InsertionsManager for ASP.NET MVC

The **`InsertionsManager`** class lets your Partial Views and Html Helpers add their scripts and style sheets into the appropriate areas of the page, often defined in a different View file (such as the "master page"). Your code also can insert hidden fields, array declarations, meta tags, JavaScript templates, and any other content in predefined locations on the page. It supports the ASP.NET Razor View Engine, but other View Engines can be expanded to support it too.

Background

Partial Views and Html Helpers inject content into the page at the location where they are defined. This is fine for adding HTML, but often you want these tools to create something more complex, like a calendar control or filtered textbox, which need JavaScript, both in files and inline, and style sheet classes which do not belong side-by-side with the HTML being inserted. They belong in specific locations in the page, often in the master page.

The Razor View Engine for ASP.NET MVC makes process of exposing these elements awkward. You often have to create `@section` groups and hope 1) that the containing page knows to load that section and 2) that you are not adding duplicate scripts and styles.

The `InsertionsManager` class extends the Razor View Engine to let your Views and Html Helpers register anything it wants inserted, and handles duplicates correctly. After Razor finishes preparing the page content, InsertionsManager will act as a post-processor to locate **Insertion Points** throughout the page and replace them with the content your Views and Html Helpers have registered.

Click on any of these topics to jump to them:

Example

Here is a typical master page (_layout.cshtml) when using the `InsertionsManager`:

```
@using InsertionsManagement;

@{
    this.InsertionsManager().AddScriptFile("~/Scripts/jquery-1.5.1.min.js");
    this.InsertionsManager().AddScriptFile("~/Scripts/modernizr-1.7.min.js");
    this.InsertionsManager().AddStyleFile("~/Content/Site.css");
}
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <meta charset="utf-8" />
   <!-- replace-with="MetaTags" -->
   <!-- replace-with="StyleFiles" -->
   <!-- replace-with="ScriptFiles" -->
</head>

<body>
   <!-- replace-with="ScriptBlocks:Upper" -->
    @RenderBody()
   <!-- replace-with="ScriptBlocks:Lower" -->
</body>
</html>
```

Understanding this code:

- The `@using InsertionsManagement` statement establishes extension classes, including the ability to use `this.InsertionsManager()`. You will add this to each View and Html Helper to interacts with the `InsertionsManager`.
- Calls made on `this.InsertionsManager()` attach the desired content. In this case, it adds two script file URLs and a style sheet URL. If you wanted to ensure a specific order to your scripts, you can pass an order value as an additional parameter:

  ```
  this.InsertionsManager().AddScriptFile("~/Scripts/jquery-1.5.1.min.js", 10);
  this.InsertionsManager().AddScriptFile("~/Scripts/modernizr-1.7.min.js", 20);
  ```

- The comment tags containing "`replace-with=`" are the Insertion Points. Each has a name identifying the type of content it will output.

Now suppose your View inserted at `@RenderBody()` needs to use jquery-validation. Its code should include:

```
@using InsertionsManagement
@model Models.MyModel

@{
   this.InsertionsManager().AddScriptFile("~/Scripts/jquery.validate.min.js", 10);
   this.InsertionsManager().AddScriptFile(
      "~/Scripts/jquery.validate.unobtrusive.min.js", 11);
}
```

Continued on the next page

Here is the resulting HTML output:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Create</title>

   <link href="/Content/Site.css" type="text/css" rel="stylesheet" />

   <script src="/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
<script src="/Scripts/modernizr-1.7.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.unobtrusive.min.js" type="text/javascript"></scri
pt>

</head>
<body>

    The view’s content goes here.

</body>
</html>
```

The comment tags that did not have content have been deleted.

Let’s add some script blocks, one assigned to the “ScriptBlocks:Upper” Insertion Point and the other to the “ScriptBlocks:Lower” Insertion Point.

```
@using InsertionsManagement
@model Models.MyModel

@{
   this.InsertionsManager().AddScriptFile("~/Scripts/jquery.validate.min.js");
   this.InsertionsManager().AddScriptFile(
      "~/Scripts/jquery.validate.unobtrusive.min.js");
   this.InsertionsManager().AddScriptBlock(null,
      "function test() {alert('hello');}", 0, "Upper");
   this.InsertionsManager().AddScriptBlock(null, "test();", 0, "Lower"););
}
```

Here is the resulting <body> tag’s output:

```
<body>
   function test() {alert('hello');}
    The view’s content goes here.
   test();
</body>
```

Adding InsertionsManager to your application

1. Add the InsertionsManagement.dll assembly to your web application. (Alternatively, add the source code project and set a reference from your application to it.)
2. Add this code to the `Application_Start()` method of Global.asax. It switches to using a subclass of the RazorView that supports the `InsertionsManager`.

```
ViewEngines.Engines.Clear();
ViewEngines.Engines.Add(new InsertionsManagement.IMRazorViewEngine());
```

Note: If you have subclassed the RazorView class, do not use step 2. Instead, see "Using your own RazorView subclass".

Adding Insertion Points to the page

An Insertion Point is an HTML comment with this pattern:

```
<!-- replace-with="identifier" -->
```

Identifiers are as follows:

- "ScriptFiles" – Creates `<script src="url" type="text/javascript" >` tags.
- "StyleFiles" – Creates `<link href="url" type="text/css" rel="stylesheet" />` tags.
- "MetaTags" – Creates `<meta name="name" content="content" />`
- "ScriptBlocks" – Creates a `<script type="text/javascript">` block to host JavaScript code. There are usually two of these on the page, one above and the other below the HTML content their scripts operate on. Use the Group names feature, below, with this Insertion Point.
  These can also create an array declaration that defines a variable name assigned to an array. These arrays are often populated by different areas of code. The idea is similar to the `System.Web.UI.ClientScriptManager.RegisterArrayDeclaration()` method.
- "HiddenFields" – Creates `<input type="hidden" name="name" value="value" />` tags. This is often placed inside a <form> tag to allow grouping all hidden fields.
- "TemplateBlocks"– Creates a `<script type="some value">` that hosts a JavaScript template as defined by jQuery-templates, Knockout.js, underscore.js, or KendoUI. *Note: This feature requires that you register the appropriate class first.*
- "Placeholder" – Creates the exact content supplied. It does not add any tags itself.

You can add Insertion Points into any area of the Views. There are five typical insertion points that are added into the master page (_layout.cshtml).

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
   <!-- replace-with="MetaTags" -->
   <!-- replace-with="StyleFiles" -->
   <!-- replace-with="ScriptFiles" -->
</head>

<body>
   <!-- replace-with="ScriptBlocks:Upper" -->
    @RenderBody()
   <!-- replace-with="ScriptBlocks:Lower" -->
</body>
</html>
```

Group names on Insertion Points

Sometimes you need several insertion points with the same identifier, but to insert different content. In that case, add each with a unique group name. The pattern is:

```
<!-- replace-with="identifier:groupname" -->
```

The "ScriptBlocks" and "Placeholder" identifiers are almost always used with group names. All others optionally use it. The example above shows ScriptBlocks using "Upper" and "Lower".

Keep in mind that you can use Razor syntax to create a variable for the group name. For example, suppose the variable *MyVariable* is defined:

```
<!-- replace-with="Placeholder:@MyVariable" -->
```

Defining the content that is inserted

Start by adding a `@using InsertionsManagement` statement to the top of your view. This brings in the extensions methods that support the remaining functionality.

```
@using InsertionsManagement
```

Then call the appropriate method on `this.InsertionsManager()` as described below.

Note: These methods use default parameters, which appear with "= value" in the definitions. They allow you to omit the parameter and get the default value passed in. The order and groupname parameters always have defaults.

Click on any of these topics to jump to them:

Adding script files

Associated Insertion Point: `<!-- replace-with="ScriptFiles" -->`

```
void AddScriptFile(string url, int order = 0, string groupName = "");
```

Parameters

- url – The value of the `src` attribute on the script tag. It can start with "~/" to indicate that the path starts with the application root folder.
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering, so dependencies occur earlier, assign this. Lower numbers appear higher in the page output.

  Suggestion: You may want to plan out order numbers throughout your script file library in advance to ensure they are consistently used.
- groupName – Use only if you define an Insertion Point with this group name defined.

Example 1

This code:

```
this.InsertionsManager().AddScriptFile("~/Scripts/jquery-1.9.1.js");
```

generates this HTML:

```
<script src="/Scripts/jquery-1.9.1js" type="text/javascript" />
```

Example 2: Ordering

jquery-#.#.#.js should always appear before jquery-validate.js.

```
this.InsertionsManager().AddScriptFile("~/Scripts/jquery-1.9.1.js", 0);
this.InsertionsManager().AddScriptFile("~/Scripts/jquery-validate.js", 10);
```

Note: One can imagine a more comprehensive tool to identify script files that knows how to insert dependencies and maintain order. InsertionsManager is designed for expansion, but you will have to implement it. See "Adding a new type of Insertion Point".

Adding style sheet files

Associated Insertion Point: `<!-- replace-with="StyleFiles" -->`

```
void AddStyleFile(string url, int order = 0, string groupName = "");
```

Parameters

- url – The value of the `href` attribute on the link tag. It can start with “~/” to indicate that the path starts with the application root folder.
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering assign this. Lower numbers appear higher in the page output.
- groupName – Use only if you define an Insertion Point with this group name defined.

Example

This code:

```
this.InsertionsManager().AddStyleFile("~/Content/StyleSheet.css");
```

generates this HTML:

```
<link href="/Content/StyleSheet.css" type="text/css" rel="stylesheet" />
```

Adding script blocks

Associated Insertion Points: `<!-- replace-with="ScriptBlocks" -->`

Typically you will have two group names, “Upper” for scripts above the HTML and “Lower” for scripts below the HTML that they operate upon.

```
void AddScriptBlock(string key, string script, int order = 0,
   string groupName = "");
```

Parameters

- key – Uniquely identifies the script with a name. If the script should always be added, leave this `null` or `""`. If the same script may be added multiple times, the key will prevent adding duplicates. You don’t have to check if there is a conflict with the key. `AddScriptBlock()` just skips adding the duplicate. If you want to check first, call `this.InsertionsManager().Contains("key")`. It returns `true` if the key is already defined.
- script – The JavaScript code that is the body of the script tag. Do not include `<script>` tags.
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering, so dependencies occur earlier, assign this. Lower numbers appear higher in the page output.
- groupName – Typically assigned to “Upper” or “Lower”.

Examples

Both of these lines:

```
this.InsertionsManager().AddScriptBlock("somekey", "callSomeFunction();", 0, "Lower");
```

generate this JavaScript:

```
callSomeFunction();
```

Adding array declarations

Associated Insertion Points: `<!-- replace-with="ScriptBlocks" -->`

Typically you will have two group names, “Upper” for scripts above the HTML and “Lower” for scripts below the HTML that they operate upon.

Array declarations create a variable name assigned to an array. Each call to these functions appends a value to the array for the given variable name. Values passed to the `ArrayDeclaration()` methods should be in their native type, such as double, int, and string. The type is used to determine how to convert it into a JavaScript value.

The `ArrayDeclarationAsCode()` method inserts the string exactly as is. It is expected to be JavaScript code that is fully compatible with being a parameter of an array. This is often used to add a value of `null`.

```
void ArrayDeclaration(string variableName, string value, bool htmlEncode = true,
   int order = 0, string groupName = "");
void ArrayDeclaration(string variableName, int value, int order = 0,
   string groupName = "");
void ArrayDeclaration(string variableName, double value, int order = 0,
   string groupName = "");
void ArrayDeclaration(string variableName, bool value, int order = 0,
   string groupName = "");
void ArrayDeclarationAsCode(string variableName, string script, int order = 0,
   string groupName = "");
```

Note: There are also declarations that support Single, Decimal, Int16, and Int64 types on the value parameter.

Parameters

- variableName – The name of the variable to add to the page. It must be a valid JavaScript identifier and is case sensitive.
- value – The value to add as the next item to add to the array.
- order – This inpacts the ordering of array names, not the elements within an array. Normally array declarations are ordered in the same order they were registered. If you want to ensure the ordering, so dependencies occur earlier, assign this. Lower numbers appear higher in the page output.
- groupName – Typically assigned to “Upper” or “Lower”.
- htmlEncode – Strings get HTML encoded when this is true. If not specified, it defaults to true.

Example

The code below creates this array:

```
var myVar = ["my string value", 100, null];
```

```
this.InsertionsManager().ArrayDeclaration("myVar", "my string value", 0, "Lower");
this.InsertionsManager().ArrayDeclaration("myVar", 100, 0, "Lower");
this.InsertionsManager().ArrayDeclarationAsCode("MyVar", "null", 0, "Lower");
```

Adding meta tags

Associated Insertion Point: `<!-- replace-with="MetaTags" -->`

```
void AddMetaTag (string name, string content, int order = 0, string groupName = "");

void AddMetaTag (MetaTagUsage usage, string name, string content, int order = 0,
   string groupName = "");
```

Parameters

- usage – Determines the attribute type that holds the value of the *name* parameter. It can be `name`, `http-equiv`, or `charset`, based on the `MetaTagUsage` enumerated type. Values are `MetaTagUsage.Name`, `MetaTagUsage.HttpEquiv`, `MetaTagUsage.CharSet`. When not supplied, it defautls to `MetaTagUsage.Name`.
- name – Defines the value of attribute specified in the usage parameter. If a duplicate is used, the previous entry is replaced.
- content - Defines the value of the `content` attribute in the meta tag
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering assign this. Lower numbers appear higher in the page output.
- groupName – Use only if you define an Insertion Point with this group name defined.

Example

This code:

```
this.InsertionsManager().AddMetaTag("description", "about my site");
this.InsertionsManager().AddMetaTag(MetaTagUsage.HttpEquiv,
   "Content-Type", "text/html; charset=iso-8859-1");
```

generates this HTML:

```
<meta name="description" content="about my site" />
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
```

Adding hidden fields

Associated Insertion Point: `<!-- replace-with="HiddenFields" -->`

```
void AddHiddenField(string name, string value, int order = 0, string groupName = "");
```

Parameters

- name – Defines the value of the `name` attribute in the `<input>` tag. If a duplicate is used, it is ignored
- value – Defines the value of the `value` attribute in the `<input>` tag.
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering, so dependencies occur earlier, assign this. Lower numbers appear higher in the page output.
- groupName – Use only if you define an Insertion Point with this group name defined.

Example

This code:

```
this.InsertionsManager().AddHiddenField("codes", "AB903F");
```

generates this HTML:

```
<input type="hidden" name="codes" value="AB903F" />
```

Adding JavaScript templates

Associated Insertion Point: `<!-- replace-with="TemplateBlocks" -->`

Before using Templates, you must register the Template engine you will be using. Add one of these to `Application_Start()`:

```
InsertionsManagement.InsertionsFactory.Default.Register(
   typeof(ITemplateBlocksInserter), typeof(KnockoutTemplateBlocksInserter));
InsertionsManagement.InsertionsFactory.Default.Register(
   typeof(ITemplateBlocksInserter), typeof(jQueryTemplateBlocksInserter));
InsertionsManagement.InsertionsFactory.Default.Register(
   typeof(ITemplateBlocksInserter), typeof(UnderscoreTemplateBlocksInserter));
InsertionsManagement.InsertionsFactory.Default.Register(
   typeof(ITemplateBlocksInserter), typeof(KendoUiTemplateBlocksInserter));
```

```
void AddTemplatesBlock(string id, string content, int order = 0,
   string groupName = "");
```

Parameters

- id – Defines the value of the `id` attribute in the `<script>` tag. If a duplicate is used, it is ignored.
- content – Defines the inner content of the `<script>` tag.
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering, so dependencies occur earlier, assign this. Lower numbers appear higher in the page output.
- groupName – Use only if you define an Insertion Point with this group name defined.

Example

This code:

```
this.InsertionsManager().AddTemplateBlock("mytemplate", "<li><% -item.value %></li>");
```

generates this HTML:

```
<script id="mytemplate" type="text/template">
   <li><% -item.value %></li>
</script>
```

Adding any other content

Associated Insertion Point: `<!-- replace-with="Placeholder" -->`

```
void AddPlaceholder(string content, int order = 0, string groupName = "");
```

Parameters

- content – This value is written out verbatim. Each call will add new content to what was already registered.
- order – Normally entries are ordered in the same order they were registered. If you want to ensure the ordering assign this. Lower numbers appear higher in the page output.
- groupName – Use only if you define an Insertion Point with this group name defined.

Example

This code:

```
this.InsertionsManager().AddPlaceholder("<!-- xyz library used under license -->");
```

Generates this HTML:

```
<!-- xyz library used under license -->
```

Expanding this framework

`InsertionsManager` is designed to be customized. Here are a few customizations.

Click on any of these topics to jump to them:

Adding a new type of Insertion Point

All Insertion Point classes implement the `InsertionsManagement.IInserter` interface. You should create a new interface definition based on `IInserter`. It will be used by the `InserterFactory` to map to your class.

Most Insertion Point classes inherit from `InsertionsManagement.BaseKeyedInserter` which holds a list of the items added, with a key field that is used to locate an existing item.

When working with `BaseKeyedInserter`, create a class to hold the value of each item added by implementing `InsertionsManagement.IKeyedInserterItem`.

Then subclass `BaseKeyedInserter` with the following modifications:

- Override `ItemContent()` to insert a string based on the item passed in (which is a `IKeyedInserterItem` class).
- Create `Add()` method(s) to accept the properties defined in your IKeyedInserterItem class and create that object. Be sure that you only add when the key is unique. You can use the `Contains()` method to see if the key exists, or `_orderedList.TryGetValue(key, out value)` to get the `IKeyedInserterItem` and modify it.

When finished, register your new class with its interface in the `InsertionsManagement.InserterFactory` like this:

```
InsertionsManagement.InserterFactory.Default.Register(typeof(interface type),
typeof(class type));
```

The identifier on Insertion Points is either the actual interface type or a name based on the interface, where the lead “I” and trailing “Inserter” have been removed. For example, "IMyInserter" supports these as its identifier: "IMyInserter" and "My".

Add extension methods to the `InsertionsManagement.InsertionManager` class that call the Add methods.

```
public static class Extensions
{
   public static void AddSomething(this InsertionsManager insertionManager,
      string value, int order = 0, string group = "")
   {
      insertionManager.Access<IScriptBlocksInserter>(group).Add(value, order);
   }
}
```

Be sure to include the namespace with your Extensions class on any page that uses this type.

Using your own RazorView subclass

Edit your RazorView subclass to ensure this functionality in the RenderView method.

```
protected override void RenderView(ViewContext viewContext,
   System.IO.TextWriter writer, object instance)
{
   using (var insertionsManager = new InsertionsManager(
       writer, viewContext.HttpContext, InserterFactory.Default))
   {
      viewContext.ViewData["InsertionsManager"] = insertionsManager;

      base.RenderView(viewContext, insertionsManager.ContentWriter, instance);

      insertionsManager.UpdatePage();
   }
}
```

Replacing string templates

Most Insertion Points embed their data into an HTML tag, such as “<meta {2}="{0}" content="{1}" />. If you have a different idea on how to format the tag, these “template strings” have been defined as public static (global) properties that you can assign in `Application_Start()`. They include:

`ScriptFilesInserter.ScriptFileTagPattern`

`StyleFilesInserter.StyleFileTagPattern`

`HiddenFieldsInserter.HiddenFieldPattern`

`MetaTagsInserter.MetaTagPattern`

`ScriptBlocksInserter.StartScriptBlockTag`

`ScriptBlocksInserter.EndScriptBlockTag`

Replacing the Insertion Point pattern

The Insertion Point pattern - `<!-- replace-with="identifier:groupname" -->` - may not be to your liking. It is matched by a regular expression that you replace. The expression is defined in a static (global) property. Here is its default.

```
InsertionsManagement.InsertionsManager.TextSearchRE =
   new Regex(@"\<\!\-\-\s+replace-with\s*=\s*[" + "\"" +
      @"'](?<name>\w+)(\:(?<group>\w+))?[" + "\"" + @"']\s*\-\-\>",
      RegexOptions.Compiled);
```

If you replace it, be sure to define two named groups, (?<name>) and (?<group>).