

[← Back to Dashboard](#) | [← Plugins Viewer](#)

## Feed Generator

## Contents

### Introduction

The ColdBox feedGenerator plug-in allows you to generate RSS 2 web feeds using data of your choice. If you wish to learn more about web feeds or RSS 2 I recommend reading the *Introducing RSS* chapter in the [RSS Tutorial for Developers](#) document.

While there are three very different forms of web feed formats currently in use on the Internet, The ColdBox plug-in only generates RSS 2 syntax. Now days it is the most popular method of creating feeds, has the largest extensible support and is usable by 99% of the feed readers out there.

A complete list of the elements usable in your feeds can be found at the [Feed Generator Elements](#) wiki entry.

### Resources

- [RSS 2.0 Specification](#)
- [RSS Tutorial For Content Publishers](#) by Mark Nottingham

### Supported Features

We have designed our feed generator to be as flexible as possible while hopefully still maintaining ease of use. We realize that many people will not need most of the functionality available to them but it is always good to know what is here.

- **Full RSS Support** Full support for all RSS tags and recommendations as specified in the [RSS 2.0.10 Best Practices Profile](#)
- **RSS Extension Support** For many of the more popular extensions.

```

+ RSS 2.0 Extensions
+ Apple iTunes Podcast Extension
+ Content 2.0 RSS module
+ Creative Commons RSS module
+ Dublin Core Metadata Element 1.1
+ Slash RSS module

```

- **Built-in Feed Debugging and Data Validation** So you know your feed data is valid and usable by the many feed readers out there.
- **Helpful and Extensive Debugging Messages** Pinpoint exactly where or what in your feed is causing a problem.
- **Unicode UTF-8 Character Encoding** So you can use multilingual characters within your data and not worry about the feed's XML being corrupted (a common problem).
- **Dynamic XML Namespaces** Detects what supported feed extensions are in use and only displays their relevant XML data.
- **Multiple XML Outputs** Either to an external XML file, a CFML variable or both.
- **White Space Trimming** Automatically trims white space from your data to keep the size of the feed to a minimum.

### Feed Elements

A complete list of the elements usable in your feeds can be found at the [Feed Generator Elements](#) wiki entry.

### Where To Start?

When you create a feed, you specify the feed content with a combination of a query object and a properties structure. The plug-in generates the feed XML and returns the results. By providing a method this result can either be in the form of a Boolean return or an onscreen dump of the generated XML code. In addition you can save the XML as a file and have it hosted on the web server. It is best practice to create, save and then point users to this generated, hosted file and only run the feedGenerator plug-in when the XML file needs updating with new content.

As mentioned the feed content is supplied using a combination of a query object and a properties structure. All the feed's Metadata, which is information describing the feed will be provided using the structure. While the feed's items such as news articles or podcast episodes, (continuously updated data) will be supplied using a query object. Whether this query object is sourced from a live database or manually created by you, this does not matter.

### Setup Your Application

Before you start feeding data into the plug-in we need to setup your application to be ready to handle the XML files. This is a very important process otherwise the character encoding may not work correctly so your XML may not display in peoples feed readers.

Create a **xml** sub-directory within the **includes** directory. This should be where the future feed XML files are stored.

```

+ApplicationRoot
|----+ includes
|----+ xml

```

Create a file with the code below and save it as **Layout.xml.cfm** in the **layouts** directory. This layout will tell ColdBox how to correctly display XML content.

```

<cfset event.showDebugPart(4)>
<cfcontent type="application/rss+xml" cfoutput="trim"(renderView(4)/#output)

```

```

+ApplicationRoot
|----+ layouts
|----+ Layout.xml.cfm

```

Create a new view file to display the RSS XML content with the code below. Name it **vwDisplayFeed.cfm** while saving it to the **views** directory.

```

<cfoutput rc.feedasxml cfoutput>

```

Then create another view file **vwGenerateFeed.cfm**, this will be used when generating or recompiling the feed.

```

<cffrc.compileFeed
<!-- Congratulations the feed was successfully updated -->
<cfelse>
<!-- There was a problem updating the feed -->
</cffrc>

```

Edit your ColdBox **ColdBox.xml.cfm** file and update your layouts declaration to use the new XML friendly layout.

```

<!-- Declare Layouts for your application here -->
<Layouts>
<!-- Declare the default layout. MANDATORY -->
<DefaultLayout> <cfdefaultlayout>
<!-- Declare XML layout for use with feeds -->
<Layout file="Layout.xml.cfm" name="xml">
</Layout>
</Layouts>

```

### Add Your Data

Create an event handler **feed.cfc** with the code below and save it to the **handlers** directory. In the code you will notice there are four functions which should be self explanatory. The **rss** function reads the feed XML file and displays it on screen as RSS-XML content. While the **generate** function sets the feed's Metadata. This Metadata is set to the **rc.feed** structure. The feed items are set to **feed.items** with the content fetched from the model **feeditems.generateItems()**. The ColdBox plug-in **feedGenerator()** is then called to convert the CFML structure/query data into a RSS formatted XML file. This file is saved to **includes/xml/my\_first\_feed.xml**.

```

<cfcomponent name="feed" output="false" extend="coldbox.system.eventhandler" wire="true">
<!-- Dependencies -->
<cfproperty name="feedItems" type="Model" scope="instance">
<cffunction name="init" access="public" returnType="feed" output="false">
<cfargument name="controller" type="any">
<cfset super.init(arguments.controller)>
<!-- Any constructor code here -->
</cfset>
</cffunction>
<!-- By default, run rss event -->
<cffunction name="index" access="public" returnType="void" output="false">
<cfargument name="Event" type="any">
<cfset rc = Event.getCollection()
<cfset runEvent("feed.rss")>
</cffunction>
<!-- Display Feed -->
<cffunction name="rss" access="public" returnType="void" output="false">
<cfargument name="event" type="any">
<cfset rc = event.getCollection()
<!-- get feed xml file -->
<cfset rc.feedasxml = getPlugin("includes.xml/my_first/Feed.xml")
<!-- view -->
<cfset event.setView("DisplayFeed")>
</cffunction>
<!-- Create Feed -->
<cffunction name="generate" output="false" returnType="void">
<cfargument name="event" type="any">
<cfset rc = event.getCollection()
<!-- feed structure -->
<cfset rc.feed = struct{}()
<!-- metadata -->
<cfset rc.feed.description = "This is an example feed showing the very basics of what constitutes a web feed"
<cfset rc.feed.link = "http://www.example.com"
<cfset rc.feed.title = "ColdBox Starter Feed"
<!-- Item's data -->
<cfset rc.feed.items = instance.feeditems.generateItems()
<!-- Process and compile the feed -->
<cfset rc.compileFeed = getPlugin("feedGenerator").createFeed(rc.feed, Output.getSetting("ApplicationPath")#includes/xml/my_first/Feed.xml">
<!-- view -->
<cfset event.setView("GenerateFeed")>
</cffunction>
</cfcomponent>

```

### RC.CompileFeed explained

```

<cfset rc.compileFeed = getPlugin("feedGenerator").createFeed(rc.feed, Output.getSetting("ApplicationPath")#includes/xml/my_first/Feed.xml">

```

- **rc.compileFeed** - Name of the variable to save the generated feed to. This is optional.
- **getPlugin("feedGenerator").createFeed()** - ColdBox feedGenerator plug-in requesting the createFeed() method.
- **feedStruct = rc.feed** - Is the structure containing the feed Metadata as well as the feed item, query data.
- **outputFile = #getSetting("ApplicationPath")#includes/xml/my\_first\_feed.xml'** - The full file path to an XML, which will be used to save the generated feed

Now everything is nearly ready for our feed. All we have to do is create our model which will contain the feed data. So save the following code to a file in the **model** directory named **feeditems.cfc**.

```

<cfcomponent name="feeditems" hint="A feed items data object">
<cfset variables.instance = struct{}()
<cfset variables.instance.items = Query.new("id,Description,Link,Title")

```

- [Feed Generator](#)
  - [Introduction](#)
  - [Supported Features](#)
  - [Feed Elements](#)
  - [Where To Start?](#)
    - [Setup Your Application](#)
    - [Add Your Data](#)
    - [RC.CompileFeed explained](#)
    - [Generate Your Feed](#)
    - [Mapping And Using Databases](#)
  - [How To Implement Elements](#)
    - [rssrcv.metadata](#)
    - [creative-commons.metadata](#)
    - [dublin-core.metadata](#)
    - [image.category](#)
    - [image.search.url](#)
    - [item.categories](#)
    - [item.enclosures](#)
    - [item.dublin.core](#)

```

<cffunction name="init" access="public" returntype="feeditems" output="false">
<cfreturn this>
</cffunction>

<cffunction name="generateItems" access="public" returntype="query" output="false">
<cfset var returnQuery = variables.instance.getItems()
<cfset QueryAddRow(returnQuery, 3)
<!--- Create our first item -->
<cfset QuerySetCell(returnQuery, 'title', 'Our first ColdBox feed item')>
<cfset QuerySetCell(returnQuery, 'description', 'Congratulations you have successfully created your first ColdBox feed.')>
<cfset QuerySetCell(returnQuery, 'author', 'bengarratt@example.com (Ben Garratt)')>
<!--- Item 2 won't have an author but that is okay as it is optional -->
<cfset QuerySetCell(returnQuery, 'title', 'Optional tags 2')>
<cfset QuerySetCell(returnQuery, 'description', 'In this item we have chosen to leave out the author tag.')>
<cfset QuerySetCell(returnQuery, 'link', 'http://www.example.com/article')>
<!--- Item 3 -->
<cfset QuerySetCell(returnQuery, 'title', 'Broken links 3')>
<cfset QuerySetCell(returnQuery, 'description', 'Don't click those example links. They are fake, but are design to to give you an example when implementing your own feeds.')>
<cfset QuerySetCell(returnQuery, 'link', 'http://www.example.com/article')>
<cfreturn returnQuery
</cffunction>
</cfcomponent>

```

### Generate Your Feed

Finally you just have to run the `feed.generate` event in your browser to compile this data into a feed.

```

// Standard link
http://localhost/coldbox/index.cfm?event=feed.generate

// With SES routing
http://localhost/coldbox/index.cfm/feed/generate

```

`{localhost}` should be replaced with the server address.

Assuming everything went okay you should have seen the message *Congratulations the feed was successfully updated.* You can now point your web browser or feed reader to the `feed.rss` event to display or use the feed.

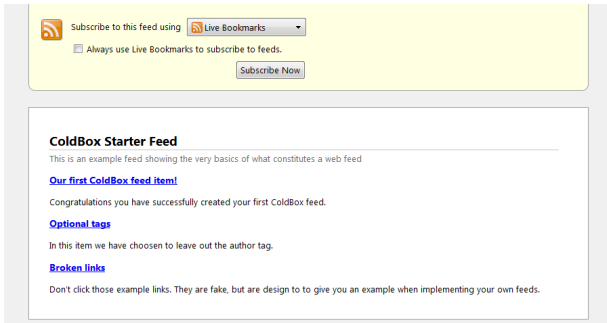


Figure: Firefox displaying our feed

### Mapping And Using Databases

In the [previous example](#), we created a feed by using items supplied with query data entered by ourselves. In this chapter we will replace that code and instead create the feed items using data supplied from a live database. To do this we create a structure known as a map. In this map, the structure key names are equal to the [item elements](#), while the structure values hold the database column names which contain the data. Once supplied, ColdBox automatically wires the database columns and the item elements together.

In this mapping example we assume you have already created the [Where To Start](#) example. We will revise that earlier example to use mapped, live data. First replace the file `feeditems.cfc` in the `model` directory with the code below. Please note this code is designed to use the `cfartgallery` example database which is included in the default install of ColdFusion. Users of other CFML engines will probably need to use their own database and would need to modify the query code accordingly.

```

<cfcomponent name="feeditems" hint="A feed items data object">
<cfset variables.instance = structNew()
<cfset variables.instance.dms = cfartgallery>
</cfset>

<cffunction name="init" access="public" returntype="feeditems" output="false">
<cfreturn this>
</cffunction>

<!--- Fetch data from a database -->
<cffunction name="generateItems" access="public" returntype="query" output="false">
<cfset var returnQuery = />
<cftry>
<!--- Query the cfartgallery database and save results to a variable -->
<cfquery name="returnQuery" datasource="variables.instance.dms">
SELECT address, city, state, postalcode, customerfirstname, customerlastname, orderdate, orderid, orderstatusid, total, tax
FROM orders
ORDER BY orderdate DESC
</cfquery>
<!--- If the database query fails we create an empty query -->
<cfcatch type="Database">
<cfset returnQuery = QueryNew( 'id', 'VarChar' )>
</cfcatch>
</cftry>
<cfreturn returnQuery
</cffunction>

<!--- Map database column names to feed item elements -->
<cffunction name="mapItems" access="public" returntype="struct" output="false">
<cfset var returnMap = structNew()
<cfset returnMap.category_tag = 'date'
<cfset returnMap.description = 'description'
<cfset returnMap.guid_string = 'orderid'
<cfset returnMap.title = 'customerlastname'
<cfset returnMap.pubdate = 'orderdate'
<cfreturn returnMap
</cffunction>
</cfcomponent>

Now that our method is complete we need to update our generate event in the feed.cfc file, located in the handlers directory to request and supply a map. Here the only real change is the rc.compileFeed variable where we now add the ColumnMap argument which points to the newly created mapItems() method.

<!--- Create Feed -->
<cffunction name="generate" output="false" returntype="void">
<cfargument name="event" type="any">
<cfset var rc = event.getCollection()
<!--- feed structure -->
<cfset rc.feed = structNew()
<!--- metadata -->
<cfset rc.feed.description = "This is an example feed showing the very basics of what constitutes a web feed"
<cfset rc.feed.link = "http://www.example.com"
<cfset rc.feed.title = "ColdBox Starter Feed"
<!--- Item's data -->
<cfset rc.feed.items = instance.feeditems.generateItems()
<!--- Process and compile the feed -->
<cfset rc.compileFeed = getPlugin(GenerateFeed).createFeed(FeedStruct=rc.feed, ColumnMap=instance.feeditems.mapItems(#rc.outputPage=ApplicationPath#)) #includes/xml/my_first/feed.xml"
<!--- view -->
<cfset event.setView(GenerateFeed)
</cffunction>

```

Now you can generate your revised feed by running the `feed.generate` event. This should create a `ColdBox.feedGenerator.InvalidFeedStructure`, which is the technical name for a feed validation error.

### Oops! Exception Encountered

```

Application Execution Exception
Error Type: ColdBox.feedGenerator.InvalidFeedStructure: [N/A]
Error Messages: The generated RSS feed has some problems which makes it incomplete

1. Item 1: The guid element '11' : is not a valid URL which is a requirement when isPermaLink attribute is ignored or 'true'
(See http://www.rssboard.org/rss-profile#element-channel-item-guid)
2. Item 2: The guid element '20' : is not a valid URL which is a requirement when isPermaLink attribute is ignored or 'true'
(See http://www.rssboard.org/rss-profile#element-channel-item-guid)
3. Item 3: The guid element '19' : is not a valid URL which is a requirement when isPermaLink attribute is ignored or 'true'
(See http://www.rssboard.org/rss-profile#element-channel-item-guid)
4. Item 4: The guid element '23' : is not a valid URL which is a requirement when isPermaLink attribute is ignored or 'true'
(See http://www.rssboard.org/rss-profile#element-channel-item-guid)
5. Item 5: The guid element '16' : is not a valid URL which is a requirement when isPermaLink attribute is ignored or 'true'
(See http://www.rssboard.org/rss-profile#element-channel-item-guid)

```

Figure: ColdBox generating a feed and throwing a validation error

To correct this problem we need to update our `feeditems` model to incorporate the `guid_permaLink` element and have it contain the value `false` for each item.

We again update the `feeditems` model, first updating `generateItems()` with a new array containing a value of `false`. We merge this array into the `returnQuery` query for use with our feed.

```

<!--- Fetch data from a database -->
<cffunction name="generateItems" access="public" returntype="query" output="false">
<cfset var returnQuery = />
<cfset var permaLinkArray = arrayNew(1)
<cfset var i = 1>
<cftry>
<!--- Query the cfartgallery database and save results to a variable -->
<cfquery name="returnQuery" datasource="variables.instance.dms">
SELECT address, city, state, postalcode, customerfirstname, customerlastname, orderdate, orderid, orderstatusid, total, tax
FROM orders
ORDER BY orderdate DESC
</cfquery>

```

```

<!-- If the database query fails we create an empty query -->
<<cfcatch type="database">
  <<cfset returnQuery = QueryNew(oid="VarChar")>
</cfcatch>
</cftry>
<!-- Create a new permalink cell, one for each query record -->
<<cfloop from="1" to="#returnQuery.RecordCount">
  <<cfset permalinkArray[i]=>
</cfloop>
<!-- Merge the permalink cells into the query -->
<<cfset QueryAddColumn(returnQuery,"permalink",permalinkArray)
<!-- return the updated query -->
<<cfreturn returnQuery
</cffunction>

```

But before we can do that we must update `mapItems` where we add a new map pointing `guid_permalink` to the newly created query column `isPermalink`.

```

<!-- Map database column names to feed item elements -->
<<cffunction name="mapItems" access="public" returntype="struct" output="false">
  <<cfset var returnMap = StructNew()
  <<cfset returnMap.category_tag="date">
  <<cfset returnMap.description="description">
  <<cfset returnMap.guid_string="id">
  <<cfset returnMap.title="customerLastName">
  <<cfset returnMap.pubDate="startDate">
  <<cfset returnMap.guid_permalink="permalink">
  <<cfreturn returnMap
</cffunction>

```

Now our feed should be complete, trouble free and ready to be regenerated. After running the event `feed.generate`, run `feed.rss` to display the feed in your reader or browser. Hopefully the new feed should contain around 23 items.



Figure: Firefox displaying our feed

## How To Implement Elements

This section will explain how to implement some of the more complex feed elements available. If you are looking for a beginner's tutorial on how to create a feed and add elements, please read the [Where To Start](#) chapter.

### category metadata

The category metadata element allows the use of multiple tags. To input these category tags, ColdBox requires a structured array. In the example below we will create 3 category tags, one of which has the optional domain attributes.

```

<<cfset rc.feed.category = Array(1)
<<cfset rc.feed.category[1] = structNew({
  <<cfset rc.feed.category[1].tag = "fruit"/>
  <<cfset rc.feed.category[1].tag = "fruit"/>
  <<cfset rc.feed.category[2] = structNew({
  <<cfset rc.feed.category[2].tag = "Business/Food_and_Related_Products/Produce/Fruits/"
  <<cfset rc.feed.category[2].domain = "dmoz"/>
  <<cfset rc.feed.category[3] = structNew({
  <<cfset rc.feed.category[3].tag = "produce"/>

```

### creative commons metadata

The creative commons element uses a list to allow multiple URLs that point to the different Creative Commons licenses.

```

<<cfset rc.feed.commonLicense = ["http://creativecommons.org/licenses/by-nc/3.0","http://creativecommons.org/licenses/by/3.0"]

```

### dublin core metadata

To implement any of the 15 Dublin Core Metadata Element Set elements, you create a `dcitem` structure. The example below uses 2 of the 15 available DC elements.

```

<<cfset rc.feed.dcitem = structNew({
  <<cfset rc.feed.dcitem.creator = "Joe Blogs">
  <<cfset rc.feed.dcitem.publisher = "The Company">

```

### itunes category

The iTunes extension category element is uniquely structured in ColdBox whereby the iTunes category is also the structure name while the sub-category is the structure value. In the example below there are two categories, *Society & Culture* and *TV & Film*. The *Society & Culture* also has the sub-category of *History*. It should be noted that only maximum of two categories are ever allowed and they must match the list contained in the [Apple iTunes Podcast specification](#).

```

<<cfset rc.feed.itunes = structNew({
  <<cfset rc.feed.itunes.category = structNew({
  <<cfset rc.feed.itunes.category["Society & Culture"] = "History"/>
  <<cfset rc.feed.itunes.category["TV & Film"] = "" />

```

### opensearch query

The OpenSearch extension Query element allows multiple instances of itself. So to allow this functionality ColdBox requires the `opensearchQuery` data to be supplied using a structured array. The example below creates three OpenSearch Queries. Two are related search terms while the other is the query request.

```

<<cfset rc.feed.opensearchQuery = arrayNew(3)
<<cfset rc.feed.opensearchQuery[1] = structNew({
  <<cfset rc.feed.opensearchQuery[1].role = "request">
  <<cfset rc.feed.opensearchQuery[1].searchTerm = "General Motors annual report">
  <<cfset rc.feed.opensearchQuery[2] = structNew({
  <<cfset rc.feed.opensearchQuery[2].role = "related">
  <<cfset rc.feed.opensearchQuery[2].searchTerm = "GM"/>
  <<cfset rc.feed.opensearchQuery[2].title = "General Motors stock symbol">
  <<cfset rc.feed.opensearchQuery[3] = structNew({
  <<cfset rc.feed.opensearchQuery[3].role = "related">
  <<cfset rc.feed.opensearchQuery[3].searchTerm = "automotive industry revenue">

```

### item category

The item category element has two inputs: `category_tag` to supply tag information and `category_domain` to supply domain. If you wanted to supply multiple categories, then you would supply the data with a comma separated, list in the `category_tag`. The same is done for `category_domain` but seeing that is an option element, individual list items can be ignored by using a `(-)` (hyphen). In the example below we have three tags and one domain. The single domain value is linked to the `Business/Food_and_Related_Products/Produce/Fruits/` category tag thanks to the use of hyphens.

```

<<cfset rc.item.category_tag = "fruit,Business/Food_and_Related_Products/Produce/Fruits/Produce"
<<cfset rc.item.category_domain = "-,dmoz,-" />

```

### item enclosures

The item enclosure element, used for attaching files has three inputs: `enclosure_url` to supply the URLs linking to the files, `enclosure_size` to supply the sizes of the files and `enclosure_type` to supply the file MIME types. If you wish to supply multiple enclosures for an item then you supply the data with a comma separated, list. All three inputs are required for each enclosure element you wish to use in an item. In the example below we have two images attached to the feed item.

```

<<cfset rc.item.enclosure_url = ["http://www.example.com/images/picture.png","http://www.example.com/images/picture2.jpg"]
<<cfset rc.item.enclosure_size = "54005,30561" />
<<cfset rc.item.enclosure_type = "image/png,image/gif" />

```

### item dublin core

To implement any of the 15 Dublin Core Metadata Element Set elements in an item, you use a `dcitem` `[term]` input. The example below uses 2 of the 15 available DC elements.

```

<<cfset rc.item.dcitem_rights = "Public Domain" />
<<cfset rc.item.dcitem_publisher = "The Company" />

```

### Categories:

- [level.intermediate](#)