

CHAPTER 43

ColdFusion and Globalization

IN THIS CHAPTER

Why Go Global? E1

What Is Globalization? E2

Going Global E3

Better G11N Practices E40

In this chapter, we'll look at how ColdFusion can help you build truly global ColdFusion applications. Before we begin, let's consider some fundamental globalization concepts.

Why Go Global?

Taking the high road, I suppose we can say that Web applications need to *go global* in order to deal with humanity's vast diversity—truth be told, it's all about money. If you've been paying attention to Internet statistics for the past few years, you already know that much of the new growth in the Internet is occurring outside the United States (in fact, North America is already at almost 75 percent Internet penetration).

Table 43.1 summarizes some important Internet usage statistics and pretty much speaks for itself. And if these statistics aren't impressive enough, you might consider the “backyard globalization” that's occurring in the U.S. (and elsewhere). According to an article in the *San Antonio Express-News*, Hispanics in the U.S. now outnumber Canadians in Canada. That's 38.8 million people in a *growing* marketplace with an estimated \$675 billion in annual purchasing power—certainly something to think about the next time you're designing a ColdFusion application.

Table 43.1 Global Internet Use

REGION	INTERNET USERS	GROWTH (2000–2009)	PENETRATION
Africa	65,903,900	1359.90%	6.7%
Asia	704,213,930	516.1%	18.50%
Europe	105,096,093	282.9%	50.10%
Middle East	47,964,146	1360.20%	23.70%
North America	251,735,500	132.9%	73.90%

Table 43.1 (CONTINUED)

REGION	INTERNET USERS	GROWTH (2000–2009)	PENETRATION
Latin America/Caribbean	175,834,439	873.10%	30.0%
Oceania/Australia	20,838,019	173.40%	60.10%

Source: Internet World Stats Web site (<http://www.internetworldstats.com/stats.htm>)
June 2009.

You can now see the “why” of going global, but what exactly does globalization mean?

What Is Globalization?

Globalization refers to deploying an I18N-ready application across several locales. What’s I18N? We’re getting ahead of ourselves a bit here, so let’s first define some terminology so we can understand the globalization concept.

Globalization Terminology

To follow the discussions in the rest of this chapter, you need to know the meaning of a few terms used in the software globalization industry.

- **Locale** is the most fundamental part of globalization. Locales are languages and other cultural norms (calendars; date, number, and currency formatting; spelling; writing system direction; and so forth) that are specific to a geographic region. The French used in Quebec is not exactly the same as the French used in Paris. Both HTML and XML define locales rather plainly as “language-country”—that is, only as a basic language identifier (though it can encompass such things as “en-scouse,” the English Liverpudlian dialect known as Scouse). Java, and by extension ColdFusion, include other cultural information that is locale specific.
- **Internationalization or I18N** (I18N is an abbreviation for the 18 letters between the *i* and the *n* in internationalization) refers to the design and development of an application so that it functions in at least two locales. You can think of I18N as making an application language or locale neutral.
- **Localization or L10N** (L10N is an abbreviation for the 10 letters between the *l* and *n* in localization) describes the process after I18N, of adapting an application to a specific locale. L10N might be best thought of as the process of skinning a locale-neutral I18N application into a specific locale.
- Finally, **globalization or G11N** (that’s right, another abbreviation) is sometimes used as a synonym for I18N. But to me, it’s the actual application implementation (L10N) across several locales after I18N—in other words, globalization is both I18N and L10N.

Now that we know what globalization really means, let’s look at an overview of how we can accomplish it.

Dancing the Globalization Jig

How you go about globalizing your ColdFusion application depends on whether you have an existing code base. (And let me emphasize here that making an existing application I18N means you have a tough row to hoe.) The process entails these basic steps:

- First you review your existing application components to identify obstacles to the I18N process. Application components include your ColdFusion code, the application's display tier (including Flex, Flash, HTML, JavaScript, and so on), ColdFusion tags, middleware, and your back-end database. An I18N obstacle is an area where the component needs either amending to make it I18N, or replacement by an I18N-capable version.
- The next step is to actually amend or replace non-I18N application components. Wholesale replacement of existing components is by far the easiest thing to do. For instance, if your database doesn't support Unicode (the ins and outs of Unicode are discussed shortly), simply replace it with one that does. Amending existing components is another story. It's a task that's often described as "mind numbing," "brutal," "death by a thousand cuts," and "torturous," as well as a few choice adjectives that the editor wouldn't let me use here. We'll get into these devilish details later on in this chapter.
- Next, you localize the application by providing translated text resources and display tier layouts, graphics, and so on.
- Finally, you document the whole mess in the appropriate locale/language.

The bottom line: Considering the amount of work required to make an existing application I18N, it's a darned good idea to design/code your application from the ground up for I18N. If you do that, all you need do is localize and document. Much simpler, isn't it?

Now that the background information's out of the way, we can get down to the real nitty-gritty of creating a G11N ColdFusion application.

Going Global

Let's start with the most fundamental part of G11N: locales.

Locales

Among the first things to consider when making a ColdFusion application G11N is what language your application's users want to use and, possibly, where the users are located. Knowing users' locale helps you better tailor your application's language response to them. In globalization, *locales* relate to users' languages and cultural norms, such as sorting conventions; formatting of currency, time and dates, and numbers; and even the spelling of common words (*colour* versus *color*, for instance). Put more simply, a locale is a language as used in a specific country or a region.

Locales are probably the most important piece of G11N—you absolutely need to get them right—and luckily for us, ColdFusion 9 really shines in this area in comparison to previous versions of ColdFusion. Since ColdFusion MX 7, ColdFusion has natively supported all the 130 or so locales

that core Java does. As a really masterful ease-of-use enhancement, in ColdFusion you can reference these locales using *standard* Java-style locale notation. You can refer to English as used in New Zealand simply as `en_NZ`, rather than `English (New Zealand)`. Besides all the typing and spelling errors this will save you, it helps streamline and standardize locale usage, not to mention making synchronization with Java `IL8N` objects easier.

Since locales are so important, we're going to take a closer look at them in this section, including the following:

- How can we determine a user's locale?
- Why do we need to maintain a user's locale choice?
- Are there any locale resources beyond what ColdFusion offers?
- What's the best Java library to support `IL8N` in ColdFusion?
- What can we do about locale-based collation (sorting)?

Determining a User's Locale

It's critically important to match a user's locale to the locales that your application supports. Matching what the user wants and what your application can actually deliver is often called *language negotiation*. So how do we do that? The quick-and-dirty answer is to simply ask them to choose from among the supported locales, maybe using a simple `HTML form select` as the very first thing they see when entering the application. The quick-and-dirty way, however, doesn't make for the best user experience; it's intrusive and disruptive, wastes users' time on things outside the real purpose of the application, often makes a bad first impression, and frankly, it's just not considered cool. In general, it's better to transparently determine a user's locale, initialize the application to use that locale, and then offer the user a way to manually change locales as part of the application's navigation interface.

TIP

Using national flag graphics as navigation aids to allow users to swap locales is generally considered bad form. For starters, it doesn't scale well; what might work with flags for 2 locales probably won't work for 52. This technique also tends to upset some folks when used with languages that cross many locales, such as English (some Brits and Aussies don't appreciate their language being represented by the U.S. flag) and, even more so, Chinese. Resist the urge to get cute.

How do we “transparently determine a user's locale”? It would be ideal if the user's ISP or browser told us precisely where the user was located—from that information we could determine their likely locale. One way to accomplish this involves geolocation, in which a user's IP address is used to look up (usually via a copy of the WHOIS database) their country. Determining locale is a common enough need that several projects, commercial and open source, have been developed to solve this problem.

For instance, the cleverly named `geoLocator CFC` does precisely this, using the open source Java IP (`InetAddress`) Locator project (<http://javainetlocator.sourceforge.net/>) as its IP/country lookup engine. The `CFC's findLocale` method takes as arguments the user's IP, CGI variable

`http_accept_language`, and a fallback locale, and returns the most likely locale (or the fallback locale if it can't decide) for that user's combination of IP (country) and `http_accept_language` (which reflects user's actual locale choices for their browser). The `geoLocator` CFC can be downloaded free from <http://www.sustainableGIS.com/projects/tz/testTZCFC.cfm>, where you'll also find details about the CFC.

So why don't we just use the CGI variable `http_accept_language` and forget the CFC? Many reasons: Older browsers don't support it, not every user has bothered to set their language and locale preferences, and some user-supplied `http_accept_language` variables are obviously made-up languages and locales (Klingon, for example). Also, since `http_accept_language` can be a list of language/locale preferences, parsing these can become problematic (especially coming from browsers on Apple computers, which produce some of the longest `http_accept_language` lists I've ever seen). The `geoLocation` method is more robust and has some added benefits; it's also useful for other things besides determining a user's locale. You can use it for country-level Web traffic analysis, screening international orders (for instance, "We won't sell betel nut to anybody living in Timbuktu"), helping to determine and price products in local currencies, and so on.

TIP

It's considered good practice to display a user's locale choices in the language of that locale (the choice for French in French, Thai in Thai, and so forth).

Locale Stickiness

Now that we know a user's locale, what do we do with it? Well, the first thing is not to forget it. Say you have a Web application supporting three locales, Thai, Russian, and U.S. English (as the default locale). The `geoLocator` CFC determines that a user in Bangkok has a `th_TH` (Thai language in Thailand) locale. This user gets the home page of the Web site in Thai, with correctly formatted Thai dates and numbers, and so on. The user then navigates to a subsection of the Web site and only sees U.S. English. The application has promptly forgotten their locale and reverted to the default.

This might seem to be a rather trivial issue, but it's an important part of developing a G11N ColdFusion application. There are several approaches to fixing this: the monolingual Web site (more on this in the later section "Better G11N Practices" section), which is more of a high-level design choice than a ColdFusion coding technique; saving the locale to shared scope variables (usually `SESSION` scope); or passing locale as part of the URL string (for example `index.cfm?locale=fr_CA`). Pick one technique; just please don't forget your user's locale.

We'll examine more uses for a user's locale later on, but next let's look at what happens when we need a locale that ColdFusion doesn't support.

CLDR: The Common Locale Data Repository

As stated earlier, ColdFusion derives its locale information from core Java. While this will provide enough locale coverage to satisfy most ColdFusion G11N applications, there will be occasions where it's not sufficient—say, when you need to support Farsi or Vietnamese. For those

situations, you'll either need to do your own locale research (and from my own personal experience, I can quite easily say bah, humbug to that idea), or you can look elsewhere for some sort of standardized locale resources. These days, "elsewhere" is the *Common Locale Data Repository (CLDR)*.

Originally a project sponsored by the Free Standards Group's OpenI18N team (<http://www.openi18n.org/>), the CLDR project was handed off to the Unicode Consortium (<http://cldr.unicode.org/>) in early 2004. CLDR's locale resources, as of version 1.7.1, cover 468 locales, including 146 languages and 159 territories. Compare that to the 150 or so locales provided by core Java, and you can understand the real significance of the CLDR. Specifically, the CLDR provides information concerning number/date/time formatting, currency values, as well as support for measurement units and text sorting order (collation). If you find yourself working with a client whose locale or language is not in the CLDR, get in touch with SETI (<http://www.seti.org/>); you might very well be dealing with an alien.

You're probably asking yourself just how to take advantage of the CLDR. The short answer is to find a tool or component that is based on the CLDR. Let's take a quick peek at IBM's ICU4J, which is, as of version 3.2, based on the CLDR.

IBM's ICU4J

One of the truly big deals of ColdFusion's move to Java was the ease of integrating Java libraries into ColdFusion applications. For G11N applications, the mother of all Java libraries has to be IBM's open source International Components for Unicode for Java, a.k.a. ICU4J (<http://site.icu-project.org/home>). The ICU4J library fills in many of the gaps in core Java's I18N functionality, such as providing non-Gregorian calendars, beefier number formatting including scientific notation and spell-out, speedier locale-based collation, international holidays, and of course all 468 CLDR locales. (We'll discuss a couple of these items in later sections.) Plain and simple, *if you do serious G11N work, you might eventually need to use this library.*

TIP

Much of the ICU4J goodness has already been encapsulated in ColdFusion CFCs. You can find these in my shop's Web site (<http://www.sustainableGIS.com/things.cfm>).

Listing 43.1 shows a simple comparison between core Java and ICU4J using Farsi locale (fa_IR, the Persian or Farsi language as used in Iran). The first thing to note is that core Java methods were used instead of ColdFusion LS functions. Why? Simply because Farsi is not one of the supported ColdFusion locales. ColdFusion behaves differently than core Java, in that ColdFusion throws an error (`coldfusion.runtime.locale.CFLocaleMgrException`) rather than using a fallback locale as core Java does. Notice that the `getDisplayName` method with a `Locale` or `ULocale` (for ICU4J) as its argument simply displays the localized name for that locale. Another major difference is the use of ICU4J's `ULocale` class rather than core Java's `Locale`.

For you to use the ICU4J library, it must first be placed on ColdFusion's class path. Download the most current version from <http://icu-project.org/download/4.2.html#ICU4J> and place the JAR files in the `coldfusion_install_location\wwwroot\WEB-INF\lib` directory. You may need to stop

and restart your ColdFusion server service to pick up the new jar file. Another option is to use the excellent javaLoader CFC to load the JAR file. You can find this CFC at RIAForge, at <http://java-loader.riaforge.org/>.

Listing 43.1 compareFarsiLocales.cfm—Comparison of ICU4J/Core Java for Farsi Locale

```
<cfprocessingDirective pageencoding="utf-8">
<!--
this example assumes that you have installed the latest version of ICU4J
-->
<cfsilent>
<!--
compares Farsi locale date formatting and name display using core java and icu4j
NOTE: made verbose for clarity
-->
<cfscript>
// full date format, common to both core java and icu4j
fullFormat=javacast("int",0);
// core java
farsiLocale=createObject("java","java.util.Locale");
farsiLocale.init("fa","IR");
coreJavaDateFormat=createObject("java","java.text.DateFormat");
coreJavaDF=coreJavaDateFormat.getDateInstance(fullFormat,farsiLocale);
////////////////////////////////////
// icu4j magic
farsiULocale=createObject("java","com.ibm.icu.util.ULocale");
farsiULocale.init("fa_IR"); // note the nifty init locale syntax
icu4jDateFormat=createObject("java","com.ibm.icu.text.DateFormat");
icu4jDF=icu4jDateFormat.getDateInstance(fullFormat,farsiULocale);
</cfscript>
</cfsilent>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>locale comparison</title>
<meta content="text/html; charset=UTF-8" http-equiv="content-type">
</head>

<body>
<!-- output what we've done -->
<cfoutput>
<b>core Java</b>: #farsiLocale.getDisplayName(farsiLocale)# #coreJavaDF.
format(now())#
<br><br>
<b>ICU4J</b>: #farsiULocale.getDisplayName(farsiULocale)# #icu4jDF.format(now())#
</cfoutput>
</body>
</html>
```

We'll need to see some output from this example (shown in Figure 43.1) in order to understand another important distinction between ColdFusion /core Java and ICU4J. Since it doesn't have any locale resource data for the `fa_IR` locale, core Java falls back on the default locale for the server (in this case, `en_US`) and produces "Persian (Iran)" as the localized name. Although the dates are exactly the same (produced using the default Gregorian calendar), the output formats are quite

different. ICU4J formats the date display using the Farsi locale resource data; that is, besides localized Farsi date part names, it also uses Arabic-Indic digits rather than European digits.

Figure 43.1

Comparison of ICU4J/core Java output for Farsi locale.

core Java: Persian (Iran) Tuesday, October 13, 2009

ICU4J: فارسی (ایران) سه‌شنبه ۱۳ آکتبر ۲۰۰۹

Is there any benefit to using ICU4J with locales that are supported by ColdFusion/core Java? In some cases there is. For example, let's compare ColdFusion to ICU4J for a locale supported by both: ar_AE, or Arabic (United Arab Emirates). This comparison is also a good example of the benefits of Java-style locale syntax. Listing 43.2 offers this simple example. Things to note are:

- ColdFusion brings simplicity that ColdFusion brings to G11N.
- A single function, `setLocale`, sets ColdFusion's locale for that page.
- The `getLocaleDisplayName` function returns a localized name for this locale similar to ICU4J's `getDisplayName` function.
- The `lsDateFormat` returns a formatted date for this locale similar to ICU4J's `format` method—and this is where we find another fly in the locale ointment.

Listing 43.2 `compareCFLocales.cfm`—Comparison of ICU4J/ColdFusion for Arabic Locale

```
<cfprocessingDirective pageencoding="utf-8">
<!--...

--->
<cfsilent>
<!--...
compares arabic locale date formatting and name display using ColdFusion and icu4j
made verbose for clarity
--->
<cfscript>
// ColdFusion , yup that's all there is to it
oldLocale=setLocale("ar_AE");
////////////////////////////////////
// full date format
fullFormat=javacast("int",0);
arabicUlocale=createObject("java","com.ibm.icu.util.ULocale");
arabicUlocale.init("ar_AE"); // nifty init syntax
icu4jDateFormat=createObject("java","com.ibm.icu.text.DateFormat");
icu4jDF=icu4jDateFormat.getDateInstance(fullFormat,arabicUlocale);
</cfscript>
</cfsilent>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>locale comparison</title>
<meta content="text/html; charset=UTF-8" http-equiv="content-type">
</head>
```


Listing 43.2 (CONTINUED)

```

<body>
<!-- output what we've done --->
<cfoutput>
<b>ColdFusion </b>: #getLocaleDisplayName("ar_AE","ar_AE")#
#lsDateFormat(now(),"full")#
<br><br>
<b>ICU4J</b>: #arabicULocale.getDisplayName(arabicULocale)# #icu4jDF.format(now())#
</cfoutput>
</body>
</html>

```

Figure 43.2 shows the output from this example. Although ColdFusion certainly gets the localized date parts (month and day of week) correct, it doesn't fully support Arabic-Indic digits for the numeric parts (year and day of month) of the date format; ICU4J, however, does. In general, for locales supported by ColdFusion/core Java, all Arabic locales in ColdFusion will yield date/time and numeric formatting incorrectly using European instead of Arabic-Indic digits. Note that this is an issue with the underlying core Java, and *not* with ColdFusion per se.

Figure 43.2

Comparison of ICU4J/
core Java output for
Arabic (United Arab
Emirates) locale.

<p>ColdFusion : ٢٠٠٩ ١٤ أكتوبر، العربية (الإمارات)</p> <p>ICU4J : ٢٠٠٩ ١٤ أكتوبر، الأربعاء، العربية (الإمارات العربية المتحدة)</p>
--

Note that some differences exist between core Java and ICU4J locale data, some of which are indeed minor. For example, ColdFusion/core Java returns named time zones (ICT for machines using Bangkok, Thailand, time zones), whereas ICU4J returns time zone information as offsets from UTC (UTC+07:00). Also, for the `da_DK`, Danish (Denmark) locale, ColdFusion/core Java returns `30. januar 2005` (no day part name), whereas ICU4J returns `søndag 30 januar 2005` (day part name, but no period after day of month part). In many cases, the meaning of the output display is still relatively clear, but again, the devil is in the details. The locale resource used by your application will depend entirely on the locales you need to support, on whether the Unicode Consortium's CLDR has any official meaning to you or your users, or perhaps on other factors such as non-Gregorian calendars.

The final piece of the locale puzzle we'll look at is collation, or sorting.

Collation

Collation is a peculiar thing. It's more or less a universal user requirement, and getting it wrong will certainly make users think less of your application. But getting it right across many locales will also certainly go unnoticed; most users think sorting is quite trivial and do it routinely almost unconsciously. Further complications arise because collation is not consistent for the same characters; for instance, people of German, French, and Swedish nationality sort the same characters differently. Collation is not even consistent within the same language, as in so-called phone-book collation as opposed to sorting in dictionaries and book indices. And that's just the alphabet-based scripts—Asian ideograph collation can be either phonetic or based on the appearance (strokes) of the characters.

Then there are the special cases based on user preferences: ignore/consider punctuation, case (*A* before/after *a*), and so on. You're looking at thousands of years of human collation baggage, so yes, it's going to be complex, even if users do think it's pretty minor. You can read more about the Unicode Consortium's take on collation at <http://www.unicode.org/reports/tr10/>.

As a rule of thumb, your application should first take advantage of your database's collation functionality. Quite a bit of research time and effort was put into this. Most of today's "big iron" databases can handle substantial collation complexity and even cast result sets to a collation other than that table or database's default. See Listing 43.3 for an example using Microsoft SQL Server's `COLLATE` clause. The subsequent discussion deals with cases where we have to sort within a ColdFusion page, as in Query-of-Query or when sorting a list or an array.

NOTE

Fine-tuning collation/sorting to a given locale is more important than many developers think. Most users would think an application plain stupid if it couldn't even sort their alphabet correctly.

Listing 43.3 `castCollation.cfm`—Casting Collation with Microsoft SQL Server

```
<!--
snippet showing MS SQL Server syntax to cast from default collation, say SQL_Latin1_
General_Cp1250_CS_AS (case & accent sensitive) to SQL_Latin1_General_Cp1250_CI_AS
(case insensitive, accent sensitive)
this should produce a resultset ordering that ignores case
-->
<cfquery name="getTaxRoll" datasource="municipalINFO">
    SELECT title+' '+firstName+' '+LastName as taxPayer
    FROM taxRoll
    ORDER BY COLLATE SQL_Latin1_General_Cp1250_CI_AS
</cfquery>
```

Suppose we have this scenario:

- Application serving German locale (`de_DE`)
- Requirement to sort an array of names
- Users bitterly complaining that results aren't being sorted correctly

Let's examine what's happening here to see what we can do about shutting up those darned users. The application is quite logically using the `arraySort` function. The problem is that the sorted results aren't at all what the user expects. Names with umlauts (Ä, Ë, Ü) are sorting together as a group *after* the unadorned characters (A, E, U), rather than as most German users would expect, which would be more along the lines of AÄËÜ (the commonly used German phone-book or DIN-2 collation).

Why is this happening? Because all of ColdFusion's collation functionality is based on sorting sequential Unicode code points (see Table 43.2 for an example). This will work for users in most locales; after all, $a < b$ is true for both lexicographical (dictionary) and Unicode orders. However, it obviously won't work for languages/locales with collation orders that differ from the Unicode code point order.

Table 43.2 Some Unicode Code Point Values

CHARACTER	DECIMAL VALUE
A	41
E	45
U	55
Ä	196
Ë	203
Ü	220

As usual, the solution to this conflict for G11N issues in ColdFusion is to make use of the underlying Java functionality—specifically, core Java’s `java.text.Collator` class or ICU4J’s `com.ibm.icu.text.Collator` class. Either of these classes allows you to perform locale-sensitive string comparison, although the ICU4J class handles collation considerably better (see <http://site.icu-project.org/charts/collation-icu4j-sun> for details). Listing 43.5 later in this chapter provides a look at the use of ICU4J to solve this problem, but before we can make sense of this example, we’ll have to examine how core Java and ICU4J actually handle collation.

In Java (both plain Java and ICU4J), collation complexity is handled using three parameters: `locale`, `strength`, and `decomposition`. The `locale` parameter is obvious; a specific locale’s collation data is used to order sorts (and searches). The `strength` parameter is used across locales (although exact strength assignments vary from locale to locale) and determines the level of difference considered significant in comparisons. There are four basic strengths:

- **PRIMARY.** Significant for base letter differences; *a* versus *b*.
- **SECONDARY.** Significant for different accented forms of the same base letter (*o* versus *ô*).
- **TERTIARY.** Significant for case differences such as *a* versus *A* (but, again, differs from locale to locale).
- **IDENTICAL.** All differences are considered significant during comparison (control characters, precomposed and combining accents, etc.).

ICU4J adds a fifth strength, **QUATERNARY**, which distinguishes words with/without punctuation.

Let’s take an example from the Java docs (<http://java.sun.com/javase/6/docs/api/java/text/Collator.html>). In Czech, *e* and *f* are considered primary differences; *e* and *è* are secondary differences; *e* and *E* are tertiary differences; and *e* and *e* are identical. Got that?

The `decomposition` parameter is just what it sounds like: Characters are decomposed for comparison. There are three basic decompositions (only two for ICU4J):

- **NO_DECOMPOSITION.** Characters are not decomposed; accented and plain characters are the same. This is the fastest collation but will only work for languages without accented (and so on) characters.

- **CANONICAL_DECOMPOSITION.** Characters that are canonical variants are decomposed for collation; that is, accents are handled.
- **FULL_DECOMPOSITION.** Not only accented characters, but also characters that have special formats are decomposed (this decomposition doesn't exist in ICU4J; **CANONICAL_DECOMPOSITION** is used instead). Basically, un-normalized text is properly handled.

TIP

The `ic4jSort.cfc` wraps up both the core Java and ICU4J versions of locale collation, including functions to sort queries. You can find it in the usual places (mentioned previously).

Now that we understand how collation works in core Java and ICU4J, let's consider the example code in Listing 43.4.

Listing 43.4 `icu4jSort.cfm`—ICU4J-Based Locale Array Sorting Function

```
<!--
authors:hiroshi okugawa <hokugawa@macromedia.com>
        paul hastings <paul@sustainableGIS.com>
notes:  this method handles sorting string arrays using locale based collation.
originally part of i18nSort.cfc. note that this code has been made verbose for
clarity.
-->
<cffunction name="icu4jSort" output="No" returntype="array" hint="returns array
sorted using ICU4J collator">
<cfargument name="toSort" type="array" required="yes">
<cfargument name="sortDir" type="string" required="no" default="Asc">
<cfargument name="thisLocale" type="string" required="no" default="en_US">
<cfargument name="thisStrength" type="string" required="no" default="TERTIARY">
<cfargument name="thisDecomposition" type="string" required="no" default="FULL_
DECOMPOSITION">
<cfscript>
var icu4jCollator=createObject("Java","com.ibm.icu.text.Collator");
var uLocale=createObject("Java","com.ibm.icu.util.ULocale");
var tmp="";
var i=0;
var strength="";
var decomposition="";
var thisCollator="";
var locale=uLocale.init(arguments.thisLocale);
// Arrays object to handle sort
var Arrays = createObject("java", "java.util.Arrays");
//set up the collation options
//strength of comparison
switch (arguments.thisStrength){
//handles base letters 'a' vs 'b'
    case "PRIMARY" :
        strength=icu4jCollator.PRIMARY;
        break;
//handles accented chars
    case "SECONDARY" :
        strength=icu4jCollator.SECONDARY;
        break;
```

Listing 43.4 (CONTINUED)

```

//handles accented chars, ignores punctuation
case "QUATERNARY" :
    strength=icu4jCollator.QUATERNARY;
    break;
//all differences, including control chars are considered
case "IDENTICAL" :
    strength=icu4jCollator.IDENTICAL;
    break;
//includes case differences, 'A' vs 'a'
default:
strength=icu4jCollator.TERTIARY;
}
//decompositions, only 2 for icu4j
//fastest sort but won't handle accented chars, etc.
if (arguments.thisDecomposition EQ "NO_DECOMPOSITION")
    decomposition=icu4jCollator.NO_DECOMPOSITION;
else //compromise, handles accented chars but not special forms
decomposition=icu4jCollator.CANONICAL_DECOMPOSITION;
//set collator to required locale
thisCollator=icu4jCollator.getInstance(locale);
thisCollator.setStrength(strength);// set strength
thisCollator.setDecomposition(decomposition);//set decomposition
tmp=arguments.toSort.toArray();
//do the array sort based on this collator
Arrays.sort(tmp,thisCollator);
if (arguments.sortDir EQ "Desc") { //need to swap array?
    arguments.toSort=arrayNew(1);
    for (i=arrayLen(tmp);i GTE 1; i=i-1) {
        arrayAppend(arguments.toSort,tmp[i]);
    }
} else arguments.toSort=tmp;
return arguments.toSort;
</cfscript>
</cffunction>

```

The first thing to note (again) is the use of ICU4J's `ULocale` class rather than core Java's `Locale` class. The next point is the use of core Java's `Arrays` class; we're using it because it can accept a `Collator` object that we begin to build by sorting out (pun intended) what strength and decomposition to use for this `Collator`. We then build the `Collator` for this locale:

```
thisCollator=icu4jCollator.getInstance(locale)
```

We next have to turn the ColdFusion Array into a Java Array (in order to use the `Arrays` object's nifty sorting methods), using:

```
tmp=arguments.toSort.toArray()
```

Now we're ready to actually do the sort using the `Arrays` object, quite simply:

```
Arrays.sort(tmp,thisCollator)
```

The last thing we have to handle is the direction of the sort (ascending or descending), swapping the array around if the calling page required descending sort direction.

What happens if the locale we're interested in isn't one of the locales for which ICU4J has actual collation data? ICU4J will silently fall back on the Unicode Collation Algorithm (UCA), which should suffice for many of these locales. You can read more about how the UCA works at <http://www.unicode.org/reports/tr10/>. You can also construct your own collation using ICU4J's `com.ibm.icu.text.RuleBasedCollator` class. Besides creating new collations, this class also allows you to combine existing collations or customize individual collations to suit specific needs.

NOTE

By now you might be starting to suspect that G11N ColdFusion code isn't exactly rocket science, and you're right. You can pretty much use any style or framework that you're comfortable with. As long as you follow most of the principles and information laid out in this chapter, you should be good to go.

The preceding discussion has given you a good handle on the ins and outs of locales, so let's examine the next G11N issue, the always-fun task of character encoding.

Character Encoding

In my experience, many (perhaps too many) ColdFusion developers get into some kind of trouble over character encoding. This section is going to provide you with the one single answer to all your character encoding problems; it goes like this: *Just use Unicode*. Let's review some of the more important aspects of character encoding as they apply to ColdFusion.

Not Unicode? Not So Smart

I suppose it would be useful to see what ColdFusion has to say about character encoding. Quoting from the *Developing ColdFusion Applications* documentation: “*Character encoding* maps each character in a character set to a numeric value that can be represented by a computer. These numbers can be represented by a single byte or multiple bytes.” Great, but what that doesn't mention is that it's not unusual for a language to have more than one encoding. For example, English has both 8-bit ISO-8859-1 or Latin-1, and 7-bit ASCII; Japanese has Shift-JIS, EUC-JP, and ISO-2022-JP encodings; and, well, we won't get into the Chinese encodings. Furthermore, not all characters for a given language are represented in every encoding used for that language. For instance, the Euro symbol (€) isn't found within the ISO-8859-1 encoding. (The ISO encoding came before the Euro was established as the default currency in the EU.)

If this weren't enough variety, some character sets appear to be equivalent (at least to some folks) but are in fact not. Many developers think ISO-8859-1 and Windows-1252 are the same character set, when in fact Windows-1252 (also called Windows Western or Windows Latin-1) is more like a superset of ISO-8859-1. The mistake of copying and pasting characters from Word documents into HTML forms using ISO-8859-1 encoding highlights this issue pretty nicely. This is particularly troublesome if no encoding metadata is available for a chunk of text. G11N projects are prone to this misstep owing to the need for translations, often done by non-IT professionals who quite often wouldn't know a character encoding if it fell on their heads.

Let's summarize some things about character sets:

- Undeniably, there are a lot of character sets floating around (see the IANA's page on character sets, <http://www.iana.org/assignments/character-sets>.) I stopped counting at 75.
- The same character encoding can be used in different languages.
- Many languages are covered by several character sets.

That kind of wild variety is one of the things I loathe as a ColdFusion G11N developer. Matching the correct encoding to a language is quite difficult when there are multiple possible encodings for a language; you're bound to get it wrong once in a while. In fact, getting it wrong happens so often that a Japanese term, *mojibake* (文字化け), literally "ghost characters" or "disguised characters," has crept into the G11N vernacular to describe this situation. The term is used to designate the nonsense text that occurs because of the original text's being corrupted by bad or missing character encoding. For instance, *mojibake.tif* becomes the *mojibake \$BJ8;z2=\$1(J when the character encoding is incorrect (this was taken from some email correspondence). Encoding has to match end-to-end, and getting that 100 percent correct 100 percent of the time isn't trivial.*

Unicode

A lot of variety means a lot of choices, and that's not always a good thing. So what can we do to simplify things? You already know the answer to that: *Just use Unicode*. So what's so hot about Unicode?

- It's a standard (synchronized with the ISO 10646 standard).
- It's Internet ready (XML, Perl, Java, JavaScript, and so on all support Unicode).
- It's multilingual (see <http://www.i18nguy.com/unicode/char-count.html>).
- It travels well (text in any language can be easily exchanged globally).
- It offers monolithic text processing (and that, of course, saves you money in development and support costs, time to market, and so forth).
- It has wide industry support (Macromedia, IBM, Microsoft, HP, Sun, Oracle, and more), making it vendor neutral where pretty much nothing else is.
- It continually evolves (it's now version 5.0.0).
- It's possible to convert from legacy code pages (see <http://www.unicode.org/Public/MAPPINGS/>).
- It's more or less apolitical (see the member list at <http://www.unicode.org/unicode/consortium/memblast.html>).
- The W3C is recommending it for I18N HTML content.

NOTE

For the real skinny on Unicode, visit www.unicode.org or www.macchiato.com.

Internally, ColdFusion uses Unicode (UCS-2), which is efficient to process because its fixed width (2 bytes per character), but economical bandwidth usage requires single-byte encoding. To me, Unicode smells *inefficient*. However, the twin goals of development simplification and long-term code management are much more important than any superficial bandwidth inefficiency.

Now before you start complaining, “Hey, that smells inefficient to me, too!” stop and consider the nature of UTF-8—a multibyte encoding in which a character can be represented by from *one* to *three* or perhaps *four* bytes. That might sound uneconomical, but bear these facts in mind:

- The vast majority of text transmitted on the Internet can be represented by ASCII, which UTF-8 encodes as 1 byte (7-bit).
- UTF-8 encodes non-ASCII characters such as those used in Western Europe and Arabic countries as 2 bytes.
- Most Asian characters are encoded as 3 bytes.

UTF-8 encoding is therefore as efficient as it needs to be (despite urban myths to the contrary).

So *just use Unicode*, introduce some simplicity to the G11N process, and make UTF-8 your application’s sole encoding. Using Unicode simplifies things tremendously. You only have to deal with one encoding on the front end and back end. You will always *know* the data’s encoding, no matter what happens to it. And, of course, you’ll be on the same page with ColdFusion.

No need to take my word for it—the latest W3C working draft on authoring I18N XHTML and HTML documents actually recommends using UTF-8 or other Unicode encoding: “Choose UTF-8 or another Unicode encoding for all content.” (See <http://www.w3.org/TR/i18n-html-tech-char/>.)

Next, let’s take a look at putting Unicode to some actual use in *resource bundles*.

Resource Bundles

What’s a resource bundle? When Java folks begin making an application I18N, they always talk about “isolating locale-specific data” and for the most part are referring to text data. The accepted technique for this is to create `ResourceBundle` objects backed by `properties` files consisting of key/value pairs (see Listing 43.5 for an example).

The concept is rather straightforward; a key (from our example, `go`) has a value (`Go`) assigned to it. Dissecting the `properties` filename, `test_en_US.properties` (shown in the example’s comments), we can see its locale (`en_US`) as well as the resource bundle name (`test`). Java `properties` files use escaped ASCII for languages with characters beyond ISO-8859-1 encoding (see the later section “Resource Bundle Tools” for more on this); Listing 43.6 shows an example for Thai (`th_TH`) locale. The value for the key `go` is replaced by escaped ASCII encoding for the Thai word for *Go* (`\u0E44\u0E1B`).

You’ve probably caught on to the fact that both `properties` files contain the same keys with different values per locale. Instead of hard-coding text in applications, we can now use resource bundle keys that will have their values substituted on a per-locale basis when the page is processed.

Listing 43.5 test_en_US.properties—en_US Locale Resource Bundle Example

```
#Resource Bundle: test_en_US.properties - File automatically generated by RBManager
at Mon Dec 08 18:08:52 UTC+07:00 2003
#Mon Dec 08 18:08:52 UTC+07:00 2003
go=Go
cancel=Cancel
```

Listing 43.6 test_th_TH.properties—th_TH Locale Resource Bundle Example

```
#Resource Bundle: test_th_TH.properties - File automatically generated by RBManager
at Mon Dec 08 19:06:07 UTC+07:00 2003
#Mon Dec 08 19:06:07 UTC+07:00 2003
go=\u0E44\u0E1B
cancel=\u0E22\u0E01\u0E40\u0E25\u0E34\u0E01
```

NOTE

Java I18N is certainly a good role model for ColdFusion G11N work. I'm not ashamed to admit that many of the ideas in this chapter are derived from Java I18N work—the Java world has been at this G11N game a lot longer than many of us ColdFusion developers.

Now let's go through a simple example converting some ColdFusion code with hard-coded text to make use of resource bundles.

Using a Resource Bundle

Suppose we have a simple login form (Listing 43.7) that we want to use across all the locales supported by our application. For this exercise, the first thing we need to do is to pick through the code and isolate the text that needs replacing with resource bundle keys. So far, so good.

Listing 43.7 noni18nLogin.cfm—Non-I18N Login Form

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Please login</title>
  <style type="text/css" media="screen">
    TABLE {
      font-size : 85%;
      font-family : "Arial,Helvetica,sans-serif";
    }
  </style>
</head>
<body text="#330000">
  <form action="authenticate.cfm" method="post" name="loginForm" id="loginForm">
  <table cellpadding="5" cellspacing="5" border="0">
  <caption>
  <font size="+1" color="#FF0000"><b>Please login</b></font>
  </caption>
  <tr>
    <td align="right">user name:</td>
    <td><input type="text" name="userName" size="10" maxlength="20"></td>
  </tr>
  <tr>
    <td align="right">password:</td>
    <td><input type="password" name="password" size="10" maxlength="20"></td>
```

Listing 43.7 (CONTINUED)

```

</tr>
<tr valign="top" bgcolor="Silver">
  <td colspan="2" align="center">
    <input type="submit" value="login">
    &nbsp;&nbsp;&nbsp;<font face=""></font>
    <input type="reset" value="clear">
  </td>
</tr>
</table>
</form>
</body>
</html>

```

Let's also suppose our application design dictates a couple of things: The application's resource bundles will all be initialized at the same time and loaded into a ColdFusion structure in the APPLICATION scope. For this example, let's call it APPLICATION.loginRB. Also, each user's locale is detected using the geoLocator CFC discussed previously and stored in a SESSION scope variable, SESSION.locale.

TIP

It's a very good idea to logically separate your resource bundles into smaller files based on your application's modules.

Next, Listing 43.8 shows what our original non-I18N login form would look like after we replace its static text with ColdFusion-flavored resource bundle keys (and in light of the application design outlined just above). To illustrate what's happening, let's dissect one key:

```
APPLICATION.loginRB[SESSION.locale].loginFormTitle
```

The APPLICATION.loginRB indicates which resource bundle we want to use. SESSION.locale indicates which locale this user is in and acts as a key into the APPLICATION.loginRB structure. And loginFormTitle is the exact resource bundle key for which we want to substitute localized text.

Listing 43.8 i18nlogin.cfm—I18N Login Form

```

<!--
NOTE: these bits WOULD NOT normally be used in this page but rather
in an initialization routine. The example assumes you have downloaded & installed
the rbJava.cfc.
-->
<cfscript>
  rB=createObject("component","rbJava");
  geoL=createObject("component","geoLocator");
  i18nUtil=createObject("component","i18nUtil");
  loginRB=structNew();
  loginRB["en_US"]=rB.getResourceBundle("loginRB","en_US");
  loginRB["th_TH"]=rB.getResourceBundle("loginRB","th_TH");

  // figure out the user's locale
  session.locale=geoL.findLocale(CGI.remote_addr,CGI.http_accept_langauge,"en_US");
  // is this a BIDI locale?
  if (i18nUtil.isBIDI(session.locale))
    SESSION.writingDir="rtl";
  Else

```


Figure 43.3

The en_US locale login form.

Figure 43.4

The th_TH locale login form.

Now that we know what a resource bundle is, let's look at what it's not.

What *Isn't* a Resource Bundle?

Listing 43.9 is an example of what a resource bundle is not. Let me put to rest the notion of using ColdFusion code in lieu of “proper” resource bundles. There are several reasons not to do this; chief among these are:

- It mixes code and text like the bad old spaghetti code days.
- It requires some knowledge of ColdFusion to manage these files—and you do not want ColdFusion developers handling the translation of, say, information about brain surgery.
- It doesn't lend itself to using any of the nifty resource bundle–management tools (see “Resource Bundle Tools” coming up) that are commonplace in the G11N world.

So using ColdFusion code instead of resource bundles is a bad habit—it might work with small files for a few languages but will eventually break down as your G11N applications become more complex and cover more locales. If you're just beginning G11N work, don't start out with this method no matter how tempting it looks. And if you're already using this approach, think about quitting while you're ahead. Mingling code and text in this way is *not* a good idea.

Listing 43.9 notRB.cfm—Not a Resource Bundle

```
<cfset loginRB=structNew(>
<cfset loginRB.en_US.loginFormTitle="Please login">
<cfset loginRB.en_US.userNameLabel="user name">
<cfset loginRB.en_US.passwordLabel="password">
<cfset loginRB.en_US.loginButton="login">
<cfset loginRB.en_US.clearButton="clear">
```

Resource Bundle Flavors

Two kinds of resource bundles can be used with ColdFusion. The first is what might be termed CFMX UTF-8, where the resource bundle is constructed similar to a traditional INI file. A

variable’s text value is written out using UTF-8–encoded human-readable text. It’s simple to implement, relying solely on ColdFusion code to parse the files. Reading it requires nothing more complex than Notepad (which to my mind makes it unsuitable for larger, more complex applications).

TIP

There are ready-made CFCs for handling resource bundles available in the previously mentioned places.

The second type of resource bundle is the proper Java-style resource bundle as outlined earlier. These resource bundles require the use of core Java classes, which entail some overhead but have the benefits of being standard and having a wealth of ready-made (mostly open source) tools to manage them. You can further subdivide this resource bundle type into two subtypes, depending on how you’re able (or want) to access these files. Pure resource bundles are accessed using the Java `ResourceBundle` class. This class provides automatic determination of resource bundles from locale, and automatic fallback locales (if it can’t find a resource bundle for a given locale, it truncates that locale back to the language identifier and searches again; if it can’t find that resource bundle, it falls back to the base one, usually `en_US`). The class does, however, require that all resource bundles be located somewhere on a Java `classpath`, which makes for some complexity in shared-hosts environments. The other subtype uses the Java `PropertyResourceBundle` class to access resource bundles. It provides none of the automatic features of the `ResourceBundle` class but does have the advantage of locating your resource bundles anywhere, although you must explicitly load each resource bundle. Table 43.3 summarizes the pros and cons of the resource bundle types.

Now let’s have a look at some tools to manage resource bundles.

Table 43.3 Resource Bundle Flavor Comparison

RESOURCE BUNDLE FLAVORS	PRO	CON
ColdFusion UTF-8	Human readable	Complex resource bundles quickly become hard to manage
	Easy to manage (Notepad, etc.)	
	Simple to implement in ColdFusion	Can’t easily use standard resource bundle tools
	Quite fast	
Java <code>ResourceBundle</code> class	Pure standard Java resource bundle solution	Not human readable.
	Handles resource bundle from standard tools	Requires that resource bundle be somewhere in <code>classpath</code>
	Self-determines resource bundle for locale	Requires <code>createObject</code> permission
	Handles complex resource bundle quite easily	Some overhead in using Java object

Table 43.3 (CONTINUED)

RESOURCE BUNDLE FLAVORS	PRO	CON
Java PropertyResourceBundle class*	Allows resource bundle to be anywhere	Not human readable
	Pure standard Java resource bundle solution	Requires caller to determine resource bundle from locale
	Handles resource bundle from standard tools	Requires <code>createObject</code> permission
	Handles complex resource bundle quite easily	Some overhead in using Java object

* See <http://www.sustainablelegis.com/unicode/resourceBundle/javaRB.cfm> for an example.

Resource Bundle Tools

It’s a fact of life that large, complex G11N applications usually generate large, complex resource bundles. Trying to manage these with Notepad and sticky notes isn’t very realistic. You have to manage the creation/editing of the resource bundle keys, manage the creation/editing of resource bundles per locale, manage keys that have been translated into certain locales, and so on. Luckily, the Java I18N world has developed several resource bundle management tools that we can use for this task. Foremost among them (and also my favorite) is ICU4J’s pure-Java Resource Bundle Manager (RB Manager). Among the things RB Manager can do to help solve day-to-day L10N problems are the following:

- Handles editing multiple language files
- Provides sophisticated resource bundle search functionality
- Checks resource bundle keys for duplicates and for proper format
- Provides a grouping of resources; individual translations are easier to find
- Provides that each language file will only display a list of resources that are untranslated (wonderful for tracking what still needs to be translated)
- Keeps track of statistics such as number of resources, untranslated items, and so on
- Handles importing and exporting of translation data into multiple formats such as XLIFF, TMX, ICU, and more
- Cuts down on development, translation, and debugging time in any internationalized setting

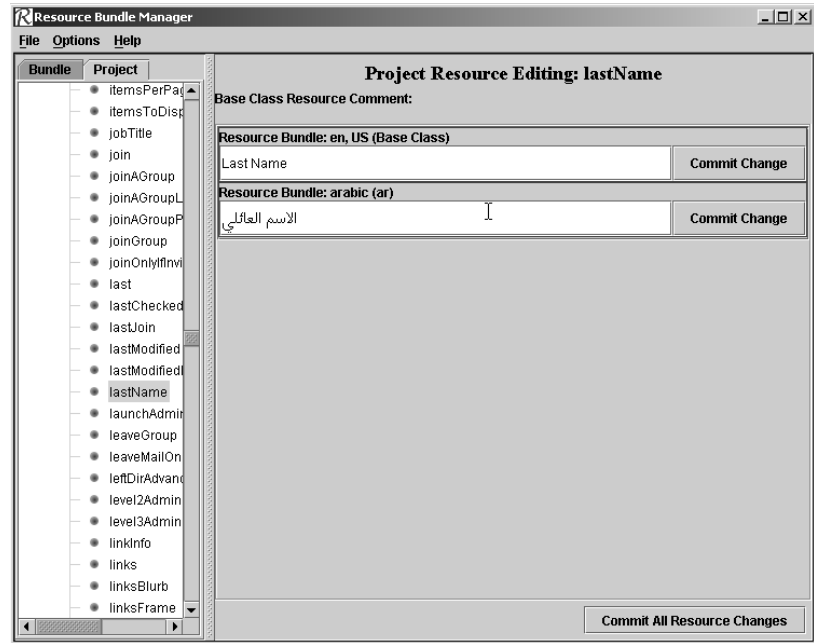
You can find a complete tutorial for RB Manager in the download file.

Figure 43.5 shows a typical resource bundle for English and Arabic languages being managed using RB Manager. In this example, the view provides a list of all resource bundle keys and their

English and Arabic translations. You can download a free copy of RB Manager from ICU4J's site, at <http://www.icu-project.org/download/rbmanager.html>.

Figure 43.5

ICU4J's pure-Java Resource Bundle Manager.



In addition to RB Manager, other resource bundle management tools are available that are more-or-less free (please review each application's licensing):

- Attesoro (<http://ostermiller.org/attesoro/>) is another pure-Java solution that can produce proper Java resource bundles.
- Zaval Java Resource Editor (<http://www.zaval.org/products/jrc-editor/>) is another Java program.
- I18nEdit (<http://www.cantamen.de/i18nedit.php?lang=en>) is another Java-based resource editor; most noteworthy is the nifty built-in Unicode character picker for those days when you're too lazy to load another locale.
- RbMan (<http://rbman.riaforge.org/>) is a ColdFusion Web-based resource bundle editor and manager solution.
- native2ascii is a command-line tool that will convert a file with native-encoded characters (the caveat being that the native encoding must be one of the Java-supported ones) to one with Unicode-encoded characters. It's found in the bin directory of your Java JRE/JDK installation.

Our next stop on the ColdFusion G11N tour deals with mailing addresses.

Addresses

Living outside the United States, one of my pet peeves is the assumption by many sites that users' addressing schemes are similar to their own. A prime example of this is the State field. Most countries do not have State as part of their addressing scheme, and adding it to your applications or, even worse, requiring it, will only confuse and possibly annoy these users. Developers need either to intimately understand a locale's addressing scheme (very possible through localization research) or to build flexibility into their address-capture routines and storage.

Developers should also not assume that postal codes (ZIP codes) confine themselves to a particular format or length. For example, Japanese postal codes can have a format such as 460-0002 (Aichi), whereas Canadian ones come in the form V2B 5S8 (Kamloops, British Columbia). Even the placement of the postal code in a mailing address can vary widely. In Laos, the postal code is to the left of the locality (01160 XAYSETHA), and in Japan it's to the left of the country (460-0002 JAPAN).

Let's look at a brief example of these ideas. Listing 43.10 shows a table design (Microsoft SQL Server data types) to hold worldwide customer information for a spatial data set product. This simple table design comes from my years of dealing with a global customer base. Its flexibility is its most important point.

Listing 43.10 galacticCustomer.txt—Galactic Customer Table Design

```
[CustomerID] [int] IDENTITY (1, 1) NOT NULL
[Salutation] [nvarchar] (100) NULL --- not fixed, as customer prefers
[FirstName] [nvarchar] (100) NOT NULL
[LastName] [nvarchar] (200) NOT NULL
[eMail] [varchar] (50) NULL --- may not have email
[PurchaseDate] [datetime] NOT NULL
[Organization] [nvarchar] (200) NULL --- company, government office, etc.
[Address] [ntext] NULL --- nTEXT will hold anything customer provides
[City] [nvarchar] (150) NULL --- may not have a city
[Locality] [nvarchar] (200) NULL --- state/province/etc. may or may not have
[Country] [varchar] (35) NOT NULL --- minimally have this, pulled from our SELECT
[PostalCode] [varchar] (40) NULL --- may or may not have
[Phone] [varchar] (50) NULL --- plenty of room
[Fax] [varchar] (50) NULL --- plenty of room
[FreeCustomer] [bit] NOT NULL --- local schools, etc. on charity list
[timestamp] [timestamp] NULL --- edit/full text indexing flag
```

NOTE

In Microsoft SQL Server's T-SQL DDL, **NOT NULL** means required data, whereas **NULL** means not required.

At first glance, there's nothing particularly remarkable about this design; however, take note of a few items. Many columns that you might normally compel a user to supply are not required, and many columns might seem overly large to someone dealing with just one locale. For example, City isn't required because in some cases there isn't an identifiable city in an address. Address, on the other hand, is an NTEXT data type capable of holding a huge amount of free-form text that might include streets, lanes, subdistricts, districts, and even directions. Notice also that SQL Server's Unicode data types (NVARCHAR and NTEXT) are used to allow the customer to supply their

own language version of name, address, and so on. For more information on address formats, see http://www.upu.int/post_code/en/addressing.shtml.

Date/Time

Addressing nuances might frustrate users, but date formatting certainly frustrates ColdFusion developers. If the ColdFusion Support Forums are any indication, even within one single locale, dates often make developers punch drunk. Even though dates are basically a simple combination of day, month, and year, there's an extensive and often confusing variety of date formats across locales. For example, 12/10/56 could be interpreted in a number of ways. In Thailand (which has a short date format of day/month/year), 12/10/56 would be taken to mean October 12, 1956. In the United States (which has a short date format of month/day/year) that date would be December 10, 1956. A similar date in Japan (where the short date format is year/month/day) would be hopelessly broken: October 56, 1912. Keeping date formats straight among locales is critical to developing G11N applications.

Our next date/time formatting issue concerns all the various calendars in use throughout the world.

Calendars

Besides date formatting, developers should not forget the types of calendars in use within a given locale. This factor can be critical; a month in one calendar might not cover the exact same time span in another. Weeks and weekends don't always start on the same day across locales using different calendars, or even within the same calendar, as in the case of Europe versus the U.S.

Out of more than 40 calendars in use around the world today, we'll examine the six most common (the "big six"), and throw in one rare calendar just for added flavoring. The "big six" discussed here are, of course, supported by the ICU4J library. The reason we're discussing these at all is to give you some background information so that you're not operating in a vacuum with these calendars behaving like some sort of mysterious black box.

NOTE

Examples of the calendars can be found at <http://www.sustainablelegis.com/projects/icu4j/calendarsTB.cfm>.

Gregorian Calendar

Pope Gregory XII introduced the Gregorian calendar in 1582 as an adaptation of the Julian calendar (named after Julius Caesar), when the 10-day difference between the actual time of year and traditional time of year on which calendar events occurred became intolerable. This calendar was constructed to give a closer approximation to the tropical year, which is the actual length of time it takes for the Earth to complete one orbit around the sun.

The actual changeover from Julian to Gregorian calendar resulted in quite an interesting "month." When England and her colonies made the change to the Gregorian in 1752 (not all countries adopted this calendar at the same time), it created a month of September something like what is shown in Figure 43.6. This move provoked widespread riots—yes, you do indeed need to pay attention to calendars.

The Gregorian calendar is in common use in Christian countries (even though some of them hadn't adopted this calendar until the early part of the twentieth century). This is the calendar most ColdFusion developers are familiar with, so I won't go into any more detail (but you can read more about this calendar here: <http://scienceworld.wolfram.com/astronomy/GregorianCalendar.html>).

Figure 43.6
Result of changeover
from Julian to
Gregorian calendar.

September 1752						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Buddhist Calendar

Behaving similarly to the Gregorian, the Buddhist calendar is identical to the Gregorian in all respects except for the year and era (B.C., A.D., etc.). Years are numbered since the birth of the Buddha in 543 B.C. (Gregorian), so that 1 A.D. (Gregorian) is equivalent to 544 B.E. (Buddhist Era) and 2005 A.D. is 2548 B.E. Quick and dirty is to simply add 543 years to the Gregorian year to arrive at the Buddhist year, and subtract 543 years to go the other way. In predominantly Buddhist countries such as Thailand (where I live these days) the Buddhist calendar is the civil calendar (the official one in general use by most folks and, of course, the government). This calendar is often used elsewhere for religious purposes.

Chinese Calendar

The traditional Chinese calendar is a *lunisolar* calendar (interestingly the same type as the Hebrew calendar). Months start with a new moon, with each month numbered according to solar events. Why? It guarantees that month 11 will always contains the winter solstice. How? Leap months are inserted in certain years. These leap months are numbered the same as the month they follow (how's that for complication?). Which month is a leap month? It depends entirely on the movements of the sun and moon.

Distinct from the Gregorian calendar, the normal Era field differs from other calendars in that it holds a 60-year cycle number rather than the usual B.C./A.D. Years are counted sequentially, numbering from the 61st year of the reign of Huang Di (more or less 2637 B.C.), which is designated year 1 on the Chinese calendar—yes, that's right, this calendar system is over 4,000 years old. Let's look at an example: 星期三 20x78-9-13

Here, 20 is the year in the current cycle, 78 is the cycle for this calendar (Era in other calendars), 9 is the month, and 13 is the day.

TIP
CFCs for handling these ICU4J-based calendars are available in the usual places (mentioned previously).

Hebrew Calendar

The Hebrew calendar is also lunisolar, which gives it what some folks would call “a number of interesting properties.” Distinct from the Gregorian calendar, months start on the day of each new moon (the ICU4J library actually makes an approximation of this). The solar year (which, as everyone knows, is 365.24 days) is not an even multiple of the lunar month (approximately 29.53 days), so an extra leap month is inserted in 7 out of every 19 years (this is beginning to sound interesting). And just to make sure everybody’s paying attention, the start of a year can be delayed by up to 3 days in order to prevent certain holidays from falling on the Sabbath (as well as to prevent illegal year lengths). The lengths of certain months can vary depending on the number of days in the year. And finally, years are counted since the creation of the world (A.M. or *anno Mundi*), believed to have taken place in 3761 B.C. Hurts my head, too—and is a compelling reason to make use of the ICU4J library and let the smart folks at IBM worry about this sort of thing.

Islamic Calendar

The Islamic calendar is also known as Hijri because it starts at the time of Mohammed’s journey, or *hijra*, to Medinah on Thursday, July 15, 622 A.D. It is the civil calendar used by most of the Arab world and is the religious calendar of the Islamic faith. This calendar is a strict lunar calendar; an Islamic year of 12 lunar months therefore does not exactly correspond to the solar year used by the Gregorian calendar system. An Islamic year averages about 354 days, so viewed from the Gregorian, each subsequent Islamic year starts about 11 days earlier.

The *civil* Islamic calendar uses a *fixed* cycle of alternating 29- and 30-day months, with a leap day added to the last month of 11 out of every 30 years. That makes the calendar predictable, so it is used as the civil calendar in a number of Arab countries.

The Islamic *religious* calendar, however, is based on the actual observation of the crescent moon. This sounds predictable and simple enough, but that observation varies based on where you are when you look (your geography), when you look (sunset varies by season), moon orbit eccentricities, and even the weather (too cloudy and you obviously can’t see the moon). All this makes it impossible to calculate in advance, so the start of a month in the religious calendar might differ from the civil calendar by up to 3 days.

Japanese Calendar

The Japanese calendar, sometimes called the Japanese Emperor Era calendar, is identical to the Gregorian calendar except for the year and era. Each Emperor’s ascension to the throne begins a new era. Each new era’s years are numbered starting with 1 (the year of ascension). What could be simpler? The modern eras began as follows:

- *Meiji*. January 8, 1868 A.D.
- *Taisho*. July 30, 1912 A.D.
- *Showa*. December 25, 1926 A.D.
- *Heisei*. January 7, 1989 A.D. (current era)

Persian Calendar

A Persian (or perhaps Iranian) calendar is the formal calendar in general use in Iran. It's also known as the solar Hijri calendar and sometimes as the Jalali calendar. I've also seen it described as the Shamsi calendar; quite frankly, I have no idea which is correct, so I'll stick with Persian.

The Persian calendar has a starting point that matches the Islamic calendar but is otherwise unrelated. The origin of this calendar can be traced back to the eleventh century when a group of astronomers (including the famous poet Omar Khayyam) created what was then called the Jalali calendar, with the modern version being adopted in 1925 A.D. Since it's one of the few calendars designed in the era of accurate positional astronomy, it's probably the most accurate solar calendar around today (we'll see why in a bit).

Like the Gregorian calendar, this calendar consists of 12 months; the first 6 are 31 days in length, the next 5 are 30 days, and the final month is 29 days in a normal year and 30 days in a leap year. To put it mildly, the Persian calendar uses a very complex leap-year structure; years are grouped into cycles that begin with 4 normal years, after which every 4th subsequent year in the cycle is a leap year. These cycles are in turn grouped into grand cycles of either 128 years (composed of cycles of 29, 33, 33, and 33 years) or 132 years (containing cycles of 29, 33, 33, and 37 years). A great grand cycle is composed of 21 consecutive 128-year grand cycles and a final 132 grand cycle, for a total of 2,820 years. The pattern of normal and leap years, which began in 1925, will not repeat until the year 4745.

Each 2,820-year great grand cycle contains 2,137 normal years of 365 days, and 683 leap years of 366 days. The average year length over the great grand cycle is 365.24219852 days, which is so close to the actual solar tropical year of 365.24219878 days that the Persian calendar accumulates an error of only 1 day in every 3.8 million years.

If this isn't enough information for you, you might have a look at this site: <http://www.tondering.dk/claus/cal/node6.html>.

Calendar CFC Use

Space doesn't permit me to post any of the code for the preceding calendar CFCs (each runs to over 1100 lines of code). What I will do instead is introduce some of the functions from these CFCs in order to help you to start thinking about using calendars in your G11N applications. (Note that many of these functions have had `i18n` added to their function name in order not to conflict with existing ColdFusion functions.)

The following are functions related to calendar math:

- `i18nDateAdd` returns a `datetime` object with units of time added. This should be used instead of ColdFusion's `dateAdd` function. Why? If you examine the output from the various calendars shown above, you will see that the same unit of time isn't equivalent across calendars. Adding 2 years to a date of 3-Feb-2005 for an Islamic calendar results in a date 709 days in the future; for the Hebrew calendar, it results in a date 739 days in the future; and for the Buddhist calendar it's 730 days.

- `18nDateDiff` returns the difference in date parts between two dates. For the same reasons outlined for `i18nDateAdd`, this method should be used instead of ColdFusion's `dateDiff` function.
- `i18nDateParse` parses a date string formatted as FULL, LONG, MEDIUM, SHORT style into a valid date object.
- `i18nIsWeekend` returns a boolean indicating whether input date falls on a weekend according to a given calendar. Weekends do not begin on the same day of the week across all calendars.
- `weekStarts` returns the first day of week for a given calendar. Weeks do not start on the same day across calendars, or even across locales within the same calendar.
- `18nDaysInMonth` returns the number of days in given month.
- `i18nDayOfWeek` returns the day of week for a given date.
- `is24HourFormat` returns 0 if not 24-hour time format, 1 if 24-hour time format in 0-23 style, or 2 if 24-hour time format in 0-24 style.
- `i18nIsLeapYear` returns true or false if a given year is a leap year.
- `getEras` returns a locale-based era (A.H., A.D., B.C., etc.).

TIP

It's not usually a good idea to use your own custom date/time formats in G11N applications. You're better off leaving that up to the standard locale-formatting functions.

These functions were designed mainly for use in page layout logic:

- `isDayFirstFormat` determines whether a given locale uses day-month or month-day format; mainly used in page layouts.
- `getTimePattern` returns locale datetime pattern string (for example, mm-dd-yy) for a given locale.
- `getDatePartOrder` is a metadata method; returns date part order (day-month-year, month-day-year, etc.) for a given calendar/locale combination.
- `getTimeDelimiter` returns a time delimiter (:/.) for a given calendar/locale combination.

The following functions are specific to individual calendars:

- `getCycle` returns the cycle for a passed date (Chinese calendar).
- `getCycleYear` returns the year in a given cycle for a passed date (Chinese calendar).
- `getExtendedYear` returns the extended year for this calendar; that is, years since start of the Chinese calendar.
- `getCycleMonth` returns the month in a cycle year for a passed date (Chinese calendar).

- `getCycleDay` returns the day in a cycle month for a passed date (Chinese calendar).
- `isLeapMonth` returns true/false if a given month is a leap month (ADAR 1) in the Hebrew calendar.
- `getEmperorEra` returns a string indicating the Japanese emperor era in which a given date falls (Japanese calendar).

Hopefully, the preceding sections have given you a firm grounding in the role of calendars in G11N. Now, let's look at one final time-related G11N issue: time zones.

Time Zones

If your application involves a global base of users, you're likely to run into issues concerning time zones. It's often the case that the application server is in one time zone while the users are in others (even non-G11N applications are affected by this). Toss daylight savings time (DST) into the mix, and things can become complicated rather quickly. Why are time zones so complicated? In theory, a time zone is an area on the earth's surface between two meridians spaced by 15 degrees of longitude (the x-axis, if you will) where the same time is adopted. Realistically, for administrative and sometimes political reasons, state or country borders often define the time zone instead of exact geographic position. For example, the following list shows the various time zone equivalents for the Asia/Bangkok (UTC+0700). These are all the same physical time zone, but simply named differently.

TIP

A CFC encapsulating the core Java time-zone functionality (`timezoneCFC`) is available in the usual places (mentioned previously).

- Antarctica/Davis
- Asia/Bangkok
- Asia/Hovd
- Asia/Jakarta
- Asia/Krasnoyarsk
- Asia/Phnom_Penh
- Asia/Pontianak
- Asia/Saigon
- Asia/Vientiane
- Etc/GMT-7
- ICT/Indo China Time
- Indian/Christmas VST

For our G11N applications, the ideal would be to allow users their own time zones and simply cast our application `datetimes` to/from the server's time zone. To further simplify things, we might also always store our application's `datetime` values in the GMT time zone. You could conceivably do this in pure ColdFusion code, but it is simpler to use either core Java's `java.util.TimeZone` class or ICU4J's `com.ibm.icu.util.TimeZone` class (handling DST changes alone in CFMX code would be quite messy, especially as there is no built-in mechanism to determine when a given time zone's DST begins and ends).

NOTE

You need to understand that for a ColdFusion server, all `datetimes` values are considered to be in that server's time zone. Consider a server in a time zone that has Daylight Savings Time (DST). A UTC `datetime` value of `2006-04-02 02:01:00.0` on those servers would never exist; it would automatically get changed to `2006-04-02 03:01:00.0` (because at one time that was when DST kicked in) because ColdFusion doesn't see your UTC time zone, only the server's DST time zone. So for an hour or so twice a year, your `datetime` values would be wrong by an hour.

An example of this method can be found at <http://www.sustainablelegis.com/projects/tz/testTZCFM.cfm>.

TIP

If your application must support multiple time zones, it's probably a good idea to maintain your `datetime` data in the UTC time zone rather than the server's or client's time zone.

Our next stop on the ColdFusion G11N tour is the topic of databases.

Databases

As far as G11N applications go, the most important factor is whether or not the database is Unicode capable. In this day and age it is rather difficult to find many popular or “big iron” databases that do *not* support Unicode. The last holdout among these was MySQL, which finally supported Unicode with the release of version 4.1 in 2005. The following is a brief review of Unicode-capable databases that you can use with ColdFusion. Consult the database's documentation for details.

Microsoft Access

Microsoft Access, within its limitations, is a suitable database for G11N applications; it supports Unicode, provided you use the Access for Unicode driver supplied with ColdFusion. You need to be aware of some quirks associated with that driver. For instance, it uses 1/0 instead of true/false for Boolean values, and it uses JET 4.0's reserved words (see <http://support.microsoft.com/?kbid=248738>).

Microsoft SQL Server

Microsoft SQL Server has been Unicode capable since version 7. It provides three data types to handle Unicode text: `NVARCHAR`, `NCHAR`, and `NTEXT`. (The N comes from the SQL-92 specification and stands for *national* data types.) Be aware that the limits for the `VARCHAR` and `CHAR` data types (8000 bytes) apply to both the standard and the Unicode variants, which effectively halves the

Unicode size limits (4000 Unicode characters). If you use Unicode data (which, of course, you should be doing at all times), also be mindful that Microsoft SQL Server requires that all Unicode text passed to it be assigned an N prefix (see <http://support.microsoft.com/kb/239530/EN-US/> for more information):

```
SELECT someColumn
FROM someTable
WHERE Greeting = N'Hello!'
```

If you use the <cfqueryparam> tag (which is a very good idea) you will need to turn on Unicode support via the ColdFusion Administrator's Advanced option for that DSN, as shown in Figure 43.7. As noted earlier in Listing 43.3, SQL Server can cast collations using the COLLATE clause, which should be your first line of attack when it comes to sorting data.

Figure 43.7

DSN Unicode support option in the ColdFusion Administrator.

Limit Connections	<input type="checkbox"/>	Restrict connections to <input type="text"/>
Maintain Connections	<input checked="" type="checkbox"/>	-- Maintain connections across client requests.
String Format	<input checked="" type="checkbox"/>	-- Enable High ASCII characters and Unicode for data sources configured for non-Latin characters

TIP

Always use your database's JDBC driver if available.

MySQL

MySQL version 4.1 and later include Unicode support as UTF-8 or UCS-2. You can assign a character set and/or collation to the server, database, table, and column. For example:

```
CREATE DATABASE dayLateDollarShort DEFAULT CHARACTER SET utf8
```

would assign the UTF-8 encoding to all CHAR and VARCHAR columns in that database. Similar to Microsoft SQL Server, you can cast collations using the COLLATE clause.

In terms of database connections, you can set the client connection character set (where ColdFusion is MySQL's client) either within MySQL itself or via the MySQL DSN's connection string option (in the Advanced option section of that DSN in the ColdFusion Administrator) using:

```
useUnicode=true&characterEncoding=utf8
```

PostgreSQL

PostgreSQL has had full Unicode support since version 7.1. Unlike MySQL, you can only set character encoding at the database level:

```
CREATE DATABASE postGISUnicode WITH ENCODING 'UNICODE'
```

Collation is also fixed at the database level—or actually at the cluster level; one instance of PostgreSQL can only have one locale.

Oracle

Oracle has supported Unicode since version 7. Oracle handles I18N issues via National Language Support (NLS), which provides database utilities, error messages, sort orders, date/time and numeric/currency formatting, and so on, adapted to relevant native languages. Oracle covers about 67 territories (locales) with 46 languages.

Oracle provides Unicode support through UTF-8 (AL31UTF8 in Oracle-talk), although the character sets differ from version 7 (AL24UTTFSS) to version 8 (AL31UTF8). AL31UTF8 handles ASCII as single-byte encoding. Similar to Microsoft SQL Server, Oracle's Unicode data types are nchar, nvarchar2, and nclob. Provided that its NLS parameters (NLS_Language, NLS_Territory) are initialized properly (server-side initialization parameters, client-side environment variables, or through the ALTER SESSION parameter), there are no serious I18N issues involving Oracle.

Display

Most ColdFusion developers tend to turn up their noses at so-called design issues like page display and layout. Display is, however, an important G11N topic, especially in locales with right-to-left (RTL) writing systems such as Arabic or Hebrew—what some folks refer to as the BIDI (bidirectional) locales. You need to understand that not just the text is RTL; the whole concept of a page in these locales is RTL. Let's look at an example.

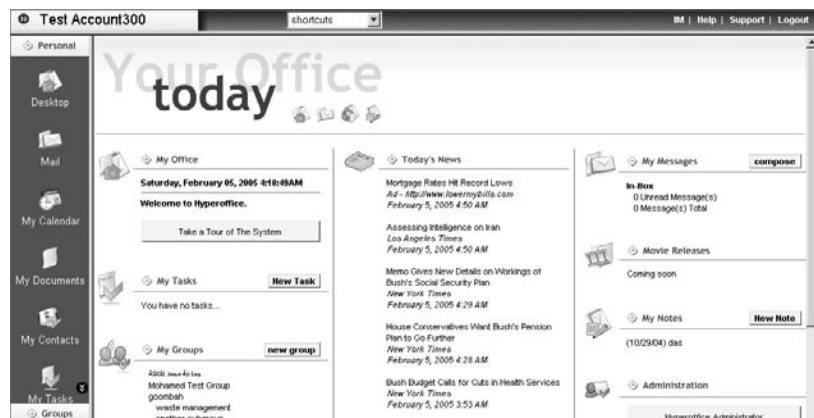
NOTE

In case you're wondering why these languages' writing systems are considered BIDI, it's because things like numbers are written left-to-right. That is, the most significant digit is leftmost, so the number 100 (one hundred) is written in Arabic or Hebrew as 100 rather than 001. Also, note that languages do not have a direction; their writing systems do.

Figure 43.8 is the desktop for a fully internationalized virtual office application (HyperOffice, <http://www.hyperoffice.com/>) for a user in the en_US, English (United States) locale. This page is laid out left-to-right (LTR), with the most important objects (menu, user name, and so on) on the left side of the page. If you look closely at the arrow icons, even these graphics are LTR (they point from the left to the right)—the devil is indeed in the details.

Figure 43.8

LTR page layout.



If we log in to this application as a user in the ar_AE, Arabic (United Arab Emirates) locale—one of the BIDI locales—you will see something like Figure 43.9. The most important objects are now on the right side of the page; the arrow icons and other graphical details are RTL as well.

As you can see, it's simply not enough to consider text handling alone; you must be concerned about every aspect of the page in locales with an RTL writing system. For more information on RTL page layout, you can visit the World Wide Web Consortium or W3C Web site (<http://www.w3.org/International/questions/qa-scripts.html>) or Tex Texin's Web site (<http://www.i18nguy.com/markup/right-to-left.html>).

Figure 43.9
RTL page layout.



So how do we go about developing a page layout to handle directionality of writing system? Leaving graphics out of it, it's actually rather easy. Recall the following line in the code of Listing 43.8:

```
<html dir="#SESSION.writingDir#" lang="#SESSION.language#">
```

That's pretty much it. It's most often recommended to set the page's writing direction in the `<html>` tag using its `dir` attribute. That's because it will also set all of the page's HTML objects' directionality as well, while leaving you with the option of changing the directionality for individual HTML objects as needed. For the page's text, this setting will have the most effect on directionally neutral text (numbers, punctuation, and so on), since most of your Unicode text will have inherent directionality (certainly another reason to *just use Unicode*).

If your page layout design tends to HTML frames, you will have to use special logic to arrange the frames in their proper sequence (see Listing 43.11 for a simple example). On the other hand, if you design with Cascading Style Sheets (CSS), no special logic is required. Starting with version 2, CSS has a `direction` property similar to the HTML `dir` attribute (see <http://www.w3.org/TR/CSS21/visuren.html#direction> for more information). CSS 3 goes a step further, adding the `block-progression` property to specify vertical flow (top-to-bottom) or horizontal flow (LTR or RTL), as well as a `writing-mode` property to act as shorthand for specifying both direction and block progression (see <http://www.w3.org/TR/css3-text/#Progression> for specifics).

Listing 43.11 frameLayout.cfm—RTL Frame Layout Logic

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<cfoutput><html dir="#SESSION.writingDir#"></cfoutput>
<head>
  <title>Bubba's Triassic Desktop</title>
</head>
<!-- frames -->
<cfif SESSION.writingDir EQ "ltr">
<!-- menu left-->
<frameset cols="20%,*">
  <frame name="menu" src="menu.cfm">
  <frame name="desktop" src="desktop.cfm">
</frameset>
<cfelse>
<!-- menu right -->
<frameset cols="80%,*">
  <frame name="desktop" src="desktop.cfm" >
  <frame name="menu" src="menu.cfm">
</frameset>
</cfif>

</html>

```

Next we look at ColdFusion's text searching as it applies to G11N applications.

Text Searching with Solr

The Lucene-based Solr engine is new to this ColdFusion release. For a detailed introduction to Solr, see Chapter 35, “Full-Text Searching.” In terms of G11N, Solr supports any language, but only the following with stemming:

- Danish
- Dutch
- Finnish
- French
- German
- Italian
- Norwegian
- Spanish
- Portuguese
- Russian
- Swedish
- Chinese
- Japanese

- Korean
- Czech
- Greek
- Thai

Now let's briefly review the new I18N bits that ColdFusion brings to the table.

What's New in ColdFusion Internationalization

The main I18N-related changes for this release of ColdFusion took place in JDK 6 (which is now the default JDK for ColdFusion).

The following CLDR-based locales are now supported:

- zh_SG Chinese (Simplified), Singapore
- en_MT English, Malta
- en_PH English, Philippines
- en_SG English, Singapore
- el_CY Greek, Cyprus
- id_ID Indonesian, Indonesia
- ga_IE Irish, Ireland
- ms_MY Malay, Malaysia
- mt_MT Maltese, Malta
- pt_BR Portuguese, Brazil
- pt_PT Portuguese, Portugal
- sr_BA Serbian, Bosnia and Herzegovina
- sr_CS Serbian, Serbia and Montenegro
- es_US Spanish, United States

Another interesting I18N change that JDK 6 provides is support for the Japanese Imperial Calendar. You can use it for date conversion and formatting by using the JP variant, `ja_JP_JP`, for your locale as shown in Listing 43.12, which should produce something similar to this: 平成19年7月26日

Listing 43.12 jec.cfm—Japanese Imperial Calendar

```
<cfprocessingDirective pageencoding="utf-8">
<cfscript>
// yes it's that simple! Don't you just love coldfusion?
setLocale("ja_JP_JP");
writeoutput("today in Japanese Imperial calendar: #lsDateFormat(now(),"FULL")#");
</cfscript>
```

The final ColdFusion 11.8N-related change we'll discuss is the addition of optional locale arguments for the following functions (these are discussed in a bit more detail in the following paragraphs); these allow you to override the current locale:

- DayOfWeekAsString
- LSCurrencyFormat
- LSDateFormat
- LSEuroCurrencyFormat
- LSIsCurrency
- LSIsDate
- LSIsNumeric
- LSNumberFormat
- LSParseCurrency
- LSParseDateTime
- LSParseEuroCurrency
- LSParseNumber
- MonthAsString

Our final stop on this tour is a brief overview of the G11N-relevant tags and functions.

Relevant ColdFusion Tags and Functions

Tables 43.4 and 43.5 list the G11N-relevant ColdFusion tags and functions. Most of these should be familiar to developers from previous versions of ColdFusion.

CharsetDecode and CharsetEncode are functions you should find useful in situations in which you're forced to convert string data to or from its binary representation. CharsetEncode is intended as a replacement for the ToString function. CharsetDecode is a shortcut function for the process of string-to-binary conversion. (In ColdFusion MX 6.1, you had to first set the string to Base64 and then use the ToBinary function to convert the string to binary data.) Both of these functions allow you to control the encoding/decoding process more finely.

Table 43.4 ColdFusion G11N Tags

FUNCTION	PARAMETER	USE
cfcontent	type	Specifies the encoding in which to return the results to the client browser
cffile	charset	Specifies how to encode data written to a file, or the encoding of a file being read

Table 43.4 (CONTINUED)

FUNCTION	PARAMETER	USE
cfheader	charset	Specifies the character encoding in which to encode the HTTP header value
cfhttp	charset	Specifies the character encoding of the HTTP request
cfhttpparam	mimeType	Specifies the MIME media type of a file; can also include the file's character encoding
cfmail	charset	Specifies the character encoding of the mail message, including the headers
cfmailpart	charset	Specifies the character encoding of one part of a multipart mail message
cfprocessingdirective	pageEncoding	Identifies the character encoding of the contents of a page to be processed by ColdFusion

NOTE

In ColdFusion, the localization functions, those that begin with **LS** like **LSDateFormat**, **LSNumberFormat**, and so on, also take an optional **locale** argument to use instead of the current locale. This argument can be helpful when you want to display data for multiple locales on the same page.

Table 43.5 ColdFusion G11N Functions

FUNCTION	PARAMETER	USE
GetLocale	–	Returns the current locale setting.
GetLocaleDisplayName	–	Returns the name of a locale in the language of a specific locale. The default value is the current locale in the locale's language.
LSCurrencyFormat	–	Converts numbers into a string in a locale-specific currency format using either the current server locale or the locale argument.
LSDateFormat	–	Converts the date part of a date/time value into a string in a locale-specific date format using either the current server locale or the locale argument.
LSEuroCurrencyFormat	–	Converts a number into a string in a locale-specific currency format using either the current server locale or the locale argument.
LSIsCurrency	–	Determines whether a string is a valid representation of a currency amount in the current locale.
LSIsDate	–	Determines whether a string is a valid representation of a date/time value in the current locale.
LSIsNumeric	–	Determines whether a string is a valid representation of a number in the current locale.

Table 43.5 (CONTINUED)

FUNCTION	PARAMETER	USE
<code>LSNumberFormat</code>	–	Converts a number into a string in a locale-specific numeric format using either the current server locale or the <code>locale</code> argument.
<code>LSParseCurrency</code>	–	Converts a string that is a currency amount in the current locale into a formatted number using either the current server locale or the <code>locale</code> argument.
<code>LSParseDateTime</code>	–	Converts a string that is a valid date/time representation into a date/time object using either the current server locale or the <code>locale</code> argument.
<code>LSParseEuroCurrency</code>	–	Converts a string that is a currency amount in the current locale or <code>locale</code> argument into a formatted number. Requires <code>Euro</code> as the currency for all countries that use the euro.
<code>LSParseNumber</code>	–	Converts a string that is a valid numeric representation in the current locale or the <code>locale</code> argument into a formatted number.
<code>LSTimeFormat</code>	–	Converts the time part of a date/time value into a string in a locale-specific date format using either the current server locale or the <code>locale</code> argument.
<code>SetLocale</code>	–	Specifies the locale setting.
<code>CharsetDecode</code>	encoding	Converts a string in the specified encoding to a binary object.
<code>CharsetEncode</code>	encoding	Converts a binary object to a string in the specified encoding.
<code>GetEncoding</code>	–	Returns the character encoding of text in the <code>FORM</code> or <code>URL</code> scope.
<code>SetEncoding</code>	charset	Specifies the character encoding of text in the <code>FORM</code> or <code>URL</code> scope. Used when the character set of the input to a form, or the character set of a URL, is not in UTF-8 encoding.
<code>ToBase64</code>	encoding	Calculates the Base64 representation of a string.
<code>ToString</code>	encoding	Returns a string encoded in the specified character encoding.
<code>URLDecode</code>	charset	Decodes a URL-encoded string.
<code>URLEncodedFormat</code>	charset	Generates a URL-encoded string.
<code>CharsetDecode</code>	encoding	Converts a string in the specified encoding to a binary object.
<code>DayOfWeekAsString</code>	locale	Returns the day of week formatted for the specified locale.
<code>MonthAsString</code>	locale	Returns the specified month as a string formatted for the passed-in locale.

Better G11N Practices

We'll wrap up this chapter by providing a set of *better* practices for developing G11N applications in ColdFusion. "Better" means better than good, but not quite ready to be the best. Why not "best practices"? My own modesty aside, mainly because there are so many ways to skin a globalized cat.

As shown throughout this chapter, the introduction of CFCs and Unicode in ColdFusion, as well as the fact that it's easier access to Java I18N libraries, certainly improved our ability to develop G11N applications. Yet G11N concepts are still relatively new to ColdFusion application developers, and I expect there is still quite a bit more to evolve as the community's heavy thinkers turn their brains to this field. What follows is what I think are better G11N practices now that ColdFusion is in town.

What Not to Do

Before we discuss what you ought to be doing, let's review some of the points made in the previous sections about what *not* to do.

- Don't use a database that doesn't support Unicode. In this day and age, it must be some kind of dinosaur anyway, so why bother?
- Don't make your application or its database monocultural. Here are some things to do to make your application more culturally sensitive:
 - Require almost nothing cultural in your forms or database (or be prepared to handle such elements on a per-locale basis).
 - Non-English terms, names, and so forth are sometimes longer than their English equivalent or consist of more than one word. Keep this in mind when designing databases, input forms, and reports.
 - Mailing address forms and database designs should be flexible in order to handle the varied address styles in use around the world (see http://www.upu.int/post_code/en/addressing.shtml for examples; note that the addressing details vary wild based on available information). Postal codes (ZIP codes in the U.S.) should not assume numeric-only data. The Street part of your address design should allow for more than one street part. Your application logic should not assume that address identifiers are always house numbers/street addresses. Also, it should not assume any patterns (left side odd, right side even) or regular sequences (house #3 might not come after house #1).
 - Don't assume global measurement units are the same ones printed on your cereal box. Your application should separate measurement units from measurement values (that is, as separate database columns or form fields). And for developers in metric countries, the whole world isn't yet metric. For applications dealing with apparel (clothing, shoes, and so on) be aware that sizes vary wildly from locale to locale.
 - If possible, always store date/time data as UTC (Universal Time Coordinate) or Greenwich Mean Time. It's a generally good idea to keep your date time data as UTC, especially if your servers or users are physically located in many time zones.

- Don't presume a Gregorian calendar system for your date/time data. There are at least six other major calendar systems in popular use today (Buddhist, Chinese, Hebrew, Islamic, Japanese, and Persian, as discussed in this chapter). While it's sometimes acceptable to use the Gregorian calendar with localized date formats, this isn't true in all locales—it's not A.D. 2010 to everyone. As noted in the "Calendars" section, this is especially critical for date-based calculations; one person's 30 days might not be another's month.
- Don't ignore CSS. Although CSS use can be complicated owing to browser quirks and the lack of a CSS police force, its use can greatly simplify G11N page layouts across locales, especially for applications that need to support BIDI text.
- Don't assume text or graphic directionality. For instance, people in Arabic/Hebrew cultures look at a page of text and graphics differently than people in Canada do.
- Don't mix text/text presentation and application code. This is, by far, the dreariest part of making an existing ColdFusion application I18N. You must search out each wayward bit of hard-coded text and replace it with ColdFusion variables of one sort or another.
- Don't fail to use UTF-8 encoding. *Just use Unicode.* It's the default encoding for ColdFusion and so offers the path of least resistance to ColdFusion globalization.

TIP

If you want to save yourself the trouble of showering your ColdFusion pages with `<cfprocessingdirective>` tags, it's wise to use an editor such as Dreamweaver, which supports a BOM (byte order mark) so that the ColdFusion server will automatically understand your page's encoding. However, if there's a chance of other users using software that might remove the BOM, it's often wiser to liberally use `<cfprocessingdirective>` tags. Note, too, that the popular Eclipse development platform doesn't support BOM in its file editing.

Let's next examine a design issue: monolingual versus multilingual Web sites.

Monolingual or Multilingual Web Sites

Some folks in the G11N field often break down Web application design to a choice between so-called monolingual or multilingual designs. In its simplest form, a *monolingual* design opts for a distinct URL for each language served by the application. For example, a Web site originally in English, say *www.iWantYourMoney.com*, might through some mechanism redirect its French users to *fr.iWantYourMoney.com*, its Thai users to *th.iWantYourMoney.com*, and so on. Such a design is obviously not well suited to static HTML text (translation and HTML file management would quickly become a nightmare). It would, however, work quite nicely as a ColdFusion-driven Web site. The downside to this approach is when an application has to serve locales rather than a single language; for example, French in Canada versus French in France. This design would either have to add increasingly more URLs to the mix it must maintain, or it would have to develop special mechanisms to handle locale within a monolingual site (something akin to a multilingual design to serve locales instead of languages within a monolingual site).

A *multilingual* design would serve all supported languages and/or locales from a single URL, www.iWantYourMoney.com. And this usually makes heavy use of resource bundles to deliver locale-specific text. Given the choice, this is my personal preference for the following reasons:

- Much more sensitive to locale. You can just as easily serve `fr` as you can `fr_FR` or `fr_CA`.
- Simplifies the application by removing the need to redirect to another URL, which also makes it more palatable to users.
- Helps simplify load-balancing schemes.
- Reduces potential issues with site structure varying across locales. (This is a pet peeve of mine. I find it particularly annoying to see a Web site's structure in one language be a thin shadow of itself in another; government Web sites are often culprits.)

As you might imagine, there are shades of these two design colors, and variations within each. There's really no right or wrong way to handle this, as long as you follow the globalization practices outlined in this chapter.

Locale Stickiness

As noted earlier in the chapter, it's very important for your ColdFusion application to remember a user's locale preference. The monolingual design is one sure way not to forget; users are basically stuck in a domain that's devoted to their language/locale. Stickiness in the multilingual design can be maintained via

- URL variables such as `index.cfm?locale=fr_CA`, which requires some mechanism to rewrite the URL variable string to append the locale on each page request
- Cookies, but these are subject to users turning them off, proxy servers trashing them, and so on
- `SESSION` variables

A frequent choice for maintained stickiness is the `SESSION` variable because it is usually the easiest to understand and maintain in an application's code. The user's locale choice is simply "there," and nothing special need be done in the application to maintain it across page requests. The downside to using these variables is the added complexity in load balancing and handling expired sessions. Of course, another option to maintain stickiness is to use `SESSION` variables with URL variables as a fallback mechanism.

HTML

ColdFusion always overrides HTML-based character encoding META tags (specifically `CONTENT-TYPE`). However, it's usually a good idea to include these, properly identified, in your ColdFusion pages. Why? The main reasons are search engines and speech synthesizers (without some kind of hint, most text-to-speech software cannot tell one language from another). The three most important HTML tags from this perspective are `<HTML>` and the `CONTENT-LANGUAGE META` and `CONTENT-TYPE META` tags.

As noted earlier in the Display section, the <HTML> tag has two important G11N attributes:

- The `lang` attribute, which specifies the base language of an element's attribute values and text content (note that you can apply the `lang` attribute to many HTML elements). Acceptable values for `lang` follow ISO639 and ISO3166, using a format such as `primary-language-code-subcode`. This is normally a two-letter country code but might also include language versions (for instance, `en-cockney`, the Cockney version of English).
- The `dir` attribute, which specifies the base direction of directionally neutral text. Like the `lang` attribute, `dir` can be applied to many HTML elements such as tables, inputs, and so forth. Acceptable values for `dir` attribute are `LTR` (left-to-right) and `RTL` (right-to-left).

The `CONTENT-LANGUAGE META` tag specifies the primary human language(s) for a page. For example, the following tag indicates that the page has English as used in the United States, Thai as used in Thailand, and French on this page:

```
<META HTTP-EQUIV="CONTENT-LANGUAGE" CONTENT="en-US,th-TH,fr">
```

The `CONTENT-TYPE META` tag specifies the content type, such as `text/html`, and the character set used on this page. This is one tag that I habitually include. The following `META` tag indicates that this page is plain text/HTML and uses a UTF-8 character set:

```
<META HTTP-EQUIV="CONTENT-TYPE" CONTENT="text/html; charset=UTF-8">
```

See Listing 43.8 for an example of these HTML tags. Even though ColdFusion will ignore them, including these three HTML tags with appropriate attribute values is a good practice.

CFML

Besides the HTML tags listed just above, your application should include the following in your `application.cfm`:

```
<!-- url and form encoding to UTF-8. -->
<cfset setEncoding("URL", "UTF-8")>
<cfset setEncoding("Form", "UTF-8")>
<!-- output encoding to UTF-8 -->
<cfcontent type="text/html; charset=UTF-8">
```

Your applications should also include:

```
<cfprocessingdirective pageEncoding="utf-8">
```

at or near the top of each and every one of your applications' templates, unless you can be 100 percent sure that each of them is properly encoded as UTF-8, including the byte order mark (BOM). Since the strict definition of UTF-8 encoding doesn't actually mention using a BOM, you might be better off including the `<cfprocessingdirective>` tag.

Resource Bundles

To completely separate text and text presentation from application code, your ColdFusion application should make use of resource bundles (discussed earlier in the chapter). The resource bundle "flavor" you use depends entirely on your application's logic and how and where it can be deployed,

although it is strongly recommended to use the Java-style resource bundle owing to the excellent set of free management tools available.

Your application should separate resource bundle files into logical groupings (for instance, `menu.properties`, `desktop.properties`, `loginForm.properties`) and then again into locale-specific files within those groupings (`menu_en_US.properties`, `menu_th_TH.properties`, and so forth). It is recommended that resource bundles be loaded into shared-scope structure variables with locale as key—for instance, `APPLICATION.menu.en_us` or, if you prefer, `APPLICATION.menu["en_US"]`.

There are generally two approaches used to initialize these resource bundles: on demand, and fire and forget. On-demand initialization, in which the application loads each locale's resource bundles as needed (that is, when a user from that locale visits the Web site), can be appropriate in situations in which the developer knows or suspects the application will have an uneven distribution of locale users, say 1 million Americans, 500,000 Italians, 10,000 Brazilians, and three guys in New Jersey who speak Zulu. (Of course, there are plenty of Zulu or IsiZulu speakers outside New Jersey, but for the purpose of this discussion, let's assume there are only three and they live in Hoboken.) Server resources are used only as and when needed—if the three Zulu speakers in New Jersey never visit the Web site, the application never loads that locale's resource bundle. Listing 43.13 shows an example of this arrangement.

Listing 43.13 onDemandRB.cfm—On-Demand Resource Bundle Loading

```
<cfscript>
// uses rbJava CFC to handle java style resource bundles
if (NOT structKeyExists(APPLICATION.commonBundle,"#SESSION.userLocale#")) {
    APPLICATION.commonBundle[SESSION.userLocale]=rB.getResourceBundle("common.
thisAppCommon",SESSION.userLocale,markDebug);
    APPLICATION.adminBundle[SESSION.userLocale]=rB.getResourceBundle("admin.
thisAppAdmin",SESSION.userLocale,markDebug);
    APPLICATION.appBundle[SESSION.userLocale]=rB.getResourceBundle("applications.
thisAppApplications",SESSION.userLocale,markDebug);
    APPLICATION.globalBundle[SESSION.userLocale]=rB.getResourceBundle("global.
thisAppGlobal",SESSION.userLocale,markDebug);
    APPLICATION.groupBundle[SESSION.userLocale]=rB.getResourceBundle("groups.
thisAppGroups",SESSION.userLocale,markDebug);
    APPLICATION.toolsBundle[SESSION.userLocale]=rB.getResourceBundle("tools.
thisAppTools",SESSION.userLocale,markDebug);
}
</cfscript>
```

This code assumes the use of the `rbJava.CFC` discussed previously, and also assumes that the relevant ColdFusion structures have been created to hold the resource bundles. The `rbJava.CFC` returns a structure holding the resource bundle's key/value pairs. The code first checks to see if one of the resource bundle structures has a key for this user's locale. If not, it then proceeds to load the various resource bundles into the appropriate structures using this user's locale as a key.

The dot-syntax naming scheme (`common.thisAppCommon`) for the resource bundle properties file is required for the `getBundle` method, which uses a `directory.fileName` notation to find the correct resource bundle property file.

The `markDebug` option for that CFC's `getResourceBundle` function indicates whether or not to mark the returned resource bundles with asterisks (*) to aid in debugging. (It helps pick out text supplied from the resource bundle versus static application text left over from the I18N process; that is, bugs.)

The fire-and-forget approach simply loads all supported-locale resource bundles when the application is initialized, rather than waiting for a user to initialize a new resource bundle for a particular locale. This method of initialization might best be applied when the developer knows that locale usage is evenly distributed or doesn't want to bother with any dynamic loading of resource bundles.

Listing 43.14 provides a simple example of this technique. The array of application-supported locales (`APPLICATION.supportLocales`) might be supplied as an application parameter or dynamically determined by parsing the locales available for any given resource bundle.

Listing 43.14 `ffRB.cfm`—Fire-and-Forget Resource Bundle Loading

```
<cfscript>
// uses rbJava CFC to handle java style resource bundles
for (i=1; i LTE arrayLen(APPLICATION.supportLocales); i=i+1) {
    //verbose for readability
    thisLocale=APPLICATION.supportLocales[i];
    APPLICATION.commonBundle[thisLocale]=rB.getResourceBundle("common.thisAppCommon",
thisLocale,markDebug);
    APPLICATION.adminBundle[thisLocale]=rB.getResourceBundle("admin.thisAppAdmin",
thisLocale,markDebug);
    APPLICATION.appBundle[thisLocale]=rB.getResourceBundle("applications.
thisAppApplications",thisLocale,markDebug);
    APPLICATION.globalBundle[thisLocale]=rB.getResourceBundle("global.thisAppGlobal",
thisLocale,markDebug);
    APPLICATION.groupBundle[thisLocale]=rB.getResourceBundle("groups.thisAppGroups",
thisLocale,markDebug);
    APPLICATION.toolsBundle[thisLocale]=rB.getResourceBundle("tools.thisAppTools",
thisLocale,markDebug);
}
</cfscript>
```

Just Use Unicode

If your application needs to support more than a few languages—especially any of the Chinese, Japanese, and Korean (CJK) languages—it needs to use Unicode as its character encoding. This isn't rocket science and, as emphasized earlier, there are no real adverse side effects to using Unicode. The only serious issue arises over legacy data using codepage encodings, and in the long run you'll be better off biting the bullet and doing the conversion early on in the application's life cycle. *Just use Unicode.*

When ColdFusion Isn't Enough

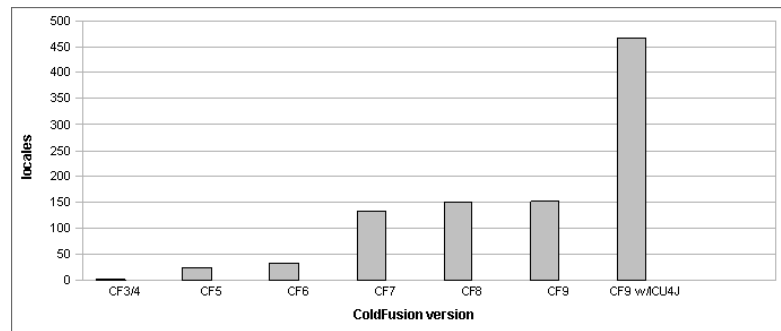
Out of the preceding sections, you'll want to recall one of the guiding principles for developing G11N applications: These applications should be *generic*. A G11N application must be distilled

down to its essence (internationalized) before it becomes specialized (localization). That old saw “Build once, run anywhere” really does apply to G11N applications.

Making generic applications in the face of all this complexity is difficult. It becomes even more difficult if you are faced with the situation where you need to support locales that ColdFusion doesn't. Figure 43.10 shows a graph of locales supported by various versions of ColdFusion, with the final value in the graph showing locales supported by ColdFusion backed by the ICU4J library. If you're at this G11N business long enough, you will eventually run into that 90+ locale difference. It becomes a question then of handling the unsupported locales as special cases, and maintaining the simplicity and speed offered by the native ColdFusion G11N tags and functions for the rest of your locales. Or you can switch to the complete use of ICU4J's G11N functions and deal with the added code complexity, but maintain a more generic application.

Figure 43.10

ColdFusion locale support, by version.



There is no right or wrong answer for this question of generic applicability. It depends a great deal on the individual developer and/or the policies of the development shop, and how long both have been in the G11N business. My own shop has more or less slid into the ICU4J-only style. We've developed enough applications in locales that ColdFusion previously didn't support (such as Thai and Arabic) that we have a rather large set of essential ICU4J-based tools that we can't live without. Developers new to G11N applications have a tough decision to make and, quite frankly, I can't offer any critical advice—other than to measure out your locale support and see if that warrants one style or the other.

You're probably thinking to yourself, “Well that was a mouthful.” Perhaps it was, but G11N concepts are not trivial, even though the code for it very often is. This chapter covered the major G11N issues including locales, character encoding, databases and text/code separation via resource bundles. It also provided information to help you over the rough spots and code as well as download resources to help you easily deal with the more common G11N needs. Finally, it set out some better G11N practices that you should follow.

“So long and thanks for all the fish.”

CHAPTER 44

Error Handling

IN THIS CHAPTER

Catching Errors as They Occur	E47
What Is an Exception?	E47
Introducing <code><cftry></code> and <code><cfcatch></code>	E48
Basic Exception Handling	E49
Understanding What Caused the Error	E53
Writing Templates That Work Around Errors	E56
Writing Templates That Recover from Errors	E59
Deciding Not to Handle an Exception	E64
Throwing and Catching Your Own Errors	E68
Handling Exceptions with Scripting	E70

Catching Errors as They Occur

Unless you are a really, really amazing developer—and an incredibly fast learner—you have seen quite a few error messages from ColdFusion, both during development and after deployment of your applications. ColdFusion is generally very good about providing diagnostic messages that help you understand what the problem is. Error messages are a developer's friends. These clever little helpers selflessly provide hints and observations about the state of your code so you can fix it as soon as possible. It's an almost romantic relationship.

Unfortunately, your users won't see error messages through the rose-colored glasses coders tend to wear. They will call or page you or flood your in-box with unfriendly email until you get the error message (which they perceive as some kind of ugly monster) under control.

Wouldn't it be great if you could have your code watch for certain types of errors and respond to them on the fly, before the user even sees them? After all, as a developer, you often know the types of problems that might occur while a particular chunk of code does its work. If you could teach your code to recover from predictable problems on its own, you could lead a less stressful and healthier (albeit somewhat lonelier) coding lifestyle.

This and more is what this chapter is all about. ColdFusion provides a small but powerful set of structured exception-handling tags, which enable you to respond to problems as they occur.

What Is an Exception?

When ColdFusion displays an error message, it's responding to an exception. Whenever a CFML tag or function is incapable of doing whatever your code has asked it to do—such as connect to a database or process a variable—it lets ColdFusion know what exactly went wrong and says why. This process of reporting a problem is called *raising an exception*. After the tag or function raises an exception, ColdFusion's job is to respond to it. Out of the box, ColdFusion responds to nearly all

exceptions in the same way, by displaying an error message that describes the exception. ColdFusion's logs also note the fact that the exception occurred. If you want, you can use its exception-handling tags to respond to an exception in some different, customized way. When you do this, you are telling ColdFusion to run special code of your own devising, instead of displaying an error message as it normally would. This process of responding to an exception with your own code is called *catching an exception*.

Introducing <cftry> and <cfcatch>

ColdFusion provides two basic CFML tags for handling exception conditions:

- The <cftry> tag—A paired tag you place around the portions of your templates that you think might fail under certain conditions. The <cftry> tag doesn't take any attributes. You simply place opening and closing <cftry> tags around the block of code you want ColdFusion to attempt to execute.
- The <cfcatch> tag—Used to catch exceptions that occur within a <cftry> block. <cfcatch> takes only one attribute, `type`, as shown in Table 44.1. `type` tells ColdFusion what type of problem you are interested in responding to. If that type of problem occurs, ColdFusion executes the code between the <cfcatch> tags. Otherwise, it ignores the code between the <cfcatch> tags.

Table 44.1 <cfcatch> Tag Attributes

ATTRIBUTE	DESCRIPTION
type	The type of exceptions to catch or respond to. It can be any of the types shown in Table 44.2. For instance, if you are interested in trying to recover from errors that occur while executing <cfquery> or another database-related operation, you would use a <cfcatch> of <code>type="Database"</code> .

Table 44.2 Predefined Exception Types

EXCEPTION TYPE	DESCRIPTION
Any	Catches any exception, even those you might not have any way of dealing with (such as an “out of memory” message). Use this exception value if you need to catch errors that don't fall into one of the other exception types in this table. If possible, use one of the more specific exception types listed in this table.
Application	Catches application-level exception conditions. Your own CFML code reports these exceptions, using the <cfthrow> tag. In other words, catch errors of <code>type="Application"</code> if you want to catch your own custom errors.
Database	Catches database errors, which could include errors such as inability to connect to a database, an incorrect column or table name, a locked database record, and so on.

Table 44.2 (CONTINUED)

EXCEPTION TYPE	DESCRIPTION
Expression	Catches errors that occur while attempting to evaluate a CFML expression. For instance, if you refer to an unknown variable name or provide a function parameter that doesn't make sense when your template actually executes, an exception of type <code>Expression</code> is thrown.
Lock	Catches errors that occur while attempting to process a <code><cflock></code> tag. Most frequently, this type of error is thrown when a lock can't be obtained within the <code>timeout</code> period specified by the <code><cflock></code> tag. This usually means that some other page request that uses a lock with the same name or scope is taking a long time to complete its work.
MissingInclude	Catches errors that arise when you use a <code><cfinclude></code> tag but the CFML template you specify for the <code>template</code> attribute can't be found.
Object	Catches errors that occur while attempting to process a <code><cfobject></code> tag or <code>createObject()</code> function, or while attempting to access a property or method of an object returned by <code><cfobject></code> or <code>createObject()</code> .
Security	Catches errors that occur while using one of ColdFusion's built-in security-related tags.
Template	Catches general application page errors that occur while processing a <code><cfinclude></code> , <code><cfmodule></code> , or <code><cferror></code> tag.
SearchEngine	Catches errors that occur while performing full-text searches or other Solr-related tasks, such as indexing. For details about Solr, see Chapter 35, "Full-Text Searching."

TIP

You can also provide your own exception types for the `type` attribute of the `<cfcatch>` tag. For details, see the "Throwing and Catching Your Own Errors" section, later in this chapter.

Basic Exception Handling

The easiest way to understand exception handling is to actually go through the process of adding `<cftry>` and `<cfcatch>` to an existing template. Listings 44.1 and 44.2 show the effects of ColdFusion's exception-handling tags, using a before-and-after scenario.

A Typical Scenario

Say you have been asked to create a page that lets users select from two separate drop-down lists. The first drop-down provides a list of films; the second shows a list of film ratings. Each drop-down list has a Go button next to it, which presumably takes the user to some type of detail page when clicked.

Here's the catch: For whatever reason, you know ahead of time that the database tables populating these drop-down lists (the `FILMS` and `FILMSRATINGS` tables) won't always be available. There might be any number of reasons for this. Perhaps you are connecting to a database server that is known

to crash often, that goes down during certain times of the day for maintenance, or that is accessed via an unreliable network connection.

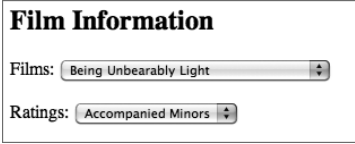
Your job is to make your new drop-down list page operate as gracefully as possible, even when the database connections fail. At the very least, users shouldn't see an ugly database error message. If you can pull off something more elegant, such as querying some kind of backup database in the event that the normal database tables are not available, that's even better.

A Basic Template, Without Exception Handling

Listing 44.1 is the basic template, before the addition of any error handling. This template works just fine as long as no problems occur while the user is connecting to the database. It runs two queries and displays two drop-down lists, as shown in Figure 44.1. If an error occurs, though, the user sees an ugly error message and can't get any further information.

Figure 44.1

This is what the drop-down page looks like, as long as no errors occur.



Film Information

Films:

Ratings:

NOTE

The examples in this chapter assume the creation of an **Application.cfc** file that sets a default data source and also turns on session management (as discussed in Chapter 18). For your convenience, the appropriate **Application.cfc** file for this chapter may be downloaded online.

Listing 44.1 ChoicePage1.cfm—A Basic Display Template, Without Any Error Handling

```
<!---
  Filename: ChoicePage1.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides navigation elements for films and ratings
-->

<h2>Film Information</h2>

<!--- Retrieve Ratings from database --->
<cfquery name="getRatings">
  SELECT RatingID, Rating
  FROM FilmsRatings
  ORDER BY Rating
</cfquery>
<!--- Retrieve Films from database --->
<cfquery name="getFilms">
  SELECT FilmID, MovieTitle
  FROM Films
  ORDER BY Films.MovieTitle
</cfquery>

<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">
```

Listing 44.1 (CONTINUED)

```

<!-- Display Film names in a drop-down list -->
<P>Films:
<cfselect query="getFilms" name="filmID" value="filmID"
    display="MovieTitle"/>

<!-- Display Rating names in a drop-down list -->
<P>Ratings:
<cfselect query="getRatings" name="ratingID" value="RatingID"
    display="Rating"/>

```

</cfform>

This template doesn't contain anything new yet. Two `<cfquery>` tags named `getRatings` and `getFilms` run, and the results from each query appear in a drop-down list that enables the user to select a film or rating. If any of this looks unfamiliar, review Chapter 14, "Using Forms to Add or Change Data," in *Adobe ColdFusion 9 Web Application Construction Kit, Volume 1: Getting Started*.

If for whatever reason `getRatings` and `getFilms` can't execute normally, an error message appears, leaving the user with nothing useful (other than a general impression that you don't maintain your site very carefully). For instance, if you go into the ColdFusion Administrator and sabotage the connection by providing an invalid filename in the Database File field for the `ows` data source, you'll see the error shown in Figure 44.2.

Figure 44.2

If an error occurs while connecting to the database, the default error message is displayed.

Film Information

The web site you are accessing has experienced an unexpected error. Please contact the website administrator.

The following information is meant for the website developer for debugging purposes.

Error Occurred While Processing Request**Error Executing Database Query.**

Database 'xUsers/ray/Documents/ColdFusion/CFWACK 8/ows' not found.

The error occurred in
 /Library/WebServer/Documents/ows/44/ChoicePage1.cfm: line 10
 8 :
 9 : <!-- Retrieve Ratings from database -->
 10 : <cfquery name="getRatings">
 11 : SELECT RatingID, Rating
 12 : FROM FilmsRatings

VENDORERRORCODE 40000
 SQLSTATE XJ004
 SQL SELECT RatingID, Rating FROM FilmsRatings ORDER BY Rating
 DATASOURCE ows
 Resources:

- Check the [ColdFusion documentation](#) to verify that you are using the correct syntax.
- Search the [Knowledge Base](#) to find a solution to your problem.

Browser Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-US)
 Remote AppleWebKit/532.9 (KHTML, like Gecko) Chrome/5.0.307.1 Safari/532.9

Address ::1

Referrer http://localhost/ows/44/

Date/Time 01-Feb-10 06:02 PM

Stack Trace
 at cfChoicePage12ecfm572873333.runPage(/Library/WebServer/Documents/ows/44/ChoicePage1.cfm:10)

Adding <cftry> and <cfcatch>

To add ColdFusion’s structured exception handling to Listing 44.1, simply wrap a pair of opening and closing <cftry> tags around the two <cfquery> tags. Then add a <cfcatch> block that specifies an exception type of Database, just before the closing </cftry> tag. The code inside the <cfcatch> tag will execute whenever either of the <cfquery> tags raises an exception.

Listing 44.2 is a revised version of Listing 44.1. The code is almost exactly the same, except for the addition of the <cftry> and <cfcatch> blocks, which display a basic “Sorry, we are not able to connect” error message. Now, if any problems occur when connecting to the database, the error message appears as shown in Figure 44.3. This really isn’t much better than what the user saw in Figure 44.2, but it’s a start.

Figure 44.3

You can use a <CFCATCH> block in its simplest form to display context-specific error messages.

Film Information

Sorry, we are not able to connect to our real-time database at the moment, due to carefully scheduled database maintenance.

Listing 44.2 ChoicePage2a.cfm—Adding a Simple <cftry> Block to Catch Database Errors

```

<!--
  Filename: ChoicePage2a.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides navigation elements for films and ratings
-->

<h2>Film Information</h2>

<cftry>
  <!-- Retrieve Ratings from database -->
  <cfquery name="getRatings">
    SELECT RatingID, Rating
    FROM FilmsRatings
    ORDER BY Rating
  </cfquery>

  <!-- Retrieve Films from database -->
  <cfquery name="getFilms">
    SELECT FilmID, MovieTitle
    FROM Films
    ORDER BY Films.MovieTitle
  </cfquery>

  <!-- If any database errors occur during above query, -->
  <cfcatch type="Database">
    <!-- Let user know that Films data can't be shown right now -->
    <i>Sorry, we are not able to connect to our real-time database at
    the moment, due to carefully scheduled database maintenance.</i><br>

    <!-- Stop processing at this point -->
    <cfabort>
  </cfcatch>
</cftry>

```

Listing 44.2 (CONTINUED)

```
</cfcatch>
</cftry>

<!-- Create self-submitting form -->
<cfform action="#CGI.script_name#" method="post">
  <!-- Display Film names in a drop-down list -->
  <P>Films:
  <cfselect query="getFilms" name="filmID" value="FilmID"
    display="MovieTitle"/>

  <!-- Display Rating names in a drop-down list -->
  <P>Ratings:
  <cfselect query="getRatings" name="ratingID" value="RatingID"
    display="Rating"/>

</cfform>
```

Even though it's rather simplistic, take a close look at Listing 44.2. First, the `<cftry>` block tells ColdFusion that you are interested in trapping exceptions. Within the `<cftry>` block, all of the queries needed by the page execute. If any type of problem occurs—anything from a crashed database server to a mere typo in your SQL code—the code inside the `<cfcatch>` block executes, displaying the static message shown in Figure 44.3. Otherwise, the `<cfcatch>` block is skipped entirely.

NOTE

It's important to note that the `<cfcatch>` block in Listing 44.2 includes a `<cfabort>` tag to halt all further processing. If not for the `<cfabort>` tag, ColdFusion would continue processing the template, including the code that follows the `<cftry>` block. This would just result in a different error message, because the `<cfselect>` tags would be referring to queries that never ended up running.

All this code does so far is display a custom error message without taking any specific action to help the user, so Listing 44.2 really isn't any better than the templates in Chapter 18, "Introducing the Web Application Framework," in Volume 1, which used the `<cferror>` tag or `onError()` method to customize the display of error messages. The advantages of using `<cftry>` and `<cfcatch>` will become apparent shortly. In practice, you will often use `<cftry>` and `<cfcatch>` along with `<cferror>` or `onError()`.

Understanding What Caused the Error

When it catches an exception, ColdFusion populates a number of special variables that contain information about the problem that actually occurred. These variables are available to you via the special `CFCATCH` scope. Your code can examine these `CFCATCH` variables to get a better understanding of what exactly went wrong, or you can just display the `CFCATCH` values to the user in a customized error message.

Table 44.3 lists the variables available to you within a `<cfcatch>` block.

Table 44.3 CFCATCH Variables Available After an Exception Is Caught

VARIABLE	DESCRIPTION
CFCATCH.Type	The type of exception that was caught. If the exception isn't a custom one, it will be one of the exception types listed in Table 44.2.
CFCATCH.Message	The text error message that goes along with the exception that was caught. Nearly all ColdFusion errors include a reasonably helpful Message value; this is the message that shows up at the top of normal error messages. For instance, the value of CFCATCH.Message for the exception shown in Figure 44.2 is Error Executing Database Query .
CFCATCH.Detail	Detail information that goes along with the caught exception. Most ColdFusion errors include a helpful Detail value. For the error shown in Figure 44.2, it's the value of CFCATCH .
CFCATCH.SqlState	Available only if type is Database . A standardized error code that should be reasonably consistent for the same type of error, even between different database systems.
CFCATCH.Sql	Available only if type is Database . The SQL statement that was used in the query.
CFCATCH.queryError	Available only if type is Database . The error message returned from the database.
CFCATCH.where	Available only if type is Database . If the query had used <cfqueryparam> tags, the where value will contain the name-value pairs used in the tags.
CFCATCH.NativeErrorCode	Available only if type is Database . The native error code reported by the database system when the problem occurred. These error codes are not usually consistent between database systems.
CFCATCH.ErrNumber	Available only if type is Expression . This code identifies the type of error that threw the exception. We recommend that you don't use this value to check for specific errors, because the values are not documented and have been known to change from version to version of ColdFusion. It's generally better to examine the text of the CFCATCH.Detail or CFCATCH.Message values.
CFCATCH.MissingFileName	Available only if type is MissingInclude . The name of the ColdFusion template that could not be found.
CFCATCH.LockName	Available only if type is Lock . The name of the lock, if any, that was provided to the <cflock> tag that failed.
CFCATCH.LockOperation	Available only if type is Lock . At this time, this value will always be Timeout , Create Mutex , or Unknown .
CFCATCH.ErrorCode	Available only when you throw your own exceptions with <cfthrow> . The value, if any, that was supplied to the errorCode attribute of the <CFTHROW> tag that threw the exception.
CFCATCH.ExtendedInfo	Available only when you throw your own exceptions with <cfthrow> . The value, if any, that was supplied to the extendedInfo attribute of the <cfthrow> tag.
CFCATCH.TagContext	An array of structures that contains information about the ColdFusion templates involved in the page request when the exception occurred. This value is used primarily for creating your own debugging templates, not for exception handling as discussed in this chapter. For details, see the ColdFusion documentation.

Listing 44.3 is a slightly revised version of Listing 44.2. This time, the message shown to the user includes information about the exception, by outputting the value of `CFCATCH.SqlState`.

Listing 44.3 ChoicePage2b.cfm—Displaying the SQL Error Code for a Database Error

```
<!--  
  Filename: ChoicePage2b.cfm  
  Created by: Nate Weiss (NMW)  
  Purpose: Provides navigation elements for films and ratings  
-->  
  
<h2>Film Information</h2>  
  
<cftry>  
  <!-- Retrieve Ratings from database -->  
  <!-- RatingIck is not a real column. -->  
  <cfquery name="getRatings">  
    SELECT RatingID, Rating  
    FROM FilmsRatings  
    ORDER BY RatingIck  
  </cfquery>  
  <!-- Retrieve Films from database -->  
  <cfquery name="getFilms">  
    SELECT FilmID, MovieTitle  
    FROM Films  
    ORDER BY Films.MovieTitle  
  </cfquery>  
  
  <!-- If any database errors occur during above query, -->  
  <cfcatch type="Database">  
    <!-- Let user know that the Films data can't be shown right now -->  
    <cfoutput>  
      <i>Sorry, we are not able to connect to our real-time database right now,  
      because of SQL Error Code <b>#CFCATCH.ErrorCode#</b>.</i>  
    </cfoutput>  
  
    <!-- Stop processing at this point -->  
    <cfabort>  
  </cfcatch>  
</cftry>  
  
<!-- Create self-submitting form -->  
<cfform action="#CGI.script_name#" method="post">  
  
  <!-- Display Film names in a drop-down list -->  
  <P>Films:  
  <cfselect query="getFilms" name="filmID" value="FilmID"  
    display="MovieTitle"/>  
  
  <!-- Display Rating names in a drop-down list -->  
  <P>Ratings:  
  <cfselect query="getRatings" name="ratingID" value="RatingID"  
    display="Rating"/>  
  
</cfform>
```

NOTE

This template only outputs the value of `CFCATCH.SqlState`. Of course, if you want to give the user more information, you are free to use the other `CFCATCH` variables from Table 44.3 that are relevant to the type of exception.

Writing Templates That Work Around Errors

As shown in the previous listing, you can use `<cftry>` and `<cfcatch>` to respond to errors by displaying a customized error message. But that's really not so different from the way you learned to use `<cferror>` and `onError()` in Chapter 18. The real value of `<cftry>` and `<cfcatch>` is the fact that they let you create templates that actively respond to or work around exception conditions. In other words, they let you write Web pages that continue to be at least somewhat helpful to your users, even after an error has occurred.

Working Around a Failed Query

Take a look at Listing 44.4, which is similar to the other listings you have seen so far in this chapter. This version isolates each `<cfquery>` within its own `<cftry>` block. Within each `<cftry>` block, if any database errors occur, the user sees a static message, but the template execution doesn't stop.

Listing 44.4 ChoicePage3a.cfm—Using `<cftry>` and `<cfcatch>` to Display Information

```
<!---
  Filename: ChoicePage3a.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides navigation elements for films and ratings
-->

<h2>Film Information</h2>
<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

  <!--- Attempt database operation --->
  <cftry>
    <!--- Retrieve Films from database --->
    <!--- MovieTitleBad is not a valid column. To fix, change to MovieTitle --->
    <cfquery name="getFilms">
      SELECT FilmID, MovieTitle
      FROM Films
      ORDER BY Films.MovieTitleBad
    </cfquery>

    <!--- If any database errors occur during above query, --->
    <cfcatch type="Database">
      <!--- Let user know that the Films data can't be shown right now --->
      <p><i>Sorry, we can't show a real-time list of Films right now.</i></p>
    </cfcatch>
  </cftry>

  <!--- Attempt database operation --->
  <cftry>
    <cfquery name="getRatings">
      SELECT RatingID, Rating
```


Listing 44.4 (CONTINUED)

```
FROM FilmsRatings
ORDER BY Rating
</cfquery>

<!-- If any database errors occur during above query, -->
<cfcatch type="Database">
<!-- Let user know that the Ratings data can't be shown right now -->
<p><i>Sorry, we can't show a real-time list of Ratings right now.</i></p>
</cfcatch>
</cftry>

<!-- If, after all is said and done, we were able to get Film data -->
<cfif isDefined("VARIABLES.getFilms")>
<!-- Display Film names in a drop-down list -->
<p>Films:
<cfselect query="getFilms" name="filmID" value="FilmID"
    display="MovieTitle"/>
<cfinput type="submit" name="go" value="Go">
</cfif>

<!-- If, after all is said and done, we were able to get Ratings data -->
<cfif isDefined("VARIABLES.GetRatings")>
<!-- Display Ratings in a drop-down list -->
<p>Ratings:
<cfselect query="getRatings" name="ratingID" value="RatingID"
    display="Rating"/>
<cfinput type="submit" name="go" value="Go">
</cfif>

</cfform>
```

If for some reason the `getFilms` query fails to run properly, a “Not able to get a real-time list of Films” message is displayed. Template execution then continues normally, right after the first `<cftry>` block. Because the `<cfquery>` couldn’t complete its work, the `getFilms` variable doesn’t exist; otherwise, everything is fine.

At the bottom of the template, an `isDefined()` check, which ascertains whether the corresponding query actually exists, protects the display of each drop-down list. If the `getFilms` query is completed without error, the first `<cfif>` test passes and the Films drop-down list is displayed. If not, the code inside the `<cfif>` block is skipped.

Since the first query causes an error, the user sees a friendly little status message, as shown in Figure 44.4, instead of the Films drop-down list. The second query can execute successfully, however, and the Ratings drop-down list still displays normally. You can easily modify the code to correct the bad SQL in the first query to see how the template changes. Also feel free to “break” the second query.

The result is a page that remains useful when a problem with the database partially disables it. In a small but very helpful way, it’s self-healing.

Listing 44.5 shows a slightly different way to structure the code. Almost all the code lines are the same as Listing 44.4, just ordered differently. Instead of placing both `<cfselect>` tags together at

the end of the template, Listing 44.5 outputs each `<cfselect>` in the corresponding `<cftry>` block, right after the query that populates it. If one of the queries fails, the `<cfswitch>` block immediately takes over, thus skipping over `<cfselect>` for the failed query.

The result is a template that behaves in exactly the same way as Listing 44.4. Depending on your preference and on the situation, placing code in self-contained `<cftry>` blocks as shown in Listing 44.5 can make your templates easier to understand and maintain. On the other hand, in many situations, the approach taken in Listing 44.4 is more sensible because it keeps the database access code separate from the HTML generation code. Use whichever approach results in the simplest-looking, most straightforward code.

Figure 44.4

ColdFusion's exception-handling capabilities enable your templates to recover gracefully from error conditions.



Listing 44.5 ChoicePage3b.cfm—An Alternative Way of Structuring the Code in Listing 44.4

```
<!---
Filename: ChoicePage3b.cfm
Created by: Nate Weiss (NMW)
Purpose: Provides navigation elements for films and ratings
--->

<h2>Film Information</h2>

<!--- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

    <!--- Attempt database operation --->
    <cftry>
    <!--- Retrieve Films from database --->
    <!--- MovieTitle2 is a bad column. Remove the "2" to fix it. --->
    <cfquery name="getFilms">
    SELECT FilmID, MovieTitle2
    FROM Films
    ORDER BY Films.MovieTitle
    </cfquery>

    <!--- Display Film names in a drop-down list --->
    <p>Films:
    <cfselect query="getFilms" name="filmID" value="FilmID"
        display="MovieTitle"/>
    <cfinput type="submit" name="go" value="Go">

    <!--- If any database errors occur during above query, --->
    <cfcatch type="Database">
    <!--- Let user know that the Films data can't be shown right now --->
    <p><i>Sorry, we can't show a real-time list of Films right now.</i></p>
    </cfcatch>
```

Listing 44.5 (CONTINUED)

```

</cftry>

<!-- Attempt database operation --->
<cftry>
<!-- Retrieve Ratings from database --->
<cfquery name="getRatings">
SELECT RatingID, Rating
FROM FilmsRatings
ORDER BY Rating
</cfquery>

<!-- Display Rating names in a drop-down list --->
<p>Ratings:
<cfselect query="getRatings" name="ratingID" value="RatingID"
    display="Rating"/>
<cfinput type="submit" name="go" value="Go">

<!-- If any database errors occur during above query, --->
<cfcatch type="Database">
<!-- Let user know that the Ratings data can't be shown right now --->
<p><i>Sorry, we can't show a real-time list of Ratings right now.</i></p>
</cfcatch>
</cftry>

</cfform>
    
```

Writing Templates That Recover from Errors

Listings 44.4 and 44.5 created templates that serve your users by skipping blocks of code that rely on failed queries. Depending on the situation, you can often go a step further, creating templates that continue to provide the basic functionality they are supposed to (albeit in some type of scaled-back manner), even when a problem occurs.

For instance, Listing 44.6 creates yet another version of the drop-down page example. It looks similar to Listing 44.4, except for the first `<cfcatch>` block, which has been expanded. The idea here is to use a backup copy of the `Films` database table, which exists for the sole purpose of providing a fallback when the primary database system can't be reached.

Listing 44.6 ChoicePage4.cfm—Querying a Backup Text File When the Primary Database Is Unavailable

```

<!--
Filename: ChoicePage4.cfm
Created by: Nate Weiss (NMW)
Purpose: Provides navigation elements for films and ratings
--->

<h2>Film Information</h2>

<cftry>
<!-- Retrieve Films from live database --->
<cfquery name="getFilms">
    
```

Listing 44.6 (CONTINUED)

```

SELECT FilmID, MovieTitles
FROM Films
ORDER BY Films.MovieTitle
</cfquery>

<!-- If any database errors occur during above query, --->
<cfcatch type="Database">

<!-- Location of our emergency backup file --->
<cfset backupFilePath = expandPath("FilmsBackup.xml")>

<!-- Read contents of the WDDX/XML in from backup file --->
<cffile action="read" file="#backupFilePath#" variable="wddxPacket">

<!-- Convert the XML back into original query object --->
<cfwddx action="WDDX2CFML" input="#wddxPacket#" output="getFilms">

<!-- Let user know that emergency version is being used --->
<p><i>NOTE:
We are not able to connect to our real-time Films database at the moment.
Instead, we are using data from our archives to display the Films list.
Please try again later today for an up to date listing.</i></p>
</cfcatch>
</cftry>

<!-- Attempt database operation --->
<cftry>
<!-- Retrieve Ratings from database --->
<cfquery name="getRatings">
SELECT RatingID, Rating
FROM FilmsRatings
ORDER BY Rating
</cfquery>

<!-- Silently catch any database errors from above query --->
<cfcatch type="Database"/>
</cftry>

<!-- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

<!-- Display Film names in a drop-down list --->
<p>Films:
<cfselect query="getFilms" name="filmID" value="FilmID"
display="MovieTitle"/>
<cfinput type="submit" name="go" value="Go">
<!-- If, after all is said and done, we were able to get Ratings data --->
<cfif isDefined("getRatings")>
<!-- Display Ratings in a drop-down list --->
<p>Ratings:
<cfselect query="getRatings" name="ratingID"
value="RatingID" display="Rating"/>
<cfinput type="submit" name="go" value="Go">
</cfif>

</cfform>

```

If the `<cfquery>` named `GetFilms` fails for whatever reason, the code in the `<cfcatch>` block will execute in an attempt to save the day. The idea behind this code is to read information about films from a previously saved XML file called `FilmsBackup.xml`. The backup file is in WDDX format, which is a special type of XML used to convert any type of data (such as a query record set) quickly and easily into a simple string format that can be saved to disk.

First, the `<cffile>` tag is used to read the contents of the `FilmsBackup.xml` file into a string variable called `wddxPacket`. At this point, the variable holds the XML code that represents the backup film data. Now the `<cfwddx>` tag is used to convert the XML code into a normal query object called `getFilms`. You can use this query object just like the results of a normal `<cfquery>` tag. In other words, the rest of this template can continue working as if the `<cfquery>` at the top of the page hadn't caused an error.

The result is a version of the template that still provides the basic functionality it's supposed to, even when the first `getFilms` query fails. Assuming that the backup file can be processed correctly, the user is presented with the expected drop-down list of films. The user also sees a message that alerts him or her to the fact that the list is populated not from the live database, but rather from an archived backup copy.

Nesting `<cftry>` Blocks

You can nest `<cftry>` blocks within one another. There are generally two ways you can nest the blocks, depending on the behavior you want.

If you nest a `<cftry>` within another `<cftry>` block (but not within a `<cfcatch>`), this doubly protects the code inside the inner `<cftry>`. This type of nesting typically follows this basic form:

```
<cftry>
  <cftry>
    <!-- Important code goes here -->

    <cfcatch></cfcatch>
  </cftry>
</cfcatch></cfcatch>
</cftry>
```

If something goes wrong, ColdFusion will see whether any of the `<cfcatch>` tags within the inner `<cftry>` are appropriate (that is, whether a `<cfcatch>` of the same type as the exception itself exists). If so, the code in that block executes. If not, ColdFusion looks to see whether the `<cfcatch>` tags within the outer `<cftry>` block are appropriate. If it doesn't find any appropriate `<cfcatch>` blocks there either, it considers the exception uncaught, so the default error message displays.

Alternatively, you can essentially create a two-step process by nesting a `<cftry>` within a `<cfcatch>` that belongs to another `<cftry>`. This second type of nesting is typically structured like this:

```
<cftry>
  <!-- important code here -->
  <cfcatch>
    <cftry>
      <!-- fallback code here -->
    </cfcatch>
```

```

<!-- last chance code here -->
</cfcatch>
</cftry>
</cfcatch>
</cftry>

```

If the code in the outer `<cftry>` fails, the inner `<cftry>` attempts to deal with the situation in some other way. If the code in the inner `<cftry>` also fails, its `<cfcatch>` tags can catch the error and perform some type of last-ditch processing (such as displaying an error message).

Listing 44.7 is an example that includes both of these forms of `<cftry>` nesting. It builds upon the previous listings by adding two `<cftry>` blocks within the larger `<cftry>` block that begins near the top of the template.

Listing 44.7 ChoicePage5.cfm—Nesting `<cftry>` Blocks Within Each Other

```

<!--
  Filename: ChoicePage5.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides navigation elements for films and ratings
-->

<h2>Film Information</h2>

<cftry>
  <!-- Location of our emergency backup file -->
  <cfset backupFilePath = expandPath("FilmsBackup.xml")>

  <!-- Retrieve Films from live database -->
  <!-- MovieTitle2 is an invalid column name -->
  <cfquery name="getFilms">
    SELECT FilmID, MovieTitle2
    FROM Films
    ORDER BY Films.MovieTitle
  </cfquery>

  <!-- If the backup file has never been created -->
  <cfif not fileExists(backupFilePath)>
    <!-- Now we'll make sure that our backup file exists -->
    <!-- If it doesn't exist yet, we'll try to create it -->
    <cftry>

      <!-- Convert the query to WDDX (an XML vocabulary) -->
      <cfwddx action="CFML2WDDX" input="#getFilms#" output="wddxPacket">

        <!-- Save the XML on server's drive, as our backup file -->
        <cffile action="write" file="#backupFilePath#" output="#wddxPacket#"

        <!-- Silently ignore any errors while creating backup file -->
        <!-- (the worst that happens is the backup file doesn't get made) -->
        <cfcatch type="any"/>
      </cftry>
    </cfif>

    <!-- If any database errors occur during above query, -->
    <cfcatch type="Database">

```

Listing 44.7 (CONTINUED)

```

<cftry>
<!-- Read contents of the WDDX/XML in from backup file -->
<cffile action="read" file="#backupFilePath#" variable="wddxPacket">
<!-- Convert the XML back into original query object -->
<cfwddx action="WDDX2CFML" input="#wddxPacket#" output="getFilms">

<!-- Let user know that emergency version is being used -->
<p><i>NOTE:
We are not able to connect to our real-time Films database at the moment.
Instead, we are using data from our archives to display the Films list.
Please try again later today for an up to date listing.</i></p>

<!-- If any problems occur while trying to use the backup file -->
<cfcatch type="Any">
<!-- Let user know that the Films data can't be shown right now -->
<i>Sorry, we are not able to provide you with a list of films.</i><br>
</cfcatch>
</cftry>

</cfcatch>
</cftry>

<!-- Attempt database operation -->
<cftry>
<!-- Retrieve Ratings from database -->
<cfquery name="getRatings">
SELECT RatingID, Rating
FROM FilmsRatings
ORDER BY Rating
</cfquery>

<!-- Silently catch any database errors from above query -->
<cfcatch type="Database"/>
</cftry>

<!-- Create self-submitting form -->
<cfform action="#CGI.script_name#" method="post">

<!-- If, after all is said and done, we were able to get Film data -->
<cfif isDefined("getFilms")>
<!-- Display Film names in a drop-down list -->
<p>Films:
<cfselect query="getFilms" name="filmID" value="FilmID" display="MovieTitle"/>
<cfinput type="submit" name="go" value="Go">
</cfif>

<!-- If, after all is said and done, we were able to get Ratings data -->
<cfif isDefined("getRatings")>
<!-- Display Ratings in a drop-down list -->
<p>Ratings:
<cfselect query="getRatings" name="ratingID"
value="RatingID" display="Rating"/>
<cfinput type="submit" name="go" value="Go">
</cfif>

</cfform>

```

The first of the nested `<cftry>` blocks attempts to create the `FilmsBackup.xml` file if it hasn't been created already, or if it has been deleted for some reason. A new version of the backup data is serialized using `<cfwddx>`, then written to the server's drive using `<cffile>`. If any errors occur during this file-creation process, they will silently be ignored (no error message is shown). In other words, if the backup file doesn't exist, this nested `<cftry>` will attempt to create it. But if `<cftry>` can't do so, it just moves on.

NOTE

Since the first nested `<cftry>` sits within an outer `<cftry>` that catches all exceptions of `TYPE="Database"`, the nested `<cftry>` will be skipped altogether if `<cfquery>` causes a database error.

NOTE

If you want to test this code, delete the `FilmsBackup.xml` file from the directory you're using for this chapter's examples, then visit this template in your browser. Provided the query itself doesn't have a problem, the backup file should be re-created.

The second nested `<cftry>` is within the outer `<cftry>` tag's `<cfcatch>` block. This is the portion of the code that attempts to read the backup file if the `<cfquery>` fails (as introduced in Listing 44.6). Remember, because of where it's now nested, this inner `<cftry>` block will only be encountered if there is a problem retrieving the live film data from the database. This inner `<cftry>` tells ColdFusion that if the database query fails and there is a problem with the emergency `<cffile>` and `<cfwddx>` code, then the template should just give up and display a "Sorry, we are not able to provide you with a list of films" message.

In other words, this version of the template has double protection: It has a fallback plan if the database query fails, and it knows how to degrade gracefully even if the fallback plan fails. This is quite an improvement over the original version of the template (refer to Listing 44.1), which could do no better than display a user-unfriendly error message when something went wrong.

Deciding Not to Handle an Exception

When your code catches an exception with `<cfcatch>`, it assumes all responsibility for dealing with the exception in an appropriate way. After your code catches the exception, ColdFusion is no longer responsible for logging an error message or halting further page processing.

For instance, look back at Listing 44.7 (`ChoicePage5.cfm`). The `<cfcatch>` tags in this template declare themselves fit to handle any and all database-related errors by specifying a `type="Database"` attribute in the `<cfcatch>` tag. No matter what type of database-related exception gets raised, those `<cfcatch>` tags will catch the error.

The purpose of the first `<cfcatch>` block in Listing 44.7 is to attempt to use the backup version of the database when the first `<cfquery>` fails. This backup plan is activated no matter what the actual problem is. Even a simple syntax error or misspelled column name in the SQL statement will cause the `FilmsBackup.xml` file to be used.

It would be a better policy for the backup version of the database to be used only when the original query failed due to a connection problem. Other types of errors, such as syntax errors, should probably not be caught and dealt with in the same way.

ColdFusion provides the `<cfrethrow>` tag for exactly this type of situation. This section explains how to use `<cfrethrow>` and when to use it.

Exceptions and the Notion of Bubbling Up

Like the error or exception constructs in many other programming languages, ColdFusion's exceptions can do something called *bubbling up*.

Let's say you have some code that includes several layers of nested `<cftry>` blocks and that an exception has taken place in the innermost block. If the exception isn't caught in the innermost `<cftry>` block, ColdFusion looks in the containing `<cftry>` block, if any, to see whether it contains any appropriate `<cfcatch>` tags. If not, the exception continues to "bubble up" through the layers of `<cftry>` blocks until it's caught. If the error bubbles up through all the layers of the `<cftry>` blocks (that is, if none of the `<cfcatch>` tags in the `<cftry>` blocks choose to catch the exception), ColdFusion then displays its default error message.

Using `<cfrethrow>`

The `<cfrethrow>` tag is basically the opposite of `<cfcatch>`. After `<cfcatch>` catches an error, `<cfrethrow>` can uncatch the error. It's then free to bubble up to the next containing `<cftry>` block, if any.

The `<cfrethrow>` tag takes no attributes and can be used only inside a `<cfcatch>` tag. Typically, you will decide to use it by testing the value of one of the special `CFCATCH` variables shown previously in Table 44.3.

For instance, with Apache Derby, if the path to the file defined in the data source is incorrect, the `CFCATCH.SQLState` variable will be set to `XJ004`. If you want your code to handle only errors of this specific type, letting all other errors bubble up normally, you would use code similar to the following within a `<cfcatch>` block:

```
<cfif CFCATCH.SQLState neq "XJ004">
  <cfrethrow>
</cfif>
```

Listing 44.8, a revision of Listing 44.7, uses the basic `<cfif>` test shown in the previous snippet to ensure that the backup data file (`FilmsBackup.xml`) is used only if the database exception has a `SQLState` value of `XJ004`.

Listing 44.8 ChoicePage6.cfm—Using `<cfrethrow>` to Process Only Database Errors of Code `XJ004`

```
<!---
  Filename: ChoicePage6.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides navigation elements for films and ratings
-->

<h2>Film Information</h2>

<cftry>
  <!--- Location of our emergency backup file -->
```

Listing 44.8 (CONTINUED)

```

<cfset backupFilePath = expandPath("FilmsBackup.xml")>

<!-- Retrieve Films from live database -->
<cfquery name="getFilms">
SELECT FilmID, MovieTitle
FROM Films
ORDER BY Films.MovieTitle
</cfquery>

<!-- Now we'll make sure that our backup file exists -->
<!-- If it doesn't exist yet, we'll try to create it -->
<!-- Also, re-create it whenever server is restarted -->
<cftry>
<!-- If the server has just been restarted, or -->
<!-- if the backup file has never been created -->
<cfif not isDefined("application.getFilmsBackupCreated")
or not fileExists(backupFilePath)>

    <!-- Convert the query to WDDX (an XML vocabulary) -->
    <cfwddx action="CFML2WDDX" input="#getFilms#" output="wddxPacket">

    <!-- Save the XML on server's drive, as our backup file -->
    <cffile action="write" file="#backupFilePath#" output="#wddxPacket#">

    <!-- Remember that we just created the emergency file -->
    <!-- This will be forgotten when server is restarted; thus, backup -->
    <!-- file will be refreshed on first successful query after restart -->
    <cfset application.getFilmsBackupCreated = True>
</cfif>

<!-- Silently ignore any errors while creating backup file -->
<!-- (the worst that happens is the backup file doesn't get made) -->
<cfcatch TYPE="Any"/>
</cftry>

<!-- If any database errors occur during above query, -->
<cfcatch type="Database">

    <!-- Unless this is SQL Error XJ004, un-catch the exception -->
    <cfif CFCATCH.SQLState neq "XJ004">
        <cfrethrow>
    <!-- If it's SQL Error, XJ004, attempt to get data from txt file -->
    <cfelse>
        <cftry>

            <!-- Read contents of the WDDX/XML in from backup file -->
            <cffile action="read" file="#backupFilePath#" variable="wddxPacket">

            <!-- Convert the XML back into original query object -->
            <cfwddx action="WDDX2CFML" input="#wddxPacket#" output="getFilms">

            <!-- Let user know that emergency version is being used -->
            <p><i>NOTE:
We are not able to connect to our real-time Films database at the
moment. Instead, we are using data from our archives to display
the Films list. Please try again later today for an up to date

```

Listing 44.8 (CONTINUED)

```

        listing.</i></p>

        <!-- If any problems occur while trying to use the backup file --->
        <cfcatch type="Any">
            <!-- Let user know that the Films data can't be shown right now --->
            <i>Sorry, we are not able to provide you with a list of films.</i><br>
        </cfcatch>
    </cftry>

    </cfif>
</cfcatch>
</cftry>

<!-- Attempt database operation --->
<cftry>

    <!-- Retrieve Ratings from database --->
    <cfquery name="getRatings">
        SELECT RatingID, Rating
        FROM FilmsRatings
        ORDER BY Rating
    </cfquery>

    <!-- Silently catch any database errors from above query --->
    <cfcatch type="Database"/>
</cftry>

<!-- Create self-submitting form --->
<cfform action="#CGI.script_name#" method="post">

    <!-- If, after all is said and done, we were able to get Film data --->
    <cfif isDefined("getFilms")>
        <!-- Display Film names in a drop-down list --->
        <p>Films:
        <cfselect query="getFilms" name="filmID" value="FilmID" display="MovieTitle"/>
        <cfinput type="submit" name="go" value="Go">
    </cfif>

    <!-- If, after all is said and done, we were able to get Ratings data --->
    <cfif isDefined("getRatings")>
        <!-- Display Ratings in a drop-down list --->
        <p>Ratings:
        <cfselect query="getRatings" name="ratingID"
        value="RatingID" display="Rating"/>
        <cfinput type="submit" name="go" value="Go">
    </cfif>

</cfform>

```

To test the exception-handling behavior of this template, try editing the DSN and editing the database folder. Now visit Listing 44.8 with your Web browser. That should cause error XJ004, so the backup version of the `Films` table will be used. Now correct the database folder and sabotage the first `<cfquery>` in some other way—for instance, by changing one of the column names to

something invalid. That should cause the `<cfrethrow>` tag to fire, which in turn causes the error to be raised again. Because there are no other `<cfcatch>` tags to handle the exception that was raised again, ColdFusion displays its usual error message (which you could customize with `<cferror>`, as discussed in Chapter 18).

You are free to have as many `<cfif>` tests as you need to make the decision whether to rethrow the error. For instance, if you wanted your `<cfcatch>` code to handle only errors of types `XJ004`, `S1005`, and `S1010` instead of only `XJ004`, you could replace the `<CFIF>` test in with something like the following:

```
<cfif listFindNoCase("XJ004,S1005,S1010", CFCATCH.SQLState) EQ 0>
<cfrethrow>
</cfif>
```

Throwing and Catching Your Own Errors

You can throw custom exceptions whenever your code encounters a situation that should be considered an error within the context of your application, perhaps because it violates some type of business rule. So custom exceptions give you a way to teach your ColdFusion code to treat certain conditions—which ColdFusion wouldn’t be capable of identifying as problematic on its own—as exceptions, just like the built-in exceptions thrown by ColdFusion itself.

Introducing `<cfthrow>`

To throw your own custom exceptions, use the `<cfthrow>` tag. The exceptions you raise with `<cfthrow>` can be caught with a `<cftry>/<cfcatch>` block, just like the exceptions ColdFusion throws internally. If your custom exception isn’t caught (or is caught and then rethrown via the `<cfrethrow>` tag), ColdFusion simply displays the text you provide for the `message` and `detail` attributes in a standard error message.

Table 44.4 lists the attributes you can provide for the `<cfthrow>` tag. All of them are optional, but it’s strongly recommended that you at least provide the `message` attribute whenever you use `<cfthrow>`.

Table 44.4 `<cfthrow>` Tag Attributes

ATTRIBUTE	DESCRIPTION
type	A string that classifies your exception into a category. You can provide either <code>type="APPLICATION"</code> , which is the default, or a <code>type</code> of your own choosing. You can’t provide any of the predefined exception types listed in Table 44.2. You can specify custom exception types that include dots, which enables you to create hierarchical families of custom exceptions. See the next section, “Throwing Custom Exceptions.”

Table 44.4 (CONTINUED)

ATTRIBUTE	DESCRIPTION
message	A text message that briefly describes the error you are raising. If the exception is caught in a <code><cfcatch></code> block, the value you provide here is available as <code>CFCATCH.Message</code> . If the exception isn't caught, ColdFusion displays this value in an error message to the user. This value is optional, but it's strongly recommended that you provide it.
detail	A second text message that describes the error in more detail, or any background information or hints that will help other people understand the cause of the error. If the exception is caught in a <code><cfcatch></code> block, the value you provide here is available as <code>CFCATCH.Detail</code> . If the exception isn't caught, ColdFusion displays this value in an error message to the user.
errorCode	An optional error code of your own devising. If the exception is caught in a <code><cfcatch></code> block, the value you provide here is available as <code>CFCATCH.ErrorCode</code> .
extendedInfo	A second optional error code of your own devising. If the exception is caught in a <code><cfcatch></code> block, the value you provide here will be available as <code>CFCATCH.ExtendedInfo</code> .
object	Allows you to throw an exception object defined in Java. The object must first be created by <code><cfobject></code> or <code>createObject()</code> before it can be used in <code><cfthrow></code> .

Throwing Custom Exceptions

Say you're working on a piece of code that performs some type of operation (such as placing an order) based on a `ContactID` value. As a sanity check, you should verify that `ContactID` is actually valid before going further. If you find that `ContactID` isn't valid, you can throw a custom error using the `<cfthrow>` tag, like so:

```
<cfthrow
  message="Invalid Contact ID"
  detail="No record exists for that Contact ID.">
```

This exception can be caught using a `<cfcatch>` tag, as shown in the following. Within this `<cfif>` block, you can take whatever action is appropriate in the face of an invalid contact ID (perhaps inserting a new record into the `Contacts` table or using `<cfmail>` to send a message to your customer service manager):

```
<cfcatch type="application">
  <cfif CFCATCH.message eq "Invalid Contact ID">
    <!-- recovery code here -->
  </cfif>
</cfcatch>
```

This `<cfcatch>` code catches the `Invalid Contact ID` exception because the exception's type is `application`, which is the default exception type used when no type attribute is provided to `<cfthrow>`. If you want, you can specify your own exception type, like so:

```
<cfthrow
  type="InvalidContactID"
  message="Invalid Contact ID"
  detail="No record exists for that Contact ID.">
```

This exception will be caught by any `<cfcatch>` tag that has a matching type, like this one:

```
<cfcatch type="InvalidContactID">
  <!-- recovery code here -->
</cfcatch>
```

NOTE

You can also use the `<cfabort>` tag with the `showError` attribute to throw a custom exception. If the exception is caught, the text you provide for `showError` becomes the `CFCATCH.Message` value. However, the use of `<cfthrow>` is preferred because you can provide more complete information about the error you are raising, via type, detail, and other attributes shown in Listing 44.5.

Handling Exceptions with Scripting

All of the examples in this chapter made use of ColdFusion tags. What if you wanted to use `cfscript` instead? The good news is that *everything* described in this chapter can be done with script syntax. That includes try and catch, throw, and rethrow. Which is better? Neither! Whether you chose to use tag-based or script-based syntax is entirely up to you. Listing 44.9 demonstrates a simple example of script-based error handling.

Listing 44.9 scriptbased.cfm—Error Handling with `cfscript`

```
<!--
  Filename: scriptbased.cfm
  Created by: Raymond Camden
  Purpose: Simple example of script based error handling.
-->

<cfscript>

try {
    writeoutput("#x#");
} catch(any e) {
    throw(message="You did not define x!", detail="Hey, why didn't you define X?");
}

try {
    z = x+1;
} catch(any e) {
    rethrow;
}
</cfscript>
```

Listing 44.9 isn't very long—one of the benefits of scripting is its terseness. The template contains two try/catch blocks. These operate the same as their tag-based counterparts, but notice the use of the `{` and `}` characters to define the code that will be tried. In the first try/catch block, a custom exception is thrown using the `throw` keyword. In the second code block, which will only execute if `x` exists and is not numeric, we simply rethrow the exception. To test this template, open it in your browser. You will get an error immediately (the custom one), but you can make that go away by simply adding `?x=1` to the URL. To see the rethrow in action, change the value of `x` to some non-numeric value.

CHAPTER 45

Using the Debugger

IN THIS CHAPTER

Overview E72

Configuring ColdFusion and the Debugger E73

Exploring the Debugger E80

ColdFusion Builder offers a powerful new developer tool in its debugger, an interactive step debugger that allows CFML developers to walk through the execution of their code, observing what lines of code it executes, what other CFML files (include files, custom tags, or CFCs) it executes, the value of all variables (optionally in all scopes) at a given point during execution, and much more. You can even change the value of variables on the fly within the debugger, such as to alter the run-time flow of execution of your code.

More than just watching the code that you run yourself, the debugger can watch the execution of any request (including requests from other users running code on your server), and also any CFML however it's requested, including calls from Ajax or Flex clients or from other servers issuing Web service requests or CFHTTP equivalents.

There are also other kinds of requests that can happen within ColdFusion through no traditional browser: scheduled tasks, requests made via the ColdFusion gateways, and events in the `Application.cfc` file, such as `onSessionEnd` and `onApplicationEnd`. With all these kinds of requests, there often is no browser to which you can easily send back traditional debugging statements.

And even with traditional browser requests, there are still times when CFML code can't display any output, such as when output has been disabled in a CFC or method, or within a `CFSILENT` tag pair, and so on.

These are just some of the scenarios in which a debugger becomes so valuable, where you use a tool to watch the execution of some code while it runs live, regardless of how it was requested by a user. Of course, there is all the requisite security to ensure that the debugger is used only by authorized developers.

The debugger is built into ColdFusion Builder, which has been introduced in other chapters in this series, especially Chapter 3, “Introducing ColdFusion Builder,” in *Adobe ColdFusion 9 Web Application Construction Kit, Volume 1: Getting Started*. Whether you're new to debugging or feel lukewarm about the concept, this chapter will help you feel comfortable and confident using the debugger as a

tool in your CFML developer toolkit. It will introduce debugging and the debugger, how to install and configure it, and how to use it.

The initial discussion of understanding, installing, and configuring the debugger (and ColdFusion) may seem a bit tedious, but it's generally a one-time effort. Once you configure the debugger, you will be able to use it quite easily and quickly.

Overview

We start with an overview of traditional and step debugging.

Traditional Forms of Debugging

As developers, we spend most of our day debugging code: whether it's new code we're writing, old code we're updating, or someone else's code we've inherited. When trying to resolve a problem in a given CFML template or CFC, you often need to understand the flow and function of the code, and to do that you have a couple choices.

First, you can try to just statically review the source code, relying on your skills at reading and interpreting the code to determine what it should be doing, eye-balling potentially many lines of code, and guessing at or tracking manually what the values of various variables will be at any given point. This can be a complicated effort, but some developers are especially well suited to the task.

More commonly we instead let ColdFusion just run the page and then we use either of two techniques to have it report what's going on in the execution of the code. The ColdFusion debugging output, which is displayed at the end of a page request (if enabled in the ColdFusion Administrator), can show high-level information about that request, including how long it took to run, what files it called (includes, custom tags, CFCs), what queries it executed (their duration, record count, SQL, and such), the variables and values in many variable scopes, and much more. This feature was discussed in Chapter 17, "Debugging and Troubleshooting," in Volume 1.

Still, there are times when either that level of detail is not enough or debugging output is not enabled (and/or cannot be). In that case most developers will resort to placing `CFOUTPUT` and `CFDUMP` tags strategically in the program code both to depict the flow of control ("What lines of code am I running?") and to determine the value of variables at a given point. This is a tried and true method, and for many developers they've known no other alternative.

Introducing Step Debugging

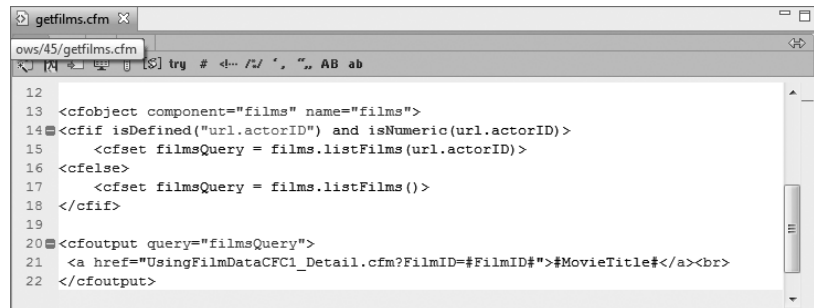
Developers who come from some other programming environments, however, are often surprised to find CFML developers having only these techniques at their disposal. Such developers have often had the advantage of using a step debugging tool to help them in this very sort of situation. With a step debugger, a developer can actually observe the lines of code as they are executed, watching each line begin and end, as well as watching the value of any and all variables in play at that point in the code.

Consider the code fragment in Figure 45.1 (from `getf1ms.cfm` in the source code directory for this chapter), which depicts some code about to be executed. As we observe the code, we can wonder

whether the first “if” test will be true, and we may wonder what goes on in the `listFilms` method that would be called. If we knew the value of the variables being tested, then of course we’d know which test would be true.

Figure 45.1

Sample CFML code being debugged.



A step debugger could tell us all that information, and more. We could tell the debugger to stop execution at any given line of code, and while there we could view all the currently defined variables (in any of the scopes we’ve configured the debugger to show). More important, we could ask the debugger to step through the code, running whatever the next line would be (line 14), and then step again, in which case we’d run either line 15 or 16, depending on the condition evaluated. No more guessing!

Beyond that, while observing this code (without a debugger) we may wonder other things: how did we get to this point in the code? Was this file called as an include file, custom tag, or CFC method? Did we run an `Application.cfm` or `Application.cfc` file before getting here?

And of course we may even wonder about the `listFilms` function itself: What does it do, where is it defined, and so on. The great news is that we could ask the debugger to step into that method, and since it’s a CFC method in another file, the debugger would find the file, open it, and stop on the first line of code inside that file. The same would be true if we stepped into included files, calls to custom tags, and so on.

All this and more can be discovered with the ColdFusion Builder debugger. Let’s look next at how to configure it; then we’ll see how to use the debugger to answer the questions just posed.

Configuring ColdFusion and the Debugger

To use the debugger against a given implementation of ColdFusion (8 or 9), you must configure that ColdFusion server to permit debugging, and then you must configure ColdFusion Builder itself. We’ll cover each of these topics in this section.

NOTE

The debugger works only with ColdFusion 8 or 9. If you’re interested in debugging ColdFusion 6 or 7 servers, look into Fusion-Debug, a commercial debugger that functions similarly to the ColdFusion Builder debugger and can debug ColdFusion 6 or higher. For more information, see www.fusiondebug.com.

Configuring the ColdFusion Administrator

You enable a server to permit debugging by way of settings in the ColdFusion Administrator. The setting for the ColdFusion debugger feature is separate from the traditional debugging output feature, which is also enabled in the Administrator. There are also important settings on other Administrator screens.

Enable Debugger Settings

The first and most important settings to enable for debugging are on a page in the Debugging & Logging section of the Administrator, in a page called Debugger Settings (Figure 45.2).

There, you must select the Allow Line Debugging option and click Submit Changes to enable debugging against this server. Note that you must restart the server upon enabling (or disabling) this setting for the it to take effect. (More on this is offered later, in the section “Special Considerations for Multiserver Deployment.”)

Figure 45.2

ColdFusion
Administrator
Debugger
Settings page.

Debugging & Logging > Debugger Settings

Enable the Allow Line Debugging option to use the ColdFusion Debugger that runs in Eclipse. Specify the port and the maximum number of simultaneous debugging sessions.

Line Debugger Settings

☒ Allow Line Debugging

Debugger Port: 5005

Maximum Simultaneous Debugging Sessions: 5

Debugging Server

The debugging server runs as a process separate from the ColdFusion Server. You can start, stop or restart the debugging server from this page, however, this is usually not necessary because the debug process is managed automatically when a debug session is started.

Stop Debugger Server Restart Debugger Server

Click the button on the right to update Debugger Settings... Submit Changes

While on that Debugger Settings page, notice the available Debugger Port setting, which you need not change unless the port is one that’s unavailable on your server.

WARNING

If you’re debugging a server that is remote to you, beware of the following issue. The debugging feature on the server actually listens for commands (from ColdFusion Builder) on a port separate from the one specified in the aforementioned setting. For the sake of security, ColdFusion will by default cause the debugger feature to use a randomly available port. This would be a problem if ColdFusion is behind a firewall and you’re trying to debug it from outside that firewall, since the firewall will block that random port.

One solution is to modify ColdFusion’s Java configuration arguments to specify a fixed debugger server port number (different from that mentioned earlier), and then modify the firewall to allow access to this port from your (or any authorized developer’s) machine.

To set a fixed debugger port number, specify the following JVM argument on the Java And JVM page of the ColdFusion Administrator (in ColdFusion Standard or Server deployments) or in `jvm.config` for ColdFusion Multiserver deployments:

`-DDEBUGGER_SERVER_PORT=portNumber`

Replace `portNumber` with the port that you wish to use, and restart ColdFusion. Be careful when modifying the Java configuration arguments, as incorrect values could prevent ColdFusion from starting. Be sure to make a copy of the `jvm.config` file before making these changes.

The next setting on the Debugger Settings page is Maximum Simultaneous Debugging Sessions, which need not be changed unless you expect to have more than the default number of developers (five) debugging against a single shared server.

I'll explain the Start Debugger Server button at the end of the chapter, in the section "Stopping the Debugger." For now, just know that you don't need to click the button to start debugging. The first debugging request will cause debugging to start on the server. (And note that when debugging has started, the button will change to Stop Debugger, and a Restart Debugger button will appear next to it.)

Configuring RDS to Secure the Server

Once the Allow Debugging option is enabled, which permits a server to be debugged, the next significant step is controlling who may debug code on the server. Adobe chose to build debugger security on top of the long-standing Remote Developer Services (RDS) feature, to provide secured access for developers using editors (including Dreamweaver and ColdFusion Builder) trying to access information from the ColdFusion server.

For a ColdFusion server to be debugged, the developer enabling debugging from ColdFusion Builder will need to provide credentials as indicated in the RDS settings of the server. The section "Configuring ColdFusion Builder" later in this chapter explains how to use these RDS credentials in ColdFusion Builder.

The ColdFusion server administrator controls whether RDS is enabled (it can be disabled at installation) and also chooses the credentials (password, if any) that will be required by developers, as configured on the Administrator's Security > RDS page.

Note that ColdFusion Enterprise and Developer editions (since ColdFusion 8) add a powerful RDS feature, allowing multiple user accounts, so that different developers can each be given their own RDS username/password pair authorizing them to access the server from development tools such as ColdFusion Builder and the debugger. RDS security is discussed further in Chapter 56, "Security in Shared and Hosted Environments," in *Adobe ColdFusion 9 Web Application Construction Kit, Volume 3: Advanced Application Development*.

Increase Maximum Request Timeouts

While debuggers are powerful in enabling you to stop on a line of code and inspect variables and so on, there is a downside. ColdFusion is often configured to forcefully timeout requests that take too long, as configured in the Administrator's Server Settings > Settings page. If the Timeout Requests after (seconds) option is checked, then ColdFusion will try to terminate requests lasting longer than the specified time limit. When debugging code on a server using the step debugger, you will need to either increase the time limit or disable this feature entirely (in which case requests are free to run as long as they need to, which may be acceptable on a developer workstation).

Special Considerations for Multiserver Deployment

A final discussion regarding ColdFusion server configuration for the debugger revolves around a difference between installing ColdFusion in the stand-alone (Server) mode versus the Multiserver (or multi-instance) or J2EE deployment mode. In the former, when you enable the aforementioned setting, Allow Line Debugging, ColdFusion will automatically modify the underlying `jvm.config` file used to control the server, placing there the appropriate information needed to enable debugging.

In the Multiserver or J2EE modes, however, you must manually place the following information into the `jvm.config` file on the existing `java.args` line provided there:

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=#portNum#
```

Here, `#portnum#` is the port number entered in the Debugger Port setting, discussed earlier. That information (and the rest of the `java.args` line) must be entered on a single line, without line breaks.

WARNING

Be sure to make a copy of the `jvm.config` file before editing it. Any mistake in the file could prevent ColdFusion restarting.

Yet another concern when you run the Multiserver edition of ColdFusion is that by default all the instances share a single `jvm.config` file. This is a problem when using the debugger, as only one instance at a time can use the debugger port specified above. You may find that at times when an instance goes down, the lock on the debugger port remains tied to one or the other instances. In this case, you should explore giving each instance its own `jvm.config` file, with its own designation of the debugger port. This can be done most easily by starting a ColdFusion instance from the command line, which may be especially suitable in a development environment. You can also configure an instance that is started as a Microsoft Windows service so that different instances use different configuration files. Each approach is discussed in the Adobe technote at http://www.adobe.com/go/tn_18206.

Upon configuration of ColdFusion, you're ready to proceed to configuring ColdFusion Builder.

Configuring ColdFusion Builder

To connect a developer's ColdFusion Builder installation to a ColdFusion server, you need to configure a few settings in the ColdFusion Builder environment, including settings to define a project, define a server connection, configure an RDS connection, and possibly add debug mappings and debugger settings, and then you need to define a ColdFusion Builder debugging configuration for ColdFusion.

NOTE

As discussed in Chapter 3, ColdFusion Builder can be installed both in a stand-alone mode and as a plug-in on an existing Eclipse installation. While debugging can be performed in either, this chapter describes steps and shows screenshots for the stand-alone approach.

Define a ColdFusion Builder Project

For you to debug a CFML template, it must exist within a ColdFusion Builder project. If you're new to Eclipse-based IDEs like ColdFusion Builder, you may not be familiar with or may not yet have defined a project. Project creation is discussed in Chapter 3, as well as in the ColdFusion Builder Help section "Managing Projects."

Define a Server Connection

The template needs to exist within a project, and in addition that project must be defined with a server connection to the ColdFusion server. You may be using projects without having defined a server connection yet. How to create a server connection is discussed in Chapter 3, as well as in the ColdFusion Builder Help section "Adding ColdFusion Servers," which discusses how to create a connection to both local and remote servers.

The server connection is also where you will specify the RDS connection and authentication information as discussed in the first section of this chapter. For additional help on configuring and testing an RDS connection, see the ColdFusion Builder Help section "ColdFusion Builder Development Perspective."

Configure Debug Mappings If Needed

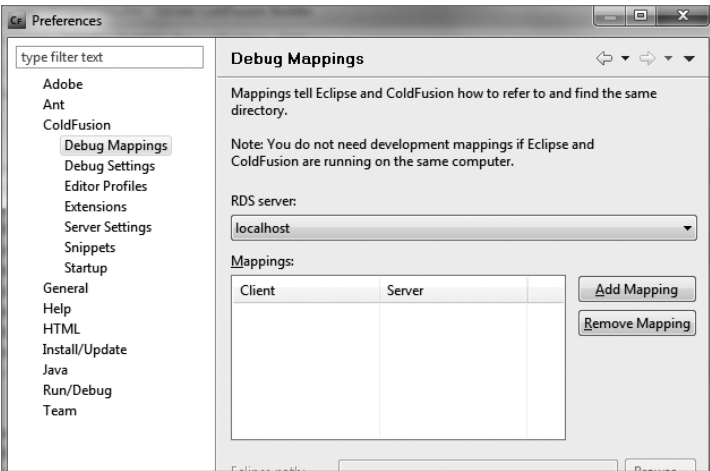
If ColdFusion and ColdFusion Builder are running on the same computer, you can skip the Mappings setting (Window > Preferences > ColdFusion > Debug Mappings). Otherwise, use this setting to map a connection between the path to files as ColdFusion Builder will see them (on your local machine typically) and the path to the same files as they will be found by the ColdFusion server. For instance, if you edit and test files locally in `C:\inetpub\wwwroot\someapp\` but later upload them to a remote or central server where you'd like to debug them, and they're identified on that server as being in `D:\webs\someapp`, then these would be the two values you'd specify on this page for the client and server mappings.

NOTE

Both paths are case-sensitive. There will be no error if the incorrect case is used here, but when you later attempt to debug against the given remote server, breakpoints simply won't fire.

To add a new set of mappings, first choose the server against which you will be debugging (in the RDS Server list, populated per the previous steps described above), then click Add Mapping and follow the instructions to fill in the Eclipse and ColdFusion paths, and click Apply to save the settings. See Figure 45.3. (Again, though, if you're running ColdFusion Builder and ColdFusion on the same computer, you don't need to add a debug mapping.)

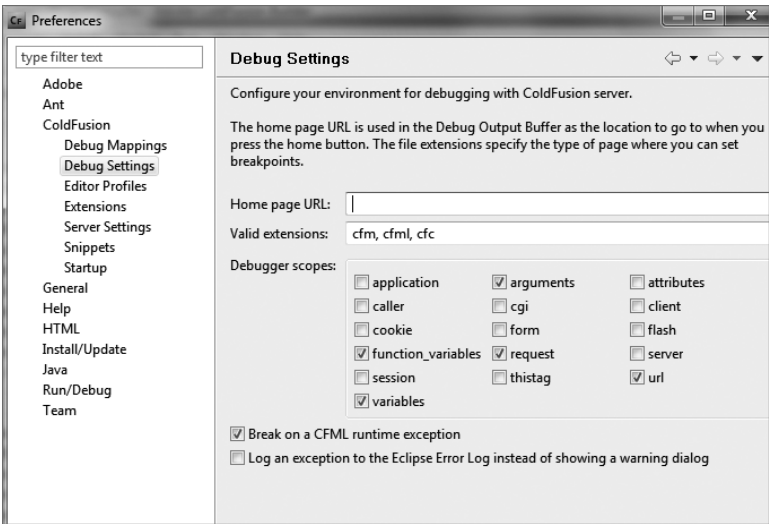
Figure 45.3
ColdFusion Builder
Debug Mappings
page.



Configure ColdFusion Builder Debug Settings

You specify the next segment of ColdFusion Builder debug settings on the Debug Settings page at Window > Preferences > ColdFusion > Debug Settings. There are four settings configurable on the page, with the final two being most important (Figure 45.4).

Figure 45.4
ColdFusion Builder
Debug Settings page.



First, you can specify (if desired) a home page URL, which will be used to point to the page that appears in the Browser pane of the Debug Output Buffer of the debugger (a feature discussed later) when you click the Home button there (not something you’ll likely do that often). Also, you can modify the extensions of the types of files that you want to debug.

More important for most is the Debugger Scopes setting, where you can specify which variable scopes you want the debugger to display when you're displaying variables (discussed later). Though it's tempting to select many scopes to see all possible information within the debugger, each will add more overhead to (and slow execution of) the debugger.

TIP

Here's an important tip: you can always view any variable in any scope using the Add Watch Expression feature, discussed later.

The fourth setting on this page is Break on a CFML Runtime Exception. With this feature enabled, the debugger will intercept an error occurring in any template within a project you are actively debugging (see the next section, "Manage Debug Configurations"). The error message will appear in an alert window, and the debugger will stop at the line that caused the error. This is a powerful feature since it precludes your needing to anticipate where code in a program might cause an error.

Remember also that because the debugger can intercept any request, you could be notified about errors happening to other users executing code in a project you are debugging. Note also that once the debugger has intercepted a request, it will not intercept another until you have completed step debugging of the first request.

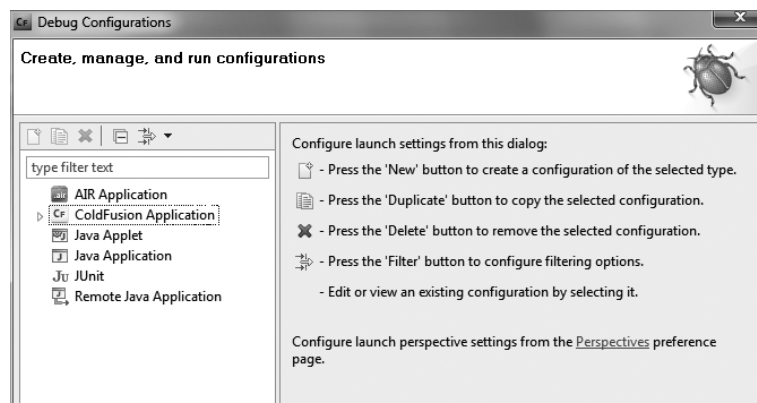
The final option is Log an Exception to the Eclipse Error Log Instead of Showing a Warning Dialog. With this option enabled, the Break on Exception feature will not display the pop-up message, although it will still intercept the request triggering the error. Instead, the error will be logged to the Eclipse Error Log (located in the `.metadata\.log` file in your Eclipse workspace).

Manage Debug Configurations

The final step in being able to debug ColdFusion code is to create a new ColdFusion Builder debugging configuration for ColdFusion, which simply indicates for a given project the ColdFusion server to which you want to connect (as defined in the "Define a Server Connection" section earlier in this chapter). In ColdFusion Builder, choose the menu command Run > Debug Configurations, which opens a window for creating, managing, and running debugging configurations (Figure 45.5).

Figure 45.5

ColdFusion Builder
Debug Configurations
page.

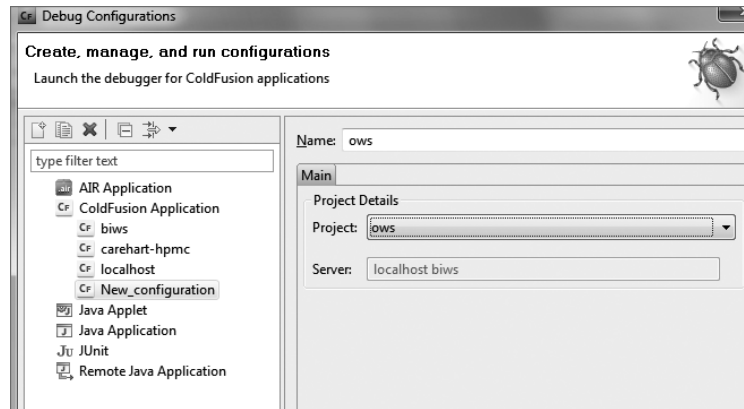


Here you could define debug configurations for different ColdFusion applications and projects. To create a new ColdFusion configuration, double-click the ColdFusion Application option, or single-click the ColdFusion Application option and, as directed on the screen, click the New button. The icon to click is depicted in the instructions shown on the screen as in Figure 45.5.

This step creates a new configuration, as shown in Figure 45.6.

Figure 45.6

Adding a new ColdFusion debug configuration.



Here you can specify a name for the debugging configuration, which can be anything you like, such as *ows*, as I have used here since I created an *ows* project for which I'll want to perform debugging. The name is simply a designation you'll use later when launching a debugging session.

Then select the project that you want to debug, using the Project drop-down list of options. These are projects that you have created previously, as discussed briefly earlier, in the section "Define a ColdFusion Builder Project." The Server field will be populated automatically with the name of whatever server configuration you defined for the project.

NOTE

If you can't select any project in this New Configuration page, that's an indication that you've already created a debug configuration for that project. You cannot create another debug configuration for the same project.

Having selected a project (and server) name, you could click Debug to cause ColdFusion Builder to attempt to start a ColdFusion debugging session for the selected project or server, but let's leave that for the next section. For now, just click Apply and then Close to save these settings.

Okay—that really was a lot of introduction to get to debugging. You may be wondering why the process is so complicated, but there simply are a lot of moving parts. Again, most of the steps to this point are one-time configuration settings. Once set, you'll never configure them again.

Exploring the Debugger

We're now finally ready to debug a CFML page (.cfm or .cfc file). In this section we'll learn how to set breakpoints (places where control should stop while debugging), start a debugging session, step through code, and observe (or change) variables, among other things.

To recap, for a CFML page to be debugged, three things are necessary (each discussed previously):

- The ColdFusion server on which the code is running must be configured to permit debugging.
- A developer must have ColdFusion Builder open with the debugger settings configured: a project created with server, RDS, and debugger configuration created for the server against which debugging will take place.
- The debug configuration for that project and server must be started (typically with a breakpoint set for the file in question, though Break on a CFML Runtime Exception can stop on a line even without a breakpoint being set.)

Switching to the Debugging Perspective

When you initially open ColdFusion Builder, you may notice that the top-right corner shows an icon labeled “ColdFusion.” This icon represents what ColdFusion Builder calls a perspective. A perspective defines a preconfigured layout of the screen, showing tabs and window panes that are appropriate for a given kind of development work.

In our case, we want to switch to what ColdFusion Builder calls the ColdFusion Debugging perspective. You can switch perspectives in a number of ways, including by choosing Window > Open Perspective > Other, or by clicking the Open Perspective icon shown to the left of the current perspective name in the top-right corner of the screen or by clicking the double-arrow icon to the right of that (if you’ve already opened another perspective). There are also some situations in which the debug perspective will open automatically, such as when the Break on Exception feature discussed previously is used.

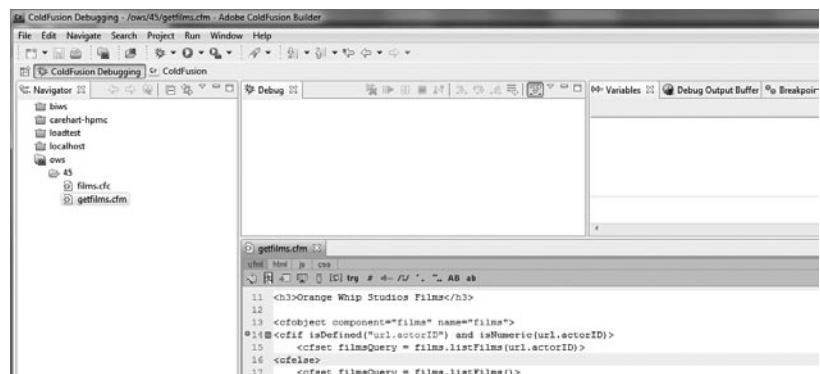
NOTE

You may find that your perspective names don’t appear in the top right of ColdFusion Builder, but instead appear in the top left, under the menu. If you want to switch the location between these choices, right-click the perspective name and choose Dock On and then select Top Left or Top Right.

You’ll notice that the interface changes (Figure 45.7) and you now see panes that are related to doing debugging. I’ll explain these panes and their use in the remainder of this chapter.

Figure 45.7

The ColdFusion Builder Debug perspective.



Opening a File

When you switch to the Debug perspective, the ColdFusion Builder Navigator remains displayed on the screen, so if you want to debug a page, you need to have opened the page from a project (again, you can debug only files in projects). Drill down in the project directory tree to find the file and double-click to open it. The file will appear in an editor like that shown in Figure 45.7.

Notice that Figure 45.7 shows numbers next to each line. These line numbers are very helpful, especially for debugging. To enable the display of line numbers, you can choose Window > Preferences > General > Editors > Text Editors and select the option Show Line Numbers, or right-click in the gutter (where the numbers are shown in the figure) and choose Show Line Numbers.

Setting a Breakpoint

With a file open, you can identify a line of code on which you want the debugger to stop when it reaches that line during execution of the request. This is called setting a breakpoint. You can right-click in the gutter, that area to the left of the line (where the line number appears, or the gray area left of that) and choose Toggle Breakpoint. You can also set a breakpoint by using the key combination Ctrl+Shift+B or by choosing using Run > Toggle Breakpoint. Figure 45.7 shows that I had already set a breakpoint on line 14.

When you do that, a blue dot will appear to the left of the line (and line number), depicting that the breakpoint has been set. Note that it only makes sense to set a breakpoint on a line with CFML (tags or expressions), though ColdFusion Builder won't stop you from trying.

The breakpoint(s) you set will also be depicted in the breakpoints tab (click the tab shown at the top right of Figure 45.7), where they are displayed with their filename and line number.

Note that breakpoints remain enabled across editing sessions. In other words, if you close ColdFusion Builder and reopen it, the breakpoints you set previously will remain enabled. That said, breakpoints are specific to your editor, so if another developer opens the file, they will not see breakpoints you have set.

This may lead to a question: If breakpoints remain enabled across editing sessions, does that mean that a request will always be intercepted once a breakpoint is set? Not quite. Setting a breakpoint is the first step. The far more important step is to start a debugging session.

Starting a Debugging Session

Even with breakpoints set, the debugger doesn't do anything until you begin a debugging session. To begin a session, open a file within a project and choose Run > Debug or press the F11 keyboard shortcut. This will cause ColdFusion Builder to start a debugging session for the project associated with the file that's open.

There are still other ways to start debugging: Choose Run > Debug As > ColdFusion Application, or click the bug icon just above the file Navigator, or right-click in the editor and choose Debug As > ColdFusion Application.

Technically, you don't need to open the Debug perspective before using any of the options to start a debugging session. If the perspective isn't open, ColdFusion Builder will prompt you to confirm the perspective switch, asking if you want to switch to the debugging perspective, with a Remember My Decision option for future debugging sessions.

Assuming that all is configured properly, this should initiate a debugging session between your editor and the ColdFusion server.

WARNING

If you do not have your cursor focus within the editor of a currently opened file, then using any of the options mentioned here to start the debugger will restart whatever was the last-executed debugger configuration. This could be very confusing as it may not be the debug configuration for the files you intend to debug. Be sure to note the debug configuration that is started, as shown in the debugging window discussed next in the section "Understanding the Debug Status Window."

If you're debugging a local server, it will also cause ColdFusion Builder to open your default browser to start executing the URL for the file you selected, again assuming that you selected an open file and the cursor focus is within that file at the time you started the debugger.

Even if a browser window isn't opened automatically, you can certainly still open a browser: any browser, whether an external one or one of the available internal browser tabs within ColdFusion Builder. This topic is discussed further in the upcoming section, "Browsing a Page to Be Debugged."

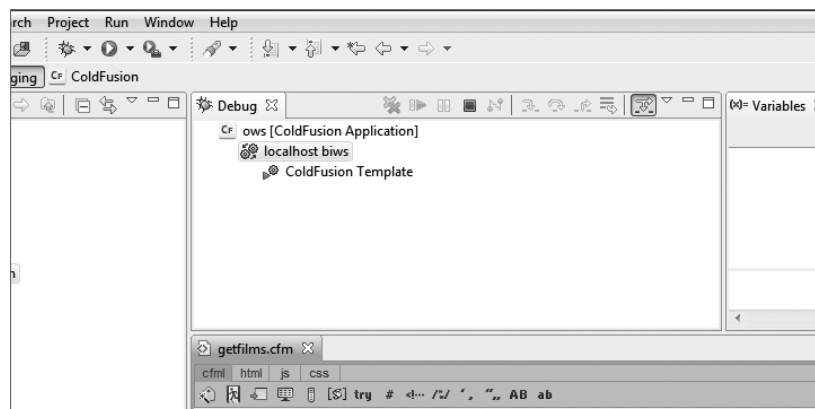
Understanding the Debug Status Window

After a debugging session is started, the debugger is now ready for you (or someone) to execute a request to be intercepted and debugged.

Figure 45.8 shows the kind of information that might be displayed in the debug window, which shows the debugger ready to accept requests for files in the selected debugging configuration. (I've selected to start my debugging configuration named *ows*. Technically, the value shown on the first line, with the CF icon to its left, is the name of the selected debugging configuration, and the next line below it is the name of the server used for the project in that configuration.)

Figure 45.8

A successfully started ColdFusion Builder debugging session.



The approach just described starts a debugging session for the specific file that was open. You can also start a debugging session for any existing debugging configuration through a couple of other approaches in ColdFusion Builder.

First, you can choose Run > Debug Configuration, which will open the same interface that we saw in Figure 45.4, where we defined debug configurations. You can select a configuration and click the Debug button in the lower-right corner.

Second, you can click the icon (just above the file navigator) that looks like a bug. If you've not previously used any of your debugging configurations, you'll be presented with the same dialog box as shown earlier to select one. And when you've selected more than one debugging session, you will be able to select one of them quickly by clicking the down arrow icon just to the left of the bug icon, which will display a list of previously selected debugging configurations. Choosing one will start that debugging configuration. Note that the menu also offers a Debug option, which also opens the interface shown in Figure 45.4.

When Starting a Debug Session Fails

Your attempt to start a debugging session can fail for a number of reasons. What are some of the things that could go wrong?

- **ColdFusion isn't started:** You'll get a pop-up saying "Connection timed out: connect" followed by another starting with "Unable to connect to the RDS server" and continuing with another message, "Connection timed out: connect." Start the ColdFusion server and try again.
- **RDS wasn't enabled in the ColdFusion server:** You'll get a pop-up starting with "Error Executing RDS command. Status code: 404" and then another reporting "Unable to connect to the RDS server." Again, the ColdFusion Builder debugger requires an RDS connection, so the server being debugged must have RDS enabled. You can learn how to enable it at http://www.adobe.com/go/tn_17276. After making the change, restart the server and try again.
- **The RDS authentication information provided was invalid:** You'll get a pop-up starting with "Unable to connect to the RDS server" and continuing with the error message "Unable to authenticate using the current username and password information." Correct the RDS connection information and try again. The RDS connection information can be edited by modifying the server definition, which you can do by right-clicking the server in the Servers view (Window > Show View > Servers). Be sure to select the server defined to be used for the project whose file you're trying to debug.
- **Allow Line Debugging wasn't enabled in the ColdFusion Administrator:** You'll get a pop-up starting with "Unable to connect to the RDS server" and continuing with the error message "Error Executing RDS command. Status code: 404." Return to the ColdFusion Administrator, as discussed in the section "Enable Debugger Settings," and ensure that Allow Line Debugging is checked. You must restart ColdFusion for this change to take effect. Try starting the debug session again.

- **The debugger port is not opened on the server:** You'll get a pop-up starting with "Unable to connect to the RDS server" and continuing with the message "Connection timed out: connect." You must open a hole in your firewall exposing the debugger port for access from the machine on which you're running the debugger, but see the important discussion of how ColdFusion uses a random debugger port and how you can resolve this in the earlier section, "Enable Debugger Settings." Then try starting the debug session again.

Browsing a Page to Be Debugged

With a debugging session enabled, it's now finally time to run your page to experience debugging. This step can actually be as confusing for some developers as any of the configuration features discussed to this point. While most debuggers may have some configuration that must be performed, as discussed earlier, with most debuggers the next step is simply to run the desired program in the IDE or in some sort of special debugging window. That's not necessary with the ColdFusion Builder debugger.

Instead, you simply run the CFML page request just as you normally would, whether from a browser, or via a Flex or Ajax client, or as a Web service call from some other environment, and so on. There's no special debugging window in which you must run your request. (Internal browsers provided by ColdFusion Builder are available; you don't have to use those, though you can.)

TIP

Since the debugger intercepts any request for a CFML page, it can even intercept some requests that have no browser or client at all, such as requests if you debugged the `onSessionEnd` method of `Application.cfc` (which would be executed when a session ended), or requests in a ColdFusion event gateway like the `DirectoryWatcher` (which would be executed when there was a change in the directory it was observing).

If you have a ColdFusion Builder debugging session enabled against a server, with a breakpoint set in a file, then when that CFML code is executed, whether by you or anyone else, the debugger will stop on that line of code.

You may be surprised that the debugger intercepts any request for a CFML page, regardless of whether you request it or someone else does. This means that in an environment with multiple users making requests against a given server that you're debugging, you could intercept someone else's request. That could be either desirable or unexpected. Just be aware.

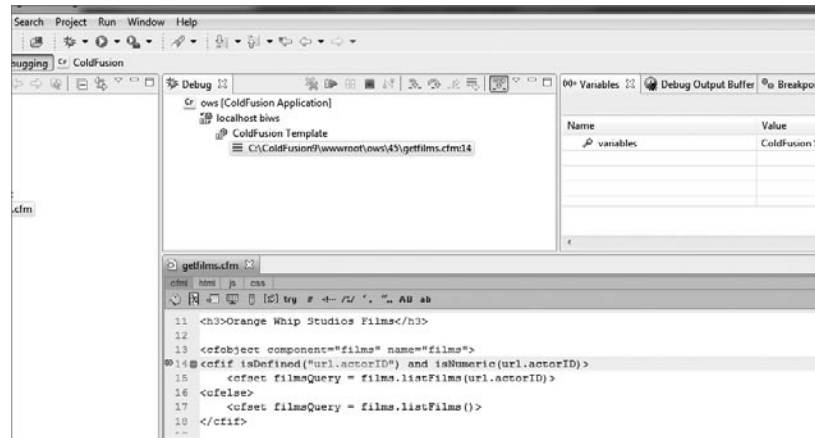
What You, the Developer, Will See

Assume that you or someone else has issued a request and it's been intercepted by the debugger at a desired breakpoint; Figure 45.9 shows how the ColdFusion Builder environment will reflect that you're stopped at that breakpoint.

First, notice that the Debug window at the top now reflects more information than we saw in Figure 45.8. It shows the complete path to the file being debugged and the line number at which execution has stopped. This complete path can help if you ever wonder what file you're viewing or debugging.

Figure 45.9

Execution stopped at a breakpoint.



Note also that in this case we see other lines there, which reflect what some may call the stack trace—or how the code we’re stopped on was called by other code. For instance, if we were stopped in a CFC, include file, or custom tag, it would show the file (and line number) of the code that called this file.

Second, in the code window, you can see that where Figure 45.7 had showed only a blue dot next to the line on which a breakpoint had been set, now we see also a blue arrow overlaid atop it. (This may be too small to observe in the screenshot.) The arrow is a current instruction pointer, which reflects where the debugger has stopped execution. It shows the line that is about to execute. We’ll learn about stepping through that and other code shortly.

What if the breakpoint doesn’t fire? You may find that when the request is executed, the breakpoint does not fire. The first step is to ensure that all previous configurations are correct (you’ve opened a file in a project and properly defined a server and RDS server for that project) and that you’ve set a breakpoint and started a debug session for the project in which the desired file is located.

Even if all these settings are correct, occasionally the debugger doesn’t pick up the breakpoint. You may even see an icon where the blue dot should have appeared (indicating that a breakpoint had been set on a line) showing instead a small question mark over the dot. In either case, try making some change to the file (while in the editor) and save it. You may even want to terminate and restart the debug session, as discussed later, in the section “Stopping the Debugger.”

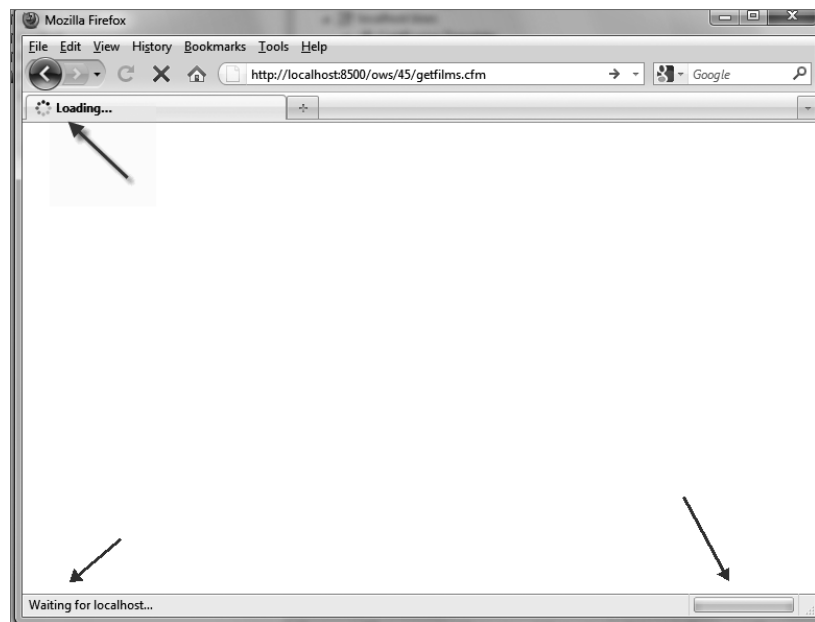
What the User Sees

While we sit here looking at the debugger, observing the fact that the debugger has intercepted the request, allowing us to look at the information above (and more to be discussed soon), you may—and should—wonder what things look like to the user making the request. If it’s a regular browser user, what they’ll see will appear as if their request is hung, as if the server is not responding, and in fact it’s not. We’ve held up the request’s processing so there is not yet a response for the user to see.

Just as when a user's request is waiting for any response from the server, they will see various indicators that they are waiting, which will vary depending on the browser being used. See Figure 45.10 for an example in Firefox, where I've highlighted with arrows the three ways that it shows that a page is waiting to complete loading. This is the page that triggered the debugging session in Figure 45.7.

Figure 45.10

The user waits for page execution to finish.



You may wonder, if the page had started to generate output before the breakpoint, whether that output would at least be sent to the browser. It is *not*. As with a normal ColdFusion request, the page output is buffered internally and sent only when the page completes (unless a CFFLUSH causes it to be sent sooner or an internal buffer size is reached). If you want to see what partial page output has been created in the output buffer, such a feature is available in the debugger, discussed later in this chapter, in the section “Observing the Debug Output Buffer.”

Of course, this discussion has assumed we were referring to a normal browser request calling the page. If the page had been an Ajax or Flex client request, those two would show the request waiting, typically using some sort of prompt to the user (which might flash by in a split second under normal circumstances).

Beware Page Timeouts

It's possible that you could leave the browser waiting so long for a response that it would consider the page abandoned and could show a message indicating that it stopped waiting.

We mentioned previously how ColdFusion itself could report that the request exceeded a configured timeout. You would receive a pop-up message in the debugger reporting, “coldfusion.runtime.RequestTimeoutException: The request has exceeded the allowable time limit.” In

this case, you should configure ColdFusion to allow CFML pages to run longer than normal, by adjusting the ColdFusion Administrator's Timeout Requests feature, discussed earlier in the section "Increase Maximum Request Timeouts."

If for some reason you can't get the Administrator setting changed, there's yet another option to prevent timeouts, at least on a page-by-page basis. When the normal execution of a page may be legitimately expected to exceed the server timeout, you can use the `CFSETTING` tag and its `RequestTimeout` attribute to specify a number of seconds that the page should be permitted to run before timing out.

Stepping Through Code

Now let's continue with the use of the debugger.

We've stopped the debugger on the line of code set as a breakpoint. There are many things we can do at the point, as we'll see in upcoming sections, but often the sole reason for using the debugger is to observe the flow of execution through the code (and among the files that the code may call upon). There are many beneficial reasons to step through the code.

Perhaps you're trying to understand why certain code is (or is not) being executed. Or maybe this is new code that you've been asked to maintain (or debug), and you want to use the debugger to become more familiar with its operation (versus statically reading the source code by hand).

A powerful feature of the debugger is that it opens files that are called upon as you step into lines that point to other files (such as includes or CFC methods). Thus it's also an excellent tool for learning where your code is going and what files are being used. Recall from Figure 45.8 that it shows you the full path to the file being debugged.

Finally, you may be new to ColdFusion in general, and the debugger is an excellent tool to help you become familiar with how CFML works.

You can step through the code on a line-by-line basis, or you can run until a next breakpoint is reached. And if you've stepped into some other file, you can have the debugger complete execution of code in that file and stop at the next line in the calling file after the line which called the other file.

Stepping Line by Line: Step Over, Step Into

The most common situation is that you want to have the debugger just proceed to execute the next line in the flow of execution. Here you have two choices.

First, you can choose to step over the line, which means simply that you want to execute the code and proceed to the next line (in the current file) that would have followed it (unless there are no more lines to execute, in which case the request will complete). This can be accomplished by choosing `Run > Step Over`, pressing the F6 key, or click the Step Over icon listed among the icons atop the debug window.

Your second option is to step into the next line of code (choose Run > Step Into, or press the F5 key, or click the Step Into icon). This option makes sense if the line of code about to be executed would call another file (such as a `CFINCLUDE`, custom tag call, or CFC method invocation) and you want to use the debugger to open that file and stop on the first line of CFML code in the newly opened file.

Let's return to our code example and follow as it steps into the call to a CFC method.

NOTE

If the tag you're interested in stepping into (which would open a file) happens to have an opening and closing tag (such as a `CFINCLUDE`) with intervening CFML tags within it, the Step Into process won't happen until you reach the closing tag.

Notice that you don't need to tell the debugger where the file is. That's one of the very powerful features of the debugger: it figures out the file that ColdFusion would use. This is a great feature when you have any doubt about what file is being opened. (The file may not always be obvious to you due to ColdFusion mappings and other mechanisms by which ColdFusion may find and open a named file; if the file can't be found, you may need to take an extra step, discussed at the end of this section.)

TIP

Here's a bonus ColdFusion Builder feature. You don't need to use the debugger to have it open the file associated with a tag or function that would point to another file. ColdFusion Builder has a handy feature that lets you hold down the Ctrl key (Windows) or Mac key (Macs) while using the mouse to hover over such a filename. An underline will appear, indicating that the mouse is pointing to a file, and clicking will open that file. This feature works with both CFML and HTML.

What if you don't want to follow the flow of execution into the file that would be opened (by the line of code about to be executed)? You don't have to open it. You can instead use the Step Over option, and the code in the current file will be executed (just as during normal execution of the page) and the current instruction pointer will be set at the next line in this page (unless there are no more lines to execute, in which case the process will complete, or if you've stepped into a file, the debugger will return to the calling file).

If you do choose to use Step Into on a tag that opens a new file, note that once you're inside that new file, you have the same options as discussed earlier, so you can use Step Over and Step Into, as well as a new option, Step Return, discussed next.

CAUTION

Be careful not to use Step Into when you don't need to (don't use it on tags, functions, or expressions that would not otherwise have any reason to open a new file). For one thing, it can cause the request to take longer than it would with a Step Over.

More important, on some tags and functions, using Step Into will actually try to open an underlying file unexpectedly. Some CFML tags, such as `CFDUMP`, `CFSAVECONTENT`, and a few more, as well as some functions, such as `writedump()`, are actually executed as CFML pages by ColdFusion. ColdFusion Builder will try to open the respective file, and you'll get an error in ColdFusion Builder reporting "source not found" and offering an Edit Source Lookup Path button. But the path used, such as `E:\cf9_final\cfusion\wwwroot\WEB-INF\cftags\dump.cfm`, does not really exist on your ColdFusion server. The path corresponds to a location where the file existed when ColdFusion itself was compiled.

When considering the idea of stepping into files, it's important to remember that for the debugger to locate a file, the debugger must be able to find the file, and it expects to find it in the project.

This creates a dilemma that often stumps new debugger users: What if the file being opened is not defined in the project (which perhaps is pointing to a single Web site's Web root)? What if the file is instead in some directory that's shared among multiple Web sites? An example may be a CFC or custom tag stored either in the default ColdFusion custom tags directory or in a directory pointed to by a custom tag mapping in the ColdFusion Administrator. While ColdFusion can certainly find that file when it needs to, how can the debugger be told?

The magic is in the definition of the project (and for remote servers, the debug mappings mentioned previously). The solution is to add a link folder definition in the project's properties. This can be done in the project's properties (right-click a project name and choose Properties); then in the list of options, select ColdFusion Project and under Additional Resource, select Add to point to the location. For more information, see the ColdFusion help topic "Link to resources outside the workspace."

Step Return and Resume

If you do step into a new file, and you decide you no longer want to step line by line through the code, you don't need to laboriously make your way through the remaining lines of code. That's what Step Return is for (Run > Step Return, or F7, or the appropriate icon). That option completes execution of the file that's been stepped into (unless there's another breakpoint later in the file) and returns control to the line of code that follows whatever tag or function called the file. All the remaining code in the file is executed, of course. You just don't step through it.

Similarly, if at any point while debugging a request you decide that you no longer want to step through the code (whether you've stepped into a file or not), you can have the debugger proceed to execute the entire remainder of the request using Resume (Run > Resume, or F8, or using the appropriate icon). Unless there's another breakpoint in the flow of execution, the debugger will finish and the completed page will be shown to the browser (or whatever client made the request for the CFML page).

The Terminate feature is discussed in the last section of this chapter, on stopping the debugger.

Observing Variables and Expressions

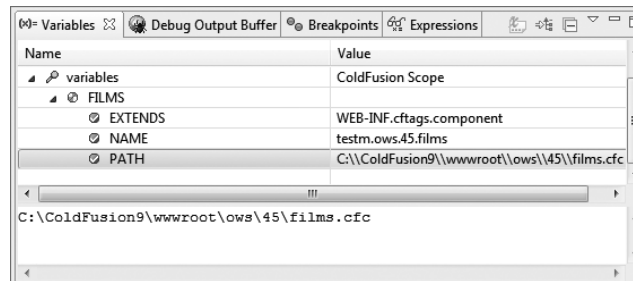
For some developers, just being able to have a granular, line-by-line depiction of the flow of execution through their pages is miraculous enough. But there's more. While you're at a breakpoint, you can also observe the value of any variable, in any of ColdFusion's scopes. This includes all the usual scopes (FORM, URL, SESSION, REQUEST, and so on), as well as things like the ARGUMENTS scope when in a function, the THIS and LOCAL scope as created by a CFC, and so on.

The Variables Pane

The first place to look is the Variables pane, which appears to the left of the Breakpoints pane as shown in Figure 45.9. If you select it, you can traverse the tree of available scopes, expanding them to see the name and value of any variables in the selected scope. See Figure 45.11, where I've selected the VARIABLES scope. Since this code just created a CFC instance that was stored by default in the VARIABLES scope (as films), I've expanded that variable, and you can see the various internal variables that ColdFusion tracks (as structure keys) for a CFC instance, such as the actual path where it was found.

Figure 45.11

Display of VARIABLES scope.

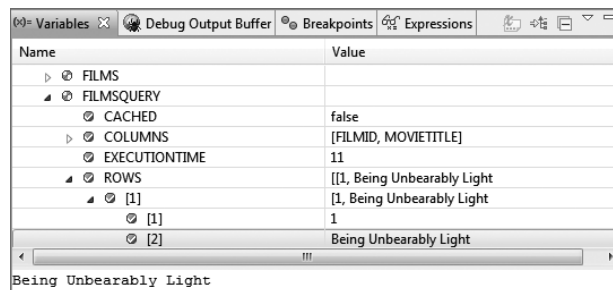


The depiction of variables in this pane is of course not limited to the simple variable values shown above. If the code had executed a query, we could expand the query to see all its related variables (recordcount, columnlist, etc., and even each row and column in the result set). If we stepped through the next line of code, which will call the CFC that returns a query, we can view the query result set. Figure 45.12 shows the query result set expanded, showing the first row and its two column values (and the column names are shown in the columns variable also displayed for the query).

Here are a couple of tips that could make working with the Variables pane more effective. First, if the value of the variable you wish to observe is too long to be displayed in the Value column provided, while you could select the column divider to make it wider, a simpler solution is that if you select the variable, you'll notice (as shown in Figure 45.12) that the selected value is also displayed in a Details window at the bottom of the Variables pane. This can be more easily scrolled to view the full value.

Figure 45.12

Display of a query.



Also, while in the Variables pane, you can scroll up and down in this list to see all the available variables in the selected scope. But note that you can also use an available Find feature, which can search through all the scopes to find a variable of a given name. Right-click the variables pane and choose Find, or left-click on it and use Ctrl-F.

Remember, too, that the list of scopes shown is limited to a default subset shown in the Debug Settings preferences (or those you chose during setup), as discussed previously in the section of the same name. Recall from that discussion that you should be careful about adding too many scopes, as that can slow the execution of the debugger. Another option is that you can instead ask the debugger to show you some specific variable using the Expressions pane, discussed next.

The Expressions Pane

You may want to observe a variable value to watch how it changes over lines of code as you step through it, or perhaps over multiple requests for the page. While you could traverse the Variables pane to find and display your desired variable each time, you could instead use the available Expressions pane. This has the benefit that you choose to name what variable you want to watch (including one in a scope that is not being displayed in the Variables pane because of Debug Settings preferences).

To add a new expression to the pane, right-click it, choose Add Watch Expression, and then provide the name of the variable. You don't need to surround the variable in pound signs (though it's okay if you do). The variables you name will be displayed in the Expressions pane (in the order in which you add any). When you're stopped at a breakpoint, its value will be shown—assuming the variable is defined at that point in the code. If it's not, the variable will be shown in red. (If you ever find that a variable which should be defined is not displaying, right-click it and choose Reevaluate Watch Expression. You can also right-click an expression and choose Edit Watch Expression.)

Note that you're not limited to naming variables with simple values. You can even name an entire scope and all its values will be displayed. You can also create a new Watch Expression by right-clicking on a variable in the Variables pane and choosing Watch.

Another benefit of the Expressions pane is that, like breakpoints, the items you create here remain across debugging sessions and editor restarts. And like the variables pane, you can use the Find command to search among them.

Changing Variables on the Fly

Here's a feature that will amaze some readers. Did you know that you can change the value of a variable on the fly, while the program is being debugged? Yes, you can. While at a breakpoint, you can right-click on a variable name in either the Variables or Expressions pane and choose Change Value. It's really that simple. The value will be changed for the remainder of the execution of the request (or until you or some other code changes it again).

Observing the Debug Output Buffer

Finally, the last feature to discuss is the Debug Output Buffer pane. I mentioned it previously as a way that you can view the generated output of the CFML code (whether you're generating HTML, XML, JSON, or whatever else you may be outputting as the page response).

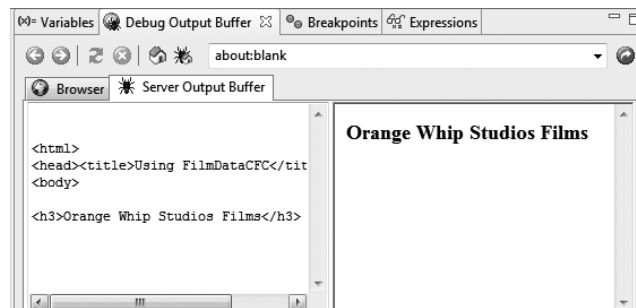
The Debug Output Buffer pane is what enables that processing. The pane appears to the right of the Variables pane in the top right of the Debugging perspective. Within that pane, select the Server Output Buffer tab (I'll discuss the Browser tab in a moment).

Server Output Buffer Tab

The Debug Output Buffer tab consists of two panes (Figure 45.13).

Figure 45.13

Server output buffer pane.



The left pane shows the raw generated browser code (HTML, in this case) that the CFML template has generated to this point. You could scroll down and see that it's not a complete page, since the request is in the middle of generating the content. Further testament of that fact is in the right pane, which shows an attempt to render that HTML in a built-in browser.

WARNING

If you open the Debug Output Buffer pane after having already started to debug a request, you may not see it displaying any of the request's output. Try completing and restarting the request to see output presented in this pane.

As you step through the code, you'll be able to observe the building of the page as it might appear in the real browser. This is just a simple rendering of the HTML in a temporary file created by the editor. It's not really executing in the context of the URL of the server, so some things (like the image shown) may not render correctly.

Also, don't be misled by the URL displayed atop the pane. That's not the URL of the page being debugged (unless you typed a URL there). That address bar is really there for the sake of the other window in this pane, the Browser tab, discussed next.

And to be clear, the server output buffer is populated regardless of the browser used to launch the request being debugged (again, it could be launched by a user other than you).

Browser Tab

I had mentioned previously that to trigger the debugger you could make a request from any browser, and I'd mentioned that there were some internal browsers available within ColdFusion Builder. This is one of them. If you wanted to, you could make your page requests using this address bar and the result will be shown in the Browser tab of this pane, not in the Debug Output Buffer pane. Indeed, it wouldn't really make sense to use that address bar while you're viewing the result of a page being debugged.)

It's a real browser. In fact, the URL that's shown (unless you entered one yourself) is the very value of the Home Page URL that was mentioned briefly in the discussion of Debug Settings, Figure 45.4. (So that's what that's for!) If you click the home page icon atop that Browser pane, that URL will be loaded.

That said, I should clarify that some might not care for this browser tab because the window is so small. But here's a tip: As with all the panes discussed here, you can resize this one and even detach it so that it floats as a window over top the ColdFusion Builder interface.

Stopping the Debugger

Finally, when you're finished debugging pages, you may wonder how you stop debugging. There are several ways, each with slightly different purposes.

Terminating Debugging Within the Debugger

First, if you close ColdFusion Builder, that will terminate your debugging session, which means fortunately that you can't leave it running by mistake (at least not by closing it without remembering to stop it).

You can also stop the debugging session while you remain in the editor. There are a few ways. You can use the Terminate button. That stops the debugging session and lets the request run to completion. You can also choose Run > Terminate.

You can also right-click in the Debug pane (where the file name and line number of what's being debugged is shown), and there choose Terminate. That will stop the debug session and leave an indication in the Debug pane that you had used that debug configuration. That way you could right-click and choose Relaunch as another way to start a session. You can also right-click and choose Terminate and Relaunch if, for instance, you made changes in the debug configuration. Finally, you'll see another option, Terminate and Remove, that will remove the debugging session from the debug pane.

Forcing Termination from the Server

I mentioned above that closing the editor would terminate any debugging sessions. But what if you instead mistakenly left the Editor open and a debug session running while you left for lunch—or for the day? This could be a real problem. Users might run a request for a page you're debugging,

and at least the first of those would have their request hang waiting for a debugging activity that might not happen for hours.

Here's where it's important to note that there's one final way to stop debugging that can be done from the server rather than from the editor. Recall the discussion of the Admin console page for Debugging & Logging > Debugger Settings. I mentioned the available Start Debugger Server button, which once debugging begins becomes a Stop Debugger button, and an associated Restart Debugger Button appears.

This is the situation where you would use those buttons. You could either use Stop Debugger (and then Start Debugger) or just use Restart Debugger. That will terminate all current debugging sessions, thus freeing up the requests that were pending waiting for a debugging session to complete. Even if you'd used Stop Debugger, the first request for a debugging session from a developer would restart the debugger server anyway.

Think of the Stop or Restart as like a forced termination of debugging from the server. All developers who were performing debugging will have their debug sessions terminated, so it's a bit brute force, but they can easily restart them. The good news is that they do need to manually restart them, so the person who left their editor (and debugger) running when they left will no longer be the cause for intercepted requests.

NOTE

The Restart option also can be used if, for any reason, ColdFusion Builder crashes during a debugging session. In that case, on restarting and launching a new debug session, you may get a breakpoint error reporting "Another session has already set a breakpoint there." Clicking the Restart button should resolve the problem.

If you wonder how to stop all debugging from happening any more on the server, clearly that's not what the Stop Debugger Server does (since the next request for a debug session will start it up). Instead, turn off (uncheck) the option on that Admin page for Allow Line Debugging. That takes effect immediately, and ColdFusion would no longer permit debugging requests against that server.

Other Features

While we've covered a broad range of features and troubleshooting topics, there are still more features that cannot be elaborated upon due to space constraints.

For instance, you can copy variables and expressions by right-clicking them in the Variables or Expressions pane. You can also export and import breakpoints (to share them with other developers) by right-clicking in the Breakpoints pane.

There are also options to temporarily disable one or more breakpoints or to skip all breakpoints (both via an icon on the Breakpoint pane's toolbar, or via Run > Skip All Breakpoints.)

That said, there are also some features on the Run menu that do not work in ColdFusion Builder (but instead are remnants of the underlying Eclipse Java debugger), such as method breakpoints, watchpoints, and step filters.

Also, when you're done debugging and want to return the display to the more typical development (rather than debugging) perspective, you can either choose **Window > Open Perspective > Other > ColdFusion** or click the icon for the ColdFusion perspective, which again (as discussed earlier) appears either in the top left or top right of the interface.

Finally, be aware that the ColdFusion 9 manual, *Developing CFML Applications*, has a section "Using the ColdFusion Debugger," but there are some differences between what this section discusses and what you'll experience in ColdFusion Builder. This manual was created prior to the release of ColdFusion Builder and discussed an older plug-in approach to debugging ColdFusion.

It's great to see how in so many ways the Adobe Engineering team has thought ahead to help not only provide a debugger but one that addresses some common challenges that CFML developers would face. For many, the biggest challenge of using a debugger is simply remembering that it's there. I hope this discussion has motivated you to consider using it.