ScriptWriter 3

ScriptWriter 3 is an open-source, CFC-based module designed to allow web developers to programmatically manage the inclusion of JavaScript and CSS.

ScriptWriter 3 supports dynamic external and inline inclusion, runtime minification using the YUI and JSMin libraries, runtime file merging for HTTP request reduction, inline to external conversion, and flexible grouping and output control for easy implementation of page load performance best practices.

This document is current for version 3.5

Document by Matt Gersting

Last Updated 11/22/2010

Basic Setup

1. Download the latest version of the ScriptWriter CFCs at RIAForge.org or from our Google Code repository and place them in your preferred CFC location.

2. If you want to use minification, install the JAR files in your ColdFusion classpath and restart the JVM. Support for JavaLoader is coming soon. A copy of the JSMin and YUI jar files are included in the "lib" folder of the ScriptWriter download package.

3. Setup "ScriptWriter" in your global (via the ColdFusion Administrator) or application-level mappings.

4. If you're going to use minification, set up the wrapper CFCs.

5. Create the Factory object
   ScriptWriter 3 is meant to be used as a singleton, so the best place to create your factory object (generally) is in the onApplicationStart() method of your Application.cfc:

```
<cfset application.ScriptWriter =
CreateObject("component","org.mgersting.scriptwriter.core.Factory").init()/>
```

Parameter	**Values**	**Required**	**Description**
writeMode	String; "lazy" or "eager"	No	See notes below.
javascriptCompressor	String; dot-delimited CFC path	No	Sets the wrapper CFC for the JavaScript compressor. Defaults to org.mgersting.scriptwriter.core.JSMin
cssCompressor	String; dot-delimited CFC path	No	Sets the wrapper CFC for the CSS compressor. Defaults to org.mgersting.scriptwriter.core.YUICssCompressor
lessEngine	String; dot-delimited CFC path	No	Sets the wrapper CFC for the LESS CSS engine. Defaults to org.mgersting.scriptwriter.core.AsualLessEngine
dataTimeout	Numeric;	No	This is the minimum time in seconds that must past before ScriptWriter will attempt

			to cleanup stranded or outdated request data. Defaults to 60.
errorMode	String; “silent”, “error”, or “comment”	No	Defines how internal errors are handled. Defaults to “silent.”

WHAT IS WRITEMODE?

WriteMode takes two possible values – “lazy” or “eager”. Lazy is best for production environments – it limits the number of file writes ScriptWriter does. Eager mode is better suited for dev environments where files are changing regularly. It forces ScriptWriter to output minified or merged files fresh for every request. This is NOT recommended for production environments, especially under load. Setting the write mode is optional and defaults to “lazy”.

IMPORTANT CHANGE STARTING IN VERSION 3.5

Prior to version 3.5, ScriptWriter required that the developer initialize objects for the CSS and JS minifiers. SW3 now does this automatically, so unless you want to use custom classes for this the arguments can be left off altogether.

6. Initialize each request.
   Because ScriptWriter runs as an application singleton, it manages the script and style references for all incoming requests. This means each request requires a unique identifier to talk back and forth with ScriptWriter. The simplest way to do this is using the onRequestStart() method of the Application.cfc in tandem with ColdFusion’s built-in CreateUUID() method:

```
<!-- Create a unique ID -->
<cfset request.sRequestID = CreateUUID() />

<!-- Use the unique ID to start “recording” items associated with this request -->
<cfset application.ScriptWriter.openRequest(request=Request.sRequestID) />
```

7. Close each request.
   Also because ScriptWriter is running as a singleton it’s important to close each request. This allows ScriptWriter to clear out unnecessary data and minimize its memory footprint. It’s suggested that you do this once, globally for each request, in the onRequestEnd() method of Application.cfc:

```
<!-- Close the request -->
<cfset application.ScriptWriter.closeRequest(request=Request.sRequestID) />
```

Adding and Outputting Your First Script

Here we are going to add a single external JavaScript file, then instruct ScriptWriter to output the relevant markup.

Our sample code is setting the region to “head”, but you can use any name you like, provided the output code in step 2 is updated to match.

1. In your controller code, use the add() method to add an item to your request. Note that we are setting request equal to the ID that was created in the onRequestStart() method.:

```
<cfset application.ScriptWriter.add(
     request=Request.sRequestID,
     type=”script”,
     region=”head”,
     path=”path/to/file.js”
)/>
```

2. In your display code, call the output method where you want the data written:

```
<cfoutput>
     #application.ScriptWriter.output(
          request=Request.sRequestID,
          type=”script”,
          region=”head”
     )#
</cfoutput>
```

If all went well your display file should now be showing:

```
<script type=”text/javascript” src=”path/to/file.js”></script>
```

OVERRIDING THE DEFAULT WRITEMODE BEHAVIOR:
We previously noted that you can set the write mode in the ScriptWriter factory. This will in turn act as the default setting for all scripts and styles that you add. However, you can override the default behavior on individual items as you see fit. Just add the writeMode argument to the add() statement.

STYLES ARE JUST AS EASY:
Adding a style is basically identical to adding a script. Only two things change. First, the type parameter is set to “style” instead of “script.” Lastly, there is an additional optional parameter available – media. By default, media is set to “screen”.

Minifying Files

Now we’re going to take nearly the same code with the same external JavaScript file and we’re going to tell it to minify the file.

1. Again, in your code, use the add() method, but this time will set some additional arguments: minify, which we will set to true, and outputPath, which tell ScriptWriter where the new minified file should be saved to:

```
<cfset application.ScriptWriter.add(
     request=request.sRequestID,
     type=”script”,
     region=”head”,
     path=”path/to/file.js”,
     minify=true,
     outputPath=”path/to/file.min.js”
)/>
```

Our output code is identical to the first example. ScriptWriter will automatically detect that the item has been successfully minified and will update the outputted path to new, minified file.

Merging Files

OK, so we've accomplished adding and minifying files. Now we're going to take it one step further. First, we're going to add two files to the head of our document. As we do this, we're going to minify them and then merge them into a single file.

1. Add two files:

```
<cfset application.ScriptWriter.add(
     request=request.sRequestID,
     type=”script”,
     region=”head”,
     path=”path/to/file1.js”,
     minify=true
)/>

<cfset application.ScriptWriter.add(
     request=request.sRequestID,
     type=”script”,
     region=”head”,
     path=”path/to/file2.js”,
     minify=true
)/>
```

2. In your controller, apply the combine() command to the group.

```
<cfset application.ScriptWriter.combine(
     request=request.sRequestID,
     type="script",
     region="head",
     outputPath="path/to/combo.min.js"
)/>
```

Once again, our output code is going to be identical to the previous examples. ScriptWriter automatically keeps track of our file mergers and will only output the results.

IMPORTANT PERFORMANCE NOTE:

Please notice that in the previous example for using the combine() method, that the two add() calls do not provide an outputPath argument. This means that ScriptWriter will not attempt to do any filesystem writes for those two add() calls. If you weren't combining them this would end up being problematic because ScriptWriter would have

no file to actually link to when you did your output() statement, but because you know you're going to combine them later on it's preferred - ScriptWriter will use the outputPath of the combine() action and in turn you will save resources and increase performance.

MERGING STYLES AND CONTROLLING THEIR MEDIA:
When merging styles there is an additional optional parameter available to the combine() method: media. By default ScriptWriter applies a value of "screen", but you can control this by passing your custom value via this argument.

Using LESS CSS

Perhaps you've heard all the buzz about the LESS CSS system? If you haven't, you may want to check out http://lesscss.org and learn about CSS with the power of variables, mix-ins, etc. If you're already using LESS CSS you can now use it with ScriptWriter 3 (as of version 3.5). And, to make things even better, you don't incur the overhead of the extra client-side LESS CSS processing. ScriptWriter 3 includes the Asual LESS CSS Java engine and processes the files at run-time. Just add() an element of type "style" and if it has .less extension it will be processed accordingly.

Advanced: Regions and Groups

Why Regions?

Regions are overall output areas. Generally speaking, most styles and scripts will be placed in one of three locations: inside the document <head>, just inside the document <body>, or immediately before the close of the document </body>.

However, rather than force all to conform to a single model, ScriptWriter makes it free form so you can name and output your own regions wherever in the document you want.

Why Groups?

Groups are subdivisions of regions. Each region can have *n* number of groups inside of it. Why do regions need to be subdivided into groups? Well, it's entirely possible that they won't, but they exist to ensure maximum compatibility *and* maximum page load performance. How?

First, we want to maximize page load performance. One of the major way we accomplish this is by reducing the number of HTTP requests required per page. ScriptWriter 3 helps us do this by using its combine() and minify() methods. However, you should never be forced to merge or minify *all* of your scripts/styles unless you explicitly choose too – you're in charge.

That's where groups come in: Groups allow you say "everything in this region goes here, but this group of items will be minified/merged and this group of items (in the same region) will not."

Advanced: Inline Elements

Why Inline Elements?

General consensus is that inline elements – be they CSS or JavaScript – are bad. And there's good reason for this: it blurs the line between data and presentation, bloats HTML files, isn't generally cacheable, makes maintenance more difficult, etc. etc.

However, as we know in the web, there is an exception to every rule, and sometimes inline code is simply unavoidable. Particularly I've found when dealing with content management systems and CSS that is heavy on background images.

No matter your particular reason, ScriptWriter is here to help in several ways. First, it provides you with a clean, standard way to include inline elements in your page, and second, it actually provides the ability – should you so choose – to take your dynamic, inline code and output it as an external file. Let's look at some samples:

Adding and Outputting Inline Elements

First, set up the inline code using a <cfsavecontent> block. We're going to keep it short and simple, adding a background color to a class.

```
<cfsavecontent variable="variables.sInlineContent">
     .sampleClass {
          background-color: red;
     }
</cfsavecontent>
```

Just like the previous examples we will be using the add() method. However, rather than providing the sourcePath argument, will provide the src argument instead:

```
<cfset application.ScriptWriter.add(
     request=request.sRequestID,
     type="style",
     region="head",
     src=variables.sInlineContent
)/>
```

The output statement first the very example still works here, with the exception that you'll want to tell it to output type "style" rather the "script":

```
<cfoutput>
     #application.ScriptWriter.output(
          request=request.sRequestID,
          type="style",
          region="head"
     )#
</cfoutput>
```

Adding an Element Inline, Outputting as External

The previous example showed how we can easily add inline code to our documents. But, if you want, you can actually have ScriptWriter take your dynamic code and save the results out to an external file, thus allowing it to be cached, and perhaps saving your soul a little bit of pain from having inline code in your documents. How do we do this? It's simple: We just provide the same add() method with an outputPath. ScriptWriter will automatically generate the file and the output method will automatically link to it rather than output the inline code.

```
<cfset Application.ScriptWriter.add(
     request=request.sRequestID,
     type="style",
     region="head",
     src=variables.sInlineContent,
     outputPath="styles/inline.sample.css"
)/>
```

And in case you were wondering – yes, you can actually have ScriptWriter minify your inline code before it saves it to disk. Simple add the minify=true argument and you're good to go.

Advanced: Using Remote Libraries

Why Remote Libraries?

Performance! Using remotely hosted JavaScript or CSS libraries can increase your site's performance with improved caching and asynchronous loading. So, how does ScriptWriter deal with these? It's quite simple actually. Simply add the item with ScriptWriter as you have been, but with a couple caveats:

- Do not set minify to true for the remote item
- Do not put the remote in a group that is going to be combined later
- Do not provide an output path for the remote item

Here's an example that would work will all of our previous sample code:

```
<cfset application.ScriptWriter.add(
     request=request.sRequestID,
     type="script",
     region="head",
     group="remote",
     path="http://www.sample.com/script.js"
)/>
```

Advanced: Templates - Passing Data to Assets

Want to create JavaScript template that you can update at runtime? It's easy to accomplish with ScriptWriter. Simply include the `keyDelimiter` and `keys` arguments when you use the `add()` method. First, let's take a look at a very simple sample CSS file:

```
@charset "utf-8";
div#header { display: block; }
```

This CSS file is minimal. We've got our charset definition and a single CSS selector. So, let's say that the value of the div ID we're selecting needs to be dynamic. To accomplish this using ScriptWriter first you must decide what format your variables will take. For the sake of this example, our variable format will be the dollar sign followed by a variable name, ala: `$myvar`. Let's update our CSS file:

```
@charset "utf-8";
div#$myvar { display: block; }
```

OK. Our CSS file is ready to have data sent to. Now, when we add this element to ScriptWriter, we add two new arguments to our add() method – keyDelimiter and keys – which you can see in the code snippet below.

```
<cfset variables.stKeysIn = {} />
<cfset variables.stKeysIn.myvar = "header" />

<cfset application.ScriptWriter.add(
     request=request.sRequestID,
     type="script",
     region="head",
     path="path/to/file1.css",
     keyDelimiter="\$"
     keys=variables.stKeysIn
)/>
```

When our code has completed running, you will now see that the file at `path/to/file1.css` has been augmented with the data from `variables.stKeysIn`.

IMPORTANT NOTE ABOUT KEY DELIMITERS AND REGULAR EXPRESSIONS:
Please note backslash in the previous keyDelimiter argument. ScriptWriter uses regular expressions to match and replace your variable placeholders with their corresponding values, so if your keyDelimiter is a special character in regular expression you'll need to escape it like we've done here.

Using the clear() method

From time to time you will find nooks and crannies of your application that diverge from the norm and require significantly different CSS or JavaScript support. Many times in these instances it's easier to start clean and fresh. ScriptWriter provides several methods to accomplish this. The first of which is the clear() method.

The clear() method takes up to four arguments (request, type, region, and group) each one adding a level of specificity to your removal.

Clearing everything from the current request stack:

```
<cfset application.ScriptWriter.clear(
     request=request.sRequestID
)/>
```

Clear all scripts or all stylesheets:

```
<cfset application.ScriptWriter.clear(
     request=request.sRequestID,
     type=”script”
)/>


<cfset application.ScriptWriter.clear(
     request=request.sRequestID,
     type=”style”
)/>
```

Clear a specific type and region:

```
<cfset application.ScriptWriter.clear(
     request=Request.sRequestID,
     type=”style”,
     region=”regionName”
)/>
```

Clear a specific group inside of a type and region:

```
<cfset application.ScriptWriter.clear(
     request=Request.sRequestID,
     type=”style”,
     region=”regionName”,
     group=”groupName”
)/>
```

License / About / Credits

ScriptWriter is made available under the GNU Public License. Please share and improve.

ScriptWriter was originally written by Matt Gersting in the fall of 2005 for the internal ColdFusion development team at Full Sail University. It was originally non-CFC based and before too long was rewritten as version 2, which was CFC-based and introduced a number of features, specifically redundant and missing file handling. Version 2 was used successfully for several years across many products and under many traffic scenarios.

In late 2009 and early 2010, the Marketing subdivision of Full Sail's development team began looking into ways to maximize the performance and scaling of their products, and among several efforts ScriptWriter 3 was devised. The goal was to improve page load performance by allowing more flexible output points (regions and groups), reducing HTTP requests (using the combine() method), and decrease total file size (minification). Additionally, the decision was made to attempt to move to a singleton-based model to reduce the number of objects being created in high traffic environments.

Special thanks to Adam Bellas, Matt Terrill, and Kat Bennett for their input. Thanks also to Tom de Manicor and Joe Roberts whose YUICompressor CFC and Combine.cfc code have informed this project extensively. Thanks to Dom Watson for help wrangling the LESS CSS Java and whose cfStatic code inspired both the move to JavaLoader and the inclusion (feature battle!) of LESS CSS processing.