

## Contents

- [CFSCRIPT Enhancements](#)
- [Components and Interfaces](#)
- [Properties](#)
- [Functions](#)
- [Tag Equivalents](#)
- [abort](#)
- [admin](#)
- [cookie](#)
- [execute](#)
- [exit](#)
- [feed](#)
- [finally](#)
- [flush](#)
- [ftp](#)
- [http](#)
- [import](#)
- [include](#)
- [location](#)
- [lock](#)
- [logout](#)
- [mail](#)
- [pageencoding](#)
- [param](#)
- [pdf](#)
- [query](#)
- [rethrow](#)
- [savecontent](#)
- [setting](#)
- [thread](#)
- [throw](#)
- [transaction](#)
- [writeDump](#)
- [writeLog](#)

## CFSCRIPT Enhancements

Railo 3.2 features a large number of enhancements to the CFSCRIPT language so that CFCs can be written entirely in CFSCRIPT. Prior to release 3.2, Railo allowed the body of a CFCOMPONENT to be written in CFSCRIPT with function arguments having types and defaults declared, as long as individual functions didn't use tags for which there was no CFSCRIPT equivalent. Those restrictions have been almost completely lifted in release 3.2 with only HTML-generating tags no longer having CFSCRIPT enhancements.

### Components and Interfaces

Both components and interfaces can be declared in CFSCRIPT now. Surrounding CFSCRIPT tags are not needed for such declarations:

```
component extends="my.base.Class" {
...
}
```

Any attributes supported by CFCOMPONENT can be specified between the **component** keyword and the opening `{`. In particular, **accessors="true"** tells Railo to automatically generate setters and getters for properties declared in the component (see next section). Any unrecognized attributes will be added to the metadata for the component.

Note that **output=** is not required since CFSCRIPT generates no output by default.

```
interface hint="ICollection is a general collection class interface" {
...
}

component implements="ICollection" hint="Set provides a collection class with unordered set semantics" }
```

```
...
}
```

## Properties

The CFPROPERTY tag has three representations in CFSCRIPT:

```
property thing; // declares a property called "thing" with type "any"
property string username; // declares a property called "username" with type "string"
property name="email" type="string" hint="A property called email with type string and a documentation hint";
```

Any attributes supported by CFPROPERTY can be specified in the third form of **property** declaration, including ORM-specific relationship and type hinting metadata. Any unrecognized attributes are added to the metadata for the property.

If **accessors="true"** is specified on the **component** declaration or **getter="true"** is specified on the **property** declaration, Railo will automatically generate a getter function as if you had written:

```
function getSomeProperty() {
return variables.someProperty;
}
```

Providing your own getter function explicitly will suppress the generated default.

If **accessors="true"** is specified on the **component** declaration or **setter="true"** is specified on the **property** declaration, Railo will automatically generate a setter function as if you had written:

```
function setSomeProperty( any someProperty ) {
variables.someProperty = someProperty;
}
```

Providing your own setter function explicitly will suppress the generated default.

## Functions

In addition to argument types and defaults, Railo 3.2 now support function access and return type declarations in CFSCRIPT, as well as additional metadata:

```
public string function someFunction( string username = "admin" ) hint="A public function that returns a string" {
...
}

private numeric function howMany( any collection ) hint="A private function that returns a number" {
...
}
```

Any attributes supported by CFFUNCTION can appear between the closing `)` of the argument list and the opening `{` of the function body. Note that **output="true"** is not required as CFSCRIPT generates no output by default. Any unrecognized attributes are added to the metadata for the function.

If the access specifier is omitted, it will default to **public**. If the return type is omitted, it will default to **any**. Arguments can be declared as **required** to indicate they must be supplied by the caller.

## Tag Equivalents

A large number of tags now have CFSCRIPT equivalents.

### abort

The following equivalents to the CFABORT tag are supported:

```
abort; // equivalent to
abort "I failed!"; // equivalent to
```

### admin

The following equivalent to the CFADMIN tag is supported:

```
admin action="getDebugData" returnvariable="data";
writeDump(data);
```

The **admin** statement supports the same attributes and usage as the CFADMIN tag.

Added in 3.2.0.004.

### cookie

The following equivalent to the CFCOOKIE tag is supported:

```
cookie name="testcookie" value="I was set by script.";
// equivalent to
```

The **cookie** statement supports the same attributes and usage as the CFCOOKIE tag.

### execute

The following equivalent to the CFEXECUTE tag is supported:

```
execute name="pwd" timeout="10" variable="workingDir";
execute name="ls" arguments="-l #workingDir#" timeout="10" variable="home";
writeOutput( '
' & home & '
' );
```

The **execute** statement supports the same attributes and usage as the CFEXECUTE tag.

### exit

The following equivalents to the CFEXIT tag are supported:

```
exit; // equivalent to
exit "exittag"; // equivalent to
exit "exittemplate"; // equivalent to
exit "loop"; // equivalent to
```

### feed

The following equivalent to the CFFEED tag is supported:

```
feed source="http://blog.getrailo.com/feeds/rss.cfm" name="railoRSS";
writeDump( railoRSS );
```

The **feed** statement supports the same attributes and usage as the CFFEED tag.

See also the auto-imported [Feed](#) CFC.

### finally

**finally** is an addition to the **try/catch** statement that declares code that should always be executed, regardless of whether an exception occurred or not. It is the equivalent of the CFFINALLY tag:

```
try {
writeOutput( "try this!" );
if ( structKeyExists( URL, "fail" ) ) throw "failure";
} catch (any e) {
writeOutput( "I failed!" );
rethrow;
} finally {
writeOutput( "I am always executed!" );
}
```

Regardless of whether an exception occurs or not, "I am always executed!" will be output -- even tho' the **catch** clause rethrows the exception!

## flush

The following equivalents to the CFFLUSH tag are supported:

```
flush; // equivalent to
flush 100; // equivalent to
```

## ftp

The following equivalent to the CFFTP tag is supported:

```
ftp action="open" connection="moz" directory="/pub/mozilla.org/"
server="ftp.mozilla.org" username="anonymous" password="sean@getrailo.com";
ftp action="listdir" directory="/pub/mozilla.org/"
connection="moz" name="mozDirectory";
ftp action="close" connection="moz";
writeDump( mozDirectory );
```

The **ftp** statement supports the same attributes and usage as the CFFTP tag.

See also the auto-imported [FTP](#) CFC.

## http

The following equivalents to the CFHTTP and CFHTTPPARAM tags are supported:

```
http url="http://ws.geonames.org/earthquakesJSON" method="post" result="quakes" {
httpparam type="formfield" name="north" value="44.1";
httpparam type="formfield" name="south" value="-9.9";
httpparam type="formfield" name="east" value="-22.4";
httpparam type="formfield" name="west" value="55.2";
};
writeDump( deserializeJSON( quakes.fileContent ) );
```

The **http** statement supports the same attributes and usage as the CFHTTP tag. The **httpparam** statement supports the same attributes and usage as the CFHTTPPARAM tag.

See also the auto-imported [HTTP](#) CFC.

## import

This is not a direct equivalent to the CFIMPORT but instead allows CFCs to be *imported* and referred to by their unqualified name:

```
import path.to.MyCFC;
obj = new MyCFC(); // can use unqualified name because it was imported;
import some.folder.*; // imports all the CFCs in /some/folder
```

The path may optionally be enclosed in quotes, i.e., it may be a string:

```
import "path.to.MyCFC";
import "some.folder.*"; // imports all the CFCs in /some/folder
```

## include

The following equivalent to the CFINCLUDE tag is supported:

```
include "path/to/included/file.cfm";
```

This is equivalent to:

```
template="path/to/included/file.cfm" />
```

## location

The following equivalents to the CFLOCATION tag are supported:

```
location("http://getrailo.com"); // url
location("http://getrailo.org", false); // url, addtoken
location( url = "http://wiki.getrailo.org" );
location( url = "http://groups.google.com/group/railo", addtoken = false );
```

## lock

The following equivalent to the CFLOCK tag is supported:

```
lock name="my_lock" type="exclusive" timeout="30" {
...
}
```

The **lock** statement supports the same attributes and usage as the CFLOCK tag.

## logout

The following equivalent to the CFLOGOUT tag is supported:

```
logout; // equivalent to
```

## mail

The following equivalents to the CFMAIL, CFMAILPARAM and CFMAILPART tags are supported:

```
mail subject="Test Email" from="sean@getrailo.com" to="sean@getrailo.org" server="localhost" {
mailparam name="X-Railo" value="Sent from cfscript";
mailpart type="html" {
  writeOutput( '

```

## Hi Sean!

```
' );
  writeOutput( '
This is an HTML email.
' );
};
mailpart type="text" wraptext="72" {
  writeOutput( 'Hi Sean!' &chr(13) );
  writeOutput( 'This is a plain text alternative email.' &chr(13) );
};
};
```

The **mail** statement supports the same attributes and usage as the CFMAIL tag. The **mailparam** statement supports the same attributes and usage as the CFMAILPARAM tag. The **mailpart** statement supports the same attributes and usage as the CFMAILPART tag.

The body of the **mail** and **mailpart** statements can contain any script code. It will be executed. Any output generated, e.g., via **writeOutput()**, becomes the text of the mail.

See also the auto-imported [Mail](#) CFC.

## pageencoding

The character encoding of a component may be set using the **pageencoding** directive as follows:

```
component {
pageencoding 'iso-8859-1';
...
}
```

The **pageencoding** directive should appear at the top of the component body, as shown in this example.

The **pageencoding** directive is equivalent to this form of the CFPROCESSINGDIRECTIVE:

```
output="false">
  pageencoding="iso-8859-1" />
```

...

**param**

The following equivalents to the CFPARAM tag are supported:

```
param name="iAmRequired";
param name="iMustBeNumeric" type="numeric";
param name="iHaveADefault" default="42";
```

The **param** statement supports the same attributes and usage as the CFPARAM tag.

**pdf**

The following equivalent to the CFPDF tag is supported:

```
pdf action="getinfo" name="info" source="/Users/sean/Documents/ebooks/couchdb.pdf";
writeDump( info );
pdf action="thumbnail" source="/Users/sean/Documents/ebooks/couchdb.pdf" destination="/Users/sean/Documents/pdf/" imagePrefix="couch";
files = directoryList( "/Users/sean/Documents/pdf/" );
writeDump( files );
```

The **pdf** statement supports the same attributes and usage as the CFPDF tag.

**query**

The following equivalent to CFQUERY and CFQUERYPARAM tags are supported:

```
query datasource="test" name="users" {
writeOutput( 'SELECT * FROM user WHERE id = ' );
queryparam value="#userId#" cfsqltype="cf_sql_integer";
}
writeDump( users );
```

The **query** statement supports the same attributes and usage as the CFQUERY tag. The **queryparam** statement supports the same attributes and usage as the CFQUERYPARAM tag.

The body of the **query** statement can contain any script code. It will be executed. Any output generated, e.g., via **writeOutput()**, becomes the text of the SQL in the query.

See also the auto-imported [Query](#) CFC.

**rethrow**

The following equivalent to the CFRETHROW tag is supported:

```
rethrow;
```

**savecontent**

The following equivalent to the CFSAVECONTENT tag is supported:

```
savecontent variable="stuff" {
writeOutput("This is saved to stuff");
x = "This is just executed (and not saved to stuff)";
if ( someCondition ) {
writeOutput(" appended because someCondition was true");
}
}
```

This is equivalent to:

```
enablecfoutputonly="true"/>
variable="stuff">
This is saved to stuff
x = "This is just executed (and not saved to stuff)" />
someCondition>
appended because someCondition was true
```

Note that you need the CFSETTING tag to restrict output to just the text within CFWRITEONLY

tags. The following would generate extra whitespace:

```
variable="stuff">
This is saved to stuff
  x = "This is just executed (and not saved to stuff)" />
  someCondition>
  appended because someCondition was true
```

## setting

The following equivalent to the CFSETTING tag is supported:

```
setting enableCFoutputOnly="Any" showDebugOutput="Boolean" requestTimeOut="number";
```

## thread

The following equivalents to the CFTHREAD tag are supported:

```
thread name="t1" {
sleep( 1000 );
thread.x = 42;
}
threadJoin( 't1' );
writeDump( t1 );
```

The **thread** statement supports the same attributes and usage as the CFTHREAD tag. The **action** attribute defaults to **"run"** and for that action a thread body may be provided as shown enclosed in { braces }. The **thread** statement may also be used to join, terminate or sleep threads - or you may use the functions shown below.

```
thread action="join" name="t1,t2";
thread action="terminate" name="t3";
thread action="sleep" duration="1000";
```

The **threadJoin** function is equivalent to the CFTHREAD tag or **thread** statement with **action="join"** and supports two arguments:

- name - comma-separated list of thread names to wait for
- timeout - optional time in milliseconds to wait for the named threads to complete, before throwing a timeout exception

There is also a **threadTerminate** function that is equivalent to the CFTHREAD tag or **thread** statement with **action="terminate"** and supports one argument:

- name - comma-separated list of thread names to terminate

In addition, the **sleep** function shown in the example is equivalent to the CFTHREAD tag or **thread** statement with **action="sleep"** and supports one argument:

- duration - the number of milliseconds to sleep for

## throw

The following equivalents to the CFTHROW tag are supported:

```
throw "Oops!"; // equivalent to
throw("Oops!"); // also equivalent to
throw("Fail","What"); // equivalent to
throw("Fail","What","Specific"); // equivalent to
throw("Fail","What","Specific","Code"); // equivalent to
throw("Fail","What","Specific","Code","More Details"); // equivalent to
```

In addition, named arguments may be used to specify any or all of those attribute equivalents. For example:

```
throw( type = "Foo", message = "What" );
throw( type = "Bar", message = "Fail!", detail = "Why we failed." );
```

## transaction

The following equivalent to the CFTRANSACTION tag is supported:

```
transaction {
...
}
```

This is the simplest form of transaction and is equivalent to:

```
...
```

The following attributes are supported:

- action - "begin", "commit", "rollback"
- isolation - "read\_uncommitted", "read\_committed", "repeatable\_read"

```
transaction isolation="repeatable_read" {
...
  if ( ... ) transaction action="commit";
...
}
```

The following functions also exist:

- transactionCommit(); - equivalent to the CFTRANSACTION tag and **transaction** statement with **action="commit"**
- transactionRollback(); - equivalent to the CFTRANSACTION tag and **transaction** statement with **action="rollback"**

Savepoint functionality is not currently supported.

As of 3.2.0.004, nested transactions no longer cause an exception. Nested transactions with actions "none" or "begin" are simply ignored. Commit and rollback apply to the outermost enclosing transaction. This allows use of code that already uses transactions internally as part of a larger block of code that needs to aggregate several operations into a single overriding transaction.

## writeDump

The following equivalent to the CFDUMP tag is supported:

```
writeDump( someVar ); // equivalent to
writeDump( someVar, false ); // equivalent to
```

The available arguments, all optional except for **var**, are as follows:

- object var, boolean expand, string format, string hide, numeric keys, string label, boolean metainfo, string output, string show, boolean showUDFs, numeric top, boolean abort

So the **writeDump** function supports the same arguments and usage as the CFDUMP tag.

In addition, named arguments may be used to specify any or all of those attribute equivalents. For example:

```
writeDump( var = someVar, expand = false );
```

## writeLog

The following equivalent to the CFLOG tag is supported:

```
writeLog( "log this!" ); // equivalent to
writeLog( "log this!", "error" ); // equivalent to
```

The available arguments, all optional except for **text**, are as follows:

- string text, string type, boolean application, string file, string log

So the **writeLog** function supports the same arguments and usage as the CFLOG tag.

In addition, named arguments may be used to specify any or all of those attribute equivalents.  
For example:

```
writeLog( text = "log this!", type = "error" );
```