web services EDGE conference&expo ANNOUNCING WEBSERVICESEDGE 2001 P.53

U.S. $8.99 (Canada $9.99)

# COLDFUSION Developer's Journal

ColdFusionJournal.com

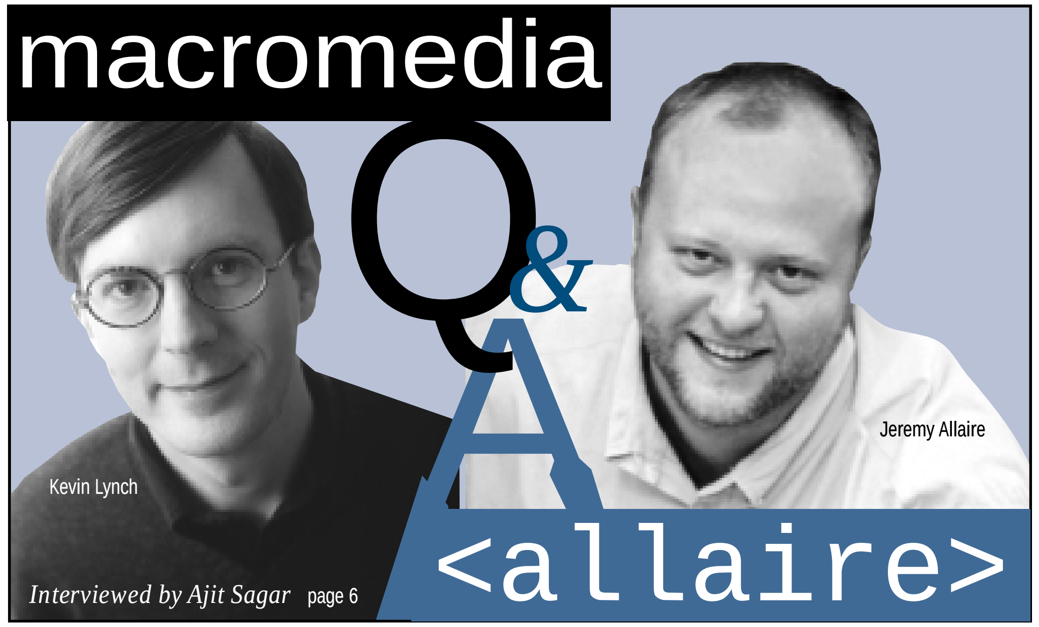
April 2001 Volume: 3 Issue: 4

Sept. 23–26, 2001 New York, NY
JAVAEDGE conference&expo 32

Oct. 22–25, 2001 Santa Clara, CA
XMLEDGE conference&expo 2001 40

SYS-CON MEDIA

macromedia Q&A <allaire>



*Interviewed by Ajit Sagar* page 6

EDITORIAL OFFICES
**SYS-CON MEDIA, INC.** 135 CHESTNUT RIDGE RD., MONTVALE, NJ 07645
**TELEPHONE:** 201 802-3000 **FAX:** 201 782-9600

COLDFUSION DEVELOPER'S JOURNAL (ISSN #1523-9101) IS PUBLISHED MONTHLY (12 TIMES A YEAR) FOR $89.99 BY SYS-CON PUBLICATIONS, INC., 135 CHESTNUT RIDGE RD.,MONTVALE, NJ 07645

POSTMASTER
**SEND ADDRESS CHANGES TO:**
COLDFUSION DEVELOPER'S JOURNAL
**SYS-CON MEDIA**
135 CHESTNUT RIDGE RD.,MONTVALE, NJ 07645

# This Month…Interrupt

BY **ROBERT DIAMOND**

A big part of the lifeblood of a good programming language is its ability to be expanded and built upon by its user community. Java, perhaps the hottest programming language of our day, certainly has this ability, along with many other languages. And CFML is certainly no exception. Extending the language, and sharing code and swappable components, helps foster the one thing all languages strive for – a living, active user community.

ColdFusion excels in many ways – the hundreds of Web sites, mailing lists such as CF-Talk and our own CFDJList, and, most of all, Allaire's Developer Exchange (http://devex.allaire.com/developer/gallery/). The Developer Exchange is Allaire's repository of custom tags, visual tools, scripts, Web content, and more….Several articles in this issue are devoted to all facets of custom tags – using the custom tags that are out there, which are the best, and why; how to create your own custom tags; and, of course, the optimization of the whole process – we've got it all. A one-stop custom tag shop, if you will.

## Looking Ahead…(with a Little Bit of Help)

On a separate note, I'm quite proud to announce the forming of our new International Advisory Board to work with me and our other in-house editors on the direction of the magazine, ranging from focus issues and editorial content to new features for our companion Web site. On the board (full details available on www.sys-con.com/coldfusion) are several names with which I'm sure you're familiar, and a few you might not be familiar with yet, but soon will be. For starters, it would be impossible to talk about ColdFusion without the leadership of those running the company behind it, both Jeremy Allaire and Kevin Lynch, the "new" Macromedia (am still practicing writing and saying Macromedia ColdFusion, will go write it on the blackboard 100 times after completing this editorial) CTO and president of products. We also have Ben Forta, a man who needs no introduction in this magazine; I'm sure you all own a copy of at least one of his books. Charles Arehart is our journeyman extraordinaire who I've heard described as a developer's developer. Steve Drucker is the CEO of FigLeaf Software and the founder of the first ColdFusion user group. Hal Helms is one of the leading Fusebox experts as well as a frequent speaker and writer on related topics. Ajit Sagar, editor-in-chief of ***XML-Journal***, and Karl Moss, a principal software developer for JRun, will help us cover some of the many technologies that CF works with. Michael Smith and Michael Dinowitz are two of the biggest community leaders; Michael Smith runs the MDCFUG and CFUNK2K events that we all know and love, and Michael Dinowitz hosts House of Fusion and Fusion Authority, and runs more mailing lists than I have room to list. Last, but certainly not least, is Bruce Van Horn, our **Ask the Training Staff** column editor, who answers many of the technical questions we get sent each and every month.

I look forward to working with all of them, as together we help ***CFDJ*** grow and evolve to better serve the expanding community of developers. So please, help us, help them, and help the magazine – if we're doing something you like or, more important, that you don't like – drop me a line.

[illegible]

ROBERT@SYS-CON.COM

ABOUT THE AUTHOR
*Robert Diamond is editor-in-chief of* **ColdFusion Developer's Journal** *as well as* **SYS-CON**'s *newest magazine,* **Wireless Business & Technology**. *Named one of the "Top thirty magazine industry executives under the age of 30" in Folio magazine's November 2000 issue, Robert is currently a senior in the School of Information Studies at Syracuse University.*

# macromedia allaire

Interview...with

**Kevin Lynch**

President of **Macromedia Products**

and

**Jeremy Allaire**

CTO of **Macromedia, Inc.**

INTERVIEWED BY **AJIT SAGAR**

<allaire> macromedia <allaire

macromedia <allaire> macromedi

Q

allaire
macromedia

*CFDJ: The biggest news of the year for Allaire and Macromedia is the merger. What are the reasons for uniting these two companies?*

Lynch: The merger of Macromedia and Allaire mirrors what's happening in Web development today. Creating the best user experience on the Web requires the combination of server-side Web application logic with client-side user interface, and team members with expertise in these areas need to work together effectively to succeed. I believe this merger is really about the marriage of two user communities, bringing the developers and designers together so they can more easily build the Web sites of tomorrow.

We also see the nature of the Internet changing as users access Web content and applications through a range of devices. This poses many challenges for developers and designers, and we want to help them be successful in delivering great user experiences across all these devices in an efficient way. It's exciting to bring these communities together, to join world-class teams with the background and experience necessary to enable this.

Allaire: This merger is as much about the specific synergies and opportunities created by combining Allaire and Macromedia as it is about a broader set of trends in the development of the Web, such as the emergence of teams of Web professionals that span a wide range of skill sets, tools, and technologies, and need to come together to deliver great Web sites and great user experiences. It's about combining dynamic content with visual authoring, and delivering Web applications to multiple devices. It's about creating the next generation of user experiences, going beyond the limitations of current Web standards. No software company has ever really combined the world of content with the world of logic and programming, and we think we're creating something special and unique.

*CFDJ: So what's the geographical spread now?*

Lynch: Both Macromedia and Allaire have offices around the world. Macromedia already has product development teams located in San Francisco, Dallas, and Minneapolis, and of course this merger now adds the Boston location.

*CFDJ: It seems to me this rapid evolution could confuse the marketplace. Is there a uniform message Macromedia/Allaire want to send out to the Web community?*

Lynch: Our mission continues to be to enable our customers to deliver the best experiences on the Web. Basically we'll have an even greater group of development teams working for our customers to help them succeed in leading Web development.

The products our customers know and love will become even stronger when combined with Allaire's server-side capabilities. A number of new solutions will emerge for our customers as the teams innovate together on supporting multiple Web devices, Web services, dynamic publishing, team collaboration, and future key trends. We'll continue to make sure that our software will be the best available and appropriately designed for each customer audience.

"I believe this merger is really about the marriage of two user communities, bringing the developers and designers together so they can more easily build the Web sites of tomorrow."

—*Kevin Lynch*

The server strategy is exciting for our customers, as together we'll be providing the most approachable way to develop dynamic Web content and applications and deploy them across industry-standard application servers, such as JRun or IBM WebSphere, and to integrate with services through Microsoft .NET. This will enable Web developers to easily and quickly create the best dynamic Web experiences and deliver them across a range of platforms.

*CFDJ: Your company now offers a large array of products. Can you briefly list what the existing products are for the benefit of our readers?*

Lynch: Our combined company will have leading products in three main categories. The first is centered around the Dreamweaver platform, which is used by over 70% of Web professionals for Web site development and includes products such as Dreamweaver, Fireworks, HomeSite, UltraDev, ColdFusion and JRun Studios, and Kawa. The second product line is based around the Macromedia Flash player, which delivers high-impact experiences to over 300 million people around the world, with content being developed with Flash, FreeHand, and Director, as well as the Flash player and the higher-end Shockwave player. As Web experiences built with these products are integrating with server-side application logic, this leads to our group of server products – ColdFusion, JRun, Spectra, and Generator. The combination of these products enables Web professionals to create the best Web experiences in the world in the most efficient way.

Alaire: Allaire is known for solutions such as ColdFusion, a popular, rapid Web application development and deployment product. JRun is a popular J2EE ser-ver known for its ease of use, affordability, and great flexibility. We also have a server product called Spectra, which is a dynamic publishing tool. Finally, we sell a few different visual development tools, including HomeSite, a popular HTML editor; the Studio products, which provide rich IDEs for server-side scripting with CFML and JSP; and Kawa, a simple yet powerful Java IDE focused on ser-ver-side Java development.

*CFDJ:* In any merger, overlaps in technology and products are inevitable. What overlaps do you see in your combined company and what are your plans for eliminating them? What integration challenges are you facing?

Allaire: Interestingly, there are few real overlaps here. For example, we obviously don't provide rich media and graphics products, nor does Macromedia provide application serving software. With the visual Web development tools we've had fundamentally different approaches to the market, reflecting different types of customers. Whereas Dreamweaver and UltraDev focused on visual authoring of HTML and dynamic applications, HomeSite, Studio, and Kawa focused exclusively on code-centric development. By bringing the world of design and the world of programming or development together, we'll really see some fantastic things. Specifically, how we'll package our visual tools, our server products, or other potential new products we're leaving to our product groups who we've empowered to build solutions that reflect the diverse needs of our now very diverse customers.

*CFDJ: What is the road map going forward?*
Lynch: We're committed to delivering on the dreams of the Allaire and Macromedia customers, and are working hard to provide the tools and technology for them to be successful. As we develop the next generation of Web development software, you'll see great, new, innovative capabilities for building dynamic Web sites that result from the cross-pollination of the Allaire and Macromedia teams, as well as even deeper integration across our product lines.

In the short term we're really excited about delivering the next major release of our servers, which builds a wide range of functionality on top of the J2EE architecture. This includes next generations of the ColdFusion and JRun technologies, as well as a next-generation application framework technology based on Spectra, Generator, and other server technologies. We'll continue to enable teams of Web professionals to work together more productively, and lead the cutting edge of user experiences and applications with the Macromedia Flash and Shockwave players.

*CFDJ: This year it seems the application server vendors are continuing their move toward being "one-stop shops" for all B2B2C frameworks. I think you're stepping away from that. Obviously your strength is in the front end and the presentation side of the middle-tier technologies. What would you say is unique about what your company offers?*
Allaire: Macromedia will be the first and only large software company focused exclusively on the needs of Web professionals. We'll be the first Web software company to provide products that support the needs of every type of professional involved in delivering a great Web site or Web application. We'll be one of the only companies that combines a rich, client-side technology (the Flash player) with great authoring and development tools and a simple, affordable server platform. We're also unique in that we're deeply committed to building open software that spans and works with other vendor platforms, whether it's Microsoft's .NET or Sun's J2EE, or whether it's deploying our server solutions on JRun or on our partners' servers, such as BEA, IBM, and Sun.

I'd also agree that our strength is in the front end of the Web application world. There are strong companies, many of them our partners, who have built great products for enterprise application integration, transaction and messaging infrastructure that reach very deep into the enterprise, and provide a level of enterprise scalability that we're not as focused on.

Given this focus, I don't think you're likely to see us try and build vertical products that target customers who are not Web professionals, such as solutions packaged for supply-chain integration, B2B marketplaces, retail and merchandising frameworks, and large-scale process integration. However, we definitely plan to expand the range of application services beyond the basic serving infrastructure necessary for delivering great Web sites and applications.

*CFDJ: Does Allaire/Macromedia plan to get into actual application design, that is, step into industry verticals, or always be application enablers? Or do you plan to continue expansion in horizontal technology offerings?*
Lynch: We're focused on enabling Web professionals build the best Web sites and user experiences in the world, and that's the community we'll continue to serve.

*CFDJ: What does this merger mean for the development communities, specifically the ColdFusion, Spectra, Flash, Dreamweaver, and Java developers?*
Allaire: I think across the board this is going to create great opportunities for developers using all these platforms. For ColdFusion, JSP, and Java developers, they can count on having fantastic visual development tools that cover both basic and advanced development. For Macromedia Flash customers, they'll be able to more easily take advantage of servers for the dynamic delivery of content and provide full application functionality. Spectra customers can look forward to better tools to create the building blocks of a dynamic publishing application. Also, all these customers can expect that we're going to try and put together technology that supports their work as a team, creating a better production workflow in how people build increasingly complex sites and applications.

*CFDJ: At Allaire, initiatives to integrate CF and Java have already begun. How are these progressing? What is the strategy to make CF available in a J2EE environment?*
Allaire: This is definitely one of our highest priorities and one of the most exciting things happening this year. We announced a project code named Neo last fall, which we demonstrated at our developer conference. Neo enables developers to build dynamic page applications using CFML – easily the most accessible and rapid development environment for Web applications – and to deploy those applications on a standard J2EE server. This is also about a higher degree of interoperability between CFML and JSP, and that's going to be important to Java developers as well. This whole project is going great, and you can expect to hear a lot more about this later this year.

A: allaire macromedia

*CFDJ: How is Spectra affected? Where does it fit in with the existing Macromedia products such as Dreamweaver and Flash?*
Lynch: Spectra is focused on helping Web developers create dynamically published Web sites, and this will become an increasingly important capability as Web sites evolve. We see many opportunities to integrate with Dreamweaver, Flash, and all our products to help create these dynamic Web sites.

*CFDJ: Some of the clients I've spoken to view Dreamweaver and ColdFusion as alternatives for a presentation layer. What's your view on this? Is there an overlap?*
Lynch: These products are very complementary – I don't see any overlap since one system authors HTML and the other is a server technology.

*CFDJ: How do you plan to address enterprise-level concerns such as scalability, robustness, and security?*
Allaire: First, we're fully committed to providing server products that are robust, that scale, and can be performance tuned, clustered, and easily managed. However, that's very different from providing the full infrastructure necessary for large-scale, distributed transaction systems that integrate with large legacy systems, provide end-to-end security, and more. For example, we don't plan to focus on providing transactional application integration software, certificate and directory servers for security, or to go into the systems management business. These are all enterprise domains with strong companies that are our partners. I would expect that customers using our software would want to take advantage of these products and partners for a range of capabilities.

*CFDJ: With Spectra, you've introduced a workflow environment that fits in the domain of business roles and synchronous workflows. How does this map to back-end workflow products, such as HP ChangeEngine, WebLogic Process Integrator, and Tibco's workflow product?*
Allaire: These other products are great solutions for people building large-scale distributed systems, systems that integrate lots of back-office applications, systems across the Internet, and large enterprises. That's also a great market, but is not something we're focused on. Spectra's workflow engine is geared toward the front-end processes involved in managing a Web site rather than back-end process integration.

*CFDJ: Do you have any offerings in the wireless market?*
Lynch: Yes, we're working to help Web professionals deliver great experiences on mobile devices. We're working to get the Macromedia Flash player onto these devices and have recently announced the developer release of the player for the PocketPC, which is available for download from our Web site. We're also working to support authoring content for these devices, and recently worked with Nokia to produce the WML Studio extensions to Dreamweaver, which are available on the Macromedia Exchange (www.macromedia.com/exchange/).

Allaire: Today, Allaire has support for HDML and WML in our visual tools, and we have a free developer-oriented WAP gateway that you can download and use with JRun (it's actually a servlet). We've also spent a lot of time helping customers understand how to build wireless applications more easily using the dynamic content capabilities of our servers.

"I think across the board this is going to create great opportunities for developers using all these platforms. For ColdFusion, JSP, and Java developers, they can count on having fantastic visual development tools that cover both basic and advanced development."
—Jeremy Allaire

*CFDJ: Does XML fit anywhere into your technology blueprint?*
Lynch: Yes, we believe XML is becoming a basic building block of Web sites, and we're supporting it in many ways across our products from authoring to players. For example, the Macromedia Flash Player 5 now supports XML content that streams directly into Flash content.

Allaire: At this point, XML is pervasive across our products. Our visual tools support extensibility through an XML model; they support working with new XML vocabularies. Our servers provide support for parsing XML and transforming XML with XSLT, and use XML internally for metadata. Spectra uses XML for content-object persistence and content syndication. As I've mentioned, support for Web services protocols built on XML is central to our future platform.

*CFDJ: For ColdFusion developers who are not familiar with Macromedia products, where's a good place to start incorporating them into existing applications?*
Allaire: One of the most exciting places they can start doing this is with Macromedia Flash. Flash expands what you can do with user interfaces on the Web, and with Flash 5 it's much easier to connect to ColdFusion using HTTP and XML, building real applications. We've put together something called the Flash UI Kit for ColdFusion that provides information on what Flash can do. It has a library of six UI controls (trees, menus, calendars, etc.) that can be put in a page using CFML tags and will be available from www.macromedia.com. This is just the tip of the iceberg in terms of what is possible.

*CFDJ: Are there any white papers or technology briefings that tell the whole story about how all the products interact and integrate? Similar to the J2EE Blueprints from Sun? Do you plan to have these in the near future?*
Lynch: The teams are currently working on the integration – I encourage you to attend the Macromedia User Conference in New York City, April 10–12, for more details. Registration information is available at www.macromedia.com.

ajit@sys-con.com

CFDJ FEATURE

BY KEDAR DESAI

# Logging a SQL Using Nested Tags

A useful debugging tool

*ColdFusion provides powerful support for building both dynamic and static SQL queries. The ability to log executed SQL statements either in a text file or in the database for security, application logging, or debugging purposes is required in some applications. However, in ColdFusion the executed SQL is not available as a variable that can be used for logging. Fortunately, ColdFusion provides a way to nest custom tags that can be used to our advantage to get the SQL string.*

Building SQL queries in CF is easy with the support that's provided through the "SQL Builder" option in Studio and the CFQuery tag. The ability to construct SQL queries that use dynamic parameters is a powerful mechanism for linking variable inputs to database queries. However, in more advanced applications you'll often want user inputs to determine not only the content of the queries, but also the structure.

Dynamic SQL enables you to dynamically determine (based on runtime parameters) which parts of a SQL statement are sent to the database. If a user leaves a search field empty, for example, you can simply omit the part of the WHERE clause that refers to that field. Or, if a user doesn't specify a sort order, the entire ORDER BY clause can be omitted.

Dynamic SQL is implemented in ColdFusion by using CFIF, CFELSE, and CFELSEIF tags that control how the SQL statement is constructed, for example:

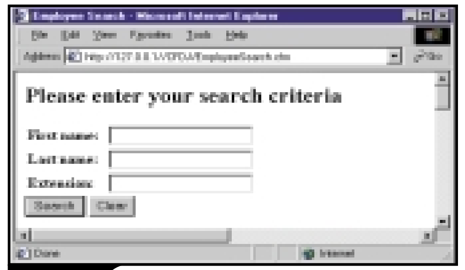
```
<CFQUERY NAME="queryname" DATASOURCE="data
  sourcename">
    ...Base SQL statement

    <CFIF value operator value >
         ...additional SQL
    </CFIF>
</CFQUERY>
```

The CFQUERY tag doesn't allow the developer to view the SQL before it's sent to the database. Knowing the SQL can be helpful when it's all built dynamically and involves a lot of tables and conditions. Since it's not available as a variable, there's no way to log a SQL if necessary.

## Demonstration Example

I've used a simple form (see Figure 1) to illustrate the SQL string-building method that I'm going to propose. This form consists of three fields and searches against the "Employees" table in the database.

There are several commonly used approaches to writing dynamic SQLs in ColdFusion that can help solve the problem introduced in this article.


Employee Search - Microsoft Internet Explorer

Please enter your search criteria

First name:
Last name:
Extension:

Search  Clear

FIGURE 1: The employee search screen

### Traditional Approach

One approach, which I call the *traditional approach*, is writing the SQL within CFQuery tags and using <CFIF>. Dynamic SQLs are written as shown in Listing 1. Using this method, the executed SQL isn't available as a string. To get the SQL string into a variable, developers often resort to the complex approach.

### Complex Approach

This approach complicates the writing of the SQL as demonstrated in Listing 2. In this solution we're building the SQL as a separate string, making it easy to log. Simply write a custom tag that takes this string as input and log it in the database or a text file.

This certainly doesn't seem neat. Moreover, the SQL is now more difficult to understand. Fortunately, there's a better way of building the SQL string without deviating much from the traditional approach of writing dynamic SQLs in ColdFusion. Before we look at this new approach, let's understand how nested tags work in ColdFusion.

### Nested Tags

ColdFusion lets you turn a custom tag into a special container that can enclose additional custom tags, thus allowing you to nest tags. Base tags are also known as *parent* tags, while the tags that base tags call are known as *sub* tags, or *child* tags.

Nested custom tags operate through three processing modes accessible through the variable "ThisTag.ExecutionMode". Table 1 shows the various modes.

It's beyond the scope of this article to cover nested tags in detail. For further reading on nested tags, refer to Allaire CF documentation (Chapter 7: Reusing Code).

| | |
|---|---|
| Start mode | The base tag is processed for the first time |
| Inactive mode | Sub tags and other code contained within the base tag are processed |
| End mode | The base tag is processed a second time |

TABLE 1 Mode processing

### Generated Content

The term *generated content* in the context of a nested tag paradigm refers to the portion of the results that's generated by the body of a given tag. It also includes any results generated by the child tags. The generated content is available as a variable called ThisTag.GeneratedContent.

Custom tags can access and modify the generated content of any of their instances using the ThisTag.GeneratedContent variable, which is always empty during the processing of a start tag.

## Building a SQL String Using Nested Tags

We've seen how difficult it is to build a dynamic SQL as a string using the complex approach explained earlier. Using "nested tags" simplifies the whole process. Writing SQL using this nested tag approach can be accomplished in the following steps.

**Step 1:** ***Write the SQL***

The SQL will be coded the normal way (traditional approach), but without the CFQuery tags around it (see Listing 3).

**Step 2:** ***Write the Base Tag***

Let's define a base tag called "Basetag.cfm", implemented as follows:

```
<CFIF ThisTag.ExecutionMode EQ "END">
     <CFSET GenContent =
       ThisTag.GeneratedContent>
     <CFSET Caller.SQLString =
       REReplace(GenContent,'[[:space:]]',
       " ","ALL")>
     <CFSET ThisTag.GeneratedContent = "">
</CFIF>
```

This tag collects the "ThisTag.GeneratedContent" variable and replaces all multiple spaces with single spaces. Notice that only the "End" mode is coded. It's during this mode that the SQL string is available.

**Step 3:** ***Wrap the SQL with the Base Tag***

Wrapping the SQL with the "BaseTag" puts the SQL as the generated content. Hence it can be accessed in the "End" mode.

```
<CF_BASETAG >
     … The SQL written in ‘step 1’ goes in
     here…
</CF_BASETAG >
```

**Step 4:** ***Write the CFQuery Tag***

The final code for the SQL now looks like this:

```
<CF_BASETAG >
     … The SQL written in ‘step 1’ goes in
     here…
</CF_BASETAG >

<!--- ‘SQLString’ is the variable returned
 by the ‘BaseTag.cfm’ custom tag. - →
<CFQUERY DATASOURCE="A2Z"
  NAME="EmployeeList">
     #PreserveSingleQuotes(SQLString)#
</ CFQUERY >
```

## Conclusion

The powerful nested tag concept enables us to log SQLs. Since the SQL string is available as a variable, it can be used for debugging purposes too. Using this new approach, developers will have access to the SQL string even before the SQL is sent to the database, a useful debugging tool.

About the Author

*Kedar Desai works with differential technologies in Fairfax, Virginia. He leads a team that focuses on combining Allaire technologies and Java to build applications using the open standards of the Web.*

desaik@yahoo.com

**Listing 1: Traditional approach to write dynamic SQLs in ColdFusion**

```
<CFSET WhereClause = "WHERE ">
<CFSET Cnj = "">
<CFQUERY DATASOURCE="A2Z" NAME="EmployeeList">
     SELECT FirstName, LastName, PhoneExtension, EmployeeID
         FROM Employees

     <CFIF FORM.FirstName IS NOT "">
           #WhereClause# #Cnj# FirstName LIKE
              '#Form.FirstName#%'
           <cfset WhereClause = "">
           <cfset Cnj = "AND ">
     </CFIF>

     <CFIF FORM.LastName IS NOT "">
           #WhereClause# #Cnj#  LastName LIKE
              '#Form.LastName#%'
           <cfset WhereClause = "">
           <cfset Cnj = "AND ">
     </CFIF>

     <CFIF FORM.PhoneExtension IS NOT "">
           #WhereClause# #Cnj#  PhoneExtension LIKE
              '#Form.PhoneExtension#%'
           <cfset WhereClause = "">
           <cfset Cnj = "AND ">
     </CFIF>
     ORDER BY LastName, FirstName
</CFQUERY>
```

**Listing 2: Complex approach to writing dynamic SQLs in ColdFusion**

```
<CFSET WhereClause = "">
<CFSET Cnj = "">

<CFSET SelectStatment = "SELECT FirstName, LastName,
 PhoneExtension, EmployeeID FROM Employees">

<CFIF FORM.FirstName IS NOT "">
     <CFSET Value = FORM.FirstName>
     <CFSET WhereClause =
          "#WhereClause#"&"#Cnj#"&"FirstName" &" = "&"'
           #Value#'">
     <CFSET Cnj = " AND ">
</CFIF>

<CFIF FORM.LastName IS NOT "">
     <CFSET Value = FORM.LastName>
     <CFSET WhereClause = "#WhereClause#"&"#Cnj#"&
           "LastName" &" = "&"'#Value#'">
     <CFSET Cnj = " AND ">
</CFIF>

<CFIF FORM.PhoneExtension IS NOT "">
     <CFSET Value = FORM.PhoneExtension>
     <CFSET WhereClause = "#WhereClause#"&"#Cnj#
            "&"PhoneExtension" &" = "&"'#Value#'">
     <CFSET Cnj = " AND ">
</CFIF>
<CFIF WhereClause IS NOT "">
      <CFSET SelectWithWhere = SelectStatment & " WHERE " &
       "#WhereClause#">
      <CFSET SelectStatment = SelectWithWhere>
</CFIF>

<CFSET  SelectStatment = SelectStatment & " ORDER BY Last
      Name, FirstName">

<CFQUERY DATASOURCE="A2Z" NAME="EmployeeList">
      #Preservesinglequotes(SelectStatment)#
</CFQUERY>
```

**Listing 3: Writing the SQL**

```
CFSET WhereClause = "WHERE ">
<CFSET Cnj = "">
<CF_BASETAG >
      <CFOUTPUT>
            SELECT FirstName, LastName, PhoneExtension,
              EmployeeID
               FROM Employees

        <CFIF FORM.FirstName IS NOT "">
            #WhereClause# #Cnj# FirstName LIKE
              '#Form.FirstName#%'
                      <cfset WhereClause = "">
                      <cfset Cnj = "AND ">
                 </CFIF>
                 <CFIF FORM.LastName IS NOT "">
                   #WhereClause# #Cnj#  LastName LIKE
              '#Form.LastName#%'
                       <cfset WhereClause = "">
                       <cfset Cnj = "AND ">
                 </CFIF>
                 <CFIF FORM.PhoneExtension IS NOT "">
                   #WhereClause# #Cnj#  PhoneExtension LIKE
              '#Form.PhoneExtension#%'
                      <cfset WhereClause = "">
                      <cfset Cnj = "AND ">
                 </CFIF>
            ORDER BY LastName, FirstName
      </ CFOUTPUT><
</CF_BASETAG >

<!--- ‘SQLString’ is the variable returned by the
‘BaseTag.cfm’ custom tag. -‡
<CFQUERY DATASOURCE="A2Z" NAME="EmployeeList">
      #PreserveSingleQuotes(SQLString)#
</ CFQUERY >
```

CODE
LISTING
The code listing for
this article can also be located at
www.ColdFusionJournal.com

# Testing for Smarties

## Test harnesses guide better code writing

BY
HAL
HELMS

*f people only knew how hard I work to gain my mastery, it wouldn't seem so wonderful at all.* — Michelangelo

Aristotle is the man usually credited with the invention of the logical device known as a *syllogism*. The syllogism takes two (hopefully) undeniable premises set forth in such a way that the conclusion is inescapable. Here is its classic formulation:

**Premise 1:**
All men are mortals.
**Premise 2:**
Socrates is a man.
**Conclusion:**
Therefore, Socrates is mortal.

On that simple foundation (premise plus premise yielding to conclusion) is built much of the last 2000 years of Western thought.

But new times call for new thinking, and I'd like to offer this updated syllogism:

**Premise 1:**
All programs have bugs.
**Premise 2:**
You write programs.
**Conclusion:**
Therefore, your program has bugs.

This syllogism's regrettably inescapable conclusion leads me to this month's topic: testing.

Testing is not something that comes naturally to most of us. I'd say the surest mark of an inexperienced programmer is an unwillingness to comment on his or her code, but it's followed closely behind by an aversion to testing it.

It's probably not coincidental that both of these tasks suffer from the "banana" problem – named after the little girl who told her teacher, "I know how to spell banana. I just don't know when to stop."

Everyone knows they should test their code, but where do you begin – and where do you stop? I think it will be helpful to peel this particular banana by looking more closely at what we mean by the term *testing*.

### Testing and Debugging

While there are some obvious connections between debugging and testing, they're two separate disciplines. *Debugging* tries to get the program running. *Testing* is more comprehensive. Debugging tells us that there are no obvious errors. Testing probes that contention, seeing if the correct execution of our program can be depended on, or is merely a happy coincidence.

The best testing is planned into the design of our program. Such programs are written so that, at any point during the development (and perhaps even afterwards), tests can be run.

In this article I'm going to relate my own testing techniques. Because I use Extended Fusebox for developing applications, the test methods I use will be Extended Fusebox-centric, but good testing principles don't subscribe to any one methodology and can be adapted to what you find most helpful.

### Levels of Testing

The most basic level of testing is the unit test. This is testing done at the level of the component (in Fusebox, this would be a fuse). What unit testing tells us is that this "unit" works as it should. It makes no larger claims about the application.

Very often unit testing is skipped or is done in an ad hoc fashion. Whatever time savings may accrue usually turns out to be a false economy. What should have been a simple case of a failed test on a single unit becomes an elusive bug in a larger component, costing far more in money and project delays to find and fix it.

Integrated testing describes the attempts to "wire together" the various lower-level units to see if, together, they work as advertised. It may be done at a subassembly level (a circuit application in Fusebox) if the application size warrants it, and it's always done on the release candidate for the finished application.

Acceptance testing is done with the client to verify that the application does what it's supposed to do. An acceptance test plan provides clear guidance on what tests should be run and what these tests should reveal in order for the project to be successfully finished. If you've ever been plagued with feature creep after delivery, acceptance tests provide a fixed point against which to evaluate any requests. It provides protection against our own version of the banana problem: "I know how to program an application. I just don't know when to stop."

### Test What?

At all three levels, we must know what to test. Otherwise we're likely to simply click the "Browse" button on ColdFusion Studio and approve it if we don't see any obvious errors. In the remainder of this article, I'm going to restrict myself to discussing unit testing. What, though, should we unit test for?

You may want to consider the idea of "testing against a contract." As the name suggests, this means that you have a contract against which to test. We can test against a contract at unit, integrated, and acceptance test levels, though the format of the contract may be quite different. Here's a contract for a simple fuse (unit level):

```
If I receive an "itemID", I update
the "Item" table with "price" and
"quantity" passed to me. Otherwise,
I insert a new item in the database.
```

If you use Fusedocs, this contract is likely to seem very familiar. It's the statement of responsibilities found in every Fusedoc. (For more reading on the Fusedoc specification, go to www.halhelms.com.)

Such a statement of responsibilities is important not only for testing, but for writing the code. Without it, how will the programmer know what to write – how to fulfill the contract, as it were?

Unlike the ones we're all too familiar with, these contracts aren't lengthy, obfuscated documents. They should be written in such a way that a competent coder can write the code without knowing anything more about the application. The more modular the code, the shorter the contract needs to be and the easier it is to write to and test against.

While Fusedocs make excellent contracts at unit level, I rely on prototyping and client feedback via DevNotes (more info at www.halhelms.com) to provide guidance on what the tests should test for.

## Test Harnesses

No matter what level of testing – unit, integrated, or acceptance – successful tests are those that can be run again and again. It is key to successful projects – certainly successful large projects – that we have a high degree of confidence in the code that's already written. When you write all the code yourself, this may not seem much of an issue, but when other people are involved, it becomes a necessity.

We are all used to ad hoc tests – tests involving alterations to the code itself, such as <cfoutput>ing variable values. Here's an example of an ad hoc test:

```
<!--- TEST CODE HERE--->
<cfoutput>
 <cflock scope="SESSION" timeout="5">
  The value of session.userID is
#session.userID#<br>
 </cflock>
</cfoutput>
```

While this is a very helpful means of testing, we want a more flexible and powerful technique. This technique is the *test harness*. The harness "fits over" our code, letting us run automated tests whenever we wish. Here's one example of the simplest test harness:

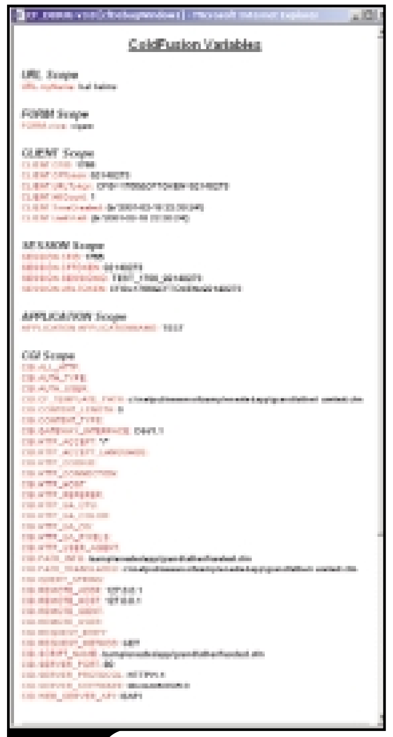
```
<cfinclude template="myCFMLpage.cfm">
```

It's nothing more than a file that includes the file to be tested. Of course, in real life, you'd probably never encounter so simple a test harness. After all, real code files often expect arguments to be passed in, and will break if they're absent. So, here's a more realistic – though still simple – test harness:

```
<cfset self="FuseTest.cfm">

<cfapplication name="TEST" session
    management="Yes" clientmanagement=
"Yes">

<cflock scope="SESSION" timeout="10"
   type ="EXCLUSIVE">
 <cfset session.userID = '111'>
</cflock>

<cfinclude template="myCFMLpage.cfm">
```

Fusebox links and forms return to that main page (the fusebox). But right now, we don't want them returning there. When we're doing unit testing, we may be sitting on a beach somewhere warm and not have access to the fusebox.

Even if this much-to-be-desired scenario isn't the case, the fusebox


ColdFusion Variables

URL Scope

FORM Scope

CLIENT Scope

SESSION Scope

APPLICATION Scope

CGI Scope

FIGURE 1: ColdFusion variables

isn't the ideal file to use when testing. It doesn't provide us with the kind of feedback we need for testing. Luckily, we can create a special file (I call mine *FuseTest*) that gives us more information. But I'm getting ahead of myself. I need to explain a little more about my Extended Fusebox methodology so you can follow what's happening.

Using Extended Fusebox, I don't hard-code links to the central file, but instead use the alias "self". Here's an example:

```
<a href="#self#?fuseaction=some_fuse
  action">click me</a>

<form action="#self#?fuseaction=
some_other_fuseaction" method="post">
...
</form>
```

This looks odd at first sight. Fuseboxers are more likely to recognize the above code in this formulation:

```
<a href="index.cfm?fuseaction=
some_fuseaction">click me</a>

<form action="index.cfm?fuseaction=
some_other_fuseaction" method="post">
...
</form>
```

There are two reasons for using the alias. First, we want our fusebox file to be the default page called when a user enters "www.myDomain.com," but how do we know that the default page on the server(s) on which our applications will be deployed will be index.cfm? Since we don't, using the alias "self" lets us resolve this at deployment.

The other reason is for just this case – when we're testing. Instead of having the fuse submit to the fusebox, we want it to submit to a page that will display the variables being created by and/or sent with this fuse. Thus we can verify that our code is adhering to the contract set out for it. All that's required is pointing "self" to FuseTest.cfm. FuseTest.cfm uses Dan Switzer's excellent tag, Debug.cfm, available at Allaire's Developer Exchange, http://devex.allaire.com/developer/gallery.

When the test harness is run, the resulting fuse will head back to FuseTest, which will call Dan's custom tag and we'll see exactly what our fuse has been up to (see Figure 1).

Now I can test the output of my fuse with the contract in my Fusedoc to make sure the fuse is doing what it's supposed to.

In addition to setting "self", you also need to set any parameters the file being unit tested is expecting. How do you know what needs to be set, apart from poring over the file? Well, if you use Fusedoc, the "attributes" section will provide you with information. Regardless of how you get the information, you'll need to have these params set prior to testing the unit file.

Finally, you call the actual file to be tested, "myCFMLpage.cfm", in the example I gave. I save these test harnesses using a "tst" (or "tst_" if you like using underscores) prefix. They can go in the same directory as the actual files themselves, ready to be used whenever tests need to be run.

## Testing Parameter Values

We can also test how robust our code is in dealing with unexpected parameter values. Suppose, for the sake of illustration, we are expecting a single parameter to be passed to our fuse. Here's the attributes section of the Fusedoc from "myCFMLpage.cfm":

```
--> attributes.quantity: an INTEGER
```

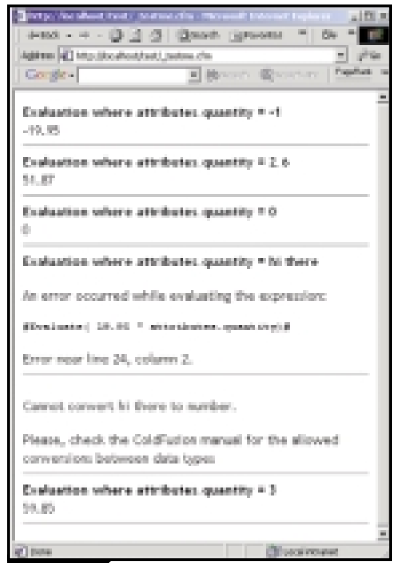
If this is completely foreign to you, here's a translation: a parameter called *quantity*, of the scope "attributes", will be passed into the fuse. The datatype of this variable will be an integer. Here's a snippet from "myCFMLpage.cfm":

```
#Evaluate( 19.95 * attributes.quantity)#
```

What kinds of things do we test for? Well, among others, we want to know how the application will handle things if the "quantity" passed in is one of these values:

- -1: A negative number
- 2.6: A noninteger number
- 0
- "hi there": A nonnumeric value
- 3: A valid value

We can rework our test harness to loop through each of these values, looking to see how the fuse handles it. Here's what the test harness would look like:


Evaluation where attributes.quantity = -1
-19.95

Evaluation where attributes.quantity = 2.6
51.87

Evaluation where attributes.quantity = 0
0

Evaluation where attributes.quantity = hi there

An error occurred while evaluating the expression:

#Evaluate( 19.95 * attributes.quantity)#

Error near line 24, column 2.

Cannot convert hi there to number.

Please, check the ColdFusion manual for the allowed conversions between data types

Evaluation where attributes.quantity = 3
59.85

FIGURE 2: Localhost test

```
<cfset quantity_List = "-1,2.6,0,hi
  there,3">

<cfloop list="#quantity_List#"
  index="aValue">
 <cfset SetVariable(
'attributes.quantity', aValue )>

 <cfoutput>
  <b>Evaluation where
attributes.quantity = #aValue#</b>
  <br>
 </cfoutput>

 <cftry>
 <cfinclude
template="myCFMLpage.cfm">
 <cfcatch>
  <cfoutput>
   #cfcatch.message#
  </cfoutput>
 </cfcatch>

 </cftry>
<hr>
</cfloop>
```

We start by putting all the values we want to try in a list. Then we loop over the list, enclosing our file to be unit tested in <cftry><cfcatch> blocks to trap any errors. Figure 2 shows the output we get.

Of course, in a real-world case, you usually wouldn't have only one variable being passed in. In that case you can use nested loops to test for every possible permutation.

```
<cfset attributes_quantity =
"-1,2.6,0,hi there">
<cfset attributes_myName =
  "Adam,Steve,Nat,Hal">

<cfloop list="#attributes_quantity#"
  index="aQuantity">
 <cfloop list="#attributes_myName#"
  index="aMyName">
  <cfset attributes.quantity = aQuantity>
  <cfset attributes.myName = aMyName>
  <cfoutput>
   <b>Results with following parameter
values:<br>
   quantity: #aQuantity#<br>
   myName: #aMyName#<br></b>
   <br>
   <cftry>
   <cfinclude
template="myCFMLpage.cfm">

   <cfcatch>
    #cfcatch.message#<br>
   </cfcatch>
   </cftry>
  </cfoutput>
  <hr>
 </cfloop>
</cfloop>
```

## Waiting for <CFDWIM>

Some time ago I asked Sim Simeonev, Allaire's chief architect, if he would add just one tag to ColdFusion. All I wanted was a <CFDWIM> tag.

"DWIM?" he asked.

"Yes, DWIM. It's an acroynm for Do What I Mean."

Sim promised me he would work on it. Until then, we're going to have to test our code. Regardless of the methodology you use for writing code, you need to ensure that your code works, and is robust enough to handle unwanted situations.

I've found that making myself write test harnesses influences the way I write code. Now I write it with an eye to testing it, and that shift in thinking has helped me write better code. One thing is sure: testing applications at the unit level allows the entire project to proceed faster and with less risk of running into error landmines when we later integrate the code. I hope you find some of these techniques and ideas help you write better code.

ABOUT THE AUTHOR
*Hal Helms (www.halhelms.com) is a Team Allaire member who provides both on-site and remote training in ColdFusion and Fusebox.*

HAL.HELMS@TEAMALLAIRE.COM

# Rolling Your Own…

## Creating custom tags in ColdFusion

BY KEVIN SCHMIDT

Ever had a complex piece of code you wanted to separate from the rest of your application? What about a piece of business logic you want to use from different templates on your site?

You can address these issues and others by writing your own custom tags in ColdFusion. Let's take a step-by-step journey through the process.

### What, Exactly, Is a Custom Tag?

A custom tag is simply a ColdFusion template. Just like any other CF template, you can use any of the ColdFusion tags or functions when creating it. You can store your custom tag in the local directory, making it available to templates in that directory only, or you can store it in the CustomTags directory (C:/CFUSION/CustomTags), thus making it available to every template.

When a custom tag is called, ColdFusion first looks in the directory where the calling template is located. If it doesn't find the tag there, ColdFusion checks the CustomTags folder in the aforementioned directory. Now that we know what custom tag is, where it goes, and what it can do, let's get creative.

### Creating a Custom Tag

Let's say we have an application that calculates mortgage payments. Rather than include the code directly in the template, we can write a custom tag to handle the calculation.

First we create the file. I created a blank page and saved it as Mortgage_Calculator.cfm in the CustomTags folder, thus making it available to any template on the site.

Now that we have the template for our custom tag set up, we'll write the code for collecting the necessary information and processing it.

### Collecting the Information

To collect the information needed, I built a form for users to input the specifics about their mortgage. It collects the amount, term (in years), and interest rate of the loan. These variables must be passed to the custom tag in order to calculate the payment.

Once the information is collected, a user clicks the "calculate" button and calculate.cfm, the page specified in the action attribute of the form, is executed (see Listing 1).

### Calling the Custom Tag

The calculate.cfm template calls the custom tag on the first line. (Listing 2 contains the calulate.cfm code.) Let's take a closer look at the code for calling the tag.

```
<CF_Mortage_Calculator
AMOUNT="#form.amount"
TERM="#form.term#"
RATE="#form.rate#">
```

When we call a custom tag, we reference it using syntax such as <CF_yourtemplate>. Thus, by typing <CF_Mortage_Calculator>, we're actually calling the Mortgage_Calculator.cfm template in the CustomTags directory.

Custom tags don't have access to read-write variables other than the ones that are passed to it. They do, however, have access to all the "read only" variables such as CGI variables. In light of this, we must specify the attributes that are to be passed so as to get our values into the custom tag. Once passed, these values are referenced in the custom tag using the attributes scope. Thus, when referencing the variable rate, we must refer to it as #attributes.rate#.

Let's move on to our Mortgage_Calculator.cfm template. (Listing 3 contains the code for Mortgage_Calculator.cfm.)

### <CF_Mortgage_Calculator>

As you can see, the first thing we do is set up our variables; this keeps us from having to reference them using the attributes scope. The code below executes this task:

```
<cfset amount = attributes.amount>
<cfset term = attributes.term>
<cfset rate = attributes.rate>
```

Once these variables are set, we're ready to do the actual calculation of the monthly payment. (For this example, we're taking only principal and interest into account; we'll leave out factors such as property taxes, PMI, and escrow money.)

To do the calculation, it's necessary to change the term and rate values. The term should be in months instead of years, and the interest rate should be broken down into its monthly percentage. The code below accomplishes this:

```
<cfset rate = ((rate/100.0)/12)>
<cfset payments = term * 12>
```

Now that the variables are in the proper format, we'll plug them, along with the amount of the loan, into the standard equation for calculating mortgage payments. In case you're wondering what that is, I've included it below:

```
Monthly Payment = amount * payment /
1 - (1 / (1 + rate) ^ payments)
```

The code for the ColdFusion calculation looks very similar to the calculation above, with only a few changes. See for yourself:

```
<cfset monthlypayment = round((amount
* rate / (1 - (1 / (1 + rate) ^ pay-
ments))) * 100)>
```

The major difference is the addition of the round() function. When we present the payment to users, it would be nice to show them the value rounded to two decimal places. Unfortunately, ColdFusion's round() function only rounds to the nearest whole integer. Thus we lose our two decimal places. This can be overcome easily, however, by multiplying the monthlypayment variable by 100 before using the round() function. Thus, if our payment is supposed to be $504.14, the monthlypayment variable would hold the value 50414.

We're almost ready to return the monthlypayment variable to the calling page; first, though, we'll have to format and scope it properly.

### Returning Values to the Calling Template

Just as custom tags don't have access to variables from the calling template, the calling template doesn't have access to variables within the custom tag. Once again we must use the appropriate scope if we want to reference variables from the custom tag in the calling template.

This is accomplished by setting the desired variable we want to reference to the caller scope. Returning to the Mortgage_Calculator, we'll format our monthlypayment variable and scope it properly. The following code illustrates the procedure:

```
<cfset caller.monthlypayment =
"$#left(monthlypayment,(len(monthly-
payment) - 2))#.#right(monthlypay-
ment, 2)#">
```

The monthlypayment variable is first set to the caller scope. Next we use the left() and right() functions to format the variable to two decimal places; then we add the decimal point and the dollar sign.

When we reference the variable in our calling template now, we should get $504.14. When we want to display this variable in our calling template, we reference it as #caller.-monthlypayment#.

### Conclusion

Custom tags are powerful and versatile. While our example is a simple one that passes values and performs a basic mathematic equation, any number of tasks can be accomplished by using custom tags. I hope I've provided the inspiration for you to create your own. To help you I've set up a demo of this article's example at www.fusemonkey.com/cf_mortgage_calculator/example1.cfm. Give it a shot!

ABOUT THE AUTHOR
*Kevin Schmidt, an Allaire-certified ColdFusion developer, is Web technology manager for pwb, inc., an integrated marketing/information design firm in Ann Arbor, Michigan. He spends most of his time developing Web applications for clients using ColdFusion and Oracle.*

SCHMIDT@PWB.COM

**Listing 1**

```
<!--- HTML form to accept the users input and pass it to
the mortgage2.cfm template --->
<table width="175" cellpadding="0" cellspacing="0">
 <tr>
  <td width="175" bgcolor="#000000">
  <table width="175" cellpadding="3" cellspacing="1">
   <tr>
    <td width="175" bgcolor="#000080" align="center">
    <font size="2" face="Arial, Helvetica"
color="#ffffff"><strong>Mortgage Calculator</strong></font>
    </td>
   </tr>
   <form name="mortgage" method="post"
action="mortgage2.cfm">
   <tr>
    <td width="175" bgcolor="#dae1e9">
    <font size="1" face="Arial, Helvetica">Loan Amount<br>
    <input type="text" name="amount" size="20"><br>
    (ex: 120000)
    </font>
    </td>
   </tr>
   <tr>
    <td width="175" bgcolor="#ffffff">
    <font size="1" face="Arial, Helvetica">Loan Term (In
years)<br>
    <input type="text" name="term" size="20"><br>
    (ex: 30)
    </font>
    </td>
   </tr>
   <tr>
    <td width="175" bgcolor="#dae1e9">
    <font size="1" face="Arial, Helvetica">Loan Interest
Rate<br>
    <input type="text" name="rate" size="20"><br>
    (ex: 7.6)
    </td>
   </tr>
   <tr>
    <td width="175" align="center" bgcolor="#000080">
    <input type="submit" value="Calculate">
    </td>
   </tr>
  </table>
  </td>
 </tr>
 </form>
</table>
```

**Listing 2**

```
<!--- Call the custom tag and pass the attributes from the
form --->
<cf_mortgage_calculator amount="#form.amount#"
term="#form.term#" rate="#form.rate#">

<!--- HTML to display the output --->
<table width="175" cellpadding="0" cellspacing="0">
 <tr>
  <td width="175" bgcolor="#000000">
  <table width="175" cellpadding="3" cellspacing="1">
   <tr>
    <td width="175" bgcolor="#000080" align="center">
    <font size="2" face="Arial, Helvetica"
color="#ffffff"><strong>Monthly Payment</strong></font>
    </td>
   </tr>
   <tr>
    <td width="175" bgcolor="#dae1e9" align="center">
    <font size="3" face="Arial, Helvetica"><strong><cfout-
put>#monthlypayment#</cfoutput><font size="2" face="Arial,
Helvetica"><sup>(1)</sup></font></strong></font>
    </td>
   </tr>
  </table>
 </tr>
 <tr>
  <td width="175" bgcolor="#FFFFFF">
  <font size="2" face="Arial,
Helvetica"><sup>(1)</sup></font>
  <font size="1" face="Arial, Helvetica">Principal and
interest only.</font>
  </td>
 </tr>
</table>
```

**Listing 3**

```
<!--- Set up variables --->
<cfset amount = attributes.amount>
<cfset term = attributes.term>
<cfset rate = attributes.rate>

<!--- do the calculation --->

<!--- get the monthly interest rate --->
<cfset rate = ((rate / 100)/ 12)>

<!--- set number of total payments --->
<cfset payments = term * 12>

<!--- calculate and round the monthly payment --->
<cfset monthlypayment = round((amount * rate / (1 - (1 /
(1 + rate) ^ payments))) * 100)>

<!--- format the amount in $XXXX.XX format and scope it so
the calling template can read it --->

<cfset caller.monthlypayment =
"$#right(monthlypayment,(len(monthlypayment) -
2))#.#left(monthlypayment, 2)#">


<!--- end --->
```

CODE LISTING

The code listing for this article can also be located at

www.ColdFusionJournal.com

# Building a Better Custom Tag

## Are you up to the challenge?

BY
BEN
FORTA

We've come a long way in the past couple of years. Not that long ago I was teaching how to write simple custom tags and encouraging developers to experiment with them.

Now Allaire's Developers Exchange has thousands of custom tags listed, and I see developers using them as part of everyday development. Considering that it's been only a few years since Jeremy Allaire asked me to write a few tags so his new Tag Gallery would contain some initial content, we've made real progress.

So now I'd like to up the ante a bit and challenge developers to build a better mousetrap (so to speak). Most of the newly available tags are new twists on old ideas, often the same old way of doing things. I'd like to see developers get creative with tag designs, making them as flexible and as reusable as possible. The better the tag design, the better the abstraction, the better the encapsulation, the better the interface – the more uses you'll find for it. That's really what custom tags are all about.

### Building Basic Tags

The best way to explain what I mean is with an example, so here goes.

I've been doing lots of WAP work recently (as I know many of you have). When generating WAP content you've probably discovered that $ (the dollar sign) is a special character in WML – it's used to prefix variables. Any time $ is used in text that's not a variable, it has to be escaped as $$ (kind of like # is escaped as ## in CFML). I found myself using code like this throughout my application:

```
#Replace(variable, "$", "$$", "ALL")#
```

This Replace() function simply replaces all "$" with "$$". Now, when CF5 ships I'll create a user-defined function called WAPSafe() that will likely look something like this:

```
<CFSCRIPT>
function WAPSafe(string)
{
 return Replace(string, "$", "$$",
"ALL");
}
</CFSCRIPT>
```

I'd then be able to simply format text as follows:

```
#WAPSafe(var)#
```

But user-defined functions are not available yet, so I created a simple custom tag using the code shown below:

```
<!--- Initialize defaults --->
<CFPARAM NAME="ATTRIBUTES.text"
DEFAULT="">

<!--- Process text --->
<CFOUTPUT>#Replace(ATTRIBUTES.text,
"$", "$$", "ALL")#</CFOUTPUT>
```

This tag, named <CF_WAPSafe>, takes a single attribute that contains the text to be processed. The code within the tag simply displays the processed text. To call this tag I'd do something like this:

```
<CF_WAPSafe TEXT="#var#">
```

So far so good.

### Dual Purpose Tags

Custom tags that process data shouldn't arbitrarily write output text. For maximum control, custom tags should return results to the caller page and let the caller code do whatever it needs with the data.

Here's the revised <CF_WAPSafe> tag:

```
<!--- Initialize defaults --->
<CFPARAM NAME="ATTRIBUTES.text"
DEFAULT="">
<CFPARAM NAME="ATTRIBUTES.variable"
DEFAULT="WAPSAFE">

<!--- Process text --->
<CFSET
output=Replace(ATTRIBUTES.text, "$",
"$$", "ALL")>

<!--- Save to CALLER variable --->
<CFSET "CALLER.#ATTRIBUTES.vari-
able#"=output>
```

The tag takes two attributes: the text to be processed and the name of a variable to be created in the CALLER scope (arbitrarily naming variables is bad form; the caller should always be able to specify the name of the variable to be created). The custom tag processes the data and then saves the converted text into a variable. To use this tag I'd do the following:

```
<CF_WAPSafe TEXT="#var#"
VARIABLE="wap_var">
<CFOUTPUT>#wap_var#</CFOUTPUT>
```

There are now two versions of the same tag: one that saves converted text to a variable and one that displays it. Even though best practices dictate that saving the results to a variable is preferred, there may be occasions when I'd want to have the processed text dumped to the output. So why not support both modes of operation as follows?

```
<!--- Initialize defaults --->
<CFPARAM NAME="ATTRIBUTES.text"
DEFAULT="">
<CFPARAM NAME="ATTRIBUTES.variable"
DEFAULT="">

<!--- Process text --->
<CFSET
output=Replace(ATTRIBUTES.text, "$",
"$$", "ALL")>

<!--- Check if VARIABLE passed --->
<CFIF ATTRIBUTES.variable IS "">
```

```
   <!--- Display it --->
   <CFOUTPUT>#output#</CFOUTPUT>
<CFELSE>
   <!--- Save to CALLER variable --->
   <CFSET "CALLER.#ATTRIBUTES.vari-
able#"=output>
</CFIF>
```

This custom tag is the same as the previous version except that the final <CFIF> statement checks if the VARIABLE attribute was specified. If it was, the processed data is saved to the specified variable; otherwise it's displayed using a regular <CFOUTPUT> block.

With minimal extra work the custom tag is now a bit more useful.

## The Tag Pair Solution

There's another version of this custom tag that would be useful. Instead of passing the text to be processed as a variable, it would be nice to be able to simply enclose it within tags, such as:

```
<CF_WAPSafe>
...
</CF_WAPSafe>
```

This way I could use all sorts of functions, expressions, tags, whatever I needed, and be sure that the generated output was safely formatted.

ColdFusion makes writing this type of custom tag very easy; here's the code for the new version:

```
<!--- Only process in End mode --->
<CFIF ThisTag.ExecutionMode IS "End">
   <!--- Process text --->
   <CFSET output=Replace(ThisTag.
GeneratedContent, "$", "$$", "ALL")>
   <!--- Update content --->
   <CFSET
ThisTag.GeneratedContent=output>
</CFIF>
```

This is a great example of using ColdFusion tag pairs. The tag itself is called twice, once when <CF_WAPSafe> (the start tag) is processed and again when </CF_WAPSafe> (the end tag) is processed. When the start tag is processed, the tag is called and ThisTag.ExecutionMode will be "Start". When the end tag is processed, the tag will be called again and ThisTag.ExecutionMode will be "End". Obviously, the custom tag needs to process the text when the end tag is reached (there would be nothing to process when the start tag is invoked) and so the entire code block is wrapped within a <CFIF> statement that checks that This-Tag.Execution-Mode is "End".

The code itself is simple. This-Tag.GeneratedContent contains all the final postprocessing content present between the start and end tags, so Replace() processes that variable. Then ThisTag.Generated-Content is overwritten with the processed text.

Clean and simple.

## Putting It All Together

I now have two versions of the tag: one that's a single tag that takes passed data and one that's used as a tag pair to process the enclosed content. Can these two tags be merged? Look at this code:

```
<!--- Only process if stand alone or
end of pair --->
<CFIF (NOT ThisTag.HasEndTag) OR
(ThisTag.ExecutionMode IS "End")>

   <!--- Initialize defaults --->
   <CFPARAM NAME="ATTRIBUTES.text"
DEFAULT="">
   <CFPARAM NAME="ATTRIBUTES.vari-
able" DEFAULT="">

   <!--- Get input --->
   <CFIF ThisTag.HasEndTag AND
ThisTag.ExecutionMode IS "End">
       <!--- If tag pair get text
between tags --->
       <CFSET
input=ThisTag.GeneratedContent>
   <CFELSE>
      <!--- Otherwise use passed
attribute --->
      <CFSET input=ATTRIBUTES.text>
   </CFIF>

   <!--- Process output text --->
   <CFSET output=Replace(input, "$",
"$$", "ALL")>

   <!--- Check if VARIABLE passed --->
   <CFIF ATTRIBUTES.variable IS "">
      <!--- No variable passed --->
      <CFIF ThisTag.HasEndTag>
         <!--- If tag pair update
text between tags --->
         <CFSET
ThisTag.GeneratedContent=output>
      <CFELSE>
         <!--- Otherwise display
output --->
         <CFOUTPUT>#output#</CFOUT-
PUT>
      </CFIF>
   <CFELSE>
      <!--- Save to CALLER variable
--->
      <CFSET
"CALLER.#ATTRIBUTES.variable#"=out-
put>
      <!--- If tag pair prevent dis-
play --->
      <CFIF ThisTag.HasEndTag>
         <CFSET
ThisTag.GeneratedContent="">
      </CFIF>
   </CFIF>

</CFIF>
```

The custom tag needs to be executed if the tag is called as a simple tag (not part of a tag pair) or as the end tag of a tag pair. The first <CFIF> statement tests for both of these conditions by checking This-Tag.Has-EndTag and ThisTag.Execution Mode. Next comes variable initialization. The text to be processed is retrieved from either the passed attribute or ThisTag.Generated-Content. Then the Replace() operation occurs. Once the text is processed, a determination is made as to whether the data is to be written out or saved to a variable. If the former, the data will be written to either ThisTag.Generated-Content or displayed as is. If the latter, This-Tag.GeneratedContent is flushed if needed.

The result? A single tag that can be used in four different ways. As a simple inline tag:

```
<CF_WAPSafe TEXT="#var#">
```

As a simple tag saving output to a specified variable:

```
<CF_WAPSafe TEXT="#var#"
VARIABLE="output">
```

As a tag pair replacing content automatically:

```
<CF_WAPSafe>
...
</CF_WAPSafe>
```

And one last option (that comes for free), to suppress output and write converted content to a variable:

```
<CF_WAPSafe VARIABLE="output">
...
</CF_WAPSafe>
```

I'll be the first to admit that this simple example probably didn't warrant such sophistication. The truth is, it's probably overkill in this situation. I used this example because it's a simple tag to understand (I didn't want to dedicate pages to explaining the tag being built). The concepts and techniques used here are sound, and the more complex and sophisticated a custom tag is, the more it can benefit from this type of design.

## Where to Go from Here

I've only scratched the surface here. Tag families (parent–child tags) present even more opportunities. For example, consider a custom tag with syntax such as:

```
<CF_Chart NAME="chart.jpg"
QUERY="data" X="1000" Y="250" Z="350"
HEIGHT="240" WIDTH="360"
COLOR="white" TEXTCOLOR="black">
```

There's nothing wrong with the syntax, but what if you were also to support the following:

```
<CF_Chart NAME="chart.jpg"
QUERY="data">
   <CF_ChartParam NAME="X"
VALUE="1000">
   <CF_ChartParam NAME="Y"
VALUE="250">
   <CF_ChartParam NAME="Z"
VALUE="350">
   <CF_ChartParam NAME="HEIGHT"
VALUE="240">
   <CF_ChartParam NAME="WIDTH"
VALUE="360">
   <CF_ChartParam NAME="COLOR"
VALUE="white">
   <CF_ChartParam NAME="TEXTCOLOR"
VALUE="black">
</CF_Chart>
```

There are advantages and disadvantages to each syntax. The former is simpler to write, works well with Tag Editors, and is what new users will feel most comfortable with. The latter is cleaner, supports the conditional inclusion of attributes, and makes working with long lists of attributes much easier.

Which do you support? Why not both? The underlying tag code is the same in both cases; all that differs is how data is passed to the tag and any subsequent initialization code. If designed properly, every attribute should be supported both ways (of course you'll need to find a way to handle conflicting values, but you should be handling that already even for simple tags).

## Summary

As you can see, there's a lot more to custom tags than a simple attribute passing. And there's lots of room for you to be creative when designing your own custom tags. Custom tags are all about encapsulation, black-boxing, and reuse. Well-designed custom tags are powerful, flexible, intuitive, and highly usable. And most important, well-designed tags are used over and over. Are you up to the challenge?

ABOUT THE AUTHOR

*Ben Forta is Allaire Corporation's product evangelist for the ColdFusion product line. He is the author of the best-selling* ColdFusion 4.0 Web Application Construction Kit *and its sequel,* Advanced ColdFusion 4.0 Development, *as well as* Allaire Spectra E-Business Construction Kit *and* Sams Teach Yourself SQL in 10 Minutes. *He recently released* WAP Development with WML and WMLScript.

BEN@FORTA.COM

# Testing Existence in Arrays

## Helpful solutions to challenging problems

BY
CHARLES
AREHART

Have you ever wanted to test if a given array element exists? Or when dealing with arrays of structures, if a given key exists?

Both are challenging if you try to use IsDefined(). This article addresses why IsDefined() fails in both cases when working with arrays.

## Part 1: Testing for Existence with an Array

The first problem arises in the following example: you have an array with four items, but the two in the middle don't have any value (not an empty string, but no array elements at two of the array positions). Consider the following code:

```
<CFSET ACart = ArrayNew(1)>
<CFSET ACart[1] ="Nokia 8150 phone">
<CFSET ACart[4] ="Carrying case">

<CFOUTPUT>
<CFLOOP FROM="1"
TO="#arraylen(ACart)#" INDEX="i">
 Item: #ACart[i]# <br>
</CFLOOP>
</CFOUTPUT>
```

Notice there are only two elements in the array at positions 1 and 4. (Let's not get hung up on why someone would do this. I've hard-coded it this way, but there are a number of programmatic situations in which you'd get an array with elements missing.) An example of this is basing the array location on the value of the table's numeric primary keys when mapping query results to an array.

The point is you may end up inserting data into some array elements, but not others. You're intentionally not adding new elements to the end of the array, because array elements are representing some other data structure. If there are no values at a given point in the original, you don't want them in the array, either.

The main problem is when you loop through the array. During the second iteration, where there's no second array element, you'll get an error trying to refer to ACart[2]:

*The element at position 2 in dimension 1 of object "ACart" cannot be found. The object has elements in positions 1 through 4. Please, modify the index expression.*

The question is: How do you test if a given element exists before trying to use it?

### Testing Existence with IsDefined()

You may try to wrap the use of the ACart array (inside the loop) in a test for IsDefined() as in:

```
<CFIF IsDefined("ACart[i]")>
  Item: #ACart[i]# <br>
<.CFIF>
```

That will fail, though, saying:

*Parameter 1 of function IsDefined which is now "ACart[i]" must be a syntactically valid variable name*

This can be frustrating because it seems a valid variable. You can output the same variable in the loop, right? If the array element being tested had a value?

### IsDefined() Expects a String

Those with more experience might point out that the problem is that the value of *i* is not being reflected as a dynamically changing number, since the variable name is in quotes (the IsDefined() function expects a string after all). That's not the solution either. Changing it to:

```
<CFIF IsDefined("ACart["&i&"]")>
```

would solve the problem if that were the issue. But this approach still fails, with:

*Parameter 1 of function IsDefined which is now "ACart[1]" must be a syntactically valid variable name*

This is frustrating because ACart[1] is a valid variable name! Moreover, there is indeed data in that array element. The simple answer is you can't test for the existence of an array element with IsDefined. But how then *do* you test for it?

### Testing Existence with IsArray()

You may be tempted to look into the IsArray() function and one of its less-known features to test for the existence of a particular dimension within the array, using:

```
IsArray(ACart,i)
```

That's incorrect thinking. This tests whether there's an array of the given dimension. Our array above is a one-dimension array. It has two elements, but it's still a one-dimension array. (See the Allaire docs or the Allaire Advanced ColdFusion class for more on multidimensional arrays.)

If you're hoping it returns false when it gets to the value of I=2 in the loop above, you'd be right. But that's because there aren't two dimensions in the array. The more critical point is that IsArray(ACart,4) would also be false. It's not testing if an element exists at the given position in the array. It's testing if it's a four-dimension array. It's not, and it would be inappropriate to make this a multidimensional array to force this IsArray() function to work for us. It's just not an appropriate solution.

### Drop Back and Punt: CFTRY/CFCATCH

We need to know if a given element in the array exists, but there seems to be no conventional way to test for that. There's a kind of brute force way around it. You could put the code that's trying to use the array elements inside a CFTRY and cause it to ignore the error that arises, as in:

```
<CFLOOP FROM="1"
TO="#arraylen(ACart)#" INDEX="i">
 <CFTRY>
  Item: #ACart[i]# <br>
  <CFCATCH>
  </CFCATCH>
 </CFTRY>
</CFLOOP>
```

This states that if the attempt to use any given array element fails, the error handler should catch it and ignore the error (we give it nothing to do in the CFCATCH, so it passes out of the CFTRY and on to the next iteration of the loop).

This way you don't get any errors, but it doesn't look obvious to a programmer who has to maintain your code (a comment would definitely be warranted!).

### CFPARAM: A Possible Solution

We mentioned earlier that prepopulating the array could solve the problem by ensuring there was always a value in each array element. That seems kludgy, especially doing some sort of loop to put a default value in each element and then testing for that default during each iteration through the array. Indeed, there are times when that wouldn't be appropriate anyway, since it may be significant that there's no value in particular array elements, or it may lead to lots of extra bytes of information in an otherwise large, sparse array.

Still, if that solution has appeal, there's still one more approach that may be at least a little less kludgy. Rather than loop through the array to prepopulate it (or deal with CFTRY/CFCATCH to detect errors), you could use CFPARAM within the loop while iterating through the array. This will assign a default empty string to any array elements that are otherwise "not defined," then test for that value to determine which array elements have value. The change to the loop above would be:

```
 <CFLOOP FROM="1"
TO="#ArrayLen(ACart)#" INDEX="i">
  <CFPARAM NAME="Acart[i]"
DEFAULT="">
  <CFIF Acart[i] is not "">
   Item: #ACart[i]# <br>
  </CFIF>
 </CFLOOP>
```

CFPARAM says, "If the named variable doesn't exist, create it and populate it with the given default value." Now, unlike the test for IsDefined(), the test that follows will never result in a runtime error, and it'll only print an array element if it has content. (If you need to distinguish between an element whose value is already an empty string and one whose value is assigned via CFPARAM, change the DEFAULT and test using some other unlikely value.)

Thanks to developer Tim Painter and instructor Emily Kim, who both shared this solution with me.

I hope this discussion of the problem is helpful for those facing this challenge. I welcome any thoughts on better approaches. My e-mail address is at the end of the article.

“ ”

**If the named variable doesn't exist, create it and populate it with the given default value**

## Part 2: Testing for Existence Within an Array of Structures

Maybe you don't have a problem with an array that's missing elements. However, a more practical extension of this problem is when you're looping over an array of structures and want to test whether an array element has all its keys.

Here's a practical example. Let's make our shopping cart array a little more advanced, using structures within it to hold multiple values of data for each item. Consider the following code:

```
<CFSET ACart = ArrayNew(1)>

<CFSET ACart[1] = StructNew()>
<CFSET ACart[1].item="Nokia 8150 phone">
<CFSET ACart[1].price="135.00">
<CFSET ACart[1].rebate="75.00">

<CFSET ACart[2] = StructNew()>
<CFSET ACart[2].item="Carrying case">
<CFSET ACart[2].price="35.00">

<CFOUTPUT>
<CFLOOP FROM="1"
TO="#arraylen(ACart)#" INDEX="i">
 Item: #ACart[i].item# <br>
 Price: #ACart[i].price#<br>
 Rebate: #ACart[i].rebate#
 <hr>
</CFLOOP>
</CFOUTPUT>
```

Note that some of those structures have keys that the others don't. This code fails when it tries to print out the value of ACart[2].rebate, because there's no rebate for the second cart item (the "carrying case").

Note that this is *not* about missing array elements. It's about missing structure keys. They're different but might seem related.

### Trying, Again, to Use IsDefined()

It's tempting to throw in a test with IsDefined() around the reference to rebate, to print a value only if it exists. Even if we're clever about the dynamic value of *i*, as in:

```
<CFIF
IsDefined("ACart["&i&"].rebate")>
 Rebate: #ACart[i].rebate#
<.CFIF>
```

we still get the error we saw above:

*Parameter 1 of function IsDefined which is now "ACart[1].rebate" must be a syntactically valid variable name*

Again, ACart[1].rebate is a valid variable name, and the first element in the array and the rebate key in the structure exist. We're trying to refer to an array inside IsDefined() and it's not allowed – whether the instance is there or not.

Given that simple syntactical limitation, some clever folks may get around it by first assigning the structure in the given array element to a simple variable. This creates a copy (by reference) of the structure. Since it's no longer an array, we can test for its keys with IsDefined(), as in:

```
<CFSET test = ACart[i]>
<CFIF IsDefined("test.rebate")>
 Rebate: #ACart[i].rebate#
<.CFIF>
```

### To the Rescue: StructKeyExists()

That two-step approach works, but there's an even easier approach using the StructKeyExists() function. This tests the existence of a given key in a structure and – here's the key point – it doesn't have a problem looking in an array of structures.

The solution would become:

```
<CFIF StructKeyExists(ACart[I],
"rebate")>
 Rebate: #ACart[i].rebate#
<.CFIF>
```

This solves the problem of a missing structure key (such as "rebate"), but it would still fail in the case of our sparse array from earlier. The ultimate solution is to also use CFPARAM, again, just inside the loop, to be able to assign a value to missing array elements. The question, then, is what to make the default.

Simply assigning an empty string to Acart[i] won't suffice, because the test for structkeyexists will complain if a given element of Acart[] is just a string (it expects a structure). The simple solution is to assign a default that creates a structure in the missing array element, as in:

```
 <CFPARAM NAME="Acart[i]"
DEFAULT="#StructNew()#">
```

Put that after the CFLOOP, and note that it's important not to leave out the pound signs around the StructNew function. Doing so would simply assign the string "StructNew()" to the array element, which would also lead to failure in the later IF test. (And you can't just leave off the quotes either: always use quotes around the value of an attribute to a CF – or indeed, HTML – tag. You may get away with it sometimes, but usually, as in this case, it won't provide the expected result.)

“ ”

**…the structure *should not be* in quotes while the key *should be***

### Structure Functions: Beware, and Some Bonus Topics

*Note:* Similar to most structure functions that work on structure keys, it expects the structure (the first parameter) to be the value of a structure and the second (the key) to be the name of the key. This means that in the example above and in most cases, the structure *should not be* in quotes while the key *should be*.

This often confuses folks when they begin working with structure functions, such as StructKeyExists and even StructDelete(). This nifty function can take advantage of the fact that sessions are indeed structures and can therefore be used to delete a given session variable (login, for instance) using:

```
#StructDelete(session,"login")#
```

Some people will be delighted to have read to this point as they've always wanted to be able to delete a particular session variable. That wasn't the point of the article, but let's consider it a bonus for being willing to consider all the ideas being offered here. (As of Release 4.5 you can also treat many variable scopes this way, including application, cgi, cookie, form, and url. You can't refer to client or server variables this way, though.)

*One last point:* Of course, if the name of the key to be deleted was stored as the value of a variable, the second parameter shouldn't be in quotes. The same is true for our StructKeyExists above. For instance, if we could loop through all the keys using <CFLOOP>'s COLLECTION attribute instead of hard-coding what we expected to find in the collection, as we did above:

```
…
<CFOUTPUT>
<CFLOOP FROM="1"
TO="#ArrayLen(Acart)#" INDEX="i">
 <CFLOOP COLLECTION="#Acart[i]#"
ITEM="y">
  <CFIF StructKeyExists(Acart[i], y)>
   #y#: #Acart[i][y]#<br>
  </CFIF>
 </CFLOOP>
 <hr>
</CFLOOP>
</CFOUTPUT>
```

the output would be:

```
ITEM: Nokia 8150 phone
PRICE: 135.00
REBATE: 75.00

ITEM: Carrying case
PRICE: 35.00
```

It's nifty if you didn't know you could loop through the structure that way. The important point is, the second parameter of StructKeyExists is "y". It's a variable driven by the CFLOOP COLLECTION, and its value will be "item", then "price", then "rebate" in the first array element, and "item", then "price" in the second.

You can loop through the variable structures we referred to earlier: session, application, cgi, cookie, form, and url. Try it using a variation of the code above; for a similar explanation see my article in last December's issue of ***CFDJ*** (Vol. 2, issue 12), "Toward Better Error Handling, Part 2," in which I showed the usefulness of looping through these structures to display their values when an error occurs.

### Summary

I hope these solutions (testing for the existence of an array element and for a key in an array of structures, as well as deleting session variables and looping through structures) have been helpful.

If you haven't had these problems and still read to this point, I congratulate you. You'll now be armed when any of these problems *do* arise.

ABOUT THE AUTHOR

*Charles Arehart is a certified Allaire trainer/developer and CTO of SysteManage, an Allaire partner. He contributes to several CF resources, is a frequent speaker at user groups throughout the country, and provides training, coaching, and consultation services.*

CAREHART@SYSTEMANAGE.COM

# The Flexibility of **Nested Custom Tags**

## Make it easy for others to use your code

BY
ANDREW
CRIPPS

Saving development costs by reusing code has been a goal of the computing industry for a long time. With a little forethought you can save time and money developing Web systems using ColdFusion by encapsulating functionality in custom tags.

This article describes how to nest custom tags (a working knowledge of these tags is assumed. If you need more information on paired custom tags, see Charles Arehart's article in ***CFDJ,*** Vol. 3, issue 1). Because of the frequent references to listings later on, I suggest you download them now from www.ColdfusionJournal.com.

Any custom tag can be turned into a container for other custom tags. The container is called the *parent* tag; the other custom tags are called *descendants* or *child* tags. As with any family tree, there can be several levels of nesting, as shown in Figure 1.

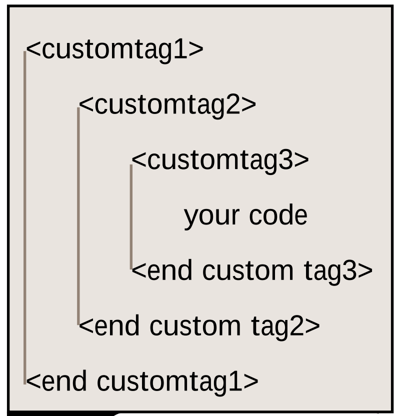


**FIGURE 1:** Custom tag hierarchy

## Example Use of Nested Tags

You may find that you've written a custom tag that needs a lot of data. You can make some custom tags a lot easier to use by turning them into nested tags. For example, consider a custom tag that will build a formatted, numbered list of items for you. You could make a custom tag such as cf_buildlist that would take perhaps two attributes: one a list of styles (e.g., liststyles= "italic,bold,plain,plain,italic"), the other a list of list items (e.g., listitems="forests, jungles, oceans, lakes, rivers").

```
<cf_buildlist
liststyles="italic,bold,plain,plain,
italic"
listitems="forests,jungles,ocean,lakes,
rivers">
```

The cf_buildlist tag would generate something like this:

1. *forests*
2. **jungles**
3. oceans
4. lakes
5. *rivers*

In this custom tag design, your users have responsibility for creating two related lists and passing them to the cf_buildlist custom tag. It would be much easier for them if they could specify the formatting for each item at the time each list item is specified.

Nested custom tags can make things easier for your users. In this example you could create a cf_buildlist tag that had a closing tag and another child tag cf_listitem that accepted list items:

```
<cf_buildlist>
 <cf_listitem item="forests"
style="italic">
 <cf_listitem item="jungles"
style="bold">
 <cf_listitem item="oceans"
style="plain">
 <cf_listitem item="lakes"
style="plain"
 <cf_listitem item="rivers"
style="italic"
</cf_buildlist>
```

Now users can more easily specify each item and the style for each item. You could extend cf_buildlist to accept other attributes about the list as a whole. For example, you might add an attribute that specifies whether the list is bulleted or numbered.

A good guideline for deciding whether to turn a custom tag into a nested tag is this: if you can identify repeating groups of information in your existing attributes, then you have a candidate for creating parent/child tags.

## Parents Getting at Child Data

Listing 1 is an example of a ColdFusion template that calls a custom tag cf_myparenttag (see Listing 2). cf_myparenttag has a child tag, cf_mysubtag (see Listing 3), and cf_mysubtag has its own child tag, cf_mysubsubtag (see Listing 4). In Listing 1 lines 16–26 make the call to cf_myparenttag. Lines 19–23 are the call to the cf_mysubtag.

CF_ASSOCIATE is used to allow parent tags to get at the attributes defined for child tags. Let's say you have a custom tag called cf_myparenttag that calls a child tag cf_mysubtag. You want cf_myparenttag to be able to access the attributes defined for cf_mysubtag. Do this is by using CF_ASSOCIATE in cf_mysubtag (see Listing 3), then accessing the attributes in cf_myparenttag (see Listing 2).

In Listing 3, lines 10–12, we say that if the execution mode is "start", then associate the attributes defined for cf_mysubtag with the parent tag, cf_myparenttag.

Now look at Listing 2, lines 19–25. Here, if the execution mode is "end", we access the array of structures made available by the cfassociate function and print out the structure key and its corresponding value.

The output of Listing 1 is given in Listing 5. You can see the nesting effect in the printed statements: "Before calling myparenttag", "Before calling mysubtag", "Before calling mysubsubtag", "After calling mysubsubtag", "After calling mysubtag", "After calling myparenttag". All the custom tags have been called in sequence.

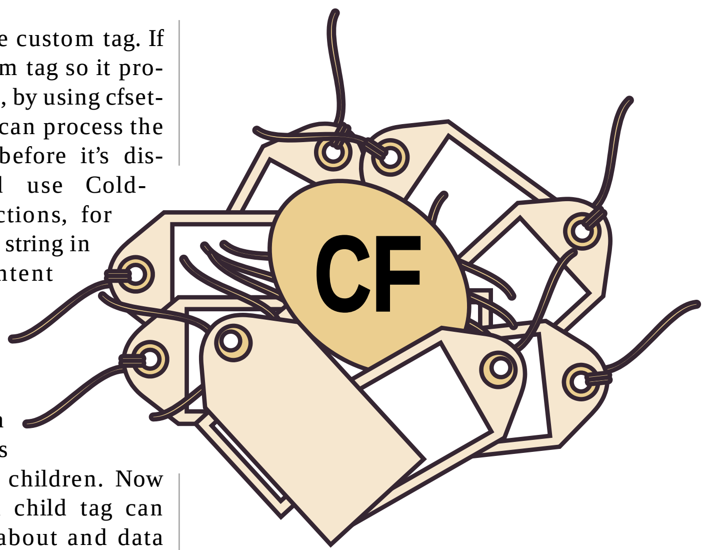
## Thistag Scope

CFML provides functions that give information about the execution and type of custom tags and about the content generated by the tags. The structure "thistag" provides access to these variables:

- ***AssocAttribs***: Associated attributes from child tags.
- ***ExecutionMode***: Valid values are start, end, and inactive.
- ***HasEndTag***: Used for code validation, it distinguishes between custom tags that have and don't have end tags for ExecutionMode=start. The name of the Boolean value is ThisTag.HasEndTag.
- ***GeneratedContent***: Content from this tag that can be processed as a variable.

We've already seen how to use CF_ASSOCIATE to populate the AssocAttribs variable. All the other variables are populated for you by ColdFusion. In Listing 5, which shows the output of running Listing 1, you can see the values for the executionmode and hasendtag variables for each custom tag call. Note that cf_mysubsubtag isn't a paired custom tag; there's no corresponding end tag </cf_mysubsubtag> for the opening <cf_mysubsubtag> call. The parent tag for cf_mysubsubtag is cf_mysubtag, and in Listing 3, line 16, the variable #thistag.hasendtag# has the value NO because there is no end tag.

The variable #thistag.generatedcontent# provides access to the content generated by the custom tag. If you code your custom tag so it produces no output (e.g., by using cfsetting or cfsilent), you can process the generated content before it's displayed. You could use ColdFusion's string functions, for example, to replace a string in the generated content before it's displayed.



## Children Getting at Parent Data

We've seen how a parent tag can access the attributes of its children. Now we'll look at how a child tag can access information about and data from its parent.

Two ColdFusion functions provide information about the execution of custom tags:

- GetBaseTagList()
- GetBaseTagData(parent tag name)

GetBaseTagList returns a comma-delimited list of execution scopes. In Listing 2, line 14, for example, the call to #getBaseTagList()# (when executed by running Listing 1) produces:

```
[CFOUTPUT, CF_MYPARENTTAG]
```

This gives the programmer the information that the current execution scopes are, first, CFOUTPUT, and then the custom tag cf_myparenttag.

Now look at Listing 3, line 17. Here the call to #getBaseTagList()# (when executed by running Listing 1) produces the list:

```
[CFOUTPUT,CF_MYSUBTAG,CF_MYPARENTTAG]
```

A programmer could now use this information to determine in what context the custom tag is executing. This might be important when the custom tag is included in an IF clause, for example. The programmer could check the list generated by #getBaseTagList# and determine whether the call to the current custom tag was made from within an IF clause.

GetBaseTagData returns a structure that contains information about the specified parent tag. Look at cf_mysubsubtag (Listing 4, lines 21–22). Here there is a call to #GetBaseTagData()# for the tag cf_mysubtag – the parent of cf_mysubsubtag. Since the parent we specified was a custom tag, we have access to the #thistag# set of variables. Line 22 accesses the #thistag.executionmode# variable for cf_mysubtag. GetBaseTagData will work for any parent tag. For example, we could add a line at line 23 to say:

```
 <cfset myparenttagdata =
#getBaseTagData("cf_myparenttag",1)#>
```

This would give access to all the thistag variables for cf_myparenttag.

Since you know the list of parent tags from #getBaseTagList#, you could write code that obtains data for each parent tag in the list. Note that some parents, such as CFIF, don't have any associated data.

## Summary

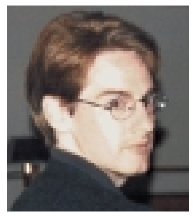
Nested custom tags provide a very flexible way to abstract complexity, and make it easy for others to use your code. You can create nested tags easily, but as the complexity of your solution grows, you may need to call on the functions ColdFusion provides to allow parent tags to access attributes defined in child tags, or to allow child tags to access information about their parents. You can control with absolute precision what ColdFusion will display to the user by working with the generated content.

ABOUT THE AUTHOR
*Andrew Cripps currently lives and works in Massachusetts. He has over 12 years of experience in the computing industry, and specializes in the design and development of Web-based systems.*

ANDREW@CRIPPS.NET

# A Cold Cup **o'Joe** Part 3 of 9

## Java object basics



BY
GUY
RISH

Without a doubt, one of the most powerful assets available from the ColdFusion and Java marriage is the simple ability to call up anything from the extensive Java class libraries.

In this article we'll look at the basics of working with Java in ColdFusion. I'll specifically talk about the different ways to create and use a Java object within your templates. As with most things, there are a few twists and turns when you mix technologies.

### Configuration

It's important to make certain that the ColdFusion Server's Java settings are properly configured. Misconfiguration can lead to poor server performance and considerable frustration for the developer. Check the settings with the information in the ColdFusion Server's documentation and in Part 1 of this series (*CFDJ*, Vol. 3, issue 1).

In this article you'll be working with some simple classes that must be placed somewhere in the ColdFusion Server's Java CLASSPATH setting. This can be configured through the CLASSPATH setting on the ColdFusion Administrator's Java Settings panel.

For example, on my Windows machine it is:

```
C:\CFusion\CFX;C:\CFDJ\javalib
```

and on my Linux machine:

```
/opt/coldfusion/cfx:/usr/local/CFDJ/
javalib
```

### Hello, Java

Creating a Java class for use in a ColdFusion template doesn't require any special handling or subclassing (see Listing 1, HelloWorld). Actually, using it doesn't require much either, just a simple call to <CFOBJECT> or the CreateObject function in <CFSCRIPT>.

### *Tag, You're It*

Through the <CFOBJECT> tag, ColdFusion can create an instance of nearly any Java class in the ColdFusion Server's Java CLASSPATH setting. This makes <CFOBJECT> one of the most powerful tags in the language's repertoire. Creating a Java object with <CFOBJECT> requires only four attributes: Type, Action, Class, and Name. The Type attribute allows <CFOBJECT> to identify which kind of object handling it will use; in this case it'll always be a value of "Java". The Action attribute specifies the instantiation method of the object, either "Create" or "Connect"; in this case it'll always be a value of "Create". The Class attribute specifies the name of the Java class to be instantiated; since it refers to a Java class name it's case sensitive and, if necessary, should be fully qualified with a package path. The Name attribute is the name of the object variable that's to be created. As you can see in the following snippet:

```
<CFOBJECT Type="java" Action="cre-
ate" Class="HelloWorld" Name="hw">
```

Accessing a method of the HelloWorld class is just as easy:

```
<CFOUTPUT>
  #hw.greeting()#
</CFOUTPUT>
```

### *Follow the Script*

Within a <CFSCRIPT> block, objects are created with the CreateObject function. The syntax for creating Java objects is pretty simple and requires only two arguments: Type and Class. The Type argument identifies which kind of object handling to use; in this case it'll always be a value of "Java". The Class argument is the name of the Java class to be instantiated. As with <CFOBJECT>, the class name is case sensitive and, if necessary, should be fully qualified with a package path. The function returns a handle to an object.

Once the object is created, the usage is comparable to the usage in a <CFOUTPUT> block:

```
<CFSCRIPT>
 hw = CreateObject("Java",
   "HelloWorld");
 WriteOutput(hw.greeting());
</CFSCRIPT>
```

### Digging Deeper

The examples from the previous snippets work smoothly enough, but seldom are classes that simple. Working with more complex classes requires a little more investigation, so we've graduated the HelloWorld class a bit (see the Hello class in Listing 2). Further, showing the intricacies of working with objects requires a busier example (see Listing 3). For the sake of convenience I used line numbers, which allow me to focus on specific aspects of object creation and usage since I'm going to jump around a bit.

### *Static Members*

Let's begin with the following snippet from Listing 3:

```
14. hw2 = CreateObject("java", "Hello");
```

This single statement causes the ColdFusion Server's JVM to load the specific class; however, it doesn't create an instance. This is an interesting state to exist in – the code that makes up a class is placed in memory but it's not executing. Not until reference to the newly created object is made is it really instantiated. It will then have its own internal state and its variable values will likely be different from other executing instances of the same class. Some classes have static members that are independent of their instantiation and can be accessed without causing the object to be created. This nuance of Java has no ColdFusion equivalent, but it can be very useful. The basic Java class library abounds with static members; in fact, the entire java.lang.Math class, used for general purpose math functions, is static. This can be seen in the following lines:

```
16.WriteOutput("<h2>Displaying a
   static member</h2>");
17.WriteOutput(hw2.DEFAULT_STYLE);
```

At this point it's still not an executing object.

### *Object Construction*

Returning to line 14, we see the creation of the Hello class. Again, there's no instantiated object, only the class is loaded. While access to static class members won't instantiate the object; as soon as a nonstatic method or property is accessed the object will be instantiated with the default class constructor. Until then CFML lets you call an alternate class constructor, as seen here:

```
19.hw2.init("Yo");
```

By calling the init method of the loaded class, the interpreter will create the object with one of the alternate constructors defined in the class. In this case, the snippet from line 19 calls the constructor that defines the greeting style for the object. Inspecting the Hello class, however, won't turn up any method called init. This phantom method is a special mechanism in CFML that aliases the class constructors. This does pose a slight problem for classes that actually define real methods called init. Calls to a "real" init method, even after object construction, will cause an exception to be raised.

### *Overloaded Methods*

After line 19 an instance of the class is created:

```
21.WriteOutput("<h2>Alternate
   Greetings</h2>");
22.WriteOutput(hw2.greeting());
23.WriteOutput("<br>");
24.WriteOutput(hw2.greeting("Dude"));
```

## MINOR ADJUSTMENTS

Not long ago I had an e-mail exchange with Steven Erat, a support engineer at Allaire, and he gave me a few pointers on configuring ColdFusion 4.5.x with Sun's JDK 1.3. As it turns out, a few adjustments to the settings need to be made before the newer JVM works politely with the ColdFusion Server on the Linux platform. I didn't have any problems under Windows 2000.

You can get the Java 2 Platform, Standard Edition v1.3 from Sun's Javasoft Web site, www.javasoft.com/j2se/1.3/. If you want access to the Enterprise APIs, you'll need to download it separately as the J2EE for 1.3 actually lies on top of the Standard Edition. The folks at Sun have done you a favor if you're running on Red Hat as the 1.3 JDK can be gotten in RPM format. Never fear, though: if you're not using a package management system or RPM you can also get it as a tarball. What's more, I've been told informally that the HotSpot compiler works better with ColdFusion than it did with previous versions.

Once installed, you'll need to make three changes: the Java Virtual Machine Path in the Administrator, the LD_LIBRARY_PATH in the ColdFusion Server's start script, and the Implementation Options, also in the Administrator.

Initially, after installing v1.3 on my machine, I just changed the path settings to point at the new JVM library in my JVM path in the Java settings panel of the Administrator. For example:

```
/usr/local/java/jre/lib/i386/hotspot/
libjvm.so
```

Then, when I did a test, a confusing message displayed in my browser:

```
The JVM library could not be found. Please
check if the file specified in the
ColdFusion Administrator actually exists.
```

This was pretty vexing after doing a directory listing a few times on an existing file and not being able to put two and two together. It wasn't until I looked in the ColdFusion logs (server.log) that I found my problem:

```
 Error occurred during initialization of VM
 Unable to load native library:
/usr/local/java/jre/lib/i386/hotspot/
libjvm.so: undefined symbol: jdk_sem_post
```

I'll attribute it to lack of caffeine but it took me more than a moment to realize that my LD_LIBRARY_PATH setting must not be right. After careful inspection I made a few adjustments:

```
JAVAPATH=/usr/local/java

LD_LIBRARY_PATH=$CFHOME/lib:$JAVAPATH/lib/
i386:$JAVAPATH/jre/lib/i386:$JAVAPATH/jre/
lib/i386/hotspot:$JAVAPATH/jre/lib/i386/
native_threads
```

At this point I was still getting an error in the browser, but the message in the log was different:

```
#
# HotSpot Virtual Machine Error, Internal
Error
# Please report this error at
# http://java.sun.com/cgi-
bin/bugreport.cgi
#
# Error ID: 4F533F4C494E55580E4350500570
#
# Problematic Thread:
"Fatal","TID=9224","02/25/01","14:02:21",
"Caught a fatal signal (11) - Aborting"
#
```

Reviewing a link that Mr. Erat sent me (http://developer.java.sun.com/developer/bugParade/bugs/4397350.html), I found I needed to add a switch in the Implementation Options setting in the Java settings panel of the Administrator:

```
-XX:+AllowUserSignalHandlers
```

Once I made this addition, everything ran as expected.

From my e-mail exchanges I've heard that these adjustments are supposed to be solid for the current betas of ColdFusion 5.0 as well. I haven't had time to verify this, though.

What's of major interest in this snippet are the two different calls to the greeting method. What might be inferred is that the greeting method has an optional argument, like many CFML functions. However, upon reviewing the Java code in Listing 2, it's found that greeting is an overloaded method. This means there are two different methods with the same name but distinctly different possible arguments. In this example there's nothing particularly special about the functionality posed by either method, but in a more complex class this is often not the case. An example of this is the getConnection method of the java.sql.DriverManager class.

According to the online documentation (*CFML Language Reference*, Chapter 1: ColdFusion Tags, CFOBJECT Type="JAVA" and Chapter 2: ColdFusion Functions, CreateObject), the current version of ColdFusion supports only overloaded methods as long as they each have a different number of arguments. As might be expected, this overloading rule applies to constructors as well. Sadly, simply changing the data type of one of the arguments in your Java source is insufficient and creates unpredictable results, almost certainly causing an exception to be raised. After some investigation I found that absolute failure is not assured, and there does seem to be some rationale to the selection of which class method the call is routed to. Since there's no absolute guarantee of an exception, this is something that should be taken into consideration during those late-night debugging sessions.

I'll leave that to your experimentation.

> "…there are two different methods with the same name but distinctly different possible arguments"

## Wrapping It Up

In this article we've investigated two different ways to construct objects in CFML, the flexibility of static members, how to use alternate constructors, and the twists with overloading.

Among the things I'll talk about in Part 4 of this series are the JavaCast function, structured exception handling, and a solution to the problem of loading and unloading Java classes from the ColdFusion Server's JVM instance.

ABOUT THE AUTHOR
*Guy Rish teaches and mentors ColdFusion and object-oriented analysis and design for Andrews Technology, a consulting firm in San Francisco. He holds instructor certifications from Rational Software and Allaire and teaches ColdFusion for SFSU in the Multimedia Studies Program.*

GRISH@CNCDSL.COM

**Listing 1**

```
public class HelloWorld
{
    public String greeting()
    {
    return(new String("Hello, World"));
    }

    public String greeting(String whom)
    {
    return(new String("Hello, " + whom));
    }
}
```

**Listing 2**

```
public class Hello
{
    public static final String DEFAULT_STYLE = "Hello";
    protected String m_style;

    public Hello()
    {
    this(DEFAULT_STYLE);
    }

    public Hello(String style)
    {
    m_style = style;
    }

    public String getStyle()
    {
    return(m_style);
    }

    public void setStyle(String style)
    {
    m_style = style;
    }

    public String greeting()
    {
    return(new String(m_style + ", World"));
    }

    public String greeting(String whom)
    {
    return(new String(m_style + ", " + whom));
    }
}
```

**Listing 3**

```
<HTML>
  <HEAD><TITLE>Listing 5</TITLE></HEAD>

  <BODY>

    <cfscript>
       hw1 = CreateObject("java", "Hello");

       WriteOutput("<h2>Basic Greetings</h2>");
       WriteOutput(hw1.greeting());
       WriteOutput("<br>");
       WriteOutput(hw1.greeting("Mr. Rish"));

       hw2 = CreateObject("java", "Hello");

       WriteOutput("<h2>Displaying a static member</h2>");
       WriteOutput(hw2.DEFAULT_STYLE);

       hw2.init("Yo");

       WriteOutput("<h2>Alternate Greetings</h2>");
       WriteOutput(hw2.greeting());
       WriteOutput("<br>");
       WriteOutput(hw2.greeting("Dude"));
     </cfscript>

   </BODY>
</HTML>
```

CODE LISTING
The code listing for this article can also be located at www.ColdFusionJournal.com

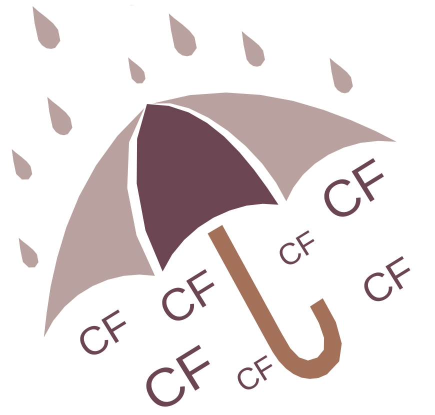
# Ask the Training Staff

## A source for your CF-related questions

BY
BRUCE
VAN HORN

Well it's April already! Can you believe it? If you're like me, you've got a full-blown case of "spring fever" and it's hard to concentrate on CF code. But spring is also a time for new projects and pressing deadlines.

Don't worry. We're here to help. Just send us your questions and we'll try to get you going in the right direction. Here are some of the questions that came in recently.



**Q:** ***I have some questions about server variables. (1) Is there a limit to the number of variables you can create in the server scope? (2) When do they expire? (3) Are they "global" (i.e., available to all sites on the server)?***

**A:** Here are your answers: (1) No, there's no limit to the number of variables you can set in the server scope (or any scope, for that matter). The only limit would be the amount of available memory on your server. (2) Server variables never expire (unlike application and session variables). Once created, they stay in the server's memory until CF is shut down. (3) Yes, that's the purpose of the server variable scope. Once created, these variables are immediately available to any CF page running on that server, regardless of the application.

A word of caution about using the scope: because server variables use a shared memory space, be sure to use CFLOCK around any read or write to the server scope!

**Q:** ***I have an application that checks to see if a user has logged in (checks for a Session.LoggedIn variable). How can I redirect a user to the page that was requested first or the page that was up when he or she got logged out because of inactivity, rather than always sending him or her to the home page after logging in?***

**A:** Great question! As with anything in CF, there are probably several ways to accomplish this, but here's how I'd do it. First we need to know what page was requested (whether it's due to an initial request for a page in the site or a user refreshing a page after being away for a while is irrelevant; it's the same process!) just prior to being kicked out to the login page. You can do this by checking the value of CGI.Script_Name. This variable is always available to CF with each page request and it tells you the name of the page the user requested. Next you need to pass the value of that variable over to your login page. Then have your login script direct the user back to that page after successfully logging in.

Listing 1 shows how to capture the value of CGI.Script_Name and pass it over to your login page using a URL variable. You would place this code in your Application.cfm page. Listing 2 shows how to embed the URL.PageRequested variable in your login form, and Listing 3 shows how to do the redirect after a successful login.

**Q:** ***I'm just starting out and have no experience with SQL. Is there a book/guide you can recommend to help me get started with basic SQL?***

**A:** Yes, I'd recommend *Sams Teach Yourself SQL in 10 Minutes* by Ben Forta. This is a great place to start, and you can read Emily Kim's review in ***CFDJ*** (Vol. 2, issue 2**)**.

**Q:** ***Do you have a Web site that has an archive of the "AskCFDJ" questions and answers?***

**A:** Yes, you can visit www.NetsiteDynamics.com/AskCFDJ.

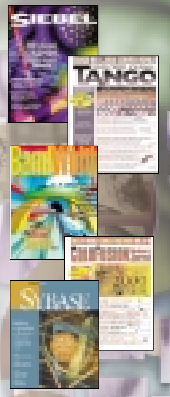
**Q:** ***Should I use the IIF() function instead of a <CFIF><CFELSE></CFIF> block?***

**A:** NO! You should avoid the IIF() function like the plague! Although this function is very familiar to those of you with traditional programming backgrounds, and it takes fewer keystrokes to code, you should always try to use the <CFIF><CFELSE></CFIF> tags because they execute much faster. Copy the code in Listing 4 into your copy of CF Studio and browse it. You'll see that the CFIF code is up to four times faster!

Please send your questions about ColdFusion (CFML, CF Server or ColdFusion Studio) to AskCFDJ @sys-con.com.

BRUCE@NETSITEDYNAMICS.COM

ABOUT THE AUTHOR
*Bruce Van Horn, a regular contributor to **CFDJ**, is president of Netsite Dynamics, LLC (an Allaire Alliance Partner), and an Allaire certified instructor.*



www.javadevelopersjournal.com
www.wbt2.com
www.xml-journal.com
www.powerbuilderjournal.com
www.coldfusionjournal.com
www.linuxbusinessweek.com
subscribe online
www.sys-con.com
or call
800 513-7111
wireless | java | xml | linux | coldfusion | powerbuilder
SYS-CON MEDIA

Q&A

**Listing 1**

```
(Application.cfm)
<!---Test for login --->
<CFIF NOT IsDefined("Session.LoggedIn")>
   <CFLOCATION URL="../Login/Login.cfm?PageRequested=
#CGI.Script_Name#">
</CFIF>
```

**Listing 2**

```
(Login_form.cfm)
<!--- Give a default value for URL.PageRequested in case
   the user came here directly --->
<CFPARAM NAME="URL.PageRequested"
   DEFAULT="../Home/index.cfm">

<form action="Login_action.cfm" method="POST">
<CFOUTPUT>
<INPUT TYPE="Hidden" NAME="PageRequested"
   VALUE="#URL.PageRequested#">
</CFOUTPUT>
<!--- the rest of your form goes here! --->
```

**Listing 3**

```
(Login_action.cfm)
<!--- if login is successful, set the session var and
   redirect the user --->
<CFIF qLogin.RecordCount>
   <CFSET Session.LoggedIn = "1">
   <CFLOCATION URL="#Form.PageRequested#">
</CFIF>
```

**Listing 4**

```
<!--- Test iif() --->
<CFSET Start = GetTickCount()>
<CFLOOP FROM="1" TO="1000" INDEX="i">
   <CFSET NewVar = iif(i mod 2,"1","0")>
</CFLOOP>
<CFSET End = GetTickCount()>
<CFSET Total = Variables.End - Variables.Start>
The IIF() block took <CFOUTPUT>#Variables.Total#</CFOUTPUT>
   milliseconds!
<BR>
<BR>
<!--- Test <CFIF><CFELSE></CFIF> --->
<CFSET Start = GetTickCount()>
<CFLOOP FROM="1" TO="1000" INDEX="i">
   <CFIF i mod 2>
       <CFSET NewVar = 1>
   <CFELSE>
       <CFSET NewVar = 0>
   </CFIF>
</CFLOOP>
<CFSET End = GetTickCount()>
<CFSET Total = Variables.End - Variables.Start>
The CFIF block took <CFOUTPUT>#Variables.Total#</CFOUTPUT>
   milliseconds!
```

CODE LISTING
The code listing for this article can also be located at www.ColdFusionJournal.com

# Use the Wheel, Don't Reinvent It

## Custom tags save you time and energy

BY MATT ROBERTSON

Building an application in ColdFusion is a quick process. If you take advantage of a magnificent tool provided by Allaire, you may be able to avoid this process completely...

If you're reading this magazine you almost certainly know ColdFusion is a great development environment, especially if you need to get the job done fast. However, the benefits inherent in ColdFusion's design are only part of the story. Allaire has built a tremendous user community whose existence is an extraordinary asset to beginning and advanced developers alike.

A major part of this community, the Allaire's Developers Exchange (http://devex.allaire.com/developer/gallery), is the subject of this article. The Exchange has thousands of open-source code snippets, free custom tags, and full-blown commercial applications available for the asking.

I use the Exchange every chance I get. Even if I can't use something directly, often I can get started by finding something that performs a function similar to what I need. Looking at another author's code can break my case of writer's block and get me going.

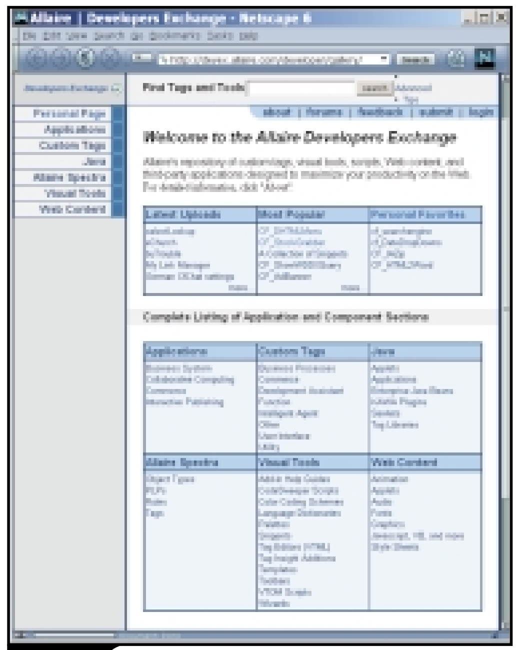


FIGURE 1: Developers Exchange intro page

## Visiting the Developers Exchange

Figure 1 shows the Developers Exchange intro page, which organizes the available material into categories. *Note:* You can personalize it, creating a list of favorite tags, marking components so you'll be notified when they're updated, and keeping a centralized list of the tags you create and post yourself.

One category, Most Popular, holds the most frequently downloaded material in the Exchange. Let's look at a couple of these tags. Before we begin, note that Figure 2 shows the tags discussed in this article, and all the code is shown in Listing 1.

## CF_DHTMLMenu by Ben Forta

With over 18,000 downloads since its original posting in 1998, the free, open-source CF_DHTMLMenu is the undisputed tag download champ. Let's look at what it does and why it's so popular.

CF_DHTMLMenu is essentially a ColdFusion "front end" to a JavaScript menu-building script. Using this tag you can quickly and easily build yourself a slick, nested drop-down menu system, running either across the top of your page like a traditional GUI menu, vertically down one side of the screen, or in just about any configuration you can think of. You can have your own system up and running in just a few minutes.

After downloading the tag, I placed the included JavaScript menu script into a folder named "js" off my development server's root (c:\inetpub\wwwroot\js\). This is the default, although a ScriptDir parameter lets you specify any location.

Once this was done, I looked at the author's usage examples located within the main tag file (DHTMLMenu.cfm). Using these examples, I quickly knocked out a working multilevel menu system, as shown in Lines 24–72.

That example system shows two separate calls to the CF_DHTMLMenu tag that create a horizontal menu running across the screen. To add additional top-level menu choices I would have simply added more CF_DHTMLMenu calls. There are settings that define the menu border, text, background, and highlight colors. Additional settings that provide still more display control are available. Speedy development is further enhanced with the inclusion of VTM tags for ColdFusion Studio users. The tag calls are simple and straightforward, and allow you to nest menus in a cascading fashion as deeply as you like.

A rose this sweet is bound to have a thorn hiding somewhere. In this case CF_DHTMLMenu didn't work in my two Netscape browsers (4.76 and 6.01). In addition, the text caption specified in the CF_DHTMLMenu call is a live link. If a user mistakenly clicks on it, they'll be taken to the default file of the current folder (index.htm, index.cfm, etc.). You'll have to trap and handle instances in which a user clicks the top-level link (coupled with a browser check and some extra programming, you can use this feature to provide an alternate navigation function to Netscape users).

All things said and done, CF_DHTMLMenu hands you a free, easy-to-install, and highly functional menu system on a silver platter.

## CF_StockGrabber by Rob Brooks-Bilson

With a respectable download count well over 12,000, CF_StockGrabber is another developer favorite on the Most Popular list. CF_StockGrabber uses CFHTTP to pull stock prices from Yahoo.com's quote service. Bear in mind, these are delayed prices. If the stakes are high, a dedicated commercial solution is a better choice.

This "tag" is actually a miniapplication and includes a fully functional demo that you can have up and running in a couple of minutes. A nifty scrolling ticker applet is included as well. After you see how the tag works in its attached demo, you can implement it in a variety of ways.

For example, CF_StockGrabber allows you to send quotes to an e-mail address. Couple this with some ColdFusion programming that evaluates the returned #Last_Traded_Price# value and you can trigger e-mail alerts to yourself when a stock price reaches a certain targeted level. Another possibility: using nothing more complicated than a META page refresh you can create a page that monitors and self-updates the current prices of your favorite stocks.

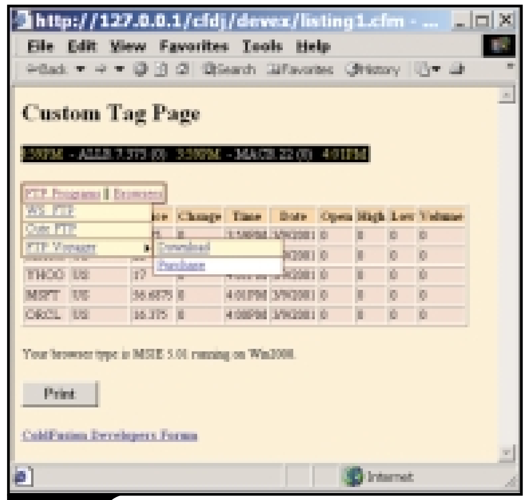
Lines 3–6 show the CF_StockGrabber tag call, which requests stock prices for five different firms. Lines 11–23 set up the scrolling ticker applet. Lines 74–101 display the tag output in HTML table format (since I created the Figure 1 screen shot on a day when markets were closed, many of the columns show zeroes).

CF_StockGrabber offers a VTM file for Studio owners, but you have everything you need by looking at the author's included demo, which simply illustrates everything the tag can do in one easy-to-understand file.

## CF_Browser v3.2 by Chris Sgaraglino

While looking at CF_DHTMLMenu, I mentioned that you'd probably want to do some browser detection when using that tag, so let's skip down the Most Popular list by a few slots and look at CF_Browser. It isn't the only browser detector in the Exchange (a quick keyword search yielded three more), but it's the most popular of the bunch with well over 7,600 downloads. Again, the tag download is loaded with examples and a VTM file.

While CF_Browser has many options available in its advanced mode, if you simply call the tag with no arguments it returns a browser type, version number, and the user's operating system. Line 1 shows the basic tag call, and Lines 102–109 show its use for display purposes and in a conditional statement. The author provides documentation on the values that CF_Browser returns, making it easy for you to set up either simple or more complex browser detection.


http://127.0.0.1/cfdj/devex/listing1.cfm - ...
File Edit View Favorites Tools Help
Custom Tag Page
Print
ColdFusion Developers Forum
Internet

Your browser type is MSIE 5.01 running on Win2000.

FIGURE 2: Custom tag page

## CF_Print by Jason Miller

As mentioned above, a conditional statement that tests the user's browser version is on Lines 107–109. If the browser version is at least version 4 (regardless of the browser type), we can run the CF_Print tag.

CF_Print is also several steps down the Most Popular list from CF_Browser, with a download count of just over 5,000. The function it provides is simple and straightforward: a print button is placed on the current template, allowing users to bring up the print selection dialog of their browser to print the current page.

Unlike the previously discussed tags, the only thing contained in CF_Print is the tag itself, which contains a few words of instruction. This simplicity is warranted, however, as implementing the tag is about as simple as it gets – install the tag file as usual and call it within FORM tags. As you can see from the tag call on Line 108, all that's necessary is the bare FORM tag itself.

Unfortunately, the tag in its present form is not functional with Netscape version 6 browsers. However, since it's open source you can easily fix this by adding the following code to the end of the CFIF statement on Line 31 of the custom tag (print.cfm):

```
OR #cgi.Http_User_Agent# CONTAINS
'Mozilla/5'
```

## Other Roads to Take

The Most Popular list isn't the only place to go for tags. As you can see in Figure 1, a number of categories are available. If you're like me, when you have a specific need, the first thing you'll do is use the Search function found at the top of any Exchange page. Both simple and complex searches are supported.

Another smart move is to visit the Exchange on a regular basis and check out the Latest Uploads. I've come across all sorts of little jewels that way and, no doubt, so will you. Sometimes new items cover things I've wished for but never got around to puzzling out myself. Sometimes they showcase great ideas that I'd never have thought up on my own. Either way, the Latest Uploads area bears watching.

In the Latest Uploads area you'll see that you can change the sort order from the default setting of descending date of posting to a variety of others. If you change the sort order to ascending date of posting, your list will have the very first tag ever submitted to the Exchange at the top.

## CF_NewWindow by Mike Andler

First uploaded on August 11, 1997, CF_NewWindow is listed as the oldest tag in the Exchange. It creates a link that opens a new window, giving display options such as window size, resizability, and whether the browser's menu or scrollbars are present. If this sounds familiar, it's because these are all typical settings for a new window created via JavaScript. CF_NewWindow is a convenient front end that writes a JavaScript new window function for you.

The tag shows its age a bit, though. A documentation file and function are included in taghelp.cfm, but attempting to access it from the also-supplied info.cfm resulted in a ColdFusion error. I was unable to work around or correct it as a result of all the files being encrypted. In addition, two other files included in the download (cleanuplist.cfm and errorhandler.cfm) are apparently unnecessary, although without any documentation and encrypted files there's no way to be certain.

I can't fault the author, who gave away CF_NewWindow for free during a time when obviously there wasn't a rule book or common standard to go by. I think, in light of the present day, this is a good example of why, unless you have a good (i.e., commercial) reason to do so, encryption can do more harm than good.

All griping aside, CF_NewWindow has held up quite nicely. By copying over nothing more than the custom tag itself (newwindow.cfm) to my custom tags directory, I was able to create a popup window (Lines 112–120) that functioned flawlessly in Internet Explorer 4.01 and 5.01 as well as Netscape 4.76 and 6.01.

## Conclusion

The thousands of tags in the Exchange didn't get there by themselves, of course. Developers with a bright idea or two posted them there for the use of others. If you have something useful, by all means post it as either a free resource or a commercial product. For my own part, I've happily downloaded a great many free tags, and I've also happily paid more than a few dollars for various commercial ones that have saved me time and energy. If you haven't tapped into this bonanza yet, do yourself a big favor and check it out.

*Code appears on next page*

ABOUT THE AUTHOR
*Matt Robertson is president and chief designer at MSB Designs, Inc., a Web development firm specializing in ColdFusion-driven applications.*

MATT@MYSECRETBASE.COM

**Robertson** *Code continued from previous page*

**Listing 1**

```
<CF_Browser>

<CF_StockGrabber
  TickerSymbols="allr,macr,yhoo,msft,orcl, "
 QueryName="GetQuotes"
 ErrorCheck="No">

<html><head><title></title></head>
<body BGCOLOR="#FFEBCD">
<H1>Custom Tag Page</H1>
<APPLET CODE="ProScroll.class"
 WIDTH=400
 HEIGHT=20
 ALIGN=TOP>
 <PARAM NAME=bgcolor VALUE="black">
 <PARAM NAME=Text VALUE="
 <CFOUTPUT QUERY="GetQuotes">##yellow##
  #Last_Traded_Time# ##white## - #Symbol#
  #Last_Traded_Price# (#Change#) </CFOUTPUT>">
 <PARAM NAME=Speed VALUE="fast">
 <PARAM NAME=style VALUE="Bold">
 <PARAM NAME=size VALUE="16">
</APPLET><P>
<TABLE BORDER="1" BORDERCOLOR="#660000"
 CELLPADDING="2" CELLSPACING="0">
<TR><TD BGCOLOR="#FFFFCC">
<CF_DHTMLMenu
 CAPTION="FTP Programs"
 TOP="135"
 LEFT="10"
 SCRIPTDIR="/js"
 BORDERCOLOR="##660000"
 BACKGROUND="##FFFFCC"
 HIGHLIGHT="##FFFFFF"
 FONTCOLOR="##3300FF">
<CF_DHTMLMenuItem
 CAPTION="WS_FTP"
 URL="http://ipswitch.com/">
<CF_DHTMLMenuItem
 CAPTION="Cute FTP"
 URL="http://cuteftp.com/">
 <CF_DHTMLMenuSubMenu
  CAPTION="FTP Voyager"
  URL="http://ftpvoyager.com">
  <CF_DHTMLMenuItem
  CAPTION="Download"
  URL="http://ftpvoyager.com/dl.htm">
  <CF_DHTMLMenuItem
  CAPTION="Purchase"
  URL="https://rhinosoft.com/ftpvoyager/buy.htm">
  </CF_DHTMLMenuSubMenu>
</CF_DHTMLMenu>
&nbsp;||
<CF_DHTMLMenu
 CAPTION="Browsers"
 TOP="135"
 LEFT="110"
 SCRIPTDIR="/js"
 BORDERCOLOR="##660000"
 BACKGROUND="##FFFFCC"
 HIGHLIGHT="##FFFFFF"
 FONTCOLOR="##3300FF">
<CF_DHTMLMenuItem
 CAPTION="Internet Explorer"
 URL="http://www.microsoft.com/ie">
<CF_DHTMLMenuItem
 CAPTION="Navigator"
 URL="http://netscape.com/download/">
<CF_DHTMLMenuItem
 CAPTION="Opera"
 URL="http://www.operasoftware.com/">
</CF_DHTMLMenu>
</TD></TR></TABLE>
<TABLE BORDER=1>
<TR>
<TH BGCOLOR="#FFCC99">Symbol</TH>
<TH BGCOLOR="#FFCC99">Exchange</TH>
<TH BGCOLOR="#FFCC99">Price</TH>
<TH BGCOLOR="#FFCC99">Change</TH>
<TH BGCOLOR="#FFCC99">Time</TH>
<TH BGCOLOR="#FFCC99">Date</TH>
<TH BGCOLOR="#FFCC99">Open</TH>
<TH BGCOLOR="#FFCC99">High</TH>
<TH BGCOLOR="#FFCC99">Low</TH>
<TH BGCOLOR="#FFCC99">Volume</TH>
</TR>
<CFOUTPUT QUERY="GetQuotes">
<TR>
<TD BGCOLOR="##EFD6C6">#Symbol#</TD>
<TD BGCOLOR="##EFD6C6">#Exchange#</TD>
<TD BGCOLOR="##EFD6C6">#Last_Traded_Price#</TD>
<TD BGCOLOR="##EFD6C6">#Change#</TD>
<TD BGCOLOR="##EFD6C6">#Last_Traded_Time#</TD>
<TD BGCOLOR="##EFD6C6">#Last_Traded_Date#</TD>
<TD BGCOLOR="##EFD6C6">#Opening_Price#</TD>
<TD BGCOLOR="##EFD6C6">#Days_High#</TD>
<TD BGCOLOR="##EFD6C6">#Days_Low#</TD>
<TD BGCOLOR="##EFD6C6">#Volume#</TD>
</TR>
</CFOUTPUT>
</TABLE>
<CFOUTPUT>
<P>Your browser type is #BrowserType# #Version#
 running on #OS#.<P>
</CFOUTPUT>

<CFIF Version GTE 4>
 <FORM><CF_Print></FORM>
</CFIF>

<B>
<CF_NewWindow
 NewWindowName="External"
 Source="http://forums.allaire.com/"
 CreateLabel="ColdFusion Developers Forum"
 Width="800"
 Height="500"
 Scrollbars="Yes"
 Menubar="No"
 Resizable="Yes">
</B>
</body></html>
```

CODE
LISTING

The code listing for this article can also be located at www.ColdFusionJournal.com

# Using ColdFusion to **Build ColdFusion**

BY
CHRIS
TWENEY

## Get two runtimes with this tag

Picture this: You're the developer of a rapidly growing content site that's beginning to bog down at peak times. You can see that pretty soon the site will be running like molasses around the clock. So you do some research.

You figure out that ColdFusion has a handy built-in mechanism that can help with performance: the CFCACHE tag. Sprinkling CFCACHE tags around strategic areas seems to help, and you rest easy.

But then your manager comes by and says, "You're going to have to make some changes. We need to put a bread-crumb trail at the top of each page that lists each page the user clicked on to get there. And John in Sales would love it if the current time and date were displayed next to the article bylines. You can do that, right?"

There is no way you'll be able to do what your manager wants with the CFCACHE system in place. Without CFCACHE your site is going to win the "Slug Site of the Year" award. What you need is a custom solution that lets you use CF to generate CF. A self-referential miracle, you say? Not at all! All you need is a simple custom tag and a careful eye for escaping your code.

The cf_outputtofile tag (see Listing 1) is really quite simple. You call it like this:

```
<cf_outputtofile>
   This text is saved to a file.
</cf_outputtofile>
```

Then the string "This text is saved to a file" gets saved to outfile.cfm in the directory of the calling template. If you now view outfile.cfm, you'll see that it contains:

```
<CFOUTPUT>
   This text is saved to a file.
</CFOUTPUT>
```

By default, cf_outputtofile wraps the text passed to it in a CFOUTPUT block. You can modify this by passing wrapwithcfoutput="No" as an attribute when you call the tag.

You can specify the output filename by passing an outfile="{filename}" attribute to the tag. You can also specify a directory by passing an outdirectory="{directory path}" attribute. Note that this is a physical path (e.g.,c:\InetPub\www- root\my App), not a logical "Web" path.

If you take a close look at the code for the tag, you'll see that it's basically just a fancy wrapper for the CFFILE tag's ACTION= "Write" method. There are 4 billion (at last count) useful things to do with this CFFILE wrapper. We'll cover just a few here, and leave the remaining billions as an exercise for the reader.

### Caching Simple Dynamic Text

To save a simple dynamic piece of text, do something like this:

```
<cf_outputtofile
   outfile="dynamic_1.cfm">
   The time is: #now()#
</cf_outputtofile>
```

When you run that code, the file myFirstFile.cfm contains this:

```
<CFOUTPUT>
   The time is: #now()#
</CFOUTPUT>
```

Obviously, when you execute myFirstFile.cfm, you're going to see something like:

```
The time is: {ts '2000-09-13
14:03:20'}
```

You must avoid the temptation to put CFOUTPUT around the #now()# function in the code above. If you do so, CF Server will parse the code at the time the cf_outputtofile tag is called, and the output file will contain something like this:

```
 The time is: {ts '2000-09-13 14:03:20'}
```

The point is that you want to store the literal text "#now()#" in your output file, not the interpreted value of "#now()#," since that value will be accurate only when you call cf_outputtofile.

### Caching the Results of a Query

Many of you reading this article may be interested in reducing the number of database calls you need to make. Listing 2 contains an example of using cf_outputtofile with a query (Listings 2 and 3 can be found on the ***CFDJ*** Web site, www.coldfusionjournal.com.) First, run a query; then, output the query's record set within a call to cf_outputtofile.

In the example given in Listing 2, URL.ID = 5, the query will fetch the Article with ID equal to 5, and cf_outputtofile will save it to article_5.cfm. You can then fetch it either directly via a URL or as a CFINCLUDEd template from another page. Of course, if you drop the WHERE clause from the query, you can use the looping construct to cache multiple articles in one pass.

### Caching More Complicated Code

If you need to cache more complicated chunks of code for runtime execution, the best way is to use an intermediary variable that stores the CF code as a string. Examine Listing 3 to see how this is done.

Let's take a closer look at the code chunk in Listing 3. I've spread the CFSET of aChunk out over several lines to make it easier to read. I've also used single quotes instead of double quotes so we can use the regular double quotes within the aChunk block. That way the code looks more normal, and is easier to debug.

The only pitfall here is that when you want to reference a variable (i.e., access a variable where you would usually use a pound sign), you must double your pound signs. Otherwise, just before you call the cf_outputtofile at caching time, the CF Server will complain that it can't find the variable "out."

### Where to Go from Here

You can use cf_outputtofile as part of a caching system that's as simple or as complicated as you need it to be. But remember that caching is only half the story of this tag. The big payoff is that you get two runtimes – one to do the heavy database-lifting logic, and one to do the user- or time-specific personalization elements. That'll keep your servers a lot less busy, and free them up for what you really want to do – serve pages to the zillions of new visitors you get every day!

ABOUT THE AUTHOR
*Chris Tweney has published articles in the* Boston Phoenix *and numerous online outlets including ZDNet. An Allaire Certified ColdFusion instructor, Chris works at roundpeg, where he provides training as well as technical leadership and development on client projects.*

CHRIS@ROUNDPEG.COM

**Listing 1**

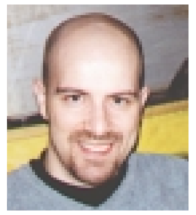
```
<cfsetting enablecfoutputonly= yes >
<!--- ----------------------------------------------------
--------- --->
<!--- cf_outputtofile --->
<!---

 Tag author: Chris Tweney         <chris@roundpeg.com>

 This tag saves text and/or CF code to a file for later
use, probably as part of a caching system that requires
certain parts of content to be executed at runtime, not
cachetime (e.g., a display of the current time at the top
of a news story).

 Usage:

  <cf_outputtofile>
   [ text to be saved to the file ]
  </cf_outputtofile>

 Arguments accepted (all are optional):

  outfile -> The filename (with extension) to which you
want to save the text.
   Default:  outfile.cfm
  outdirectory -> The directory (physical path) to prefix
to the outfile.
   Default: The current directory of the calling template.

  addnewline -> Use CFFILE's  addnewline  option, Yes/No.
Default  Yes
  wrapwithcfoutput -> Wrap the text in a CFOUTPUT block,
Yes/No. Default  Yes

  returnvar -> Variable name to return the text to the
caller. If not specified, the caller s returnvar is not
set. Use if you want to do something else with the text in
addition to caching it to a file.
```

**Listing 1 continued on www.coldfusionjournal.com**

CODE LISTING
The code listings for this article are continued on www.ColdFusionJournal.com

# Extending ColdFusion's **Caching Features**

## Speed up the parts of your pages that need it



BY RONNY PASCH

Users of Web applications want nothing but the best when it comes to the performance of a Web application. The servers they run on work hard to deliver the applications as efficiently and quickly as possible. Several elements and techniques aid in this process, such as caching.

Caching is an invaluable technique when it comes to speeding up your applications. The idea is simple: store the output of a given task for a certain amount of time. On subsequent requests, check if previously stored output is available and use that instead of doing the same task over and over again. Caching will save you hundreds of valuable milliseconds and should be utilized wherever possible within your applications.

Now don't get too excited and start caching everything. Use caching with care and always make sure the right information is cached for the right amount of time.

ColdFusion currently provides two ways of caching, each serving a different purpose.

1. ***QueryCaching:*** You can cache the resultsets returned by queries using the cachedwithin and cachedafter attributes of the <CFQUERY> tag.
2. ***PageCaching:*** You can cache the full output of a page using ColdFusion's <CFCACHE> tag.

QueryCaching is a powerful caching technique in which ColdFusion stores an entire resultset that was returned by a query. Subsequent queries with the same SQL statement won't result in ColdFusion requesting the data from the database again, as it just returns the previously stored resultset. The list of states example is frequently used to illustrate the use of QueryCaching. It's been quite a few years now since a new state was added, so caching this would be a good idea since this data is not about to change soon.

PageCaching is a good caching technique, but it also has its downside – it caches the output of the entire page. In many cases this type of caching can be useful; however, you can't use it for every page. Some pages simply have information that needs to be up to date at all times. Unfortunately, the <CFCACHE> tag only caches full pages and, in my opinion, this is where Allaire went wrong when they created this tag. The <CFCACHE> tag could have been much more powerful if, instead of caching a full page, it cached only parts. There are many pages that contain content that can't be cached because it must be up to date at all times as well as and content that can be cached, thus speeding up performance.

Fortunately for us, you can extend ColdFusion's functionality by creating your own so-called custom tags. What ColdFusion's <CFCACHE> tag lacks, we can fix by writing our own.

## Behold! <CF_CACHE>

The way a custom tag works and the data ColdFusion provides to a custom tag's code is exactly what we need to create our own caching mechanism that's capable of caching parts of a page. It even provides a feature that stores the output data in a variablescope of your choice for greater flexibility and functionality. Later I'll explain why the scope can play an important role, but first let me explain how to use the tag.

The <CF_CACHE> tag (see Listing 1) has a start and end tag, and it's the output of the code between the two tags that will be cached. The <CF_CACHE> tag has four attributes that control it.

1. ***ID:*** Every cacheblock in your code needs to be identified by a

unique ID. <CF_CACHE> needs this ID because it uses it as the key to retrieve your stored output whenever it needs to be displayed again.

2. ***Expires:*** Can be any DateTime Object. Stored data is valid until the Date/Time you specify with this attribute (default = 5 minutes).
3. ***Scope:*** Defines which variablescopes to use to store the output in (default = server).
4. ***Refresh:*** Dynamically refreshes the cached output, regardless of whether or not the expiry Date/Time is reached (default = no).

Only the ID attribute is required for the <CF_CACHE> tag to function. All other attributes are optional. Let's provide an example of this tag (see Listing 2).

What this example does is retrieve all states from a database and displays them using a <SELECT> formfield. The output of this code is then stored in server memory. Since I specified that it expires one day from now (using DateAdd('d', 1, now())), for the next 24 hours whenever this code is executed it won't do the cfquery nor will it process the cfoutput code; it retrieves only the output that was stored in server memory and displays that.

As you can see, the caching technique provided by this custom tag can, in some cases, be more efficient than QueryCaching. With QueryCaching, ColdFusion will still execute the cfoutput code in Listing 2, whereas the <CF_CACHE> tag will simply output one variable.

## Great! But How Exactly Does It Work?

Let's dive into the inner works. <CF_CACHE> makes use of structures to store the output and expiry data. A full explanation of structures and their functionality is beyond the scope of this article, though some information is necessary to understand how they work.

Structures enable programmers to store data in key/value pairs. The key is needed to access its value. For example, you could have a structure named *user*. In this structure a key called "firstname" has the value "Ronny." Another key in the same structure is called "lastname" and has "Pasch" as its value. To output this person's full name, use <CFOUTPUT>#user.firstname# #user.lastname#</CFOUTPUT>.

"

Wouldn't it be great if we could cache five news items for one user and five completely different news items for another?

The structures that <CF_CACHE> uses are called cf_cache_output blocks and cf_cache_timeouts. These names already state that they contain the data of the generated output of the code and their timeouts. The keys we use in both structures are the ID we provide in the ID attribute so, in Listing 2, both structures have a key called "StatesDropDown". The output blocks structure holds the actual generated output and the timeouts structure holds the Date/Time until this cached outputblock is valid. As you can see, it's important that the ID attribute you provide is unique because if you have two different <CF_CACHE> tags in your code, both storing totally different output, some unexpected things may appear on your screen.

<CF_CACHE> executes twice. Once for the start tag and once for the end tag. Let's look at what the start and end tags do.

### Start Tag

1. Checks to see if a key with this ID exists, if the current date/time isn't past the expiry date/time, and if no refresh of the cache was issued using a REFRESH="yes" attribute.
2. If all the above conditions are met, displays the stored output and exits the tag using <CFEXIT METHOD="exittag">, which prevents the code between the <CF_CACHE> tags and the END </CF_CACHE> TAG from executing.
3. If one of the above conditions isn't met, the code between the <CF_CACHE> tags should execute since its output either doesn't exist or has expired, or a refresh was issued using the REFRESH= "yes" attribute.

### End Tag

1. The code of the end tag executes only when one of the above-58 mentioned conditions (Step 1 of the Start Tag) isn't met. All we need to do is store the expiry Date/Time given with the EXPIRES attribute together with the content generated by the code between the <CF_CACHE> tag.

The code is simple and the technique straightforward, but it provides a powerful, flexible caching tool.

## Choosing the Right Scope for the Job

The flexibility of this tag doesn't stop at just caching parts on a page. Using different variablescopes enables you to secure your data (which I'll explain in a bit) and provides great flexibility when it comes to caching data on a per-user basis.

Imagine a Web site that uses personalization. A user defines his or her interests and, based on this data, the Web site delivers the top five news items that meet the user's preference. Wouldn't it be great if we could cache five news items for one user and five completely different news items for another?

There are two approaches to achieving this. The first approach is to use the CFID and CFTOKEN variables or any other means you have that uniquely identifies the user to your application. You could combine a name with the CFID CFTOKEN combination and use this as the value for the ID attribute in the <CF_CACHE> tag.

```
<CF_CACHE ID="Top 5 News Items
#CFID##CFTOKEN#">
```

The second approach is to specify "session" as the variablescope. That way the structures used by the <CF_CACHE> tag are created in the session scope of the user and will be unique. The client scope is not supported by the <CF_CACHE> tag because <CFLOCK> doesn't support this scope. The <CFLOCK> tag is needed around read/writes of the variables in the different variablescopes to ensure data integrity. I could have written the code in such a way that the client scope could also be used; however, I chose to keep the code as efficient as possible. If you really need to use the client scope, the code can be easily changed to support this.

```
<CF_CACHE ID="Top 5 News Items"
SCOPE="session">
```

As I mentioned earlier, using different scopes also provides a way to secure your data. If you don't define a scope, the tag will store your data in the server scope by default. Any other application running on the same server could access this data, so if your application is not the only one running on the same server, it's recommended that you use the application scope to store your cached data. Worst case scenario: when using the server scope and more than one application is running on the same server, you could end up displaying data from a different application and vice versa. Of course, you want to prevent such a situation at all costs, so be careful about the scope you choose. Make sure you have full control when you specify the server scope. In any other case, I recommend using the application scope instead.

## Conclusion

I hope all of you find great use for this tag. I know I have and it has helped me make sure the applications I write perform as efficiently as they can. In my opinion, Allaire should make some changes to their <CFCACHE> tag but until then... this will do!

ABOUT THE AUTHOR
*Ronny Pasch, a certified ColdFusion developer, has over 10 years of programming experience in various programming languages.*

INFO@RONNIEO.COM

**Listing 1**

```
<!--- THROW AN ERROR IF NO END TAG WAS FOUND. --->
<cfif NOT ThisTag.HasEndTag>
 <cfthrow message="No endtag found for the CF_CACHE tag">
</cfif>

<!--- THROW AN ERROR IF NO ID ATTRIBUTE WAS SPECIFIED. --->
<cfif NOT IsDefined("attributes.ID")>
 <cfthrow message="No ID attribute specified for the
CF_CACHE tag">
</cfif>

<!--- DEFAULTS FOR VARIABLES IF NOT SPECIFIED --->
<cfparam name="attributes.scope" default="server">
<cfparam name="attributes.expires"
    default="#DateAdd('n',5,now())#">
<cfparam name="attributes.refresh" default="no">

<!--- CHECK IF THE SCOPE IS VALID --->
<cfif NOT ListFindNoCase("server,application,session",
attributes.scope)>
 <cfthrow message="Invalid scope defined in the
CF_CACHE tag">
</cfif>

<!--- THROW AN ERROR IF EXPIRES ATTRIBUTE IS NOT A
DATETIME OBJECT --->
<cfif NOT IsDate(attributes.expires)>
 <cfthrow message="Invalid value specified as Expires
   attribute for the CF_CACHE TAG">
</cfif>

<!--- CREATE REFERENCES TO THE RIGHT SCOPE AND INITIALIZE
STRUCTURES FOR THE CHOSEN SCOPE IF NECESSARY--->
<cflock timeout="20" throwontimeout="No" type="READONLY"
scope="#attributes.scope#">
 <cfset tmp = attributes.scope & ".cf_cache_output_blocks">
 <cfparam name="#tmp#" default="#StructNew()#">
 <cfset output_blocks = Evaluate(tmp)>
 <cfset tmp = attributes.scope & ".cf_cache_timeouts">
 <cfparam name="#tmp#" default="#StructNew()#">
 <cfset timeouts = Evaluate(tmp)>
</cflock>

<!--- SWITCH TO EITHER START- OR ENDTAG CODE --->
<cfswitch expression="#ThisTag.ExecutionMode#">

 <cfcase value="START">
 <!--- IF AN OUTPUT BLOCK WITH THIS ID EXISTS, HAS NOT
EXPIRED YET AND A REFRESH OF THE CACHE IS NOT FORCED. --->
 <!--- THEN DISPLAY THE CACHED OUTPUT AND CONTINUE
EXECUTING AFTER THE END TAG. --->
  <cflock timeout="20" throwontimeout="No" type="READONLY"
scope="#attributes.scope#">
   <cfif StructKeyExists(output_blocks,attributes.ID) AND
         StructFind(timeouts,attributes.ID) GT Now() AND
             attributes.refresh EQ "no">
    <cfoutput>#StructFind(output_blocks,attributes.ID)#
       </cfoutput>
    <cfexit method="EXITTAG">
   </cfif>
  </cflock>
 </cfcase>

 <!--- THE ENDTAG ONLY EXECUTES WHEN A BLOCK HAS JUST
EXECUTED AND NEEDS TO BE CACHED --->
 <cfcase value="END">
  <!--- INSERT GENERATED CONTENT AND TIMEOUT INTO THE
      STRUCTURES --->
  <cflock timeout="20" throwontimeout="No" type="EXCLUSIVE"
      scope="#attributes.scope#">
   <cfset StructInsert(output_blocks, attributes.ID,
      ThisTag.GeneratedContent, "true")>
   <cfset StructInsert(timeouts, attributes.ID,
      attributes.expires, "true")>
  </cflock>
 </cfcase>

</cfswitch>
```

**Listing 2**

```
<CF_CACHE ID="StatesDropDown" EXPIRES="#DateAdd
( d , 1, now())#">
   <CFQUERY Name="states" datasource="DSN">
   SELECT StateID, StateName
   FROM States
   ORDER BY StateName
   </CFQUERY>
   <select name="state">
   <CFOUTPUT Query="states">
   <option value="#states.StateID#">#states.StateName#
     </option>
   </CFOUTPUT>
   </select>
   </CF_CACHE>
```

CODE LISTING

The code listing for this article can also be located at www.ColdFusionJournal.com

# CFDJnews

## ETI Kicks Off ColdFusion Resource Site

(*Alexandria, VA*) – Enhanced Technologies Inc. has spun off CFHost.net, a Web site devoted to ColdFusion hosting. The site features ColdFusion hosting options, links to developer resources, and over 50 Web applications such as e-commerce packages, intranet components, and human resource tools. www.enhtech.com

## CSMi Uses ColdFusion Products for SEC Site

(*Alexandria, VA*) – CSMi, a technology solutions provider for government, corporate, and nonprofit clients, has announced the Web site launch of its most notable federal client, the U.S. Securities and Exchange Commission (www.sec.gov). CSMi, an Allaire Alliance Partner, selected Allaire ColdFusion and ColdFusion Studio applications to design, prototype, and migrate the former SEC site, containing over 15,000 pages and receiving over 20 million page views per month. www.csmi.com

## Allaire Launches OEM Initiative

(*Newton, MA*) – Allaire Corporation is offering a streamlined program, OEM Jumpstart, to make it easier for independent software vendors to bundle Allaire's standards-based technologies with their own products, shortening production time, controlling development costs, and helping to ensure the future operability of their finished commercial software.

The program includes a license agreement, an online purchasing process with simplified repeat ordering, and redistribution rights. www.allaire.com

## *AppliedTheory* Joins *Allaire* Certified Hosting Partner Program

(*NY, NY*) – AppliedTheory, an Internet knowledge, development, and managed hosting partner for hundreds of large corporations, has qualified as a certified hosting partner for Allaire Corporation.

Allaire recently launched the program to recognize hosting companies that offer top-quality, value-added services around the Allaire platform. AppliedTheory has qualified for the program because it supports ColdFusion in its managed hosting services, offers 24-hour customer support and redundant network operations centers, holds an Allaire Hosting Provider License, and provides a national network. www.appliedtheory.com

## ADVERTISER INDEX

| ADVERTISER | URL | PH | PG |
|---|---|---|---|
| ABLECOMMERCE | WWW.ABLECOMMERCE.COM | 360.253.4142 | 2 |
| ACTIVEPDF | WWW.ACTIVEPDF.COM | 888.389.1653 | 25 |
| ALLAIRE CERTIFIED PROGRAM | WWW.VUE.COM/ALLAIRE | 877.460.8679 | 29 |
| ALLAIRE CORPORATION | WWW.ALLAIRE.COM/DEVELOPER.TRAINING.CFM | 888.939.2545 | 17 |
| ALLAIRE CORPORATION | WWW.ALLAIRE.COM/USERGROUPS | | 23 |
| AMERICAN CYBERNETICS | WWW.MULTIEDIT.COM | 800.899.0100 | 27 |
| CAREER OPPORTUNITIES | CAREERS.SYS-CON.COM | 201.802.3028 | 56 |
| CFDYNAMICS | WWW.CFDYNAMICS.COM | 800.422.7957 | 51 |
| CFXHOSTING | WWW.CFXHOSTING.COM | 866.CFX-HOST | 47 |
| CFXTRAS | WWW.CFXTRAS.COM | 704.408.6186 | 39 |
| CODECHARGE | WWW.CODECHARGE.COM | 650.754.9810 | 4 |
| CONCEPTWARE AG | WWW.CONCEPTWARE.COM/EYE | 011.781.275.6171 | 49 |
| CORDA TECHNOLOGIES | WWW.CORDA.COM | 801.805.9400 | 9 |
| ECONOMIST.COM | WWW.ECONOMIST.COM | 44(0)20 7436 4424 | 57 |
| EMPRIX | WWW.EMPRIX.COM | 781.993.8500 | 7 |
| EVOLUTION B | WWW.EVOLUTIONB.COM/SYNERGYFORFREE | 877.622.7551 | 59 |
| HOSTCENTRIC | WWW.HOSTCENTRIC.COM/CFDJ | 888.932.9376 | 15 |
| HOSTMYSITE.COM | WWW.HOSTMYSITE.COM | 877.215.HOST | 38 |
| INTERMEDIA.NET | WWW.INTERMEDIA.NET | 800.379.7729 | 60 |
| JAVAEDGE CALL FOR PAPERS | WWW.SYS-CON.COM/JAVAEDGE | 201.802.3069 | 45 |
| JAVAEDGE CONFERENCE & EXPO | WWW.SYS-CON.COM/JAVAEDGE | 201.802.3069 | 32-33 |
| JDJ STORE | WWW.JDJSTORE.COM | 888.303.JAVA | 46 |
| PACIFIC ONLINE | WWW.PACONLINE.NET | 877.503.9870 | 39 |
| PAPERTHIN | WWW.PAPERTHIN.COM | 800.940.3087 | 21 |
| SYS-CON CUSTOM MEDIA | WWW.SYS-CON.COM | 925.244.9109 | 43 |
| TECHIEGOLD | WWW.TECHIEGOLD.COM | 617.267.4700 | 56 |
| TOPHOSTS.COM | WWW.TOPHOSTS.COM | 306.546.3778 | 3 |
| WEB SERVICES EDGE 2001 | WWW.WEBSERVICESEDGE2001.COM | 617.267.4700 | 56 |
| WIRELESS BUSINESS & TECH | WWW.WIRELESS-MAGAZINE.COM. | 201.802.3069 | 53 |
| WOODBOURNE SOLUTIONS | WWW.WOODBOURNESOLUTIONS.COM | 877.668.8575 | 51 |
| XMLEDGE CONFERENCE & EXPO | WWW.SYS-CON.COM/XMLEDGE | 201.802.3069 | 40-41 |

## Next Month...

*Here's a sneak peek...*

*Welcome to Allaire Spectra 1.5*
*A close look at some features*
*by Raymond Camden*

*Spectra and Flash*
*A powerful combination*
*by Matt Tatam*

*Java for ColdFusion Developers*
*What do EJB, JTA, JMS, JSP, and servlets mean to ColdFusion developers?*
*by Kevin Hoyt*

*VTML by Example*
*How to successfully extend the ColdFusion Studio IDE*
*by Christian Schneider*

COLDFUSION Developer's Journal

*Don't miss the May issue!*

?