Content Injector for ASP.NET MVC

Content Injector lets you build smarter Views, Partial Views, and Html Helpers which dictate the scripts and style sheets they need. They can safely inject content into the page in areas already rendered, such as the `<head>` tag, where style sheets and some scripts should be located. Content Injector will also prevent outputting duplicate tags that load your style sheets and scripts.

Your Views and Helpers also can inject hidden fields, array declarations, `<meta>` tags, JavaScript templates, and any other content to predefined locations on the page.

It compliments other tools, like Web Optimization Framework, by letting them collect and manage data, while letting Content Injector inject the data into the page.

It supports the ASP.NET Razor View Engine, and is customizable.

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 injected. They belong in specific locations in the page, often in the master page.

The Razor View Engine for ASP.NET MVC does not make this easy.

- You often have to create `@section` groups and hope that the containing page knows to load that section. It's even trickier when working within an Html Helper.
- The same content can be inserted multiple times, such as two Views which add the same `<script>` tag.
- Razor's one-pass rendering engine prevents inserting your content in areas of the page that were already rendered. For example, if your View needs a `<link>` tag in the `<head>`, it has to know about that before it does its rendering.

Microsoft's Web Optimization framework enhances the collection of scripts and style sheets, but still suffers from the above problems.

Content Injector extends the Razor View Engine to let your Views and Html Helpers register anything they want injected into the page, and handles duplicates correctly. After Razor finishes preparing the page content, Content Injector will act as a post-processor to locate **Injection 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:

Forums: http://hosted.comm100.com/Forum/Support,-Discussions-and-Feedback_f576.aspx?siteid=128400

Example

Here is a typical master page (_layout.cshtml) when using Contents Injector:

```
@{
    Injector.ScriptFile("~/Scripts/jquery-1.5.1.min.js");
    Injector.ScriptFile("~/Scripts/modernizr-1.7.min.js");
    Injector.StyleFile("~/Content/Site.css");
}
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    @Injector.InjectionPoint("MetaTags")
    @Injector.InjectionPoint("StyleFiles")
    @Injector.InjectionPoint("ScriptFiles")
</head>

<body>
    @Injector.InjectionPoint("ScriptBlocks", "Upper")
    @RenderBody()
    @Injector.InjectionPoint("ScriptBlocks", "Lower")
</body>
</html>
```

Understanding this code:

- The `Injector` element is property on your page that hosts the methods you use to interact with Content Injector. In this case, the user has added two scripts files 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:

  ```
  Injector.ScriptFile("~/Scripts/jquery-1.5.1.min.js", 10);
  Injector.ScriptFile("~/Scripts/modernizr-1.7.min.js", 20);
  ```

- The `Injector.InjectionPoint()` method marks the location for your content associated with the name given. This call does not output your content. Instead, it just leaves a marker for the Content Injector's post processor to cleanup.

- There are two Injection Points for "ScriptBlocks". Each has been given a unique grouping name as the second parameter. Any injection of script blocks must use the same group name to be injected into the page. For script blocks, there are typically blocks before and after the HTML the scripts operate upon. We've given them group names of "Upper" and "Lower" here.

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

```
@model Models.MyModel

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

Continued on the next page

Here is the resulting HTML output:

```
<!DOCTYPE html>
<html>
<head>
    <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" Injection Point and the other to the "ScriptBlocks:Lower" Injection Point.

```
@model Models.MyModel

@{
   Injector.ScriptFile("~/Scripts/jquery.validate.min.js");
   Injector.ScriptFile("~/Scripts/jquery.validate.unobtrusive.min.js");
   Injector.ScriptBlock("function test() {alert('hello');}", 0, "Upper");
   Injector.ScriptBlock("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 Content Injector to your application

1. Add the **ContentInjector.dll** assembly to your web application. (Alternatively, add the source code project and set a reference from your application to it.)

   You can use either of these approaches to get the assembly:

   - NuGet Package Manager in Visual Studio. Search for "Content Injector".
   - Get the assembly file here and use the Add Reference command in Visual Studio: https://github.com/plblum/ContentInjector/blob/master/Assemblies/ContentInjector.dll?raw=true

2. Open the **Views\web.config** file and locate the `<system.web.mvc.razor>` section.

   - Replace the value of **pageBaseType** with "ContentInjector.CIWebViewPage".

     Note: If you have subclassed System.Web.MVC.WebViewPage, continue to use your class and see "Using your own WebViewPage subclass".

   - Add `<add namespace="ContentInjector" />` into the <namespaces> block.

```
<system.web.webPages.razor>
   <pages pageBaseType="ContentInjector.CIWebViewPage">
      <namespaces>
         <add namespace="System.Web.Mvc" />
         <add namespace="System.Web.Mvc.Ajax" />
         <add namespace="System.Web.Mvc.Html" />
         <add namespace="System.Web.Routing" />
         <add namespace="ContentInjector" />
      </namespaces>
   </pages>
</system.web.webPages.razor>
```

3. Add this code to the `Application_Start()` method of **Global.asax**. It switches to using a subclass of the RazorView that supports Content Injector.

```
ViewEngines.Engines.Clear();
ViewEngines.Engines.Add(new ContentInjector.CIRazorViewEngine());
```

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

4. When content is added but no Injection Point is found on the page, an exception is thrown by default. You can override this by either sending the error to the ASP.NET trace log or ignoring it. Either set the global property `ContentInjector.ContentManager.ReportErrorsMode` in `Application_Start()` or add one of these to the `<appSettings>` section of **web.config**:

```
<add key="CIReportErrorsMode" value="Trace"/>
<add key="CIReportErrorsMode" value="None"/>
```

Adding Injection Points to the page

An Injection Point is established with the Razor syntax by calling `@Injector.InjectionPoint()`:

```
@Injector.InjectionPoint("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 Injection 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 Injection Points into any area of the Views. There are five typical Injection Points that are added into the master page (_layout.cshtml).

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    @Injector.InjectionPoint("MetaTags")
    @Injector.InjectionPoint("StyleFiles")
    @Injector.InjectionPoint("ScriptFiles")
</head>

<body>
    @Injector.InjectionPoint("ScriptBlocks", "Upper")
    @RenderBody()
    @Injector.InjectionPoint("ScriptBlocks", "Lower")
</body>
</html>
```

Group names on Injection Points

Sometimes you need several Injection Points with the same identifier, but to inject different content. In that case, add each with a unique group name. The code is:

```
@Injector.InjectionPoint("identifier", "group name")
```

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".

When Injection Points are missing

When content is added but no Injection Point is found on the page, an exception is thrown by default. You can override this by either sending the error to the ASP.NET trace log or ignoring it. Either set the global property `ContentInjector.ContentManager.ReportErrorsMode` in `Application_Start()` or add one of these to the `<appSettings>` section of **web.config**:

```
<add key="CIReportErrorsMode" value="Trace"/>
<add key="CIReportErrorsMode" value="None"/>
```

Defining the content that is injected

Call the appropriate method on the **Injector** property 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.

Note: These methods are extension methods on the ContentInjector.ContentManager class. If there are compiler errors that cannot identify the method by name, ensure you have either added the namespace ContentInjector to the web.config file (see "Adding Content Injector to your application") or in a using clause on the page.

Click on any of these topics to jump to them:

Adding script files

Associated Injection Point: `@Injector.InjectionPoint("ScriptFiles"[, "group name"])`

Call `Injector.ScriptFile()`. It has this method definition:

```
void ScriptFile(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 Injection Point with this group name defined.

Example 1

This code:

```
@{ Injector.ScriptFile("~/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.

```
@{
   Injector.ScriptFile("~/Scripts/jquery-1.9.1.js", 0)
   Injector.ScriptFile("~/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. Content Injector is designed for expansion, but you will have to implement it. See "Adding a new type of Injection Point".

Adding style sheet files

Associated Injection Point: `@Injector.InjectionPoint("StyleFiles"[, "group name"])`

Call `Injector.StyleFile()`. It has this method definition:

```
void StyleFile(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 Injection Point with this group name defined.

Example

This code:

```
@{ Injector.StyleFile("~/Content/StyleSheet.css"); }
```

generates this HTML:

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

Adding script blocks

Associated Injection Point: `@Injector.InjectionPoint("ScriptBlocks"[, "group name"])`

Typically you will two injection points where group name = “Upper” for scripts above the HTML and group name = “Lower” for scripts below the HTML that they operate upon.

Call `Injector.ScriptBlock()`. It has these method definitions:

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

void ScriptBlock(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 omit the parameter. 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 `Injector.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”. However, you can use any names you like.

Examples

This code:

```
@{ Injector.ScriptBlock("somekey", "callSomeFunction();", 0, "Lower"); }
```

generates this JavaScript:

```
callSomeFunction();
```

Adding array declarations

Associated Injection Point: `@Injector.InjectionPoint("ScriptBlocks"[, "group name"])`

Array declarations are part of script blocks which is why the Injection Point is the same as Script Blocks. Typically you will two injection points where group name = “Upper” for scripts above the HTML and group name = “Lower” for scripts below the HTML that they operate upon.

Call `Injector.ArrayDeclaration()` or `Injector.ArrayDeclarationAsCode()`.

Array declarations create a variable name assigned to an array. Each call to these methods 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`.

It has several method definitions:

```
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];
```

```
@{
   Injector.ArrayDeclaration("myVar", "my string value", 0, "Lower")
   Injector.ArrayDeclaration("myVar", 100, 0, "Lower")
   Injector.ArrayDeclarationAsCode("MyVar", "null", 0, "Lower")
}
```

Adding meta tags

Associated Injection Point: `@Injector.InjectionPoint("MetaTags"[, "group name"])`

Call `Injector.MetaTag()`. It has this method definition:

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

void MetaTag(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 Injection Point with this group name defined.

Example

This code:

```
@{
   Injector.MetaTag("description", "about my site")
   Injector.MetaTag(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 Injection Point: `@Injector.InjectionPoint("HiddenFields"[, "group name"])`

Call `Injector.HiddenField()`. It has this method definition:

```
void HiddenField(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 Injection Point with this group name defined.

Example

This code:

```
@{ Injector.HiddenField("codes", "AB903F"); }
```

generates this HTML:

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

Adding JavaScript templates

Associated Injection Point: `@Injector.InjectionPoint("TemplateBlocks"[, "group name"])`

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

```
ContentInjector.ContentManagerExtensions.DefaultTemplateType =
   typeof(KnockoutTemplateInjectorItem);
ContentInjector.ContentManagerExtensions.DefaultTemplateType =
   typeof(jQueryTemplateInjectorItem);
ContentInjector.ContentManagerExtensions.DefaultTemplateType =
   typeof(UnderscoreTemplateInjectorItem);
ContentInjector.ContentManagerExtensions.DefaultTemplateType =
   typeof(KendoUiTemplateInjectorItem);
```

Call `Injector.TemplateBlock()`. It has this method definition:

```
void TemplateBlock(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 Injection Point with this group name defined.

Example

This code:

```
@{ Injector.TemplateBlock("mytemplate", "<li><% -item.value %></li>"); }
```

generates this HTML:

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

Adding any other content

Associated Injection Point: `@Injector.InjectionPoint("Placeholder"[, "group name"])`

Call `Injector.Placeholder()`. It has this method definition:

```
void Placeholder(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 Injection Point with this group name defined.

Example

This code:

```
@{ Injector.Placeholder("<!-- xyz library used under license -->"); }
```

generates this HTML:

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

Expanding this framework

Content Injector 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 Injection Point

You will be creating three new types:

- Interface based on `ContentInjector.IInjector`.
- Class based on `ContentInjector.IKeyedInjectorItem`.
- Class based on `ContentInjector.BaseKeyedInjector` that implements the above interface

Interface based on IInjector

All Injection Point classes implement the `ContentInjector.IInjector` interface. You should create a new interface definition based on `IInjector`. It will be used by the `InjectorFactory` to map to your class.

There is a recommended naming convention. “I” + *the desired name* + "Injector".

Class based on IKeyedInjectorItem

Create a class to hold the value of each item added by implementing `ContentInjector.IKeyedInjectorItem`.

There is a recommended naming convention: *the desired name* + “InjectorItem”.

Your implementation should have the following:

- Add properties that define the values to store.
- Determine which string-type property represents a unique key when in a list of these objects owned by your new Injector class. Implement the methods `GetKey()` and `SetKey()` to map to this value.
- Implement the `Merge()` method to determine what to do when a duplicate key is added by the user. The original object will always be retained, never replaced. Your `Merge()` method can either ignore the value of the new object or transfer its values into the original. For example, HiddenFieldInjectorItem replaces the Value property while ScriptBlockInjectorItem ignores the new object’s script.
- Implement the `GetContent()` method to return a string that is injected into the page representing the data.

Class based on BaseKeyedInjector

Create a class that inherits from `ContentInjector.BaseKeyedInjector` which holds a list of the items added, with a key field that is used to locate an existing item.

There is a recommended naming convention: *the desired name* + “Injector”.

`BaseKeyedInjector` takes your IKeyedInjectorItem implemenation for its generic value. Also include the interface definition described at the top of this section.

```
public MyClassInjector : BaseKeyedInjector<MyClassInjectorItem>, IMyClassInjector
```

Expand your interface to have an `Add()` method that takes two parameters, the InjectorItem class and order.

```
void Add(MyClassInjector item, int order = 0);
```

Exposing your new types

When finished, register your new Injector class with its interface in the `ContentInjector.InjectorFactory` in `Application_Start()` like this:

```
ContentInjector.InjectorFactory.Default.Register(
   typeof(interface type), typeof(class type));
```

Add extension methods to the `ContentInjector.ContentManager` class that call the `Add()` method. Your extensions class should be in the namespace ContentInjector, or if you use an alternative namespace, it must be registered in **View\web.config** in the `<namespaces>` section of `<system.web.mvc.razor>`.

```
public static class Extensions
{
   public static void functionname(this ContentManager contentManager,
      string value, int order = 0, string group = "")
   {
      contentManager.Access<IMyClassInjector>(group).Add(
         new MyClassInjectorItem(value), order);
   }
}
```

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

Expanding ScriptBlockInjector

The ScriptBlockInjector already knows how to collect scripts in two formats: raw (using `Injector.ScriptBlock`) and array declaration (using `Injector.ArrayDeclaration`). ArrayDeclaration is a script generator, converting a list of calls into a single script. You can add support for other script generates. For example, creating a jquery-ui widget's registeration, which looks like this:

```
$("#id").widgetname({options});
```

Let's use this example to show how to add support for this new generator. Here is the big picture:

- Create a class that implements `ContentInjector.IScriptBlockInjectorItem`.
- Create an extension method on the `ContentInjector.ContentManager` class to add your new class to the `ScriptBlockInjector` class instead already on the ContentManager.

```
using ContentInjector;
…
public class jQueryWidgetInjectorItem : IScriptBlockInjectorItem
{
   public jQueryWidgetInjectorItem(string selector, string widgetName)
   {
      if (String.IsNullOrEmpty(selector) || String.IsNullOrEmpty(widgetName))
         throw new ArgumentNullException();

      jQuerySelector = selector;
      WidgetName = widgetName;
      Options = new Dictionary<string, string>();
   }

/// <summary>
/// Passed to jquery as a selector. If it specifies a specific element
/// by ID, it should be "#id".
/// </summary>
   public string jQuerySelector { get; protected set; }

/// <summary>
/// The name of the extension or widget on the jquery object.
/// Case sensitive.
/// </summary>
   public string WidgetName { get; protected set; }

/// <summary>
/// The options that will appear in JSON format as the parameter to the widget's function.
/// Each key is the property name supported by that widget. Each value is the exact
/// string to be output into JSON. It has already been cleaned up using
/// the ValuesInJavaScript.ToString() function.
/// </summary>
   protected Dictionary<string, string> Options { get; set; }

#region IKeyedInjectorItem Members


   public string GetKey()
   {
      return jQuerySelector + "|" + WidgetName;
   }

   public void SetKey(string key)
   {
      String[] parts = key.Split('|');
      if (parts.Length != 2)
         throw new ArgumentException("Must be pipe delimited with two parts:" +
            " jquery selector and widget name: \"#TextBox1|validate\".");
      jQuerySelector = parts[0];
      WidgetName = parts[1];
   }
```

```
    public bool Merge(IKeyedInjectorItem item)
    {
// adds or replaces options found in items
        foreach (var option in ((jQueryWidgetInjectorItem)item).Options)
        {
            this.Options[option.Key] = option.Value;
        }
        return true;
    }

    public string GetContent(System.Web.HttpContextBase httpContext)
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("$(\"");
        sb.AppendFormat("$('{0}').{1}(", jQuerySelector, WidgetName);

        bool first = true;
        foreach (var option in Options)
        {
            if (!first)
                sb.Append(", ");
            sb.AppendFormat("'{0}': {1}", option.Key, option.Value);

            first = true;
        }
        sb.Append(");");
        return sb.ToString();
    }

    public void AddOption(string optionName, object optionValue)
    { // ValuesInJavaScript is supplied within the ContentInjector namespace
        this.Options[optionName] = ValuesInJavaScript.ToScript(optionValue);
    }

#endregion
}
```

For our extension method:

```
public static class Extensions
{
    public static void jQueryWidget(this ContentManager contentManager,
        string selector, string widgetName, Dictionary<string, object> options,
        int order = 0, string groupName = "")
    {
        IScriptBlocksInjector injector =
            contentManager.Access<IScriptBlocksInjector>(groupName);
        jQueryWidgetInjectorItem item = new jQueryWidgetInjectorItem(selector, widgetName);
        if (options != null)
            foreach (var option in options)
                item.AddOption(option.Key, option.Value);
        injector.Add(item, order);
    }
}
```

Now users will have this available on the View's `Injector` property.

```
@{ Injector.jQueryWidget("#TextBox1", "datePicker", null, 1000, "Lower"); }
```

You may want the defaults for *order* and *groupName* to differ, such as always positioning later in the "Lower" group. `int order = 1000, string groupName = "Lower"`

```
@{ Injector.jQueryWidget("#TextBox1", "datePicker", null); }
```

Using your own WebViewPage subclass

Edit your System.Web.Mvc.WebViewPage subclasses to expose the **Injector** property, which is an instance of the ContentInjector.ContentManager that is hosted in the ViewData as “ContentManager”.

```
public virtual ContentInjector.ContentManager Injector
{
   get
   {
      if (_injector == null)
      {
         _injector = ContentInjector.CIRazorViewExtensions.ContentManager(this);
      }
      return _injector;
   }
}
private ContentInjector.ContentManager _injector;
```

Note: You probably have at least two WebViewPage classes, with one associated with <TModel>. All need to be updated.

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 contentManager = new ContentInjector.ContentManager(
       writer, viewContext.HttpContext, ContentInjector.InjectorFactory.Default))
   {
      viewContext.ViewData["ContentManager"] = contentManager;

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

      contentManager.UpdatePage();
   }
}
```

Replacing string templates

Most Injection 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:

`ScriptFileInjectorItem.DefaultScriptFileTagFormat`

`StyleFileInjectorItem.DefaultStyleFileTagFormat`

`HiddenFieldInjectorItem.DefaultHiddenFieldFormat`

`MetaTagInjectorItem.DefaultMetaTagFormat`

`BaseTemplateInjectorItem.DefaultTemplateTagFormat`

`ScriptBlocksInjector.StartScriptBlockTag`

`ScriptBlocksInjector.EndScriptBlockTag`