

Writing an RSS Aggregator

Completing the task



By Jeffrey Houser

Two months ago I put together an article about building an RSS aggregator (CFDJ, Vol. 8, issue 5). Before reading this you might want to refresh your mind on the original article. Go over here – <http://coldfusion.sys-con.com/read/235976.htm> – to read it.

I discussed what an aggregator is and why we care to write one. RSS is a version of XML that is used to make syndicating data easy. Most blogs have an RSS feed attached to them. An aggregator takes a bunch of blog feeds and combines them. Weblogs.macromedia.com and FullAsaGoog.com are two great examples of aggregators in the ColdFusion community.

The last article stepped you through the thought process of designing the database and object model. We built two of the components: an RSSCategory component that is used to categorize the RSS feeds and an RSSFeed component that is used to enter an RSS feed into the database. We also wrote some admin code to enter a new RSS feed into the database. The article was, unfortunately, lacking the real meat of things, which is the RSSAggregator component. In this article, we'll flesh it out along with the item component and the scheduled task for running in. Before we do that, I did find one bug so let's fix that.

One Quick Bug Fix

While testing the code from the last article, I discovered a bug. It happens to the best of us, right? When entering the feed into the database (Feedip.cfm), I had written some code to retrieve the feed using cfhttp and then parse the XML to get the feed's title. This worked fine for standard RSS feeds, and referenced the title like this:

```
MyXMLVar.rss.channel.title.xmltext
```

The problem here is that the root element, RSS, is hard coded. When I tried to run this code against the weblogs.macromedia.com site, it didn't work. The reason is the RSS feed offered by weblogs.macromedia.com is RDF. The root element isn't RSS, it is rdf:RDF. The fix for this was easy:

```
MyXMLVar.xmlroot.channel.description.xmltext
```

Instead of hard coding the RSS root name, I used the xmlroot value. RDF and RSS handle items differently too, so this will come into play in some of the code from this article.

Writing the Item Component

In RSS, an item is the equivalent of a single blog post. For each item, we are storing the link to the original item, the title, and the description. There can be a lot more data associated with this, but for the sake of these articles, I decided to keep it simple. The component also has some internal values, such as a primary key named ItemID and a foreign key named RSSFeedID. The RSSFeedID tells us which item the RSS Feed has. The Item component code is shown in Listing 1.

The component starts with the cfcomponent tag (of course) and the pseudo constructor code. The pseudo constructor sets up the instance variables of the component. Once again, our components are borrowing Hal Helms basecomponent from <http://halhelms.com/webresources/BaseComponent.cfc>. I use this instead of writing manual getter and setter methods.

Other than inherited methods, this component contains an init method and a commit method. The init method takes an ItemID and the datasource and loads all the relevant information from a database. The commit method will insert, or update, the information in the database as needed.

Creating the Aggregator

The RSSAggregator component is shown in Listing 2. This component is a bit different than many I usually write, since its purpose is not to get data in and out of a database. It is pulling data from some far away off place and putting it in our database. This component does not contain an instance variable, and as such does not have a pseudo constructor. There are three methods, which I can explain in more detail.

The first method is GetAllFeeds. It is a private method, so it cannot be called outside of the CFC. It runs a query to retrieve all the feeds that are being watched in the database. The method returns the query. There is nothing special about this.

The second method it called ItemExists. It accepts an item component (which you learned about in the previous section of this article) and checks to see whether this item already exists in the database. If it does, it returns true, otherwise it returns false. I made the assumption that each item has a unique URL pointing to it, so that is the value the code checks to see if the item is unique.

The third method is an init method. This is the one that retrieves the feeds and stores the data in the database, if relevant. This is the only public method in the component; the `getallfeeds` and `ItemExists` methods are used by `init`. The `init` method starts by setting some local variables. These are the local variables:

- **GetAllFeeds:** This variable calls the `GetAllFeeds` function. It contains a list of all feeds that we want to pull data from.
- **Error:** The second local variable is the error variable. The most likely cause of an error when running this code will be that a feed times out. If the feed times out, we won't want to attempt to process the data. It defaults to false, which is no error.
- **TempItem:** As we loop over the feed data, this item object will be used to decide whether the data is duplicated or needs to be saved. The item component also contains the SQL for saving the item component.
- **ItemArray:** The `ItemArray` value is a temporary value that will contain an array of all items in the current feed.
- **TempItemindex:** When we look over the `ItemArray` array, a counter will be needed to keep track of which item is being examined at the moment. This is the counter variable.
- **MyXMLVar:** The `MyXMLVar` will contain the returned XML feed.

While initializing the local variables, the `GetAllFeeds` query was executed. The code starts by looping over it. The error variable is initialized to false. You want to make sure to initialize it each time through the loop, so that you aren't processing the current feed based on the result of the previous feed.

Next comes a try block. Inside the try block is code to retrieve the RSS feed. If the feed times out, a catch block switches the error value to true. Exiting the try block, if the error is false, the code processes the feed. If the error is true, skip the processing and go right to the next feed. Although left out at this time, there should probably be some sort of logging for feeds that cause errors.

Earlier I spoke about the differences between RSS and RDF. Most blogs I read pass out data in the RSS format, but weblogs. macromedia.com was using RDF. You can read more about RDF at <http://www.w3.org/RDF/>. In RSS, items are stored inside the channel. In RDF they are not. The `ItemArray` is initialized differently depending on the root. (If this code tries to parse another flavor of XML, it will cause problems.) The next code block uses `cfloop` to loop over the `ItemArray`. It creates an item object using the `tempitem` variable. It sets the relevant instance data, then it uses the `ItemExists` function to check whether or not the item exists yet. If it doesn't, the `commit` method is run to save the data. Otherwise, nothing happens. The loop ends, and the

method returns true. This is simple stuff, right?

The Scheduled Task


There is one last bit of code to examine in this article and that is the actual scheduled task code. The bulk of the code is located in the component, so all the scheduled task does is create an instance of the `RSSAggregator` and run the `init` method. The code looks like this:

```
<cfscript>
variables.RSSAggregator = CreateObject("component", "#request.Component-
Loc#.RSSAggregator");
variables.RSSAggregator.init(request.dsn);
</cfscript>
```

It is probably one of the easiest scheduled tasks I've ever written.

Conclusion

This app is far from complete. Most RSS feeds contain a lot more data than just link, description, and title. A full-featured app would address those features. I left them out in the interest of length. Also, we've only built code for collecting RSS data. There is no "view" portion of this app. The only way to view the data you're collecting is to open up the database. That isn't conducive to good usability.

Every good project needs a code name, and I decided to give this project one. After some deep soul-searching, I've decided to name this project `MyFriend`. There are two reasons for this. The first is that I modeled the whole idea after the `LiveJournal` friends list. The second is that, `My Friend` is the name of a song that my first band recorded the first time we went into a recording studio. It was my first time in a professional recording studio, and I had been playing bass for less than a month. The results came out better than you might have expected, really. I'm a sentimental freak. Check out my www.jeffryhouser.com for the latest version of this code and let me know what you think. 

About the Author

Jeff Houser has been working with computers for over 20 years. He owns a `DotComIt`, a web consulting company, manages the `CT Macromedia User Group`, and routinely speaks and writes about development issues. You can find out what he's up to by checking his Blog at www.jeffryhouser.com.

jeff@instantcoldfusion.com

Listing 1

```
<cfcomponent extends="basecomponent">
```

```
<cfscript>
variables.instance.ItemID = "";
variables.instance.RSSFeedID = "";
variables.instance.Title = "";
variables.instance.link = "";
variables.instance.description = "";
</cfscript>
```

```
<cffunction name="Init" access="public" returnType="Boolean">
<cfargument name="ItemID" type="string" required="true">
```

```
<cfargument name="dsn" type="string" required="true">
<cfset var qGetItem = "">

<cfquery name="qGetItem" datasource="#arguments.dsn#">
select *
from items
where items.ItemID = <cfqueryparam value="#arguments.itemID#"
cfsqltype="cf_sql_varchar">
</cfquery>

<cfif qGetItem.recordcount is 0>
<cfreturn false>
</cfif>
```

```

<cfscript>
variables.instance.ItemID = qGetItem.ItemID;
variables.instance.RSSFeedID = qGetItem.RSSFeedID;
variables.instance.Title = qGetItem.Title;
variables.instance.link = qGetItem.link;
variables.instance.description = qGetItem.description;
</cfscript>

<cfreturn true>
</cffunction>

<cffunction name="Commit" access="public" returntype="Boolean">
<cfargument name="dsn" type="string" required="true">

<cfset var qUpdateItem = "">

<cfif variables.instance.ItemID is "">
<cfset variables.instance.ItemID = createuuid()>
<cfquery name="qUpdateItem" datasource="#arguments.dsn#">
insert into items (ItemID, RSSFeedID, Title, Link, Description)
values (
'#variables.instance.ItemID#',
<cfqueryparam value="#variables.instance.RSSFeedID#"
cfsqltype="cf_sql_varchar">,
<cfqueryparam value="#variables.instance.Title#" cfsqltype="cf_
sql_varchar">,
<cfqueryparam value="#variables.instance.Link#" cfsqltype="cf_sql_
varchar">,
<cfqueryparam value="#variables.instance.Description#"
cfsqltype="cf_sql_varchar"> )
</cfquery>
<cfelse>
<cfquery name="qUpdateItem" datasource="#arguments.dsn#">
update items
set RSSFeedID = <cfqueryparam value="#variables.instance.RSS-
FeedID#" cfsqltype="cf_sql_varchar">,
Title = <cfqueryparam value="#variables.instance.Title#"
cfsqltype="cf_sql_varchar">,
Link = <cfqueryparam value="#variables.instance.Link#"
cfsqltype="cf_sql_varchar">,
Description = <cfqueryparam value="#variables.instance.Descrip-
tion#" cfsqltype="cf_sql_varchar">

where ItemID = <cfqueryparam value="#variables.instance.ItemID#"
cfsqltype="cf_sql_varchar">

</cfquery>
</cfif>

<cfreturn true>
</cffunction>
</cfcomponent>

Listing 2
<cfcomponent extends="basecomponent">

<cffunction name="GetAllFeeds" access="private" returntype="query">
<cfargument name="dsn" type="string" required="true">

<cfset var qGetFeeds = "">

<cfquery name="qGetFeeds" datasource="#arguments.dsn#">
select * from RSSFeeds
</cfquery>

<cfreturn qGetFeeds>
</cffunction>

<cffunction name="ItemExists" access="private" returntype="Boolean">
<cfargument name="item" type="item" required="true">
<cfargument name="dsn" type="string" required="true">

<cfset var qCheckItem = "">

```

```

<cfquery name="qCheckItem" datasource="#arguments.dsn#">
select itemID
from items
where items.link = <cfqueryparam value="#arguments.item.
get('link')#" cfsqltype="cf_sql_varchar">
</cfquery>

<cfif qCheckItem.recordcount gte 1>
<cfreturn true>
</cfif>
<cfreturn false>

</cffunction>

<cffunction name="Init" access="public" returntype="void" hint="Gets
all RSS feeds, parses them, and saves new items in the database">
<cfargument name="dsn" type="string" required="true">

<cfset var GetAllFeeds = GetAllFeeds(arguments.dsn)>
<cfset var error = false>
<cfset var TempItem = CreateObject('component', 'item')>
<cfset var ItemArray = "">
<cfset var TempItemindex = 0>
<cfset var MyXMLVar = "">

<cfloop query="GetAllFeeds">
<cfset error = false>
<cftry>
<cfhttp url="#GetAllFeeds.link#" timeout="30"></cfhttp>

<cfcatch type="any">
<!-- most likely cause of error is server timeout ---->
<cfset error = true>

</cfcatch>

</cftry>

<cfif error is false>
<cfset MyXMLVar = xmlparse(cfhttp.filecontent)>

<cfif MyXMLVar.xmlroot.xmlname is "rss">
<cfset ItemArray = MyXMLVar.xmlroot.channel.item>
<cfelseif MyXMLVar.xmlroot.xmlname is "rdf:RDF">
<cfset ItemArray = MyXMLVar.xmlroot.item>
</cfif>

<cfloop from="1" to="#arraylen(ItemArray)#" index="TempItemindex">

<cfset TempItem = CreateObject('component', 'item')>
<cfset TempItem.set('RSSFeedID', GetAllFeeds.RSSFeedID)>
<cfset TempItem.set('Title', ItemArray[TempItemindex].Title.
xmlText)>
<cfset TempItem.set('link', ItemArray[TempItemindex].link.
xmlText)>
<cfset TempItem.set('description', ItemArray[TempItemindex].
description.xmlText)>
<cfif not ItemExists(TempItem ,arguments.dsn)>
<cfset TempItem.commit(arguments.dsn)>
</cfif>

</cfloop>
<cfelse>
<!-- should add some type of error handling here ---->
</cfif>

</cfloop>

</cffunction>
</cfcomponent>

```

Download the Code...
Go to <http://coldfusion.sys-con.com>