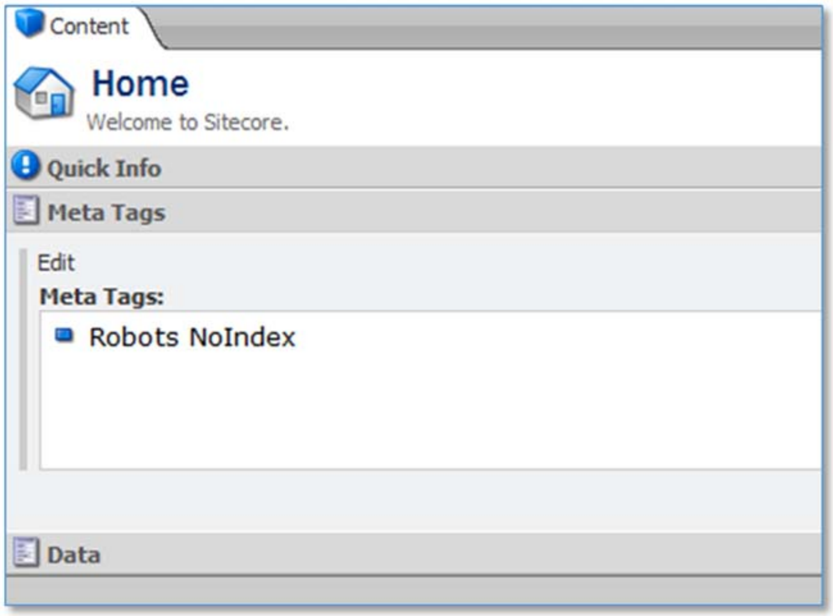
No coding is required to emit these tags to the page. The pre-defined pipeline processor `Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.GlobalTags` is responsible for emitting these tags to the page.

Adding page-level tags in the content editor

Tags defined in the OptionalTags folder of the MetaTag module are not automatically added, but are available to be added to any page by content authors.

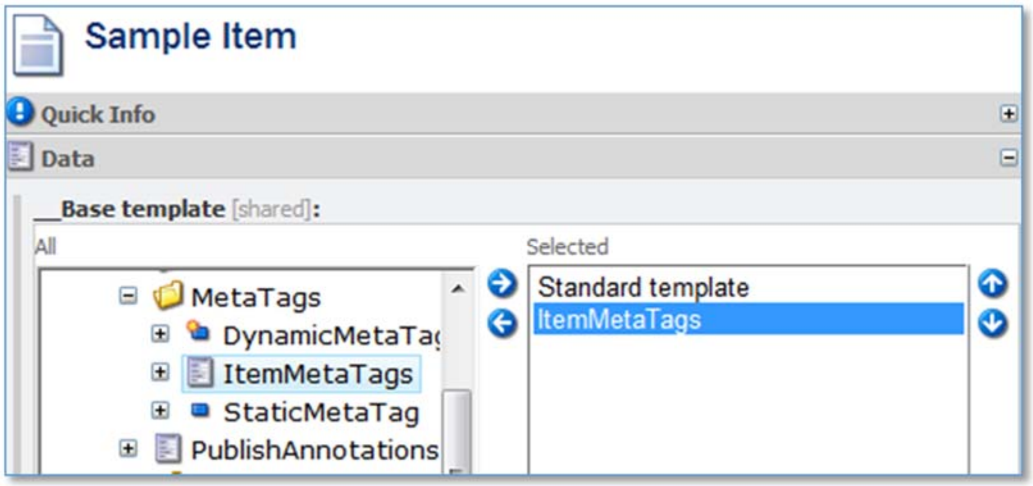


To allow content authors to include optional tags, you can modify any template to inherit from the `ItemMetaTags` template. This will put a Meta Tags multilist field in the item.



The ItemMetaTags template

A base template called `ItemMetaTags` is included with the module. Any template can inherit from this base template, allowing content authors to add optional tags to any page.



Once a template inherits from `ItemMetaTags`, no other coding is necessary to implement these tags. The pre-defined pipeline processor `Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags.ItemTags` is responsible for gathering and inserting these tags.

Adding page-level tags in code.

Coders may need to add meta tags to the page from some point in their code. The code for a layout, sublayout or rendering may need to add a meta tag to the page, for example, for data needed by a script, crawler, or metrics reporting.

See *Adding meta tags from code using convenience methods*.

Coding tags

Coding InjectMetaTags pipeline processors

The most flexible (and least convenient) way to add meta tags using this module is to create a custom processor for the `InjectMetaTags` pipeline.

While other tag processors add exactly one tag, custom processors can add as many tags as they wish, or none at all. For example, a processor might only add tags for based on specific templates, or the position of the context item on the content tree, time of day, or any other criteria.

Custom processors inherit from `IInjectMetaTagsPipelineProcessor`, and implement the usual `Process` method.

The process method accepts an `InjectMetaTagsPipelineArgs` argument, which provides access to the `MetaTags` collection. Logic in the Process method can add any number of tags to this collection. These can be `StaticTags`, `MethodTags` or `PropertyTags` but are typically `StaticTags`.

For this example, consider a solution that uses a Google Search Appliance for search services. The GSA can be configured to gather information from meta tags when crawling, and later use this data to filter search results. This solution uses a `SearchScopes` meta tag, which includes the GUID of every ancestor of the current item. In this way, any item in the site could be used as a search scope; that is, search results could be limited to that item or its descendents.

```
namespace Arke.SharedSource.MetaTags.Pipelines.InjectMetaTags
{
  public class SearchScopes : IInjectMetaTagsPipelineProcessor
  {
    public void Process(InjectMetaTagsPipelineArgs args)
    {
      Sitecore.Diagnostics.Profiler.StartOperation(
        "Adding SearchScopes tag.");
      List<string> scopes = new List<string>();
      try
      {
        scopes.Add(Sitecore.Context.Item.ID.ToString());

        Sitecore.Sites.SiteContext site = Sitecore.Context.Site;
        string startPath = String.Empty;
        if (site != null)
        {
          startPath = site.StartPath;
        }
        foreach (Item item in Sitecore.Context.Item.Axes.GetAncestors())
        {
          if (item.Paths.FullPath.Length >= startPath.Length)
          {
            scopes.Add(item.ID.ToString());
          }
        }
        Tags.StaticTag tag = new Tags.StaticTag(
          MetaTagType.name,
          "SearchScopes",
          Sitecore.StringUtil.Join(scopes.ToArray(), " "));
        args.MetaTags.Add(tag);
      }
      catch (Exception ex)
      {
        Sitecore.Diagnostics.Tracer.Error(
          "SearchScopes MetaTag failed.", ex);
      }
      finally
      {
        Profiler.EndOperation(scopes.Count.ToString() + " scopes added.");
      }
    }
  }
}
```

Coding Custom meta tags

To code a custom tag, create a class that interits from `Arke.SharedSource.MetaTags.Tags.BaseTag`. These tags can be leveraged either in the Content Editor, or in the InjectMetaTags pipeline.

Custom tags set three properties (defined in `BaseTag`) from the constructor:

- **`TagType`**: Use "Name" for a meta tag that expresses the key as a "name" attribute, or "Property: for a meta tag that expresses the key as a "property" attribute.
- **`Key`**: The value for the "name" or "property" attribute (depending on the "TagType").
- **`Value`**: The value of the "content" attribute.

For example, there is a pre-defined class for "`ItemID`":

```
namespace Arke.SharedSource.MetaTags.Tags.Custom
{
  public class ItemID : BaseTag
  {
    public ItemID()
    {
      TagType = MetaTagType.name;
      Key = "ItemID";
      Value = Sitecore.Context.Item.ID.ToString();
    }
  }
}
```

The example above generates this tag:

```
<meta name="ItemID" content="{110D559F-DEA5-42EA-9C1C-
8A5DF7E70EF9}"/>
```

Adding meta tags from code using convenience methods

There times when it is necessary to add meta tags to the page from code, perhaps in a layout or sublayout, or from a rendering, or from any other logic executing during the page lifecycle. To facilitate this, the module includes a `PageTags` class that contains a convenience method called `AddTag()`.

There are two flavors of the `AddTag()` method, to simplify adding tags from code:

- `AddTag(MetaTagType TagType, string Key, string Value)`
  Use this method to add a static tag on the fly by supplying the necessary parameters.

- `AddTag(BaseTag Tag)`
  Use this method to add a pre-built tag (either a `StaticTag`, a `PropertyTag` or a `MethodTag`).

Note that the module allows the same kind of tag to be added multiple times. It is legal for a page to have multiple tags with the same key (name or property), so the module does not try to make sure you only add "unique" tags.

The MetaTag control and Tag objects

At the end of the pipeline, the FlushMetaTags processor adds all of the tags accumulated in the AllTags collection to the page. This is accomplished using a custom control called MetaTag. This control exposes properties for `TagType`, `Key` and `Value`. The Render method assembles an html `meta` tag using these properties.

The module includes objects for `StaticTag`, `MethodTag` and `PropertyTag`, all of which inherit from `BaseTag`. These objects are used by pipeline processors to add tags to the `MetaTags` collection. Each of these objects contains constructors to simplify the creation of the various types of tags. Each also exposes a `GetControl` method, which returns a `MetaTag` control.

The InjectMetaTags Pipeline

The core of the module is the `InjectMetaTags` pipeline. This pipeline manages the gathering, formation and injection of all tags that were created by the module's other functions.

The first two processors ensure that there is an existing context item, and that the `head` section of the page has a `runat=server` attribute. Without these, the pipeline aborts.

The final processor iterates over the context collection, creates renderings, and inserts them into the head of the page.

The middle processors add tags to the current context, which are flushed to the page by the `FlushMetaTags` processor. See Defining tags in the InjectMetaTags pipeline.