

CHAPTER 30

Extending ColdFusion with CFX

IN THIS CHAPTER

- What Are CFX Tags? 1
- Introducing the CFX API 3
- Writing CFX Tags with Java 8
- Writing CFX Tags with Visual C++ 24
- Generating Debug Output 38
- Returning Structures from CFX Tags 39
- Using CFX Tags Safely: Locking and Thread Safety 43
- Using CFX Tags with Previous Versions of ColdFusion 44

As you have learned in previous chapters, you have the capability to write your own custom tags, user defined functions, and components—all using ColdFusion’s native language, CFML. It’s possible to extend ColdFusion’s capabilities even farther with CFX tags. The main difference is that you don’t use ColdFusion’s markup language to create the tag; rather, you use an external programming language (Java or C++) to create the tags.

What Are CFX Tags?

You have already become familiar with CFML custom tags, which always start with `<cf_>`, the code for which appears in a similarly named `.cfm` file. In this chapter, you’ll become familiar with tags that start with `<cfx_>`. These tags are compiled to a Dynamic Link Library (`.dll`) or a Java Class (`.class`) file. Table 30.1 describes the differences between `<cf_>` and `<cfx_>` tags.

Table 30.1 Quick Comparison: CFML Custom Tags Versus CFX Tags

CFML CUSTOM TAGS	CFX TAGS
Start with <code>cf_</code> .	Start with <code>cfx_</code> .
Written using normal ColdFusion tags and functions.	Written in C++ or Java.
Actual code is in a <code>.cfm</code> file.	Actual code is in a C++ <code>.dll</code> file or a Java <code>.class</code> file.
Can only do things that CFML code can do.	Can do just about anything that the language (Java or C++) allows.
Don’t need to be registered in ColdFusion Administrator.	Must be registered in the ColdFusion Administrator, in a step much like creating a data source or Verity collection.

When To Write (and Avoid Writing) a CFX Tag

Given the choice between creating a CFX tag or creating a CFML-based extension such as a custom tag, UDF, or CFC, it is generally best to consider one of the CFML-based alternatives first. Why? Mainly because CFML is so simple, and generally provides a rich enough feature set to get most tasks done with ease. Whenever possible, I recommend that you consider creating a CFML-based extension first.

That said, there are plenty of situations where creating a CFX tag makes lots of sense. In general, most CFX tags are created for one of the following reasons:

- To take advantage of a third-party C++ or Java API that makes something possible that isn't otherwise possible in ColdFusion. Perhaps the API knows how to connect to some kind of legacy mainframe database system, or knows how to create a special type of image file. In this situation, the CFX tag can be thought of as wrapper around the API.
- To take advantage of legacy code, open source code, or other snippets that have already been written in Java or C++. If you're working on a complex project, and you already have Java or C++ code available that can provide part of the functionality, it may make sense to just turn that code into a CFX tag, even if it would be possible to port the logic to CFML. If it's faster and easier for you and your project, go for it!
- To take advantage of faster processing. In general, your regular ColdFusion pages are compiled into Java classes that perform quite well. However, there will always be niche situations where you could write custom Java or C++ code that runs more efficiently.
- To circumvent method-selection problems or other errors when using Java classes (or Beans). Most Java classes can be used via `<cfobject>` or `CreateObject()` as discussed in Chapter 29, "Extending ColdFusion with Java". However, some Java methods and constructors are overloaded in such a way that ColdFusion can't determine which form to use at runtime, even if you use `JavaCast()`. In such a situation, you can create a CFX tag that acts as a sort of "poor man's proxy" between ColdFusion and the Java class.

Choosing Between C++ and Java

You can program a CFX tag in either Java or C++. Much of the time, the decision will have been made for you, because whatever third-party API or other existing code you are planning to use will force you to use one or the other.

The most obvious advantage to programming the CFX in Java is that it will be able to run on a ColdFusion server using any operating system (OS). Tags created with C++, however, will need to be recompiled for each OS you want to support. Of course, cross-platform support doesn't mean much if the nature of your project is such that the tag will be bound to one OS or the other anyway, so the importance of this advantage may or may not mean a whole lot to you. For instance, if you want the CFX tag to call native Windows APIs, then a C++ tag is the obvious choice; you'll probably only have to compile it once for any Win32 platform.

Another reason to use C++ is the fact that it is generally thought to be faster at extremely CPU-intensive tasks. However, recent advances in Java Virtual Machine technology have made Java code more likely than ever to perform nearly as well as comparable C++ code. Also, the ColdFusion server can invoke Java CFXs directly, but must go through an additional layer (the Java Native Interface, or JNI) when invoking a C++ tag. In my opinion, sheer performance issues should only guide you toward C++ in extreme situations.

All things considered, I recommend that you go with Java if you really have a choice between the two. The code will probably be faster, easier to write, and will almost certainly be easier to debug. Even if you don't anticipate using to use the tag on multiple platforms now, it will still be nice to know that your choice of Java keeps that option open for the future.

Introducing the CFX API

Regardless of the language you use to create a CFX tag, you need a way to pass information back and forth between the tag code and the ColdFusion page that is using the tag. You also need ways to safely throw exceptions (errors), create query objects, and include output in the current Web page. Macromedia defines a set of objects and functions that provides all this functionality and more. The objects and functions are known collectively as the *CFX API*.

The following sections will introduce you to the various functions included in the CFX API, organized by subject. Within each section, the Java syntax will be shown in one table, followed by the C++ syntax in a second table. You will notice that the Java and C++ versions of the API are very similar. For the most part, there is a one-to-one relationship between the methods provided by both versions of the API. This is great because once you design a few tags in Java, you will be well on your way to understanding what's possible with C++, and vice versa.

Working with the Tag's Attributes

Like native ColdFusion tags or CFML custom tags, your CFX tags can accept any number of attributes. You can easily add code to your tag so that some of its attributes are required while others remain optional. The CFX API defines several functions that a tag can use to accept and validate attributes from the calling ColdFusion page. Table 30.2 shows the Java version of these functions, and Table 30.3 shows the C++ version.

Table 30.2 Java Methods for Working with Attributes

METHOD	DESCRIPTION
<code>request.attributeExists(name)</code>	Determines whether the specified attribute was actually passed to the CFX tag. Returns a boolean value.
<code>request.getAttribute(name)</code>	Returns the value of the specified attribute, as it was passed to the CFX tag. Note that all attribute values are received by the CFX as <code>String</code> values.
<code>request.getAttributeList(name)</code>	Returns the names of all attributes that were actually passed to the CFX tag. The names are returned as an array of strings.

Table 30.3 C++ Methods for Working with Attributes

METHOD	DESCRIPTION
<code>pRequest->AttributeExists(lpszName)</code>	Determines whether the specified attribute was actually passed to the CFX tag. Returns a <code>BOOL</code> .
<code>pRequest->GetAttribute(lpszName)</code>	Returns the value of the specified attribute, as it was passed to the CFX tag. Note that all attribute values are received by the CFX as <code>LPCSTR</code> string values.
<code>pRequest->GetAttributeList()</code>	Returns the names of all attributes that were actually passed to the CFX tag. The names are returned as a <code>CFXStringSet</code> , which is similar to an array of strings.

Returning Variables and Text

Your CFX tag has the ability to include dynamically generated content in the ColdFusion page that is calling the tag. It can also return variables to the calling page. Table 30.4 shows the Java methods for these tasks, and Table 30.5 shows the C++ version.

Table 30.4 Java Methods for Returning Variables and Text

METHOD	DESCRIPTION
<code>request.setVariable(name, value)</code>	Returns a variable to ColdFusion with the specified name and <code>value</code> . Both name and <code>value</code> must be specified as strings.
<code>request.write(text)</code>	Includes <code>text</code> in the current page, just as if the text was generated by a <code><cfoutput></code> block. The <code>text</code> must be specified as a string.

Table 30.5 C++ Methods for Returning Variables and Text

METHOD	DESCRIPTION
<code>pRequest->SetVariable(name, value)</code>	Returns a variable to ColdFusion with the specified name and <code>value</code> . Both must be specified as <code>LPCSTR</code> compatible values.
<code>pRequest->Write(text)</code>	Includes <code>text</code> in the current page, just as if the text was generated by a <code><cfoutput></code> block. The <code>text</code> must be specified as a <code>LPCSTR</code> compatible value.

Working with Queries

Any CFX tag may return queries to the ColdFusion page that is calling the tag. The queries can then be used in ColdFusion code, just like the results of a `<cfquery>` or any other tag that returns a recordset. For instance, the queries can be used as the `query` parameter of a `<cfloop>` or `<cfoutput>` tag; they can even be re-queried or joined with other recordsets using in-memory-queries (where you set `dbtype="query"` in a separate `<cfquery>` tag; also known as “query-of-queries”). Table 30.6 shows the Java methods for working with queries, and Table 30.7 shows the C++ versions.

NOTE

Any CFX tag can return any number of queries to the calling template, but it can only access the data in one query that already exists in the calling template. To put it another way, multiple queries can be passed out from the CFX, but only one can be passed in.

Table 30.6 Java Methods for Working with Queries

METHOD	DESCRIPTION
<code>response.addQuery(name, columns)</code>	Creates a new query, which will be available in the calling ColdFusion page when the CFX tag finishes executing. Specify the name of the query as a string, and specify its <code>columns</code> as an array of strings.
<code>query.addRow()</code>	Adds a row to the query. You can then fill the individual cells (columns) of the new row using <code>setData()</code> . Returns the row number of the new row, as an <code>int</code> .
<code>query.setData(row, col, value)</code>	Places the <code>value</code> (which must be a string) into the query at the row and column position you specify. Rows and columns are both numbered beginning with 1. You can get the value for <code>col</code> using <code>getColumnIndex()</code> .
<code>request.getQuery()</code>	Retrieves the query (if any) that was passed to the CFX tag. Returns a <code>Query</code> object, or <code>null</code> if no <code>QUERY</code> attribute was provided.
<code>query.getColumnIndex(name)</code>	Returns the column index (position) of the column with the given name. You can then use the index to specify columns to <code>getData()</code> and <code>setData()</code> . Returns <code>-1</code> if the column doesn't exist.
<code>query.getColumns()</code>	Returns the names of the query's columns, as an array of strings.
<code>query.getData(row, col)</code>	Returns the value in the query at the given row and column position. The value is always returned as a string.
<code>query.getName()</code>	Returns the name of the query (as a string), as it is known in the calling ColdFusion page.
<code>query.getRowCount()</code>	Returns the number of rows in the query, as an <code>int</code> .

Table 30.7 C++ Methods for Working with Queries

METHOD	DESCRIPTION
<code>pRequest->AddQuery(name, columns)</code>	Creates a new query. Specify the name of the query as a LPCSTR, and <code>columns</code> as a <code>CCFXStringSet</code> (similar to an array of strings).
<code>pQuery->AddRow()</code>	Adds a row to the query.
<code>pQuery->SetData(row, col, value)</code>	Places the <code>value</code> (which must be a LPCSTR) into the query at the row and column position you specify.
<code>pRequest->GetQuery()</code>	Retrieves the query (if any) that was passed to the CFX tag. Returns a <code>CCFXQuery</code> object, or <code>null</code> if no <code>QUERY</code> attribute was provided to the tag.
<code>pQuery->GetColumns()</code>	Returns the names of the query's columns, as a <code>CFXStringSet</code> , which is similar conceptually to an array of strings. You can use any of the methods listed in Table 30.8 on the returned <code>CFXStringSet</code> . For instance, you can use <code>GetIndexForString()</code> to find the index position of a column by name; that index position could then be used in the <code>GetData()</code> or <code>SetData()</code> functions.
<code>pQuery->GetData(row, col)</code>	Returns the value in the query at the given row and column position. The value is always returned as a LPCSTR.
<code>pQuery->GetName()</code>	Returns the name of the query (as a LPCSTR), as it is known in the calling ColdFusion page.
<code>pQuery->GetRowCount()</code>	Returns the number of rows in the query, as an <code>int</code> .

To pass a query to a CFX tag, the name of the query must be specified as an attribute named `QUERY` (you don't get to determine the name of the attribute). So, if a `query="FilmsQuery"` attribute is supplied when the tag is called, then the `getQuery()` method will grab the query for use inside of the CFX tag. You can then use `getRecordCount()`, `getColumns()`, and `getData()` to refer to the data in the query.

Because standard C++ doesn't have a built-in notion of an ordered set of strings, the CFX API includes an additional class type called `CFXStringSet`, which is similar conceptually to an array of strings in Java. An object of this type is returned if you call the `GetAttributes()` method listed in Table 30.3 or the `GetColumns()` method from Table 30.7. Similarly, to create a new query object, you first create a new `CFXStringSet`, add column names to the string set using `AddString()`, then pass the string set to the `AddQuery()` method (also listed in Table 30.7). The methods supported by this class are listed in Table 30.8.

NOTE

This part of the CFX API is only included in the C++ version. There is no need for it in Java, which already includes the notion of a string set (the `String[]` type).

Table 30.8 C++ Methods for Working with String Sets

METHOD	DESCRIPTION
<code>stringset->AddString(string)</code>	Adds a LPCSTR value to the string set. Most commonly, this method is used to add column names to a string set that will later be passed to <code>pQuery->AddQuery()</code> . Returns the index position of the string that was just added.
<code>stringset->GetCount()</code>	Returns the number of strings in the string set, as an int. If you call this method on the result of the <code>GetAttributes()</code> method, you get the number of attributes passed to the tag. If you call it on the result of <code>GetColumns()</code> , you get the number of columns in the query.
<code>stringset->GetString(index)</code>	Returns the string (as a LPCSTR) at the index position you specify. The first string is at position 1, the second string is at position 2, and so on.
<code>stringset->GetIndexForString(string)</code>	Returns the index position of the string you specify. The search is not case sensitive. If the string is not in the set, the constant <code>CFX_STRING_NOT_FOUND</code> (defined as -1 in <code>cfx.h</code>) is returned.

Working with Exceptions and Debugging

If anything goes wrong while your CFX tag is doing its work, you need a way to display a helpful error message in the ColdFusion page that is using the tag. To that end, the CFX API provides mechanisms for throwing exceptions that behave just like ColdFusion's own exceptions, and just like exceptions thrown by `<cfthrow>` tag in normal ColdFusion code. In addition, methods are provided that allow your tag to include debug messages that will be included in the current Web page when ColdFusion's debugging options are turned on. Table 30.9 shows the Java version of this part of the CFX API, and Table 30.10 shows the C++ version.

Table 30.9 Java Methods for Exceptions and Debugging

METHOD	DESCRIPTION
<code>throw new Exception(message)</code>	Creates an exception (error message), which will be displayed in ColdFusion just like any other error message. This is similar conceptually to the <code><cfthrow></code> tag in CFML. If it wishes, the calling template can use <code><cftry></code> and <code><cfcatch></code> to handle the exception gracefully. This isn't really part of the CFX API; it's standard Java syntax. I'm including it in this table for consistency's sake.

Table 30.9 (CONTINUED)

METHOD	DESCRIPTION
<code>request.debug()</code>	Determines whether the tag has been called with the <code>debug</code> attribute turned on. If so, you would presumably include some kind of debugging message in the current Web page with <code>writeDebug()</code> . For details, see the “Generating Debug Output” section later in this chapter.
<code>response.writeDebug(text)</code>	Includes a text message in the current Web page. Very similar to <code>response.write()</code> , except that the text is only included in the page if the tag has been called with a <code>debug</code> attribute.

Table 30.10 C++ Methods for Exceptions and Debugging

METHOD	DESCRIPTION
<code>pRequest->ThrowException (message, detail)</code>	Creates an exception (error message), similar to the <code><cfthrow></code> tag in CFML. The <code>message</code> and <code>detail</code> must be specified as LPCSTR compatible values.
<code>pRequest->Debug()</code>	Determines whether the tag has been called with the <code>debug</code> attribute turned on. If so, you would presumably include some kind of debugging message in the current Web page with <code>WriteDebug()</code> .
<code>pRequest->WriteDebug(text)</code>	Includes a text message in the current Web page. Very similar to <code>pRequest->Write()</code> , except that the text is only included in the page if the CFX is used with a <code>debug</code> attribute.

Writing CFX Tags with Java

Now that you’ve gotten an introduction to the methods available in the CFX API, you can get started creating CFX tags with Java or C++. Because I assume that more people will be writing tags in Java than C++ going forward, we’ll start off with Java. If you plan to work with C++, just take a quick look through this section, then skip ahead to the “Writing CFX Tags with Visual C++” section, later in this chapter.

Getting Started

The process of creating a Java CFX tag can be broken down into the following steps:

1. Writing the Java code in a text editor or Java IDE

2. Compiling the Java code into the corresponding .class file
3. Registering the new CFX tag in the ColdFusion Administrator

In the next few pages, you'll learn how to perform each of these steps, producing a very simple CFX tag along the way. More complicated, fully featured examples will follow later in this chapter.

Writing the Java Code

The actual code for your CFX will be contained within an ordinary Java class file. Like any Java class, you can write the code in any text editor, such as the Windows Notepad, Macromedia Dreamweaver, or the editor of your choice. Once the code is written, you can use the `javac` utility from the Java SDK to compile the class (discussed below, in the “Compiling the CFX Tag” section).

NOTE

Of course, you are also free to use a free or commercial Java Integrated Development Environment (IDE), which will be able to offer such niceties as automatic code completion, automatic compilation, and integrated help. One such IDE is Sun's ONE Studio (formerly called Forte for Java); another is Borland's JBuilder product.

This chapter's first example will be a simple CFX tag called `<CFX>HelloWorld>`. Listing 30.1 shows the Java source code for the tag.

Listing 30.1 HelloWorld.java—Creating the `<CFX>HelloWorld>` Tag with Java

```
import com.allaire.cfx.* ;

public class HelloWorld implements CustomTag {
    public void processRequest( Request request, Response response )
        throws Exception
    {
        // Make sure a NAME attribute is passed to the tag
        if ( !request.attributeExists("NAME") ) {
            throw new Exception("The NAME attribute is required!");
        };

        // Make sure an AGE attribute is passed as well
        if ( !request.attributeExists("AGE") ) {
            throw new Exception("The AGE attribute is required!");
        };

        // Respond by inserting a text message in the calling ColdFusion file
        // The text will appear in place of the CFX tag in the final Web page
        response.write("<P>Hello there " + request.getAttribute("NAME") + "!");
        response.write("<BR>I hear you turned " + request.getAttribute("AGE"));
        response.write(" sometime during the past year. Happy birthday!<BR>");
    }
}
```

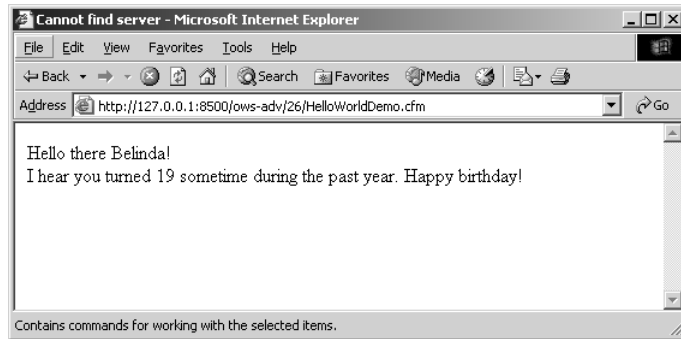
Even if you've never seen a line of Java code before, you can probably guess what this tag will do when it is used in a ColdFusion page. That's right, it will display a “Hello, World” type of message which contains the values of the `NAME` and `AGE` attributes that get passed to the tag.

For instance, the tag can be called as shown in the following code snippet. The resulting message will incorporate the words `Belinda` and `19`, just as you would expect (Figure 30.1):

```
<CFX_HelloWorld
  NAME="Belinda"
  AGE="19">
```

Figure 30.1

The `<CFX_HelloWorld>` tag accepts two attributes and displays their values in a simple text message.



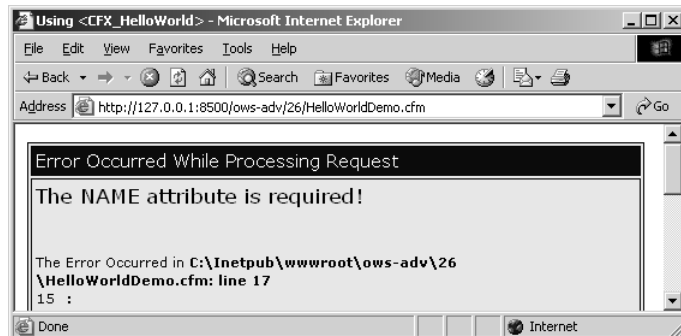
Of course, you can provide dynamic expressions to the tag's attributes, just like any native CFML tag. For instance, you could supply values from the `SESSION` and `FORM` scopes like so:

```
<CFX_HelloWorld
  NAME="#SESSION.FirstName#"
  AGE="#FORM.Age#">
```

If a ColdFusion page tries to call the tag without supplying the `NAME` or `AGE` attributes, an exception will be thrown, resulting in a standard error message (Figure 30.2).

Figure 30.2

An error message is displayed if required attributes are not provided.



Understanding the Java Code

Even though it's a very simple example, I'd like to spend a few moments going over a few key elements of the `<CFX_HelloWorld>` source code. For starters, take another look at the first few lines of the example (Listing 30.1); these are the lines that make it possible for ColdFusion to treat the class as a CFX tag.

Basically, every Java CFX tag needs to be a public class that implements the `CustomTag` interface in the `com.allaire.cfx` package. There isn't a whole lot to the `CustomTag` interface; you can think of it as a simple agreement which says that every custom tag must include a `processRequest()` function like the one shown in Listing 30.1. Each time the CFX tag is used in a ColdFusion page, the server will invoke the `processRequest()` method, passing information about the current page request to the `request` and `response` arguments. (The guts of the tag can then call the various methods exposed by `request` and `response`, as listed in the tables from the first part of this chapter.)

When you add these modest requirements up, it means that every Java CFX tag must include the following code skeleton, nearly verbatim. The only thing that will change from tag to tag is the name of the class (here, it's `HelloWorld`):

```
import com.allaire.cfx.* ;

public class HelloWorld implements CustomTag {
    public void processRequest( Request request, Response response )
        throws Exception
    {
        ...CFX tag logic goes here...
    }
}
```

The remainder of Listing 30.1 is simple. The `request.attributeExists()` method from Table 30.2 is used to make sure that the `NAME` and `AGE` attributes are passed to the tag each time it is used. If not, the standard Java `throw` statement is used to create an exception, which will bubble up to the ColdFusion server for display on the calling page.

Assuming the attributes have been provided, execution proceeds to the `response.write()` lines. As explained in Table 30.4, the `response.write()` method inserts any string into the current ColdFusion page, much like `<cfoutput>` or `WriteOutput()` in CFML (or `response.getWriter().write()` in a Java Servlet). In this case, the strings are a combination of static text and the actual values of the `NAME` and `AGE` attributes, available from the `request.getAttribute()` method (see Table 30.2).

That's it!

Compiling the CFX Tag

With the Java code written, the next step is to compile it into a Java class file. If you're using a dedicated Java IDE, you can probably just hit a `Compile` button on some kind of toolbar, but for this discussion I'll assume you need to compile the class manually using Sun's `javac` compiler.

If you don't have a copy of Sun's Java SDK (also known as the JDK) on your system, you will need to download one now. You can download the SDK at no charge from <http://java.sun.com>. Go ahead and run the installation program, making a mental note of the directory that the SDK gets installed into as you go.

NOTE

You can use any modern version of the SDK to compile your CFX tags, but for consistency's sake I would recommend using a version equal to (or later than) the version of the JRE that ColdFusion is using as its runtime engine. For the initial release of ColdFusion, that means version 1.3.1_01 or later. If you use a later version, just keep in mind that ColdFusion won't be able to run the CFX tag if it uses built-in classes that aren't supported by the version of the JRE that ColdFusion is using.

To compile your CFX, use `javac` on the command line to compile the `HelloWorld.java` file as shown below, adjusting the paths to the `cfx.jar` and `HelloWorld.java` files as appropriate for your system:

```
javac -classpath c:\CFusionMX7\lib\cfx.jar HelloWorld.java
```

The `cfx.jar` file contains the `com.allaire.cfx` package, which defines the `Request` and `Response` interfaces referred to in the skeletal snippet shown in the previous section (and in Listing 30.1). The `cfx.jar` file is installed automatically to the `CFusionMX7/lib` folder when you install ColdFusion. The compiler needs to be able to find this file in order to be able to compile the class; typically, you provide the location using the `-classpath` switch as shown above. Consult the Java SDK documentation for the `javac` utility for details on the `-classpath` switch.

NOTE

Depending on how your system is configured, you may also need to provide a full path to the `javac` executable, which is located in the `bin` subfolder within the folder to which the Java SDK was installed.

After you compile the tag, a file named `HelloWorld.class` will appear in the same folder where the `HelloWorld.java` file is located. Copy this file to the `WEB-INF/classes` folder within ColdFusion's program root (typically `c:\CFusionMX7\wwwroot\WEB-INF\classes` if you're using a Windows server).

NOTE

I'm having you place the class file in the `WEB-INF/classes` folder because that is recommended by Macromedia as a good place for CFX class files during testing and development. Once you have finished developing the tag, you can place the class on your production servers in just about any location. The only caveat is that the class needs to be somewhere within the Class Path specified in the Java and JVM page of the ColdFusion Administrator.

NOTE

The ColdFusion documentation also refers to the `WEB-INF/classes` folder as a good place to put your class files during development. The documentation suggests that one of these folders (the `WEB-INF/classes` folder or the `CFusionMX7/lib` folder) is meant to ease development by eliminating the need to restart ColdFusion after a Java CFX has been recompiled.

If you want, you can tell `javac` to place the class file directly in the `lib` folder as it is being compiled. As an example, I used the command shown below to compile the class (Figure 30.3). Of course, I typed this all on one line, but it's too long to print on one line in this book:

```
c:\jdk1.3.1-01\bin\javac c:\inetpub\wwwroot\ows_adv\30\HelloWorld.java
-classpath c:\CFusionMX7\lib\cfx.jar
-d c:\CFusionMX7\wwwroot\WEB-INF\classes -verbose
```

This command assumes that the Java SDK was installed to the `c:\jdk1.3.1_01` folder, that you are using the built-in Web server, that the `HelloWorld.java` file is stored with the other listings for this chapter (within the `ows_adv` branch of the Web server document root), and that ColdFusion was installed to the default `c:\CFusionMX7` location. Make whatever path adjustments are necessary for your system. You can leave off the `-verbose` switch if you want; it just causes `javac` to display some additional messages while it is compiling the class.

Figure 30.3

Java CFX tags are compiled using Sun's standard javac compiler.

```

C:\>c:\jdk1.3.1_01\bin\javac c:\inetpub\wwwroot\ows-adv\26\HelloWorld.java -class
spath c:\CFusionMX\lib\cfx.jar -d c:\CFusionMX\wwwroot\WEB-INF\classes -verbose
[compiling started c:\inetpub\wwwroot\ows-adv\26\HelloWorld.java]
[compiling completed 131ms]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/Object.class>]
[loading c:\CFusionMX\lib\cfx.jar<com/allaire/cfx/CustomTag.class>]
[loading c:\CFusionMX\lib\cfx.jar<com/allaire/cfx/Request.class>]
[loading c:\CFusionMX\lib\cfx.jar<com/allaire/cfx/Response.class>]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/Exception.class>]
[checking HelloWorld]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/Error.class>]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/Throwable.class>]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/RuntimeIOException.class>]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/String.class>]
[loading c:\jdk1.3.1_01\jre\lib\rt.jar<java/lang/StringBuffer.class>]
[write c:\CFusionMX\wwwroot\WEB-INF\classes\HelloWorld.class]
[total 852ms]

C:\>

```

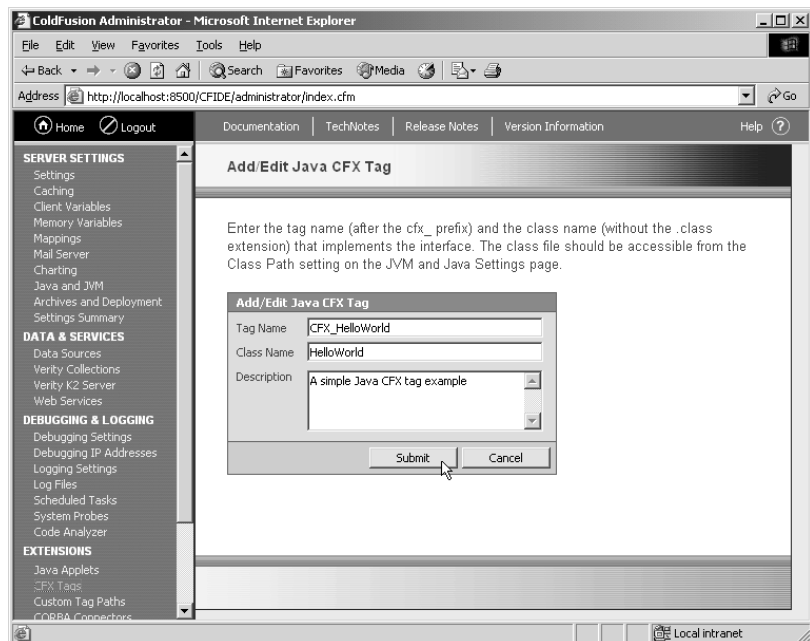
Registering the New Tag

Now that the class has been compiled and placed into ColdFusion's lib folder, the only thing left to do is to register the new tag in the ColdFusion Administrator. Follow these steps to register the tag:

1. Navigate to the CFX Tags page of the ColdFusion Administrator.
2. Press the Register Java CFX button. The Add/Edit Java CFX Tag page appears (Figure 30.4).

Figure 30.4

Tags need to be registered in the Administrator before they can be used in ColdFusion pages.



3. For the Tag Name field, enter the name of the tag, including the `CFX_` part but without the angle brackets (for this example, you will enter `CFX>HelloWorld`). The names of CFX tags are not case sensitive, so you don't need to get the capitalization exactly right. That said, it makes sense to use the same capitalization that you plan to use in your ColdFusion code.
4. For the Class Name field, enter the name of the class file, but without the `.class` extension. This part is case sensitive, so the capitalization you use here must match the capitalization of the class file (which, in turn, always matches the name of the class as specified in the Java code itself).
5. If you wish, enter a Description for the tag. The description comes in handy during development if several people on your team are creating different CFX tags.
6. Click the submit button to register the tag. The tag will now appear in the Registered CFX Tags list, indicating that it is ready for use in your ColdFusion pages.

Using the New Tag

Now that the `<CFX>HelloWorld` tag has been written, compiled, and registered in the ColdFusion Administrator, it's ready to be used in your pages. Listing 30.2 shows the code for the simple test page shown previously in Figure 30.1.

Listing 30.2 HelloWorldDemo.cfm—Using the `<CFX>HelloWorld` Tag

```

<!---
  Filename: HelloWorldDemo.cfm
  Author: Nate Weiss (NMW)
  Purpose: Demonstrates how to use a CFX tag in a ColdFusion page
  --->

<html>
<head>
<title>Using &lt;CFX>HelloWorld&gt;</title>
</head>
<body>

<!--- Invoke the Java CFX tag --->
<CFX>HelloWorld
  NAME="Belinda"
  AGE="19">

</body>
</html>

```

A Word on Exceptions

It's worth noting that the exceptions created by the `throw` statement in Listing 30.1 become true ColdFusion exceptions that behave just like exceptions thrown by native CFML tags, or by the `<cfthrow>` tag. The look and feel of the error messages can be customized with the `<cferror>` tag.

You can even catch exceptions thrown by a CFX and handle them intelligently, using `<cftry>` and `<cfcatch>`, like so:

```
<cftry>
  <CFX_HelloWorld
    NAME="#SESSION.FirstName#"
    AGE="#FORM.Age#">

  <cfcatch type="Any">
    ...error handling code goes here...
  </cfcatch>
</cftry>
```

Returning Query Objects

Now that you’ve seen how to create a simple CFX tag with Java, it’s time to move on to something a bit more complicated and useful. Our next project will be a CFX tag called `<CFX_DatabaseMetaData>`, which can be used to inspect the structure of a database on the fly. To use the tag in a ColdFusion page, you will specify the type of information you want (a list of tables, columns, indexes, or the like), and the tag will return a query object filled with items like column names, table names, data types, and so on.

Table 30.11 shows the syntax that will be supported by the completed tag.

Table 30.11 `<CFX_DatabaseMetaData>` Tag Syntax

ATTRIBUTE	DESCRIPTION
action	Required. A string indicating the type of information you are interested in, such as <code>GetTables</code> or <code>GetColumns</code> . You can provide any of the actions listed in Table 30.12.
driver	Required. The name of the JDBC driver to use. For ODBC databases, you can use Sun’s ODBC Bridge driver, as in <code>driver="sun.jdbc.odbc.JdbcOdbcDriver."</code>
connect	Required. Whatever connection information the driver needs to connect to the database. In the case of ODBC datasources, you can use a string in the form <code>jdbc:odbc:dsn_name</code> , as in <code>connect="jdbc:odbc:ows"</code> for the Orange Whip Studios example database.
username	Required. The username for the data source (may be case sensitive).
password	Required. The password for the data source (may be case sensitive).
name	Required. Like the <code>name</code> attribute for <code><cfquery></code> , a name for the query object that the tag returns. The returned query will contain different columns depending on the <code>action</code> you choose, as listed in Table 30.12.
dbcatalog	Optional. The catalog name that you want to get information about. With most database systems, the catalog name is the name or filename of the database. If you don’t specify a catalog name, all information is returned (for all available catalogs or databases). Can be used for all actions except <code>GetCatalogs</code> and <code>GetSchemas</code> .

Table 30.11 (CONTINUED)

ATTRIBUTE	DESCRIPTION
dbschema	Optional. The name of the database schema. Not all database systems have the notion of a schema; for Access databases, specifying this attribute will generate an error. For SQLServer databases, the schema is often dbo. Can be used for all actions except GetCatalogs and GetSchemas.
dbtablename	Optional. The name of the table you want to get information about. Omit or leave blank for all tables. Can be used for GetColumns, GetTables, GetViews, GetIndexInfo, and GetPrimaryKeys.
dbcolumname	Optional. The name of the column you want information about. Omit or leave blank for all columns. Can be used for GetColumns and GetProcedureColumns.
dbprocedurename	Optional. The name of the stored procedure you want information about; omit or leave blank for all procedures. Can be used for GetProcedures or GetProcedureColumns.

Table 30.12 shows the various actions you can specify for the tag's action attribute. The query object returned by the tag will contain different information based on which action you choose, as shown in this table.

Table 30.12 Information Returned by <CFX_DatabaseMetaData> Actions

ACTION	DESCRIPTION	COLUMNS RETURNED
GetTables	A listing of tables in the database. The returned query includes these columns: By default, all tables are returned; you can filter the list using the DBCATALOG, DBSCHEMA, or DBTABLENAME attributes.	TABLE_CAT, TABLE_NAME, TABLE_SCHEM, and TABLE_TYPE
GetViews	Same as GetTables, except for views.	TABLE_CAT, TABLE_NAME, TABLE_SCHEM, and TABLE_TYPE
GetColumns	A listing of columns. All columns in the entire database will be returned unless you specify a DBTABLENAME (or DBSCHEMA or DBCATALOG).	COLUMN_NAME, COLUMN_SIZE, IS_NULLABLE, SQL_DATA_TYPE, TABLE_CAT, TABLE_NAME, TABLE_SCHEM, and TYPE_NAME
GetCatalogs	A listing of catalogs (databases).	TABLE_CAT
GetSchemas	A listing of schemas.	TABLE_SCHEM
GetProcedures	A listing of all stored procedures in the database.	PROCEDURE_NAME

Table 30.12 (CONTINUED)

ACTION	DESCRIPTION	COLUMNS RETURNED
GetProcedureColumns	A listing of columns returned by stored procedures. Also includes information about the procedure's input and output parameters.	PROCEDURE_NAME, COLUMN_NAME, COLUMN_TYPE
GetPrimaryKeys	A description of primary keys.	TABLE_NAME, COLUMN_NAME, KEY_SEQ
GetIndexInfo	A description of the indexes in the database.	ASC_OR_DESC, COLUMN_NAME, TABLE_NAME, INDEX_NAME, NON_UNIQUE, TYPE

NOTE

You can get more specific information about the columns returned by each action by looking through the Java SDK documentation for the `java.sql.DatabaseMetaData` interface.

Writing the Java Code

The `<CFX_DatabaseMetaData>` tag will use a number of classes and methods from the `java.sql` package, which are standard, built-in classes supported by the Java 2 SDK and JRE. It's not possible for me to explain everything about these classes in these pages, but here is a quick introduction to the most important classes used in the next example listing:

- `java.sql.DriverManager`. This static class is used to obtain connections to data sources. Its most important method is `getConnection()`, which connects to a database based on a set of connection arguments (the name and location of the database and so on). Before the `getConnection()` method is called, it is traditional to load a database driver using `Class.forName()` (see the Java SDK documentation for details).
- `java.sql.Connection`. This class is returned by `getConnection()` and represents an active connection to a database. In this example, we will be most interested in its `getMetaData()` method, which returns a `DatabaseMetaData` object (discussed next).
- `java.sql.DatabaseMetaData`. This is the class that the CFX tag is most interested in using. The various actions supported by the tag (see Table 30.12) map directly to many of this class's methods, such as `getTables()` and `getColumns()`. These methods all return `ResultSet` objects containing the requested information.
- `java.sql.ResultSet`. This class is the Java equivalent to a ColdFusion query recordset. Within the CFX tag, the `ResultSet` object returned by `DatabaseMetaData` is converted to a ColdFusion query using the methods outlined in Table 30.6.
- `java.sql.ResultSetMetaData`. This class is conceptually similar to `DatabaseMetaData` in that it returns metadata (column names, data types, and so on), but is smaller in scope and only returns metadata about a particular `ResultSet`, rather than the database as a whole. This CFX tag uses this class to get the names of the columns in the `ResultSet` returned by `DatabaseMetaData`.

Listing 30.3 shows how these objects can be used together to create the <CFX_DatabaseMetaData> tag. This is a relatively long listing, but if you look at it part by part you'll see that the code is really quite simple and straightforward. To a large extent, you will find examples in the Java SDK documentation that contain lines quite similar to many of the lines in this listing.

Listing 30.3 DatabaseMetaData.java—Using Functionality from the java.sql Package in a CFX Tag

```
import com.allaire.cfx.* ;
import java.sql.* ;

public class DatabaseMetaData implements CustomTag
{
    // Constant string for error messages
    final String msgError = "Error occurred in a <CFX_DATABASEMETADATA> tag. ";

    // This gets called each time the tag is used in a ColdFusion page
    public void processRequest( Request request, Response response )
    throws Exception
    {
        // Obtain the values for the tag's attributes, and throw exceptions
        // if any of the required attributes have not been provided.
        String strAction = getTagAttr("ACTION", request);
        String userName = getTagAttr("USERNAME", request);
        String password = getTagAttr("PASSWORD", request);
        String driver = getTagAttr("DRIVER", request);
        String connect = getTagAttr("CONNECT", request);
        String strQueryName = getTagAttr("NAME", request);
        // These are optional attributes
        String DBCatalog = getTagAttr("DBCATALOG", request, null);
        String DBSchema = getTagAttr("DBSCHEMA", request, null);
        String DBTableName = getTagAttr("DBTABLENAME", request, "");
        String DBColumnName = getTagAttr("DBCOLUMNNAME", request, "");
        String DBProcedureName = getTagAttr("DBPROCEDURENAME", request, "");

        // For the DBTableName attribute, consider an empty string to mean null
        //if (DBTableName.equals("")) DBTableName = null;

        // Load the specified database driver
        Class.forName(driver).newInstance();

        // Attempt to connect to the data source
        Connection conn = DriverManager.getConnection(connect, userName, password);

        // Get the metadata object from the database connection
        java.sql.DatabaseMetaData dbmd = conn.getMetaData();

        // This ResultSet will be returned to ColdFusion as a query object
        ResultSet rs = null;

        // Handle ACTION="GetTables" (user wants a list of tables)
        if ( strAction.equalsIgnoreCase("GetTables") ) {
            String[] types = {"TABLE"};
            rs = dbmd.getTables(DBCatalog, DBSchema, DBTableName, types);

            // Handle ACTION="GetViews" (user wants a list of views)
```

Listing 30.3 (CONTINUED)

```

} else if ( strAction.equalsIgnoreCase("GetViews") ) {
String[] types = {"VIEW"};
rs = dbmd.getTables(DBCatalog, DBSchema, DBTableName, types);

// Handle ACTION="GetColumns" (user wants a list of columns)
} else if ( strAction.equalsIgnoreCase("GetColumns") ) {
rs = dbmd.getColumns(DBCatalog, DBSchema, DBTableName, DBColumnName);

// Handle ACTION="GetCatalogs" (user wants a list of catalogs)
} else if ( strAction.equalsIgnoreCase("GetCatalogs") ) {
rs = dbmd.getCatalogs();

// Handle ACTION="GetSchemas" (user wants a list of schemas)
} else if ( strAction.equalsIgnoreCase("GetSchemas") ) {
rs = dbmd.getSchemas();

// Handle ACTION="GetProcedures" (a list of stored procedures)
} else if ( strAction.equalsIgnoreCase("GetProcedures") ) {
rs = dbmd.getProcedures(DBCatalog, DBSchema, DBProcedureName);

// Handle ACTION="GetProcedureColumns" (list of stored procedure columns)
} else if ( strAction.equalsIgnoreCase("GetProcedureColumns") ) {
rs = dbmd.getProcedureColumns(DBCatalog, DBSchema, DBProcedureName, DBColumnName);

// Handle ACTION="GetPrimaryKeys"
} else if ( strAction.equalsIgnoreCase("GetPrimaryKeys") ) {
rs = dbmd.getPrimaryKeys(DBCatalog, DBSchema, DBTableName);

// Handle ACTION="GetIndexInfo"
} else if ( strAction.equalsIgnoreCase("GetIndexInfo") ) {
rs = dbmd.getIndexInfo(DBCatalog, DBSchema, DBTableName, false, false);

// Throw an error if the ACTION attribute was not recognized
} else {
throw new Exception("Unknown ACTION attribute!");
}

// The rs ResultSet now contains the metadata info about tables/columns
// Return it to the calling ColdFusion page as a query
returnAsQuery(rs, strQueryName, response);
}

// Helper function that accepts any java.sql.ResultSet
// and returns it to ColdFusion as a CFML query object
private void returnAsQuery(ResultSet rs, String name, Response response)
throws Exception
{
// Get metadata about the ResultSet (so we can find out its column names)
ResultSetMetaData rsmd = rs.getMetaData();

// Create an array of strings to hold the ResultSet's column names
String[] arColumns = new String[rsmd.getColumnCount()];

```

Listing 30.3 (CONTINUED)

```

// Fill the array with this ResultSet's column names
for (int col = 1; col <= rsmd.getColumnCount(); col++ ) {
    arColumns[col-1] = rsmd.getColumnName(col);
};

// Create a new com.allaire.Query object
Query q = response.addQuery(name, arColumns);

// For each row of the ResultSet...
while ( rs.next() ) {
    // Add a row to the CFML query
    int row = q.addRow();

    // For each column of the ResultSet...
    for (int col = 1; col <= rsmd.getColumnCount(); col++) {
        // Copy the data at this row/column from ResultSet to Query
        // (an error will be thrown if the data can't be read as a string)
        q.setData(row, col, rs.getString(col));
    };
};

}

// Helper function that returns the value of an attribute
// An exception is thrown if the attribute was not provided
private String getTagAttr(String name, Request request)
throws Exception
{
    if ( request.attributeExists(name) ) {
        return request.getAttribute(name);
    } else {
        throw new Exception(msgError + "The "+name+" attribute is required.");
    };
}

// Helper function that returns the value of an attribute
// If the attribute is not provided, it defaults to def.
private String getTagAttr(String name, Request request, String def)
throws Exception
{
    return request.attributeExists(name) ? request.getAttribute(name) : def;
}

}

```

Just by scanning over this example listing with your eye, you can divide this tag into three conceptual parts:

- The `processRequest()` handler method, which is executed by ColdFusion whenever the CFX tag is used in a ColdFusion page. This portion of the code does the work of connecting to the database and retrieving the requested information as a `java.sql.ResultSet` object.

- The `returnAsQuery()` function, which accepts any Java-style `ResultSet` object and makes it available to the calling page as a ColdFusion-style query object. You can cut and paste this code into other CFX tags that need to return `ResultSet` objects as CFML queries.
- The two forms of the `getTagAttr()` function, which wraps around the `attributeExists()` and `getAttribute()` methods from the CFX API that were used previously in Listing 30.1. The first form requires that the attribute be passed to the tag, and the second form makes an attribute be considered optional by using a default value when it is not provided. Again, you can cut and paste these functions into your own CFX tags if you find them convenient.

Within `processRequest()`, the first portion calls `getTagAttr()` repeatedly to establish the required and optional attributes for the tag. If any required attributes are missing, an error message will be shown; optional attributes will be assigned default value when not provided. The next few lines load the JDBC driver class specified in the tag's `driver` attribute with `Class.forName()`, then connect to the data source by supplying the `connect`, `username`, and `password` attributes to `getConnection()`, then create an object called `dbmd` to represent the data source's metadata structure.

Next, a series of `if` and `else if` tests are used to call the appropriate `DatabaseMetaData` method depending on the `action` attribute that the tag is being called with. No matter which method ends up being called (`getTables()`, `getColumns()`, or the like), the result is always a `ResultSet` object called `rs`. The `returnAsQuery()` function is then called to return the data in the `ResultSet` to the calling ColdFusion page.

The main task of `returnAsQuery()` is to create the new CFML query object with the `addQuery()` method. That method requires the new query's column names to be specified as an array of strings, so a few lines of code are required first to obtain the recordset's column names using `getMetaData()`, `getColumnCount()`, and `getColumnName()` (all described in the Java SDK documentation). Once the query has been created, it is a relatively simple task to loop through the `ResultSet`, adding rows to the CFML query and filling it with data from the `ResultSet` along the way. See Table 30.6, earlier in this chapter, for details about the `addQuery()`, `addRow()`, and `setData()` methods used in this portion of the listing.

NOTE

Please consult the Java SDK documentation for more information on the classes and methods used in this listing (except for the ones specific to CFX development; those are detailed in the tables at the beginning of this chapter). Most of them are in the `java.sql` package.

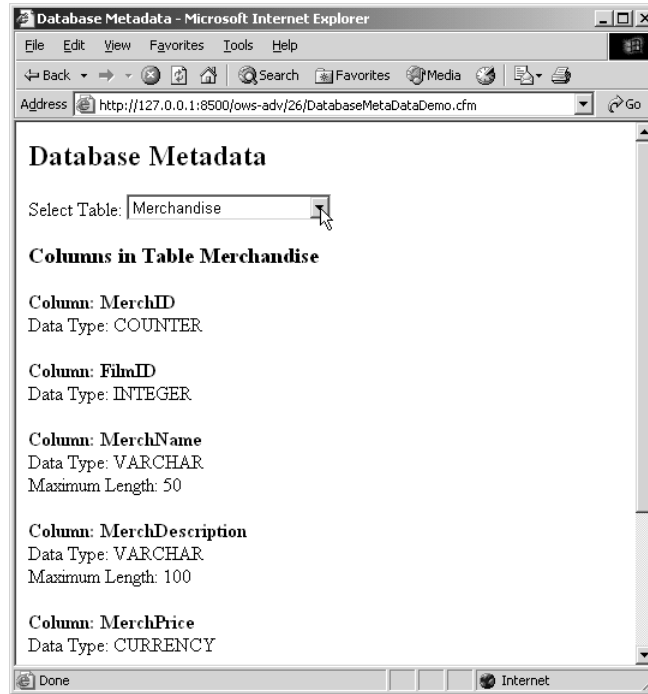
Using the New Tag in ColdFusion

To use the new `<CFX_DatabaseMetaData>` tag in a ColdFusion page, you need to use `javac` to compile Listing 30.1, then register the resulting `DatabaseMetaData.class` file with the ColdFusion Administrator. These steps are described in the “Getting Started” section earlier in this chapter.

You can then use the tag in any CFML page, such as the simple example shown in Listing 30.4. This listing creates a page that allows the user to select from the tables in the `ows.mdb` database file using a drop-down list. When the user selects a table, the names, types, and widths for each of the table's columns are displayed (Figure 30.5).

Figure 30.5

This simple example allows users to browse through the database's structure.

**NOTE**

Before this listing will work, you need to create an ODBC data source called `ows` that points to the `ows.mdb` sample database. If you are not using a Windows server, you can use other JDBC database drivers; just adjust the `DRIVER` and `CONNECT` attributes accordingly. The documentation for the driver should help you understand what values to provide for these attributes.

CAUTION

Be careful if you place pages such as this on a public Web server. This example won't allow you to view the structure of any table that is not in the `ows` sample database, but code similar to this may make it possible for people to gain unwanted access to the structure of your databases. The tag is not a security risk in and of itself, but you should use it carefully so that your users don't see more than you want them to see.

Listing 30.4 DatabaseMetaDataDemo.cfm—Using `<CFX_DatabaseMetaData>` in a ColdFusion Page

```
<!---
  Filename: DatabaseMetaDataDemo.cfm
  Author:  Nate Weiss (NMW)
  Purpose: Uses <CFX_DatabaseMetaData> to display
           column names and data types
  -->

<!--- The database table to display information about --->
<cfparam name="FORM.TableName"
         type="string"
         default="">
```

Listing 30.4 (CONTINUED)

```

<html>
<head>
<title>Database Metadata</title>
</head>

<body>
<h2>Database Metadata</h2>

<!---
Get information about the tables in the database
-->
<CFX_DatabaseMetaData
  ACTION="GetTables"
  USERNAME="Admin"
  PASSWORD=""
  DRIVER="sun.jdbc.odbc.JdbcOdbcDriver"
  CONNECT="jdbc:odbc:ows"
  NAME="TablesQuery">

<!---
Provide drop-down list of tables
-->
<cfform action="DatabaseMetaDataDemo.cfm"
  method="POST">

  <p>
  Select Table:
  <cfselect name="TableName"
    query="TablesQuery"
    value="TABLE_NAME"
    selected="#FORM.TableName#"
    onChange="this.form.submit()"/>

</cfform>

<!---
If a table name has been selected
-->
<cfif FORM.TableName NEQ "">
  <cfoutput>
  <h3>Columns in Table #TableName#</h3>
  </cfoutput>

  <!---
  Get information about the selected table's columns
  -->
  <CFX_DatabaseMetaData action="GetColumns"
    username="Admin"
    password=""
    driver="sun.jdbc.odbc.JdbcOdbcDriver"
    connect="jdbc:odbc:ows"
    dbtablename="#TableName#"
    name="FilmsMetaData">

  <!---
  Display information returned by the CFX tag
  -->

```

Listing 30.4 (CONTINUED)

```

<cfoutput query="FilmsMetaData">
  <p>
    <strong>Column: #COLUMN_NAME#</strong>
    <br>
    Data Type: #TYPE_NAME#<br>
    <cfif TYPE_NAME contains "char">
      Maximum Length: #COLUMN_SIZE#<br>
    </cfif>
  </cfoutput>
</cfif>

</body>
</html>

```

Writing CFX Tags with Visual C++

The first part of this chapter introduced you to the various methods that make up the CFX API. The second part showed you how to work with the methods in Java code.

NOTE

The specific instructions and figures in this chapter assume that you are using version 6.0 of Visual C++. If you are using a different version of the product, some of the steps and screens may look slightly different.

NOTE

Technically, you can use the CFX API for C++ to create CFX tags in other languages and development environments. For instance, it is possible to create CFX tags using Borland's Delphi product; you can find information about this on the Web. That said, Java and C++ are the only languages that the API is officially designed for.

Getting Started

The next few pages will walk you through the process of creating a new tag with the C++ version of the CFX API. The basic steps are similar to the steps you follow when creating a tag with Java:

1. Writing the C++ code in Visual C++
2. Compiling the C++ code into the corresponding .dll file
3. Registering the new CFX tag in the ColdFusion Administrator

Installing the Tag Wizard and Libraries

The best way to start a new CFX tag is with the ColdFusion Tag Wizard, which is a simple tool that plugs into the New Project dialog in Visual C++. The wizard is not distributed with ColdFusion, but is freely available from the Macromedia Web site. It is also included with this chapter's listings for your convenience.

To install the CFX Tag Wizard, follow these steps:

1. If Visual C++ is currently running, close it.
2. Copy the `cfxwiz_vc50.awx` file to Visual Studio's Template folder. For a typical Visual Studio 6.0 installation, that folder is located at: `C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Templates`.
3. Copy the `cfx.h` file from the `CFusionMX7\cfx\include` folder to the Visual C++ Include folder. For a typical Visual Studio 6.0 installation, that folder is located at: `C:\Program Files\Microsoft Visual Studio\VC98\Include`.
4. Locate the `CreateRegistryKey.reg` file (included with this chapter's listings) and double-click on it. If you get a confirmation message, click OK to accept the change to the registry. After a moment, you should receive a message saying the information was imported successfully.
5. Start Visual C++. The ColdFusion Tag Wizard should appear if you choose File, New, then select the Projects tab.

NOTE

A bit of explanation about that `.reg` file: at the time of this writing, the Tag Wizard had not been updated to reflect the fact that ColdFusion no longer stores information about registered CFX tags in the Windows Registry (this information is now kept in an XML configuration file within the CFusion directory tree). Therefore, a small change to the registry needs to be made before the Wizard will work properly. The `CreateRegistryKey.reg` file referred to in the steps listed above simply adds a registry key (folder) called `HKEY_LOCAL_MACHINE\SOFTWARE\Allaire\ColdFusion\CurrentVersion\CustomTags`, which is what the Wizard needs to work properly. If you prefer to add the key to the registry manually using the Registry Editor that ships with Windows, go ahead. Just be careful because making an incorrect change could seriously mess up your system!

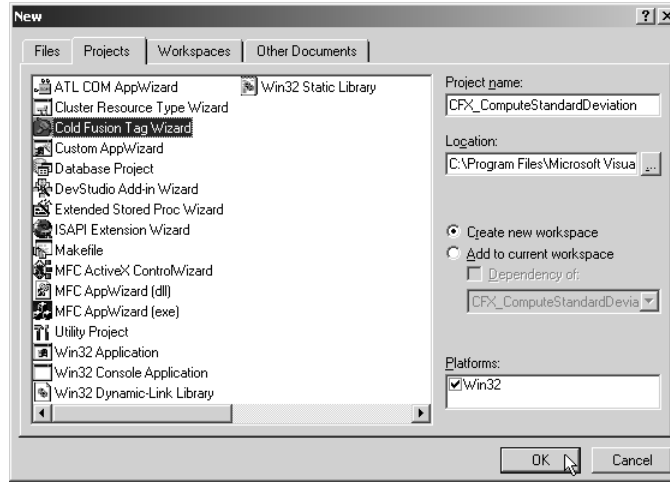
Starting a New CFX Tag Project

Now that the ColdFusion Tag Wizard has been installed, you can use it to quickly create skeleton code for a new CFX tag. Just follow these steps:

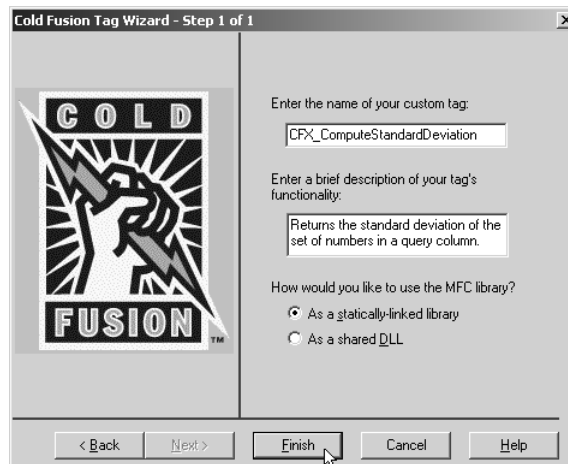
1. Launch Visual C++.
2. Choose File, New.... The New dialog box appears.
3. Select the Projects tab, then select the Cold Fusion Tag Wizard (Figure 30.6).
4. Enter a name for your new project in the Project name field. I recommend using the name of the CFX tag. For this example, use `CFX_ComputeStandardDeviation` as the project name.
5. Leave the Create new workspace and Win32 options checked, as shown in Figure 30.6.
6. Click OK. The ColdFusion Tag Wizard appears (Figure 30.7).

Figure 30.6

You will start a new project for each CFX tag you create.

**Figure 30.7**

The ColdFusion Tag Wizard makes it easy to get started with a new CFX tag.

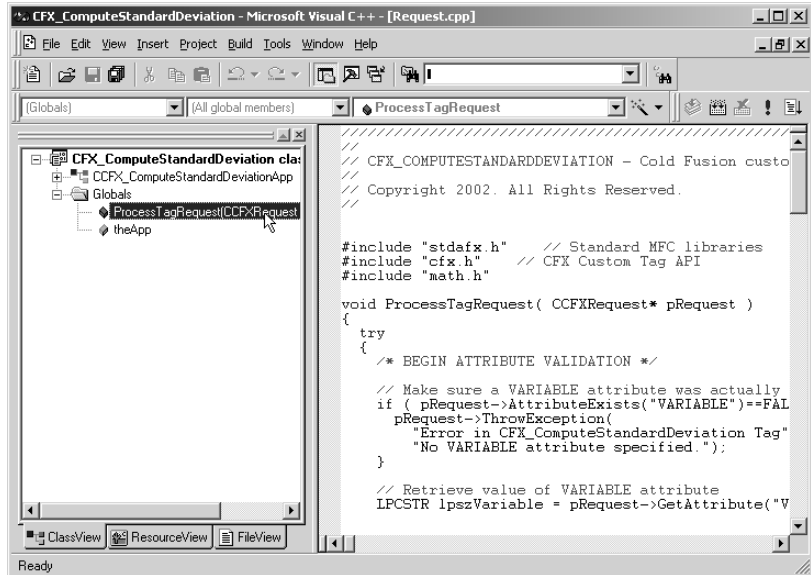


7. Enter the name of your CFX tag, including the CFX_ part. If you wish, you may also enter a description.
8. Unless you have a specific reason for not doing so, leave the As A Statically Linked Library option checked.
9. Click the Finish button to create your new project.

You should now have a new project workspace that includes a number of new files. For the most part, you need only concern yourself with the `ProcessTagRequest()` function, which is implemented in the `Request.cpp` file generated by the Wizard. You can get to this function easily by clicking on the `ProcessTagRequest` item in the `ClassView` tree (Figure 30.8).

Figure 30.8

Your new project workspace should look something like this.



Listing 30.5 shows the code generated by the ColdFusion Tag Wizard. In the next section, this code will be modified to create a CFX tag that actually does something useful.

Listing 30.5 Code Generated Automatically by the Tag Wizard

```

////////////////////////////////////
//
// CFX_COMPUTESTANDARDDEVIATION - Cold Fusion custom tag
//
// Copyright 2002. All Rights Reserved.
//

#include "stdafx.h" // Standard MFC libraries
#include "afx.h" // CFX Custom Tag API

void ProcessTagRequest( CCFXRequest* pRequest )
{
    try
    {
        // Retrieve attributes passed to the tag
        // For example:
        // LPCSTR lpszColor = pRequest->GetAttribute("COLOR") ;

        // Write output back to the user here...
        pRequest->Write( "Hello from CFX_COMPUTESTANDARDDEVIATION!" ) ;
    }
}

```

Listing 30.5 (CONTINUED)

```

// Output optional debug info
if ( pRequest->Debug() )
{
pRequest->WriteDebug( "Debug info..." );
}
}

// Catch Cold Fusion exceptions & re-raise them
catch( CFXException* e )
{
pRequest->ReThrowException( e );
}

// Catch ALL other exceptions and throw them as
// Cold Fusion exceptions (DO NOT REMOVE! --
// this prevents the server from crashing in
// case of an unexpected exception)
catch( ... )
{
pRequest->ThrowException(
"Error occurred in tag CFX_COMPUTESTANDARDDEVIATION",
"Unexpected error occurred while processing tag." );
}
}

```

Writing the C++ Code

At this point, you could compile and register the new tag using the steps described in the next section. If you do so, the code generated by the Wizard will simply display a “Hello, World” type of message when the tag is actually used in a ColdFusion page. Of course, to actually give your new tag its functionality, you need to remove the line that outputs the “Hello, World” message and add your own C++ code that accomplishes whatever task you want the tag to perform.

The code example we’ll be discussing in this chapter creates a CFX tag called `<CFX_ComputeStandardDeviation>`. The tag accepts a query recordset and computes the *standard deviation* of the numbers in one of the query’s columns. Tools like Excel provide built-in functions for computing standard deviations and other common statistical tasks, but CFML does not. The problem can be solved fairly easily with C++, so it makes for a good CFX tag example.

NOTE

If you’re not familiar with what a standard deviation is, it’s a statistical measure commonly used for analyzing data. Basically, the standard deviation can be thought of as a way of describing the similarity of a set of numbers. You’re no doubt already familiar the concept of a mathematical average (also called the mean), right? Well, the standard deviation measures how much the individual numbers tend to deviate from the average. A set of numbers that fluctuates wildly from number to number will have a higher standard deviation; numbers that fluctuate less will have a lower standard deviation. If all the numbers are the same (which is another way of saying that they all conform exactly to the average), the standard deviation is zero. In any case, the standard deviation (and the related concept of variance) forms the foundation of many strategies for analyzing trends in data, so this CFX tag can put you on the path of creating pages that perform relatively sophisticated data analysis.

NOTE

The tag computes the population standard deviation, which means that it is most appropriate for analyzing a complete set of data. You could easily alter it to compute the sample standard deviation instead, which would make it more appropriate for analyzing a random subset of the data.

The attributes for the new tag are listed in Table 30.13. If you want to see how this tag is used in a ColdFusion page, skip ahead to Listing 30.7.

Table 30.13 <CFX_ComputeStandardDeviation> Tag Syntax

ATTRIBUTE	DESCRIPTION
query	The name of a query, such as the result of a <cfquery> tag, that contains numbers for which to compute the standard deviation.
column	The column of the query that contains the numbers for which to compute the standard deviation.
variable	A variable name; the tag will return the computed standard deviation (a single number) in the variable you specify here.

Listing 30.6 shows the C++ source code used to create the <CFX_ComputeStandardDeviation> tag. The code is quite simple. About half of it is concerned only with the actual mathematical computations, which have little to do with the CFX API itself. The remainder (mostly at the top and bottom of the listing) are straightforward CFX calls that use the methods described in the first section of this chapter to exchange information with the ColdFusion page using the tag.

Listing 30.6 Request.cpp—C++ Source Code For the <CFX_ComputeStandardDeviation> Tag

```

////////////////////////////////////
//
// CFX_COMPUTESTANDARDDEVIATION - Cold Fusion custom tag
//
// Copyright 2002. All Rights Reserved.
//

#include "stdafx.h" // Standard MFC libraries
#include "cfx.h" // CFX Custom Tag API
#include "math.h"

void ProcessTagRequest( CCFXRequest* pRequest )
{
    try
    {
        /* BEGIN ATTRIBUTE VALIDATION */

        // Make sure a VARIABLE attribute was actually passed to the tag
        if ( pRequest->AttributeExists("VARIABLE")==FALSE ) {
            pRequest->ThrowException(
                "Error in CFX_ComputeStandardDeviation Tag",
                "No VARIABLE attribute specified.");
        }
    }
}

```

Listing 30.5 (CONTINUED)

```

// Retrieve value of VARIABLE attribute
LPCSTR lpszVariable = pRequest->GetAttribute("VARIABLE") ;

// Make sure a VARIABLE attribute was actually passed to the tag
if ( pRequest->AttributeExists("COLUMN")==FALSE ) {
pRequest->ThrowException(
"Error in CFX_ComputeStandardDeviation Tag",
"No COLUMN attribute specified.");
}

// Retrieve value of VARIABLE attribute
LPCSTR lpszColumn = pRequest->GetAttribute("COLUMN") ;

// Retrieve the query specified in the tag's QUERY attribute
CCFXQuery* pQuery = pRequest->GetQuery();

// Make sure a query was actually passed to the tag
if ( pQuery == NULL ) {
pRequest->ThrowException(
"Error in CFX_ComputeStandardDeviation Tag",
"No QUERY provided.");
}

// Find the index position of the column named in the COLUMN attribute
int colIndex = pQuery->GetColumns()->GetIndexForString( lpszColumn );

// If the column specified in COLUMN attribute does not exist in query
if ( colIndex == CFX_STRING_NOT_FOUND ) {
pRequest->ThrowException(
"Error in CFX_ComputeStandardDeviation Tag",
"The COLUMN attribute does not match up to a column in the query.");
}

/* FINISHED WITH ATTRIBUTE VALIDATION */

// Local variables
double dTotal = 0;
double dMean = 0;
double dStdDev = 0;
double dSumSq = 0;
double dVariance = 0;
int iValueCount = 0;

// Compute the total of all the numbers in the data set
for ( int Row = 1; Row <= pQuery->GetRowCount(); Row++) {

// Get the value (as a string) from the query
LPCSTR thisVal = pQuery->GetData(Row, colIndex);

// Assuming the value is not an empty string
if ( strlen(thisVal) > 0) {

```

Listing 30.5 (CONTINUED)

```
// Add this value to the total
dTotal += atof(thisVal);

// Increment the counter of non-null values
iValueCount++;

// Include optional debug messages for skipped rows
} else if ( pRequest->Debug() ) {
char buffer[50];
sprintf(buffer, "Skipping row %i because value is empty/NULL.", Row);
pRequest->WriteDebug(buffer);

}
}

// The rest of the computations will be dividing by iValueCount,
// so check it to protect ourselves against a divide-by-zero situation
if ( iValueCount > 0 ) {

// Compute the mean (average) of the numbers
dMean = dTotal / iValueCount;

// Begin calculating the variance
for ( Row = 1; Row <= pQuery->GetRowCount(); Row++) {
// Get the value (as a string) from the query
LPCSTR thisVal = pQuery->GetData(Row, colIndex);
// Assuming the value is not an empty string
if ( strlen(thisVal) > 0 ) {
dSumSq += pow( dMean - atof(thisVal), 2);
}
}
// Finish computing the variance
dVariance = dSumSq / iValueCount;

// The standard deviation is the square root of the variance
dStdDev = sqrt(dVariance);
}

// Convert the standard deviation to a string,
// then return it to the calling ColdFusion page
char result[20];
gcvt(dStdDev, 10, result);
pRequest->SetVariable( lpszVariable, result);
}

// Catch Cold Fusion exceptions & re-raise them
catch( CCFXException* e )
{
pRequest->ReThrowException( e ) ;
}

// Catch ALL other exceptions and throw them as
// Cold Fusion exceptions (DO NOT REMOVE! --
// this prevents the server from crashing in
```

Listing 30.5 (CONTINUED)

```

// case of an unexpected exception)
catch( ... )
{
    pRequest->ThrowException(
        "Error occurred in tag CFX_COMPUTESTANDARDDEVIATION",
        "Unexpected error occurred while processing tag." );
}
}

```

As you can see, the basic skeleton of the `ProcessTagRequest()` function has not been changed from the code generated by the Tag Wizard (Listing 30.5). An additional `#include` directive for the standard `math.h` library has been added at the top of the listing; all the other changes are within the body of the `try` block within the `ProcessTagRequest()` function. The `ProcessTagRequest()` function serves the same purpose as the `processRequest()` method in the Java interface; basically, ColdFusion will call this function each time the CFX tag is actually used in a ColdFusion page. Think of this function as being similar conceptually to the ubiquitous `main()` function that would appear in a standalone C++ program.

NOTE

Actually, you can change the name of the `ProcessTagRequest()` function if you wish, as long as you specify the same name in the Procedure field when you register the tag with the ColdFusion Administrator (discussed in the next section). You can even create several custom tags in a single C++ file (and thus in one compiled .dll file), simply by including several functions that have the same basic form as `ProcessTagRequest()` (including the `try` and `catch` blocks as shown in Listing 30.5). However, the customary thing is to leave the name of the function alone and simply create each custom tag as a separate C++ project. I recommend that you do the same unless there is some specific reason why you really want them all compiled into the same DLL.

At the top of `ProcessTagRequest()`, the `AttributeExists()` and `GetAttribute()` methods are used to make sure that the appropriate attributes have been passed to the tag (refer back to Table 30.13). If an attribute is missing, the `ThrowException()` method is used to throw a ColdFusion-style exception, which in turn displays an error message in the calling page (unless the exception is caught with `<cftry>` and `<cfcatch>`).

NOTE

Please refer to the "A Word on Exceptions" section, earlier in this chapter, for a few additional notes on throwing exceptions from CFX tags.

Some of the attribute validation code at the top of this listing involves making sure that a valid ColdFusion query has been passed to the tag. As explained near Table 30.7 (near the beginning of this chapter), the only way to pass a query to a CFX tag is to provide a query attribute to the tag; the query is then accessible to the tag via the `pRequest->GetQuery()` method, as shown in Listing 30.6. Once `GetQuery()` has been used to get a reference to the query, the `GetRowCount()`, `GetData()`, and other methods listed in Table 30.7 can be used to retrieve or modify the data in the query. For instance, this tag uses `pQuery->GetColumns()->GetIndexForString()` to verify that the column name specified in the `COLUMN` attribute of the tag actually exists in the query that was passed to the tag. It also uses `pQuery->GetRowCount()` to make sure that the query includes at least one row.

NOTE

Queries cannot be passed using any attribute name other than `query`; therefore, only one query can be passed to any given CFX tag. In contrast, CFX tags are free to return as many queries as they wish, simply by calling `pRequest->AddQuery()` multiple times (using a different query name each time).

Once the tag's attributes have been reasonably validated, execution continues to the next portion of the code (beginning with the series of `double` declarations). After defining a few local variables, this portion of the code uses two `for` loops to perform the necessary mathematical calculations. Each of the loops iterates from 1 to the number of rows in the query; within the loops, the value of the `Row` variable indicates the row currently being processed. The first thing each of these loops does is to populate a string variable called `thisVal` with the current value of the column specified in the tag's `COLUMN` attribute. It's worth noting that values retrieved from queries are always retrieved as strings; if you want to work with values as numbers, you need to use `atof()` or some similar function to convert the string to the desired numeric data type.

Speaking of which, the first loop uses an `if` test to make sure that the length of the current row's value is not zero (that is, to make sure that the value is not an empty string). If it is, that probably means that a `NULL` is occupying that row of the query. ColdFusion represents null values as empty strings whenever such a value is accessed as string; since `GetData()` always accesses all data as strings, you will always get an empty string for any null value in a query.

If the value is not an empty string, the tag adds the current numeric value to the `dTotal` variable and increments the `iValueCount` variable by one. When the first `for` loop is complete, `dTotal` will hold the total (sum) of all numbers in the data set, and `iValueCount` will hold the number of data points included in the total (that is, the number of rows that were not skipped because of suspected null values).

If, on the other hand, a particular value is determined to be an empty string, the code checks to see if the tag has been called with a `DEBUG` attribute. If so, it uses `pRequest->WriteDebug()` to display a debugging message whenever an empty string is found in the data set. See the "Generating Debug Output" section, later in this chapter, to see what the debugging messages look like.

In any case, as long as at least one valid data point was found in the query, the next line of code computes the mathematical average (mean) of the data points by dividing the `dTotal` by `iValueCount`. The average is stored in the variable called `dMean` (which is not meant to imply that it's demeaning to be average). Next, another `for` loop is used to iterate over the set of numbers again, this time subtracting each value from the mean and squaring the result. After this loop finishes, the average of these results is computed by dividing by the number of data points; this is called the *variance*. The square root of the variance is then computed with the standard `sqrt()` function; this is the standard deviation.

The tag has now completed its work, so the only thing left to do is to return the standard deviation to the calling ColdFusion page using `pRequest->SetVariable()`. Since `SetVariable()` only accepts string values, the standard deviation must first be converted to a string using the standard `gcvt()` function.

NOTE

The `atoi()` function returns 0 if it encounters a string that can't be parsed into a number. It doesn't generate an error. This means that this CFX tag will treat anything other than a number as a 0, which will affect the tag's computations. In other words, the tag is smart enough to skip over null values, but it is not smart enough to skip over other non-numeric data points. This should not be a problem because a column would not typically include both numbers and strings.

Compiling Your New CFX Tag

To compile the tag, simply choose the Build option from the Build menu in Visual C++, or use the Build toolbar button. Assuming there are no syntax or other problems with the code, the compiler should be able to compile the tag in a few seconds. The result will be a Dynamic Link Library (.dll) file, located in either the *project*\Debug or *project*\Release subfolder of Visual Studio's My Projects folder (where the *project* part is most likely the name of your CFX tag).

It's worth noting that you will be building a Debug version of the DLL by default. I recommend switching to the Win32 Release configuration before compiling the tag. You may use the default Win32 Debug configuration if you wish during testing, but you will probably be doing most of your debugging via `pRequest->WriteDebug()` rather than conventional debugging methods, so the Win32 Debug build is unlikely to do you much good in terms of actual debugging. To switch to the Win32 Release configuration, choose Build, Set Active Configuration from the Visual C++ menu.

NOTE

Either type of build should perform the same way in your ColdFusion pages, but the debug version will be larger and cannot be deployed to ColdFusion servers that don't contain Microsoft's debugging symbols. In general, the debugging symbols are only installed on machines that have Visual C++ installed on them, which means that debug builds of CFX tags will probably only work on the ColdFusion server installed on your local machine (if any).

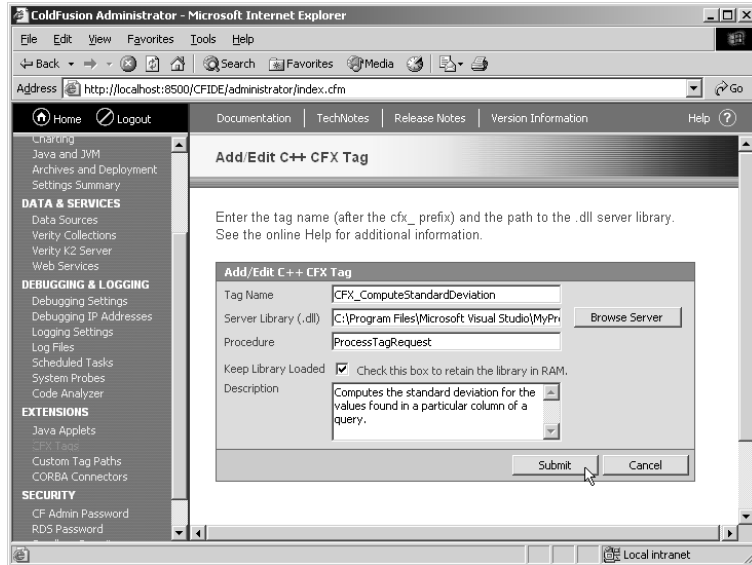
Registering the New Tag

Once the tag has been compiled, you need to register it in the ColdFusion Administrator. The registration process is simple and very similar to the process for Java CFX tags. Just follow these steps:

1. Open the ColdFusion Administrator and navigate to the CFX Tags page.
2. Click the Register C++ CFX button. The Add/Edit C++ CFX Tag page appears (Figure 30.9).
3. In the Tag Name field, enter the name of the tag, including the `CFX_` part. This does not have to be the same as the tag name you used while creating the C++ code, but I recommend that you keep them consistent to keep from getting confused. The tag name is not case sensitive.
4. In the Server Library field, provide the location of the .dll file that represents the compiled version of your CFX tag. If you are registering the tag on a ColdFusion server installed on the same machine that you compiled the tag on, you can just specify the DLL's current location within Visual Studio's directory tree. That way, whenever you rebuild the tag, the server will automatically be looking at the newly built version. If the server is not on your local development machine, you will need to copy the DLL to the server's local drive first (you can place the DLL in whatever location you wish).

Figure 30.9

Like Java CFX tags, tags written in C++ need to be registered in the ColdFusion Administrator.



5. Leave the Procedure field set to ProcessTagRequest unless you changed the name of the ProcessTagRequest() function produced by the ColdFusion Tag Wizard when you created your project's workspace.
6. Leave the Keep Library Loaded unless your DLL will occupy a large amount of space in the server's memory and will be used infrequently. Another reason to uncheck this box is if your DLL is not thread safe. See the "Using CFX Tags Safely: Locking and Thread Safety" section later in this chapter for details.
7. If you wish, fill in the optional Description field.
8. Press OK to register the tag. It will now appear in the list of registered tags in the CFX Tags page, and is ready for use in your ColdFusion pages.

Using the New Tag

Listing 30.7 shows how the new <CFX_ComputeStandardDeviation> tag can be used in a ColdFusion page. First, data from the Films table is retrieved using an ordinary SELECT query. Then the CFX tag is called, specifying the RatingID column of the query as the column for which to compute the standard deviation. After the tag executes, the computed standard deviation will be available to ColdFusion in the FilmsStdDev variable. Similar steps are used to compute the standard deviation of the prices in the Merchandise table.

Listing 30.7 ComputeStandardDeviationDemo.cfm—Computing the Standard Deviation

```
<!---
Filename: ComputeStandardDeviationDemo.cfm
Author:   Nate Weiss (NMW)
Purpose:  Computes standard deviations
```

Listing 30.7 (CONTINUED)

```

--->

<!---
Retrieve film information from database
--->
<cfquery datasource="ows"
           name="GetFilms">
  SELECT * FROM Films
</cfquery>

<!---
Compute the standard deviation of the ratings for each film
--->
<CFX_ComputeStandardDeviation query="GetFilms"
                             column="RatingID"
                             variable="FilmsStdDev">

<!---
Retrieve merchandise information from database
--->
<cfquery datasource="ows"
           name="GetMerch">
  SELECT * FROM Merchandise
</cfquery>

<!---
Insert empty row to prove tag is smart enough to skip NULL values
--->
<cfset QueryAddRow(GetMerch)>

<!---
Compute the standard deviation of the merchandise prices
--->
<CFX_ComputeStandardDeviation query="GetMerch"
                             column="MerchPrice"
                             variable="MerchStdDev">

<html>
<head>
<title>Orange Whip Statistics</title>
</head>

<body>
<h2>Orange Whip Statistics</h2>

<!---
Display the standard deviation
--->
<cfoutput>
  <p>
    Standard deviation of the films' ratings:
    <strong>#NumberFormat(FilmsStdDev, "0.00")#</strong>
  <br>

```

Listing 30.7 (CONTINUED)

```
<p>
Standard deviation of the merchandise prices:
<strong>#NumberFormat(MerchStdDev, "0.00")#</strong>
<br>

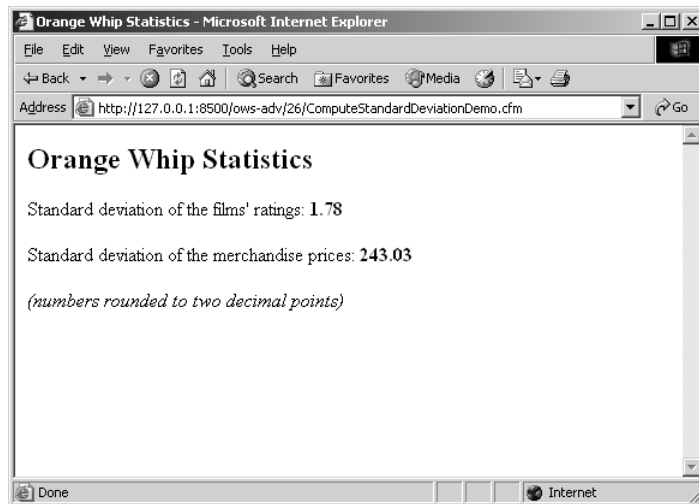
<p>
<emp>(numbers rounded to two decimal points)</emp>
<br>
</cfoutput>

</body>
</html>
```

Because the merchandise prices vary greatly (there are only about a dozen data points in the sample database, but the prices range from \$7.50 to \$950.00), we will expect to see a relatively high standard deviation. Conversely, since the film ratings vary only slightly (many of the rating values are exactly the same from film to film, and the total range is only from 1 to 6), we will expect to see a low standard deviation. Happily, the page generated by Listing 30.7 is consistent with these expectations (Figure 30.10).

Figure 30.10

CFX tags can return CFML variables that can be displayed like any other variable.

**Other C++ Examples**

Other examples of CFX tags created with C++ can be found in the CFusionMX7\cfx\examples folder that gets installed automatically with ColdFusion. The examples include a `<CFX_DirectoryList>` tag that operates similarly to the built-in `<cfdirectory>` tag, and a `<cfx_nt_userdb>` tag that allows you to add and remove users from a Windows domain or workgroup.

Generating Debug Output

If your CFX tag encounters any problems during its execution, or if you need help figuring out why your tag isn't doing what you expect it to, you can have your tag include debug messages in the calling template. Debug messages can include whatever text you want; usually you use them to display some kind of diagnostic information that indicates whether the tag was able to connect to some kind of data source, was denied access to some kind of file, or encountered some other type of unexpected condition.

In the `<CFX_ComputeStandardDeviation>` example, the tag is expecting to find only numbers in the query column passed to the tag. It knows to skip over null values, and generally does so silently. But since it's avoiding a potential problem, you might want to be able to see which values are being skipped over. The code in Listing 30.6 uses the following lines to output debug messages:

```
if ( pRequest->Debug() ) {
    char buffer[50];
    sprintf(buffer, "Skipping row %i because value is empty/NULL.", Row);
    pRequest->WriteDebug(buffer);
}
```

The `pRequest->Debug()` method always returns `FALSE` unless the tag is being called in debug mode, so these lines are usually skipped and thus add almost no overhead to normal tag execution. To call a CFX tag in debug mode, add a debug flag (attribute) to the CFX tag when you use it, like so:

```
<!-- Compute the standard deviation of the merchandise prices -->
<CFX_ComputeStandardDeviation query="GetMerch"
    column="MerchPrice"
    variable="MerchStdDev"
    debug>
```

Whenever a tag is called with the debug flag, the `pRequest->Debug()` method (or the `request.debug()` method if you're using Java) always returns `TRUE`, and the `WriteDebug()` method includes the specified text along with the tag name when the page is visited with a browser (Figure 30.11). You can use this simple but effective debugging facility whenever you want to include special messages for developers but not for the rest of the world.

NOTE

`debug` behaves like a flag rather than a Boolean (Yes or No) attribute, so the only thing that matters is whether it is present. The value after the = sign is ignored, so `debug="Yes"` and `debug="No"` both result in debug messages being shown. You must remove the `debug` flag altogether to stop the messages.

NOTE

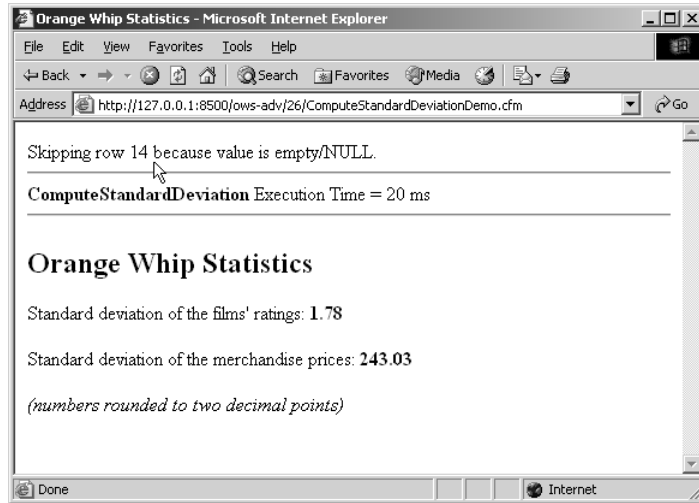
The `debug` functionality is unfortunately not integrated with the Debugging Settings page of the ColdFusion Administrator, so the settings on that page have no bearing on whether debug messages from CFX tags are shown. In particular, debug messages from CFX tags are not restricted by the Debugging IP Addresses list in the Administrator.

NOTE

The CFX API for Java also includes an additional set of classes for debugging Java CFX tags. You can use the classes to test CFX tags without actually supplying them with runtime attributes in ColdFusion pages. If you find this idea interesting, consult the "ColdFusion Java CFX Reference" section of the CFML Reference that ships with ColdFusion.

Figure 30.11

CFX tags can include debugging messages if unexpected conditions arise.



Returning Structures from CFX Tags

As you probably already know, ColdFusion supports a feature referred to in the documentation as Smart Structs. The idea behind the Smart Structs feature is to make it easier to construct complex data structures in CFML code. Basically, if you use `<cfset>` or some other mechanism to create a variable, and the variable name contains dots or square brackets such that the new variable would be nested within a structure of structures, ColdFusion will automatically create any intermediary structures necessary to allow the `<cfset>` to execute without errors.

To illustrate, consider the following line:

```
<CFSET OrangeWhip.Actresses.HotList.July = "Belinda Foxile">
```

This line will execute successfully even if the `OrangeWhip` structure does not exist yet. The same goes for the `Actresses` or `HotList` parts.

CFX Tags Are Smart, Too

So what does this have to do with CFX tags? It turns out that the CFX interface uses the same internal functions within the ColdFusion server to set variables. So, if you supply a variable name that contains dots to `response.setVariable()` (in Java) or `Request->SetVariable()` (in C++), ColdFusion will automatically create the intermediary structures needed to set the variable with the name you specify.

For instance, the code listings for this chapter include a file called `Request2.cpp`, which contains the source code for a new CFX tag called `<CFX_ComputeStatistics>`. This listing is very similar to the `<CFX_ComputeStandardDeviation>` tag created earlier in this chapter and takes the same attributes. The difference is that instead of returning a single number, it returns a structure with five keys (values), named `StandardDeviation`, `Variance`, `Mean`, `Sum`, and `Count` (the number of non-null data points).

Aside from the name of the tag, the code is exactly the same as Listing 30.6 except that these lines:

```
// Convert the standard deviation to a string,
// then return it to the calling ColdFusion page
char result[20];
gcvt(dStdDev, 10, result);
pRequest->SetVariable( lpszVariable, result);
```

have been replaced with the following lines:

```
// Return statistics to the calling ColdFusion page
char result[20];
gcvt(dStdDev, 10, result);
pRequest->SetVariable(lpszVariable + ".StandardDeviation", result);
gcvt(dVariance, 10, result);
pRequest->SetVariable(lpszVariable + ".Variance", result);
gcvt(dMean, 10, result);
pRequest->SetVariable(lpszVariable + ".Mean", result);
gcvt(dTotal, 10, result);
pRequest->SetVariable(lpszVariable + ".Sum", result);
gcvt(iValueCount, 10, result);
pRequest->SetVariable(lpszVariable + ".Count", result);
```

NOTE

Because it is a bit long and because all the other lines are the same, I am not including the `<CFX_ComputeStatistics>` source code as a separate printed listing here. However, the `Request2.cpp` file is included with this chapter's listings, and the Windows-compiled version of the finished tag (`CFX_ComputeStatistics.dll`) is included as well.

Listing 30.8 shows how the new `<CFX_ComputeStatistics>` tag can be used in a ColdFusion page. The resulting Web page shows the number of data points, the average, the statistical variance, and the standard deviation for each set of data (Figure 30.12). Note that either dots or square brackets can be used to refer to the members of the returned structure, just like any other CFML structure.

Listing 30.8 ComputeStatisticsDemo.cfm—Using Structures Returned by a CFX Tag

```
<!---
  Filename: ComputeStatisticsDemo.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Computes standard deviations
  --->

<!---
Retrieve film information from database
--->
<cfquery datasource="ows"
           name="GetFilms">
  SELECT * FROM Films
</cfquery>

<!---
Compute statistics for the ratings of each film
--->
<CFX_ComputeStatistics query="GetFilms"
                       column="RatingID"
                       variable="FilmsStats">
```


Listing 30.8 (CONTINUED)

```
<!---
Retrieve merchandise information from database
-->
<cfquery datasource="ows"
           name="GetMerch">
    SELECT * FROM Merchandise
</cfquery>

<!---
Insert empty row to prove tag is smart enough to skip NULL values
-->
<cfset QueryAddRow(GetMerch)>

<!---
Compute statistics for the merchandise prices
-->
<CFX_ComputeStatistics query="GetMerch"
                      column="MerchPrice"
                      variable="MerchStats">

<html>
<head>
<title>Orange Whip Statistics</title>
</head>

<body>
<h2>Orange Whip Statistics</h2>

<!--- Display the statistics --->
<cfoutput>
  <!--- First for films --->
  <h3>Statistics for the film ratings</h3>
  Number of data points:
  <strong>#NumberFormat(FilmsStats["Count"], "0")#</strong>
  <br>
  Average (Mean):
  <strong>#NumberFormat(FilmsStats["Mean"], ",0.00")#</strong>
  <br>
  Variance:
  <strong>#NumberFormat(FilmsStats.Variance, ",0.00")#</strong>
  <br>
  Standard Deviation:
  <strong>#NumberFormat(FilmsStats.StandardDeviation, ",0.00")#</strong>
  <br>

  <!--- Then for merchandise --->
  <h3>Statistics for the merchandise prices</h3>
  Number of data points:
  <strong>#NumberFormat(MerchStats["Count"], "0")#</strong>
  <br>
  Average:
  <strong>#NumberFormat(MerchStats["Mean"], ",0.00")#</strong>
  <br>
  Variance:
  <strong>#NumberFormat(MerchStats.Variance, ",0.00")#</strong>
```

Listing 30.8 (CONTINUED)

```

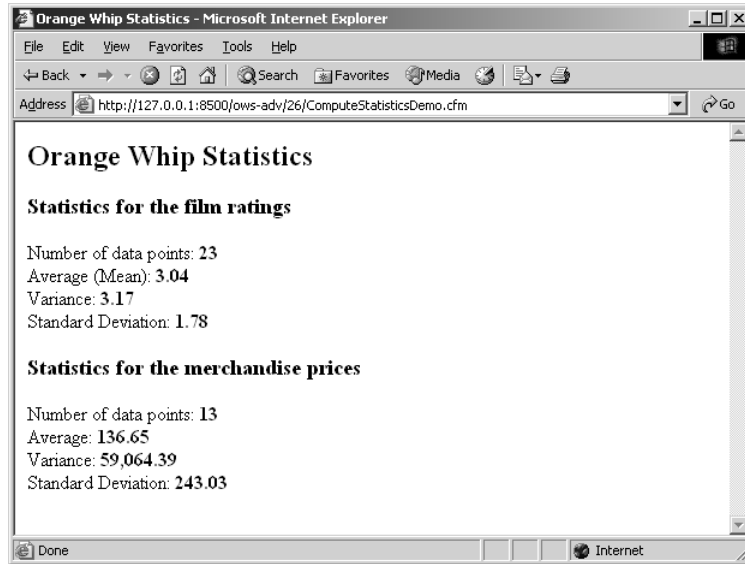
<br>
Standard Deviation:
<strong>#NumberFormat(MerchStats.StandardDeviation, ",0.00")#</strong>
<br>
</cfoutput>

</body>
</html>

```

Figure 30.12

CFX tags can return multifaceted information to ColdFusion.

**An Important Warning**

If this all sounds too good to be true, it is—kind of. This whole strategy will fail if there is already a variable called `FilmsStats` in the calling ColdFusion page and that variable is anything other than a structure. In such a case, ColdFusion will crash if a C++ CFX attempts to reference the variable as a structure. With a Java CFX, the program won't crash, but the tag will still throw an error. To protect yourself against such a possibility, you could use a line like one of the following, just before calling the CFX tag each time.

This line simply deletes the `FilmsStats` variable if it exists already (if it doesn't exist, nothing happens, so this line does no harm):

```
<cfset StructDelete(VARIABLES, "FilmsStats")>
```

Alternatively, this line creates a new, empty `FilmsStats` structure (regardless of whether it already exists):

```
<cfset FilmsStats = StructNew()>
```

Either line of code, used immediately before the CFX tag, will guard against any potential problems if the CFX tag tries to add nested structures to an existing variable that is not itself a structure. Whew, that was close!

Using CFX Tags Safely: Locking and Thread Safety

Because the ColdFusion server responds to page requests in a multithreaded fashion (meaning that it can process more than one page request at the same time), it is possible that two instances of a CFX tag may execute on the same server at the same time. If the CFX tag is not thread-safe in terms of how it deals with memory or how it works logically, problems might arise unless you take steps to avoid them. Like other concurrency issues, such problems are likely to show up only under load, making it appear that the tag (or ColdFusion itself) does not scale well, when in fact the problem could be avoided by making sure the tag does not execute in more than one page request at once.

Understanding Thread Safety

A full discussion of what it means for a compiled program to be thread-safe is well beyond the scope of what can be explained in this chapter. For purposes of this discussion, it will have to suffice that *thread-safe* basically means that the tag has been coded in such a way that it can run in several threads at the same time without any possibility of the various threads being able to create, change, edit—or, in some instances, merely access—a shared resource (where *shared resource* means any shared variable, memory, file, or the like).

Generally speaking, a CFX tag is probably thread-safe if it does not use any global variables (that is, if none of the variables are declared outside of the `processRequest()` function in Java or the `ProcessTagRequest()` function in C++) and also does not access any external or third-party APIs that themselves are not thread-safe.

NOTE

If you're not familiar with what a thread is, don't worry about it too much right now. In the context of ColdFusion, a thread is basically the same thing conceptually as a page request from a user (because each simultaneous page request is processed by a different worker thread within the server).

Locking CFX Tags with `<cflock>`

You can still use CFX tags even if you know they aren't thread-safe. Just wrap the `<cflock>` tag around every use of the tag, using `type="Exclusive"` and a name attribute equal to the name of the CFX tag (or some other name, as long as it is always the same for every single use of the tag). For instance, if you knew (or suspected) that the `<CFX_ComputeStatistics>` tag was not thread-safe, you would use code similar to the following:

```
<cflock type="Exclusive"
        name="CFX_ComputeStatistics"
        timeout="10">
  <!-- Compute statistics for the ratings of each film -->
  <CFX_ComputeStatistics query="GetFilms"
                        column="RatingID"
                        variable="FilmsStats">
</cflock>
```

If the CFX Tag Works with Files

If the guts of the CFX tag are thread-safe, but you are asking the tag to do something like create or manipulate a file on the server's drive, then you might want to lock access to the file, rather than to the tag itself. The `<cflock>` tags would still surround the CFX tag in your ColdFusion code, but you would most likely use the complete filename as the `NAME` of the lock, rather than the tag. If the tag is merely reading the file, you could use `type="ReadOnly"`; if the tag might be creating or changing the file, you would have to use `type="Exclusive"`.

You would also want to surround other portions of your CFML code that access the same file—regardless of whether they involve CFX tags—with `<cflock>` tags. The `<cflock>` tags should have the same name as the ones around the CFX tag itself, and should use `type="Exclusive"` for creation or modifications and `type="ReadOnly"` if merely reading the file.

Keeping the Page Itself in Mind

It's worth noting that just because a tag is thread-safe internally, that doesn't automatically mean that a ColdFusion page's *use* of the tag is thread-safe. For instance, if you were supplying `APPLICATION` or `SESSION` variables to the various attributes of a CFX tag, or if you were asking the CFX tag to set variables in the `APPLICATION` or `SESSION` scope, then it is theoretically possible for a logical “race condition” problem to occur if one page request changes those variables at the same time, perhaps resulting in some kind of data loss or logical data corruption. If the nature of your application is such that this could be a problem (and only you can determine that), you should continue to lock.

NOTE

As you probably already know, the need for locking dramatically decreased as of ColdFusion MX. That doesn't change the fact that any access to shared resources in CFML pages—whether dealing with CFX tags or not—must be locked if the nature of the application logic (that means your stuff, not ColdFusion's internals) is such that problems could arise if multiple threads execute the code at the same time. For more details on this topic, see the “Introducing the Web Application Framework” chapter in our companion volume, *The Macromedia ColdFusion MX 7 Web Application Construction Kit* (Macromedia Press).

Using CFX Tags with Previous Versions of ColdFusion

CFX tags are not something that can only be used with current versions of ColdFusion. The CFX API was actually the means of extending the server introduced first in the product's history, predating CFML custom tags, user-defined functions, and CFCs.

CFX tags written with C++ can be used with any version of the product after (drumroll, please) ColdFusion 2.0.

CFX tags written with Java can be used with ColdFusion 4.5 or later without any additional software. You can use Java CFX tags with version 4.0 using the `<CFX_J>` add-on package, which was freely available from Macromedia's Web site as of this writing.

Of course, the exception to this rule is the newfound ability for CFX tags to return structures to ColdFusion pages (which was first introduced in ColdFusion MX). See the “Returning Structures from CFX Tags” section earlier in this chapter for details.