CHAPTER 30

Working with Stored Procedures

IN THIS CHAPTER

Most server-based database systems—SQL Server, Oracle, and Sybase—support *stored procedures*. A stored procedure is a chunk of SQL code that's given a name and stored as a part of your database, along with your actual data tables. After the stored procedure has been created, you can invoke it in your ColdFusion templates using the `<CFSTOREDPROC>` tag.

NOTE

The use of stored procedures in database applications is a relatively advanced topic. As the saying goes, it's not exactly rocket science, but it probably makes sense to start working with stored procedures only after you have become pretty familiar with the SQL concepts introduced in the previous chapter or have a specific reason for using stored procedures in a particular ColdFusion application.

NOTE

At the time of this writing, stored procedures are supported by most server-based database systems (such as SQL Server, Oracle, and Sybase) but are not generally supported by file-based databases (such as Access, FoxPro, Paradox, dBASE, and so on). If you don't plan on using a server-based database system, you can skip this chapter without missing out on anything essential.

Why Use Stored Procedures?

Stored procedures provide a way to consolidate any number of SQL statements—such as `SELECT`, `INSERT`, `UPDATE`, and so on—into a little package that encapsulates a complete operation to be carried out in your application.

For instance, consider Orange Whip Studio's online store. When a customer places an order, the application needs to verify that the customer's account is in good standing and then carry out `INSERT` statements to the `MerchandiseOrders` and `MerchandiseOrdersItems` tables to record the actual order. In the future, the application might be expanded to first ensure that the selected merchandise is in stock. If not, the application would need to display some type of "sorry, out of stock" message for the user, and it might need to update another table somewhere else to indicate that the merchandise needs to be reordered from the supplier.

This type of complex, causally related set of checks and record-keeping is often referred to generally as a business rule or business process. Using the techniques you've learned so far in this book, you already know that you could accomplish the steps with several <CFQUERY> tags and some conditional processing using <CFIF> and <CFELSE> tags. In fact, in Chapter 29, "Online Commerce," you learned how all the relevant processing can be bundled up into a CFML custom tag called <CF_PlaceOrder>.

In this chapter, you see that you could wrap up all the database-related actions required to place an order into a single stored procedure called, say, PlaceOrder, thus encapsulating the entire business process into one smart routine your ColdFusion templates need only refer to.

This shifts the responsibility for ensuring that the steps are followed properly away from your ColdFusion code and places that responsibility in the hands of the database server itself. This shift generally requires a little bit of extra work on your part up front but offers some real advantages later.

Advantages of Using Stored Procedures

Depending on the situation, using a stored procedure in your application (rather than a series of <CFQUERY> and <CFIF> tags) can offer a number of significant advantages. I will introduce you to the most common advantages in this section. You might want to consult your database server documentation for further details and remarks on the advantages of using stored procedures.

Advantage: Modularity Is a Good Thing

When developing any type of application, it generally pays to keep pieces of code—whether it be CFML code, SQL statements, or some other type of code—broken into small, self-explanatory modules that know how to perform a specific task on their own. This type of practice keeps your code readable, easier to maintain, and easier to reuse in other applications you might need to write later. You learned how to do this for CFML code in Chapter 19, "Building User Defined Functions" and Chapter 20, "Building Reusable Components".

Keeping your code in separate, abstract chunks can also enable several people to more easily work on the same project at the same time without stepping on each other's toes. With respect to stored procedures specifically, you might have a scenario where Developer A says to Developer B, "Make me a stored procedure that does such-and-such. Meanwhile, I'll be putting together the ColdFusion templates that use the procedure." Because the stored procedure runs independently of the templates and vice versa, neither developer needs to wait for the other to get started. The result can be an application that gets up and running more quickly.

Advantage: Sharing Code Between Applications

After a stored procedure has been created, it enables you to share code between different types of applications, even applications written with different development tools. For instance, a ColdFusion template, a Visual Basic program, and a Java application might all refer to the same stored procedure on your server. Clearly, this cuts down on development time and ensures that all three applications enforce the various business rules in exactly the same way.

This type of consistent enforcement of business rules can be particularly important if the three applications are being developed by different teams or developers. In addition, if the business rules change for order-taking in the future, it's likely that only the stored procedure would need to be adapted, rather than the code in all three applications needing to be revised.

Advantage: Increasing Performance

Depending on the situation, using stored procedures can often cause an application to perform better. There are two ways in which stored procedures can help speed your application.

First, most database systems do some type of precompilation of the stored procedure so that it runs more quickly when it is actually used (this is analogous to how ColdFusion MX compiles your ColdFusion pages into Java classes for you). For instance, Microsoft SQL Server makes all its performance-optimizing decisions (such as which indexes and which join algorithms to use) the first time a stored procedure is run. Subsequent executions of the stored procedure do not need to be parsed and analyzed, which causes the procedure to run somewhat more quickly than if you executed its SQL statements in an ad hoc fashion every time. Generally, the more steps the procedure represents, the more of a difference this precompilation makes. Oracle servers do something very similar.

Second, if you compare the idea of having one stored procedure versus several `<CFQUERY>` and `<CFIF>` tags in a template, the stored procedure approach is often more efficient because much less communication is necessary between ColdFusion and the database server. Instead of sending the information about the number of books in stock and customer account status back and forth between the ColdFusion server and the database server, all the querying and decision-making steps are kept in one place. Because less data needs to move between the two systems, and because the database drivers aren't really involved at all until the very end, you are likely to have a slightly faster process in the end if you use the stored-procedure approach. Generally, the more data that needs to be examined to enforce the various business rules, the more of a difference the use of stored procedures is likely to have on overall application performance.

NOTE

In general, the performance increases mentioned here will be relatively modest. Of course, your mileage will vary from application to application and from database to database, but you should expect speed increases of 10% or so as a ballpark figure in your mind. In other words, don't expect your application to work 20 times faster just because you put all your SQL code into stored procedures.

Advantage: Making Table Schemas Irrelevant

Because a stored procedure is invoked simply by referring to it by name, a database administrator can shield whoever is actually using a stored procedure from having to be familiar with the database tables' underlying structure. In fact, in a team environment, it's easy to imagine a situation in which you are developing a ColdFusion application but don't even know the names of the tables in which the various data is being stored.

You can get your work done more quickly if whoever designed the database provides you with a number of stored procedures you can use to get your job done without having to learn the various table and column names involved. In return, the database designer can be comforted by the knowledge that if the relationships between the tables change at some point in the future, only the stored procedure will need to be changed, rather than the ColdFusion code you're working on.

Advantage: Addressing Security Concerns

Depending on the type of database server you're using, stored procedures can provide advantages in terms of security. For instance, you might have some confidential data in some of your database tables, so the database administrator might not have granted `SELECT` or `INSERT` privileges to whatever database username ColdFusion uses to interact with the database server. By creating a few stored procedures, your administrator could provide the needed information to ColdFusion without granting more general privileges than are necessary.

By only granting privileges to the stored procedures rather than enabling the actual tables to be queried or updated, the database administrator can be confident that the data in the tables does not become compromised or lose its real-world meaning. Because all database servers enable the administrator to grant `SELECT`, `UPDATE`, `DELETE`, and `INSERT` privileges separately, the administrator could enable your ColdFusion application to retrieve data but make changes only via stored procedures. That gives the administrator a lot of control and keeps you from worrying about harming the organization's data because of some mistake in your code.

Such a policy would enable the ColdFusion developer, the database administrator, and the company in general to feel very confident about the development of a ColdFusion application. Everyone involved can see that nothing crazy is going to happen as the company shifts toward Web-based applications.

Comparing Stored Procedures to CFML Custom Tags

It's worth noting that stored procedures are similar in many conceptual ways to CFML custom tags (also known as modules), which you learned about in Chapter 20, "Building Reusable Components." You can think about a stored procedure as the database server equivalent of a custom tag, which might help you get a handle on when you should consider writing a stored procedure.

Here are some of the ways stored procedures are similar to custom tags:

- Both stored procedures and custom tags enable you to take a bunch of code, wrap it up, slap a name on it, and use it as you develop your applications almost as if it were part of the language all along.
- Both encourage code reuse, cutting development times and costs in the long run.
- Both can accept parameters as input.
- Both can set ColdFusion variables in the calling template that can be used in subsequent CFML code.

- Both can generate one or more query result sets for the calling template.
- Both make integrating other people's work into your own easier.

Calling Stored Procedures from ColdFusion Templates

Now that you have an idea about what kinds of things stored procedures can be used for, this is a good time to see how to integrate them into your ColdFusion templates. For the moment, assume that several stored procedures have already been created and are ready for use within your ColdFusion application. Perhaps you put the stored procedures together yourself, or maybe they were created by another developer or by the database administrator (DBA).

At this point, all you care about is what the name of the stored procedure is and what it does. You learn how to actually create stored procedures later in this chapter.

Two Ways to Execute Stored Procedures

There are two ways to execute a stored procedure from your ColdFusion templates. You can use the <CFSTOREDPROC> tag, which is obviously designed specifically for stored procedures, or can use the <CFQUERY> tag that you already know and love.

With the <CFSTOREDPROC> Tag

The <CFSTOREDPROC> tag is the formal, recommended way to call stored procedures. You can pass input parameters to the stored procedure, collect return codes and output parameters passed back by the procedure, and use any recordsets (queries) the procedure returns. In theory, <CFSTOREDPROC> should also result in the fastest performance.

However, using <CFSTOREDPROC> has two disadvantages:

- You can't use ColdFusion's query-caching feature with the recordsets the procedure returns.
- For some data sources, the parameters you pass to the stored procedure via <CFSTOREDPROC> are referenced by ordinal position, rather than by name. This means your ColdFusion code can break if parameters are added to the procedure in the future. This is a particularly unfortunate limitation and can be enough to keep you from using the tag altogether, especially if you are using Microsoft SQLServer. For details, see the section "Ordinal Versus Named Parameter Positioning," later in this chapter.

With the <CFQUERY> Tag

As an alternative to <CFSTOREDPROC>, you can execute stored procedures via ordinary <CFQUERY> tags. Recordsets returned by stored procedures called in this way can be cached via ColdFusion's CACHEDWITHIN attribute (see Chapter 22, "Improving Performance"). Also, this method allows you to refer to the procedure's parameter by name instead of by ordinal position.

Unfortunately, this method has two big disadvantages as well:

- You can't capture any output parameters or result codes returned by the procedure (only query-style output). You'll learn what these items are later in this chapter, in the "Stored Procedures That Take Parameters and Return Status Codes" section.
- If the procedure returns multiple recordsets, you can capture only one of them for use in your ColdFusion code. See the section, "Calling Procedures with <CFQUERY> Instead of <CFSTOREDPROC>," later in this chapter.

Using the <CFSTOREDPROC> Tag

To call an existing stored procedure from a ColdFusion template, you refer to the procedure by name using the <CFSTOREDPROC> tag. The <CFSTOREDPROC> tag is similar to the <CFQUERY> tag in that it knows how to interact with data sources you've defined in the ColdFusion Administrator. However, rather than accepting ad hoc SQL query statements (such as SELECT and DELETE), <CFSTOREDPROC> is very structured, optimized specifically for dealing with stored procedures.

The <CFSTOREDPROC> tag takes a number of relatively simple parameters, as listed in Table 30.1.

Table 30.1 Important <CFSTOREDPROC> Attributes

ATTRIBUTE	PURPOSE
PROCEDURE	The name of the stored procedure you want to execute. You usually can just provide the procedure name directly, as in PROCEDURE="MyProcedure". Depending on the database system you are using, however, you might need to qualify the procedure name further using dot notation.
DATASOURCE	The appropriate data source name. Just like the DATASOURCE attribute for <CFQUERY>, this can be the name of any data source listed in the ColdFusion Administrator.
RETURNCODE	Optional. Yes or No. This attribute determines whether ColdFusion should capture the status code (sometimes called the return code or return value) reported by the stored procedure after it executes. If you set this attribute to Yes, the status code will be available to you in a special variable called CFSTOREDPROC.StatusCode. See the section "Stored Procedures That Take Parameters and Return Status Codes," later in this chapter.

NOTE

The <CFSTOREDPROC> tag also supports USERNAME, PASSWORD, BLOCKFACTOR, and DEBUG attributes. All these attributes work similarly to the corresponding attributes of the <CFQUERY> tag. See Appendix B, "ColdFusion Tag Reference," for details.

For simple stored procedures, you can just use its PROCEDURE and DATASOURCE attributes. When the template is executed in a Web browser, the stored procedure executes on the database server, accomplishing whatever it was designed to accomplish as it goes.

For instance, suppose you have access to an imaginary stored procedure called `PerformInventoryMaintenance`. It has been explained to you that this stored procedure performs some type of internal maintenance on the data in the `Merchandise` table, and that people need some way to execute the procedure via the company intranet. Listing 30.1 shows a ColdFusion template called `InventoryMaintenanceRun.cfm`, which does exactly that.

NOTE

The code listings in this chapter refer to a data source called `owsSqlServer` to indicate a copy of the `ows` sample database sitting on a Microsoft SQLServer or Sybase server, and a data source called `owsOracle` to indicate a version of the database sitting on an Oracle server. You can't work through the examples in this chapter using the Access version of the `ows` database used elsewhere in this book (because Access doesn't support stored procedures). Microsoft and Oracle each provide migration tools that can import the Access version of the `ows` database into the database server. See the section titled "Creating Stored Procedures," later in this chapter.

Listing 30.1 `InventoryMaintenanceRun.cfm`—Calling a Stored Procedure with `<CFSTOREDPROC>`

```
<!---
  Filename: InventoryMaintenanceRun.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of the <CFSTOREDPROC> tag
--->


<HTML>
<HEAD><TITLE>Inventory Maintenance</TITLE></HEAD>
<BODY>
<H2>Inventory Maintenance</H2>

<!--- If the submit button was not just pressed, display form --->
<CFIF NOT IsDefined("Form.ExecuteNow")>
  <!--- Provide button to start stored procedure --->
  <CFFORM ACTION="#CGI.SCRIPT_NAME#" METHOD="POST">
     <INPUT TYPE="SUBMIT" NAME="ExecuteNow" VALUE="Perform Inventory Maintenance">
  </CFFORM>

<!--- If the user just clicked the submit button --->
<CFELSE>
   <P>Executing stored procedure...</P>

   <!--- Go ahead and execute the stored procedure --->
   <CFSTOREDPROC
     PROCEDURE="PerformInventoryMaintenance"
     DATASOURCE="owsSqlServer">

   <P>Done executing stored procedure!</P>

</CFIF>

</BODY>
</HTML>
```

As you can see, the code to execute the stored procedure is extremely simple. You see in a moment how to use stored procedures that receive input and respond by providing output back to you. This particular stored procedure doesn't require any information to be provided to it, so the only things you must specify are the PROCEDURE and DATASOURCE parameters.

The PROCEDURE parameter tells ColdFusion what the name of the stored procedure is. The DATASOURCE parameter works just like it does for the <CFQUERY> tag; it must be the name of the appropriate data source as defined in the ColdFusion Administrator.

NOTE

Listing 30.1 uses a simple piece of <CFIF> logic that tests to see whether a form parameter called ExecuteNow exists. Assuming that it does not exist, the template puts a simple form–with a single submit button–on the page. Because the submit button is named ExecuteNow, the <CFIF> logic executes the second part of the template when the form is submitted; the second part contains the <CFSTOREDPROC> tag that executes the stored procedure. The template is put together this way so you can see the form code and the form-processing code in one listing. See Chapter 13, "Using Forms to Add or Change Data," for further discussion of this type of single-template technique.

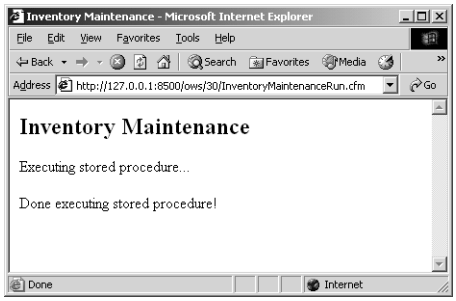
When the InventoryMaintenanceRun.cfm template from Listing 30.1 is first brought up in a Web browser, it displays a simple form (Figure 30.1). When the Perform Inventory Maintenance button is clicked, the procedure is executed and a simple message is displayed to the user to indicate that the work has been done (Figure 30.2).

Figure 30.1

The Inventory MaintenanceRun.cfm template first displays a simple form.

Figure 30.2

The stored procedure executes when the form is submitted.



Stored Procedures That Return Recordsets

Stored procedures also can return recordsets to your ColdFusion templates. As far as your ColdFusion code is concerned, recordsets are just like query results. They contain columns and rows of data—usually from one or more of the database's tables—that you can then use the way you'd use rows of data fetched by a <CFQUERY> tag.

For instance, consider a stored procedure called FetchRatingsList that returns rating information straight from the FilmsRatings table in the owsSqlServer database. This stored procedure sends back its information as a recordset, just as if you had performed a simple SELECT type of query with a <CFQUERY> tag.

It has been explained to you by whomever created the stored procedure that the recordset will contain two columns: RatingID and Rating, which correspond to the columns of the FilmsRatings table. In other words, executing this particular stored procedure is similar to running an ordinary SELECT * FROM FilmsRating query (you will see more complex examples shortly).

The <CFPROCRESULT> Tag

To use a recordset returned by a stored procedure, you must use the <CFPROCRESULT> tag, which tells ColdFusion to capture the recordset as the stored procedure is executed. The <CFPROCRESULT> tag takes just a few attributes and is easy to use.

Table 30.2 lists the attributes supported by <CFPROCRESULT>. Most of the time, you will need to use only the NAME attribute.

Table 30.2 <CFPROCRESULT> Tag Attributes

ATTRIBUTE	PURPOSE
NAME	A name for the recordset. The recordset will become available as a query object with whatever name you specify; the query object can be used just like any other (like the results of a <CFQUERY> or <CFPOP> tag).
RESULTSET	An optional number that indicates which recordset you are referring to. This attribute is important only for stored procedures that return several recordsets. See the section, "Multiple Recordsets Are Fully Supported," later in this chapter. Defaults to 1.
MAXROWS	Optional; similar to the MAXROWS attribute of the <CFQUERY> tag. The maximum number of rows ColdFusion should retrieve from the database server as the stored procedure executes. If not provided, ColdFusion simply retrieves all the rows.

TIP

Because a recordset captured from a stored procedure is made available to your ColdFusion templates as an ordinary CFML query object, the query object has the same RecordCount, ColumnList, and CurrentRow properties that the result of a normal <CFQUERY> tag would. You can use these to find out how much data the recordset contains. See Appendix D, "Special ColdFusion Variables and Result Codes," for details about these properties.

NOTE

The <CFPROCRESULT> tag must be used between opening and closing <CFSTOREDPROC> tags, as shown in Listing 30.2.

Using <CFPROCRESULT>

The FilmEntry1.cfm template in Listing 30.2 shows how to retrieve and use a recordset returned from a stored procedure. As you can see, a <CFSTOREDPROC> tag is used to refer to the stored procedure itself. Then the <CFPROCRESULT> tag is used to capture the recordset returned by the procedure.

NOTE

Remember, it is assumed that you have created a version of the Orange Whip Studios sample database for SQLServer (or whatever database server you are using), and that the database is accessible via the data source name owsSqlServer.

NOTE

In addition, for this template to work, you must have created the FetchRatingsList stored procedure. See the "Creating Stored Procedures" section in this chapter for details.

Listing 30.2 FilmEntry1.cfm—Retrieving a Recordset from a Stored Procedure

```
<!---
  Filename: FilmEntry1.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of the <CFSTOREDPROC> tag
--->

<!--- Get list of ratings from database --->
<CFSTOREDPROC PROCEDURE="FetchRatingsList" DATASOURCE="owsSqlServer">
  <CFPROCRESULT NAME="GetRatings">
</CFSTOREDPROC>

<HTML>
<HEAD><TITLE>Film Entry Form</TITLE></HEAD>
<BODY>
<H2>Film Entry Form</H2>

<!--- Data entry form --->
<CFFORM
  ACTION="#CGI.SCRIPT_NAME#"
  METHOD="POST"
  PRESERVEDATA="Yes">

  <!--- Text entry field for film title --->
  <P><B>Title for New Film:</B><BR>
  <CFINPUT
    NAME="MovieTitle"
    SIZE="50"
    MAXLENGTH="50"
    REQUIRED="Yes"
    MESSAGE="Please don't leave the film's title blank."><BR>
```

Listing 30.2 (continued)

```
    <!--- Text entry field for pitch text --->
    <P><B>Short Description / One-Liner:</B><BR>
    <CFINPUT
      NAME="PitchText"
      SIZE="50"
      MAXLENGTH="100"
      REQUIRED="Yes"
      MESSAGE="Please don't leave the one-liner blank."><BR>

    <!--- Text entry field for expense description --->
    <P><B>New Film Budget:</B><BR>
    <CFINPUT
      NAME="AmountBudgeted"
      SIZE="15"
      REQUIRED="Yes"
      MESSAGE="Please enter a valid number for the film's budget."
      VALIDATE="float"><BR>

    <!--- Drop-down list of ratings --->
    <P><B>Rating:</B><BR>
    <CFSELECT
      NAME="RatingID"
      QUERY="GetRatings"
      VALUE="RatingID"
      DISPLAY="Rating"/>

    <!--- Text areas for summary --->
    <P><B>Summary:</B><BR>
    <TEXTAREA NAME="Summary" COLS="40" ROWS="3" WRAP="soft"></TEXTAREA>

    <!--- Submit button for form --->
    <P><INPUT TYPE="Submit" VALUE="Submit New Film">
  </CFFORM>

  </BODY>
  </HTML>
```

NOTE

The syntax shown in Listing 30.2 should work with most database server systems, including Microsoft SQLServer and Sybase. If you are using Oracle, you will need to make a minor change, as discussed in the next section.

Because this listing specifies `NAME="GetRatings"` in the `<CFPROCRESULT>` tag, the rest of the template is free to refer to `GetRatings` just like the results of a `<CFQUERY>`. Here, the `GetRatings` query object is provided to a `<CFSELECT>` tag to populate a drop-down list (Figure 30.3).

TIP

If you are using Dreamweaver MX to edit your templates, you can use the Tag Inspector or the `<CFPROCRESULT>` Tag Editor to help you add the `<CFPROCRESULT>` tag to your code (Figure 30.4).

NOTE

If you don't know the column names the procedure outputs, you could run the procedure and output the value of the automatic `#GetRatings.ColumnList#` variable, just as you could do for any normal query resultset.

Figure 30.3

The <CFPROCRESULT> tag captures a record-set returned by a stored procedure.

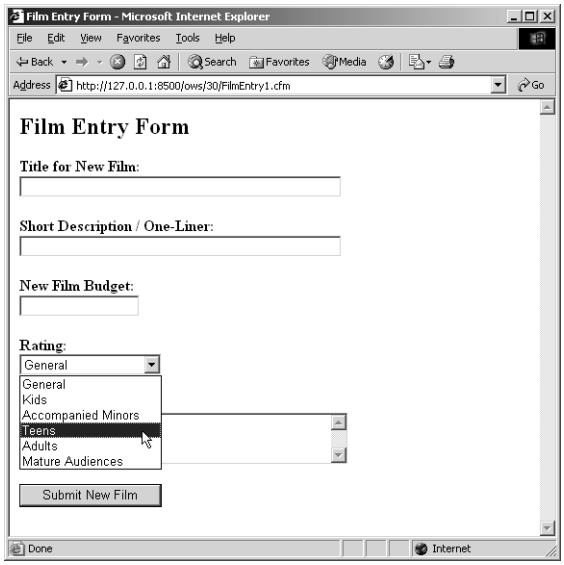
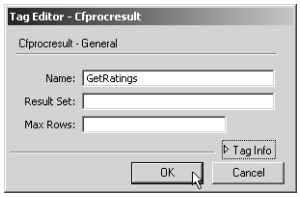


Figure 30.4

Dreamweaver MX makes adding <CFPROCRESULT> tags to your code even easier.



Using <CFPROCRESULT> with Oracle

With Oracle databases, stored procedures can't return recordsets in the traditional sense. They can return recordset-like data, but they do so via something called a *reference cursor*. You'll see an example of how to create a stored procedure that returns a reference cursor later in this chapter, but you will need to consult your Oracle documentation if you want to learn all the ins and outs about reference cursors and the various ways in which they can be used.

ColdFusion makes using the data returned by reference cursors in Oracle stored procedures easy. Basically, you just add a <CFPROCRESULT> tag to capture the data from each reference cursor.

NOTE

This is a change from previous versions of ColdFusion, which required you to use a <CFPROCPARAM> tag of CFSQLTYPE="CF_SQL_REFCURSOR" in order to use the data in a reference cursor as a CFML recordset. In ColdFusion MX, you simply use the <CFPROCRESULT> tag. At the time of this writing, the ColdFusion MX documentation claims that you still need the additional <CFPROCPARAM> tag; that is an error in the documentation.

Listing 30.3 is almost the same as the previous version of this template (refer to Listing 30.2), except that the `DATASOURCE` and the name of the procedure has been adjusted appropriately. Assuming that a procedure called `owsWeb.FetchRatingsList` exists on your Oracle server, and assuming that it has a single output parameter that exposes information from the `FilmsRatings` table as a reference cursor, this template will behave just like the previous listing (refer to Figure 30.3).

NOTE

Remember, it is assumed that you have created a version of the Orange Whip Studios sample database for Oracle, and that the database is accessible via a native Oracle data source named `owsOracle`.

NOTE

In addition, for this template to work, you must have created the `owsWeb.FetchRatingsList` stored procedure. See the "Creating Stored Procedures with Oracle" section in this chapter for details.

Listing 30.3 `FilmEntry1Oracle.cfm`—Retrieving Data from a Reference Cursor in an Oracle Stored Procedure

```
<!---
  Filename: FilmEntry1Oracle.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of stored procedures
--->

<!--- Get list of ratings from database --->
<CFSTOREDPROC PROCEDURE="OWSWEB.FetchRatingsList" DATASOURCE="owsOracle">
  <CFPROCRESULT NAME="GetRatings">
</CFSTOREDPROC>

<HTML>
<HEAD><TITLE>Film Entry Form</TITLE></HEAD>
<BODY>
<H2>Film Entry Form</H2>

<!--- Data entry form --->
<CFFORM
  ACTION="#CGI.SCRIPT_NAME#"
  METHOD="POST"
  PRESERVEDATA="Yes">

  <!--- Text entry field for film title --->
  <P><B>Title for New Film:</B><BR>
  <CFINPUT
    NAME="MovieTitle"
    SIZE="50"
    MAXLENGTH="50"
    REQUIRED="Yes"
    MESSAGE="Please don't leave the film's title blank."><BR>

  <!--- Text entry field for pitch text --->
  <P><B>Short Description / One-Liner:</B><BR>
```

Listing 30.3 (continued)

```
    <CFINPUT
      NAME="PitchText"
      SIZE="50"
      MAXLENGTH="100"
      REQUIRED="Yes"
      MESSAGE="Please don't leave the one-liner blank."><BR>

    <!--- Text entry field for expense description --->
    <P><B>New Film Budget:</B><BR>
    <CFINPUT
      NAME="AmountBudgeted"
      SIZE="15"
      REQUIRED="Yes"
      MESSAGE="Please enter a valid number for the film's budget."
      VALIDATE="float"><BR>

    <!--- Drop-down list of ratings --->
    <P><B>Rating:</B><BR>
    <CFSELECT
      NAME="RatingID"
      QUERY="GetRatings"
      VALUE="RatingID"
      DISPLAY="Rating"/>

    <!--- Text areas for summary --->
    <P><B>Summary:</B><BR>
    <TEXTAREA NAME="Summary" COLS="40" ROWS="3" WRAP="soft"></TEXTAREA>

    <!--- Submit button for form --->
    <P><INPUT TYPE="Submit" VALUE="Submit New Film">
</CFFORM>


</BODY>
</HTML>
```

NOTE

In the stored procedure name `owsWeb.FetchRatingsList`, the `owsWeb` part refers to the name of the package that the stored procedure is in, and `FetchRatingsList` is the name of the procedure itself. For more information about packages, see your Oracle documentation.

NOTE

To see the Oracle code required to create the `owsWeb.FetchRatingsList` Procedures referred to in this template, see the section, "Creating Stored Procedures with Oracle," later in this chapter.

Stored Procedures That Take Parameters and Return Status Codes

So far you've been working with stored procedures that always work the same exact way each time they are called. The `PerformInventoryMaintenance` and `FetchRatingsList` procedures used in the first two examples doesn't accept any input from the calling application (in this case, a ColdFusion template) to do their work.

Most stored procedures, however, take input parameters or output parameters and also can return status codes. Here's what each of these terms means:

- **Input parameters.** Values you supply by name when you execute a stored procedure, similar to providing parameters to a ColdFusion tag. The stored procedure can then use the values of the input parameters internally, similar to variables. The stored procedure can take as many input parameters as needed for the task at hand. Similar to attributes for a ColdFusion tag, some input parameters are required, and others are optional. To supply an input parameter to a stored procedure, you use the `<CFPROCPARAM>` tag (see Table 30.3 in the next section).
- **Output parameters.** Values the procedure can pass back to ColdFusion, or whatever other program might call the stored procedure. Each output parameter has a name, and the stored procedure can return as many output parameters as necessary. To capture the value of an output parameter from a stored procedure, use the `<CFPROCPARAM>` tag with `TYPE="OUT"` (see Table 30.3).
- **Status codes.** Values returned by a stored procedure after it executes. Each stored procedure can return only one status code; the code is usually used to indicate whether the stored procedure was capable of successfully carrying out its work. With most database systems, the status code must be numeric, and defaults to `0`. To capture a procedure's status code, use `RETURNCODE="Yes"` in the `<CFSTOREDPROC>` tag (refer to Table 30.1, earlier in this chapter).

For instance, consider a new stored procedure called `InsertFilm` that enables the user to add a new film to Orange Whip Studio's `Films` table. Whoever created the stored procedure set it up to require five input parameters called `@MovieTitle`, `@PitchText`, `@AmountBudgeted`, `@RatingID`, and `@Summary`, which supply information about the new film. The procedure uses these values to insert a new record into the `Films` table.

The procedure also has one output parameter called `@NewFilmID`, which passes the ID number of the newly inserted film record back to ColdFusion (or whatever program is executing the stored procedure).

NOTE

Stored procedure parameter names start with an `@` sign with Microsoft SQLServer and Sybase databases. Other databases systems, such as Oracle, do not use the `@` sign in this way.

In addition, the stored procedure has been designed to perform a few sanity checks before blindly recording the new film:

- First, it ensures that the film doesn't already exist in the Films table. If a film with the same title is already in the table, the procedure stops with a return value of `-1`.
- It ensures that the rating specified by the `@RatingID` parameter is valid. If the rating number does not exist in the `Ratings` table, the procedure stops with a return value of `-2`.

Provided that both of the tests are okay, the procedure inserts a new row in the `Inventory` table using the values of the supplied parameters. Finally, it sends a return value of `1` to indicate success.

Providing Parameters with <CFPROCPARAM>

ColdFusion makes supplying input and output parameters to a stored procedure easy, via the <CFPROCPARAM> tag. Table 30.3 shows the attributes supported by the <CFPROCPARAM> tag.

Table 30.3 <CFPROCPARAM> Tag Attributes

ATTRIBUTE	PURPOSE
TYPE	In, Out, or InOut. Use TYPE="In" (the default) to supply an input parameter to the stored procedure. Use TYPE="Out" to capture the value of an output parameter. Use TYPE="InOut" if the parameter behaves as both an input and output parameter (such parameters are rather rare).
VALUE	For input parameters only. The actual value you want to provide to the procedure.
NULL	For input parameters only. Yes or No. If NULL="Yes", the VALUE attribute is ignored; instead, a null value is supplied as the parameter's value. For more information about null values, see the section, "Working with NULL Values" in Chapter 29, "More About SQL and Queries."
VARIABLE	For output parameters only. A variable name you want ColdFusion to place the value of the output parameter into. For instance, if you provide VARIABLE="InsertedFilmID", you can output the value using #InsertedFilmID# after the <CFSTOREDPROC> tag executes.
CFSQLTYPE	Required. The parameter's data type. Unlike ColdFusion variables, stored procedure parameters are strongly typed, so you have to specify whether the parameter expects a numeric, string, date, or other type of value. The list of data types that you can supply to this attribute is the same as for the <CFQUERYPARAM> tag; see Table 29.5 in Chapter 29 for the list of data types.
MAXLENGTH	Optional. The maximum length of the parameter's value.
SCALE	Optional. The number of significant decimal places for the parameter. Relevant only for numeric parameters.

NOTE

Previous versions of ColdFusion also supported a DBVARNAME attribute for the <CFPROCPARAM> tag, which allowed you to specify a procedure's parameters by name. The attribute worked only for database drivers native to ColdFusion. Because ColdFusion MX uses JDBC drivers for all data sources, the DBVARNAME attribute no longer has any effect at all.

NOTE

If you're using Oracle, don't include a <CFPROCPARAM> tag for output parameters that return reference cursors. Use the <CFPROCRESULT> tag to capture that type of output, as discussed in the previous section.

FilmEntry2.cfm, shown in Listing 30.4, builds on the previous version of the data entry template from Listing 30.3. It creates a simple form a user can use to fill in the title, description, budget, rating, and summary for a new book. The form collects the information from the user and posts it back to the same template, which executes the stored procedure, feeding the user's entries to the procedure's input parameters.

Listing 30.4 `FilmEntry2.cfm`—A Simple Form for Collecting Input Parameters

```
<!---
  Filename: FilmEntry2.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of stored procedures
--->


<!--- Is the form being submitted? --->
<CFSET WasFormSubmitted = IsDefined("FORM.RatingID")>

<!--- Insert film into database when form is submitted --->
<CFIF WasFormSubmitted>
  <CFSTOREDPROC
    PROCEDURE="InsertFilm"
    DATASOURCE="owsSqlServer"
    RETURNCODE="Yes">

    <!--- Provide form values to the procedure's input parameters --->
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="50"
      CFSQLTYPE="CF_SQL_VARCHAR"
      VALUE="#Form.MovieTitle#">
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="100"
      CFSQLTYPE="CF_SQL_VARCHAR"
      VALUE="#Form.PitchText#">
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="100"
      CFSQLTYPE="CF_SQL_MONEY"
      VALUE="#Form.AmountBudgeted#"
      NULL="#YesNoFormat(FORM.AmountBudgeted EQ '')#">
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="100"
      CFSQLTYPE="CF_SQL_INTEGER"
      VALUE="#Form.RatingID#">
    <CFPROCPARAM
      TYPE="In"
      CFSQLTYPE="CF_SQL_LONGVARCHAR"
      VALUE="#Form.Summary#">

    <!--- Capture @NewFilmID output parameter --->
    <!--- Value will be available in CFML variable named #InsertedFilmID# --->
    <CFPROCPARAM
      TYPE="Out"
      CFSQLTYPE="CF_SQL_INTEGER"
      VARIABLE="InsertedFilmID">

  </CFSTOREDPROC>

  <!--- Remember the status code returned by the stored procedure --->
  <CFSET InsertStatus = CFSTOREDPROC.StatusCode>
</CFIF>
```

Listing 30.4 (continued)

```
<!--- Get list of ratings from database --->
<CFSTOREDPROC PROCEDURE="FetchRatingsList" DATASOURCE="owsSqlServer">
  <CFPROCRESULT NAME="GetRatings">
</CFSTOREDPROC>


<HTML>
<HEAD><TITLE>Film Entry Form</TITLE></HEAD>
<BODY>
<H2>Film Entry Form</H2>

<!--- Data entry form --->
<CFFORM
  ACTION="#CGI.SCRIPT_NAME#"
  METHOD="POST"
  PRESERVEDATA="Yes">

  <!--- Text entry field for film title --->
  <P><B>Title for New Film:</B><BR>
  <CFINPUT
    NAME="MovieTitle"
    SIZE="50"
    MAXLENGTH="50"
    REQUIRED="Yes"
    MESSAGE="Please don't leave the film's title blank."><BR>

  <!--- Text entry field for pitch text --->
  <P><B>Short Description / One-Liner:</B><BR>
  <CFINPUT
    NAME="PitchText"
    SIZE="50"
    MAXLENGTH="100"
    REQUIRED="Yes"
    MESSAGE="Please don't leave the one-liner blank."><BR>

  <!--- Text entry field for expense description --->
  <P><B>New Film Budget:</B><BR>
  <CFINPUT
    NAME="AmountBudgeted"
    SIZE="15"
    REQUIRED="No"
    MESSAGE="Only numbers may be provided for the film's budget."
    VALIDATE="float"> (leave blank if unknown)<BR>

  <!--- Drop-down list of ratings --->
  <P><B>Rating:</B><BR>
  <CFSELECT
    NAME="RatingID"
    QUERY="GetRatings"
    VALUE="RatingID"
    DISPLAY="Rating"/>

  <!--- Text areas for summary --->
  <P><B>Summary:</B><BR>
  <TEXTAREA NAME="Summary" COLS="40" ROWS="3" WRAP="soft"></TEXTAREA>
```

Listing 30.4 (continued)

```
    <!--- Submit button for form --->
    <P><INPUT TYPE="Submit" VALUE="Submit New Film">
  </CFFORM>


  <!--- If we executed the stored procedure --->
  <CFIF WasFormSubmitted>
    <!--- Display message based on status code reported by stored procedure --->
    <CFSWITCH EXPRESSION="#InsertStatus#">
      <!--- If the stored procedure returned a "success" status --->
      <CFCASE VALUE="1">
        <CFOUTPUT>
          <P>Film "#Form.MovieTitle#" was inserted as Film ID #InsertedFilmID#.<BR>
        </CFOUTPUT>
      </CFCASE>
      <!--- If the status code was -1 --->
      <CFCASE VALUE="-1">
        <CFOUTPUT>
          <P>Film "#Form.MovieTitle#" already exists in the database.<BR>
        </CFOUTPUT>
      </CFCASE>
      <!--- If the status code was -2 --->
      <CFCASE VALUE="-2">
        <P>An invalid rating was provided.<BR>
      </CFCASE>
      <!--- If any other status code was returned --->
      <CFDEFAULTCASE>
        <P>The procedure returned an unknown status code.<BR>
      </CFDEFAULTCASE>
    </CFSWITCH>
  </CFIF>

  </BODY>
  </HTML>
```

When the form is submitted, the `<CFIF>` block at the top of the template is executed, which executes the `InsertFilm` stored procedure via the `<CFSTOREDPROC>` tag. Within the `<CFSTOREDPROC>` tag, six `<CFPROCPARAM>` tags are used. The first five provide values for the `@MovieTitle`, `@PitchText`, and other input parameters. The last `<CFPROCPARAM>` captures the value of the output parameter called `@NewFilmID`.

Note that the `CFSQLTYPE` for each of the parameters has been set to the correct value for the type of information being passed. Also, the `NULL` attribute is used for the fifth `<CFPROCPARAM>`, so that the film's budget will be recorded as a null value if the user leaves the budget blank on the form. After the procedure executes, the status code reported by the procedure is placed into the `InsertStatus` variable.

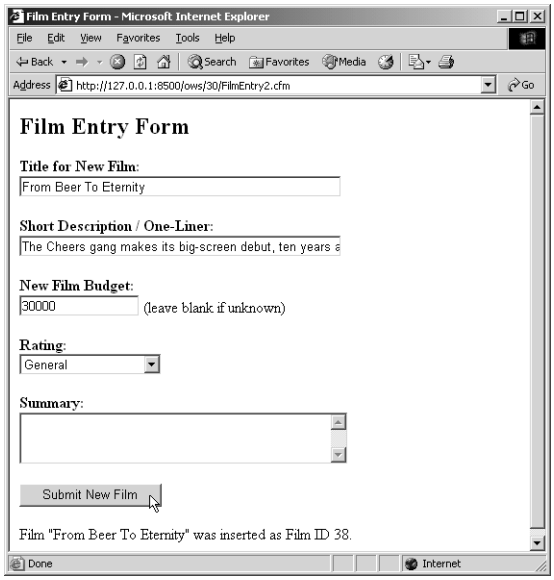
At the bottom of the template, a `<CFSWITCH>` block is used to output a message to the user depending on the status code reported by the stored procedure. If the procedure returns a value of `1`, a success message is displayed, along with the new film's `FilmID` number—which was captured by the first `<CFPROCPARAM>` tag at the top of the template (Figure 30.5). If the procedure returns some other status code, it is assumed that something went wrong, so the status code is shown to the user.

NOTE

In an actual application, you would probably do more than just display the new `FilmID`. For instance, you might provide some type of Click Here link to a screen on which the user could do further data entry about the film. The point is that a stored procedure can give back whatever information its creator desires as output parameters, and you can use that information in just about any way you please.

Figure 30.5

Parameters can be passed in and out of stored procedures.



NOTE

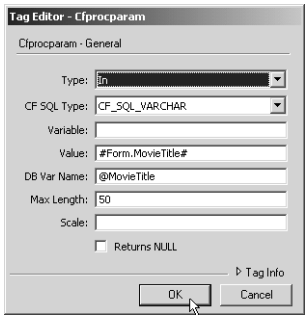
The template in Listing 30.4 uses ColdFusion's `<CFSWITCH>`, `<CFCASE>`, and `<CFDEFAULTCASE>` tags to analyze the return code from the stored procedure. Making decisions based on return codes is a perfect situation to use these case-switching tags because a single expression (the return code) exists that will have one of several predetermined values. See Appendix B for more information about `<CFSWITCH>` and related tags.

TIP

If you are using Dreamweaver MX to edit your templates, you can use the Tag Inspector or the `<CFPROCPARAM>` Tag Editor to help you add `<CFPROCPARAM>` tags to your code (Figure 30.6).

Figure 30.6

Using Dreamweaver MX as your editor makes coding `<CFPROCPARAM>` tags easy.



Ordinal Versus Named Parameter Positioning

When you use the <CFPROCPARAM> tag to provide parameters to a stored procedure, you need to provide the <CFPROCPARAM> tags in the correct order. That is, the first <CFPROCPARAM> tag needs to correspond to the first parameter in the stored procedure's definition on the database server, the second <CFPROCPARAM> tag needs to correspond to the second parameter, and so on. This is called *ordinal positioning*, meaning that the order of parameters is significant.

Unfortunately, the fact that <CFSTOREDPROC> uses ordinal positioning internally means that if the order of the parameters changes on the database server, your ColdFusion templates will likely need to be edited to match. Shortly, you will learn that you may be able to use your database server's native syntax for calling stored procedures within a normal <CFQUERY> block. If that is possible for your database system, you will likely be able to refer to the parameters by name instead of ordinal position, which means that your code will still work if the order of a procedure's parameters is changed on the database side. For this reason, you might want to consider using <CFQUERY> instead of <CFSTOREDPROC> whenever possible, if you expect the parameters for your procedures to change in the future. For details, see the section, "Calling Procedures with <CFQUERY> Instead of <CFSTOREDPROC>," later in this chapter).

NOTE

If one of the stored procedure's parameters is optional (that is, if it was given a default value when the procedure was created), you can omit the corresponding <CFPROCPARAM> tag. However, if you need to provide a value to any parameters that come after the optional parameter, then you will need to provide a <CFPROCPARAM> for the optional parameter as well. In other words, there is no way to tell <CFPROCPARAM> to skip a parameter conceptually.

Parameter Data Types

As you saw in Listing 30.4, when you provide parameters to a stored procedure with the <CFPROCPARAM> tag, you must specify the data type of the parameter, as defined by whomever created the procedure. ColdFusion requires that you provide the data type for each parameter you refer to in your templates, so that it does not have to determine the data type itself on-the-fly each time the template runs. That would require a number of extra steps for ColdFusion, which in turn would slow your application.

It's important to specify the correct data type for each parameter. If you use the wrong data type, you might run into problems. The data type you provide for CFSQLTYPE in a <CFPROCPARAM> tag must be one of ColdFusion's SQL data types, as listed in Table 29.5 in Chapter 29. These data types are based on the data types defined by the ODBC standard—one of them will map to each of the database-specific data types used when your stored procedure was created.

For instance, if you have a stored procedure sitting on a Microsoft SQL Server that takes a parameter of SQL Server data type int, you should specify the CF_SQL_INTEGER data type in the corresponding <CFPROCPARAM> tag.

Wrapping a Stored Procedure Call in a Custom Tag

If you have a stored procedure that you plan to use often in your ColdFusion applications, you may want to consider creating a CFML Custom Tag that wraps all the needed `<CFSTOREDPROC>` and related code into an easier-to-use module. For instance, you might create a custom tag called `<CF_spInsertFilm>` that calls the `InsertFilm` stored procedure.

The `spInsertFilm.cfm` template included on the CD-ROM for this chapter creates such a custom tag, which could be used like this:

```
<!--- Insert film via Stored Procedure (via custom tag) --->
<CF_spInsertFilm
  MovieTitle="#FORM.MovieTitle#"
  PitchText="#FORM.PitchText#"
  RatingID="#FORM.RatingID#"
  AmountBudgeted="#FORM.AmountBudgeted#"
  Summary="#FORM.Summary#"
  ReturnFilmID="InsertedFilmID"
  ReturnStatusCode="InsertStatus">
```

The FilmEntry3.cfm template (also on the CD-ROM) is a revision of Listing 30.4. Instead of calling the `<CFSTOREDPROC>` tag directly, the template simply calls the custom tag when the form is submitted. For more information about how to create and use CFML custom tags, see Chapter 20.

Multiple Recordsets Are Fully Supported

Some stored procedures return more than one recordset. For instance, consider a stored procedure called `FetchFilmInfo`. You are told that the procedure accepts one input parameter called `@FilmID`, and the procedure responds by returning five recordsets of information related to the specified film. The first recordset contains information about the film record itself; the second returns related records from the `Expenses` table; the third returns related records from the `Actors` table; the fourth returns related records from `Directors`; and the fifth returns information from the `Merchandise` table.

As you can see in Listing 30.5, the key to receiving more than one recordset from a stored procedure is to include one `<CFPROCRESULT>` tag for each recordset, specifying `RESULTSET="1"` for the first recordset, `RESULTSET="2"` for the second, and so on.

Listing 30.5 `ShowFilmExpenses.cfm`—Dealing with Multiple Recordsets from a Single Stored Procedure

```
<!---
  Filename: ShowFilmExpenses.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of stored procedures
--->


<!--- Execute stored procedure to fetch film information --->
<CFSTOREDPROC PROCEDURE="FetchFilmInfo" DATASOURCE="owsSqlServer">
  <!--- Provide the FilmID parameter --->
  <CFPROCPARAM
    TYPE="In"
    CFSQLTYPE="CF_SQL_INTEGER"
    VALUE="#URL.FilmID#">
```

Listing 30.5 (continued)

```
    <!--- Film information --->
    <CFPROCRESULT NAME="GetFilm" RESULTSET="1">
    <!--- Expense information --->
    <CFPROCRESULT NAME="GetExpenses" RESULTSET="2">
    <!--- Actor information --->
    <CFPROCRESULT NAME="GetActors" RESULTSET="3">
    <!--- Director information --->
    <CFPROCRESULT NAME="GetDirectors" RESULTSET="4">
    <!--- Director information --->
    <CFPROCRESULT NAME="GetMerch" RESULTSET="5">
</CFSTOREDPROC>

<!--- Get subtotals from the recordsets returned by stored procedure --->
<CFSET ExpenseSum  = ArraySum(ListToArray(ValueList(GetExpenses.ExpenseAmount)))>
<CFSET ActorSum    = ArraySum(ListToArray(ValueList(GetActors.Salary)))>
<CFSET DirectorSum = ArraySum(ListToArray(ValueList(GetDirectors.Salary)))>
<CFSET MerchSum    = ArraySum(ListToArray(ValueList(GetMerch.TotalSales)))>

<!--- Add up all expenses --->
<CFSET TotalExpenses = ExpenseSum + ActorSum + DirectorSum - MerchSum>
<!--- Determine how much money is left in the budget --->
<CFSET LeftInBudget = GetFilm.AmountBudgeted - TotalExpenses>

<HTML>
<HEAD><TITLE>Film Expenses</TITLE></HEAD>
<BODY>

<!--- Company logo and page title --->
<IMG SRC="../images/logo_b.gif" WIDTH="73" HEIGHT="73" ALIGN="absmiddle">
<FONT SIZE="+2"><B>Film Expenses</B></FONT><BR CLEAR="all">

<CFOUTPUT>
  <!--- Show film title--->
  <P><B>Film:</B> #GetFilm.MovieTitle#<BR>

  <!--- Film budget, expense total, and amount left in budget --->
  <P><B>Budget:</B> #LSCurrencyFormat(GetFilm.AmountBudgeted)#<BR>
  <B>Expenses:</B> #LSCurrencyFormat(TotalExpenses)#<BR>
  <B>Currently:</B> #LSCurrencyFormat(LeftInBudget)#
  <!--- Are we currently over or under budget? --->
  #IIF(LeftInBudget LT 0, "'over budget'", "'under budget'")#<BR>

  <!--- Output information about actors --->
  <P><B>Actors:</B>
  <CFLOOP QUERY="GetActors">
    <LI>#NameFirst# #NameLast# (Salary: #LSCurrencyFormat(Salary)#)
  </CFLOOP>

  <!--- Output information about directors --->
  <P><B>Directors:</B>
  <CFLOOP QUERY="GetDirectors">
    <LI>#FirstName# #LastName# (Salary: #LSCurrencyFormat(Salary)#)
  </CFLOOP>

  <!--- Output information about expenses --->
  <P><B>Other Expenses:</B>
```

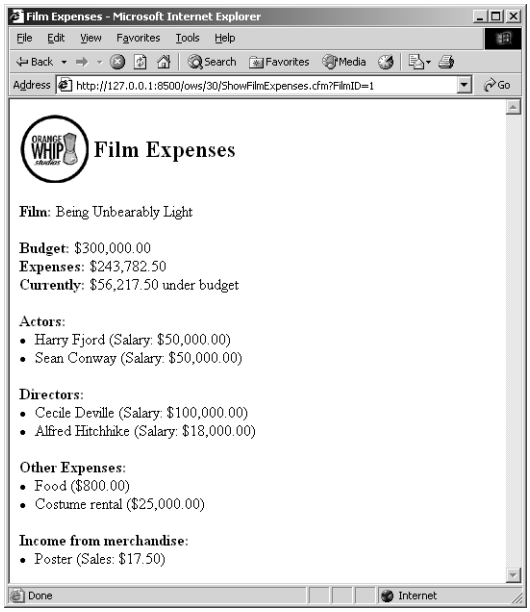
Listing 30.5 (continued)

```
    <CFLOOP QUERY="GetExpenses">
      <LI>#Description# (#LSCurrencyFormat(ExpenseAmount)#)
    </CFLOOP>

    <!--- Output information about merchandise --->
    <P><B>Income from merchandise:</B>
    <CFLOOP QUERY="GetMerch">
      <LI>#MerchName# (Sales: #LSCurrencyFormat(TotalSales)#)
    </CFLOOP>
  </CFOUTPUT>

  </BODY>
  </HTML>
```

After the `<CFSTOREDPROC>` tag, the ColdFusion template is free to refer to the five recordsets named in the `<CFPROCRESULT>` tags just as if they were the results of five separate `<CFQUERY>`-type queries (Figure 30.7). Because only one communication between ColdFusion and the database server needed to take place, though, you can expect performance to be faster using the single stored procedure.

Figure 30.7

ColdFusion makes capturing multiple recordsets from a single stored procedure easy.



NOTE

Your template doesn't have to handle or receive all the recordsets a stored procedure spits out. For instance, if you weren't interested in the second recordset from the `FetchFilmInfo` procedure, you could simply leave out the second `<CFPROCRESULT>` tag. Neither ColdFusion nor your database server will mind.

If you are using Oracle, then `RESULTSET="1"` refers to the first output parameter of type `REF CURSOR`, `RESULTSET="2"` refers to the second such parameter, and so on. So, if you had a procedure called `owsWeb.FetchFilmInfo` on your Oracle server that exposed five reference cursors (such as the five

recordsets returned by the SQLServer used in the previous listing), hardly anything needs to change. You would simply use the same <CFPROCRESULT> tags to capture the data from the reference cursors, as shown in Listing 30.6.

Listing 30.6 ShowFilmExpensesOracle.cfm—Dealing with Multiple Reference Cursors from an Oracle Stored Procedure

```
<!---
  Filename: ShowFilmExpensesOracle.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of stored procedures
--->

<!--- Execute stored procedure to fetch film information --->
<CFSTOREDPROC PROCEDURE="OWSWEB.FetchFilmInfo" DATASOURCE="owsOracle">
  <!--- Provide the FilmID parameter --->
  <CFPROCPARAM
    TYPE="In"
    CFSQLTYPE="CF_SQL_INTEGER"
    VALUE="#URL.FilmID#">

  <!--- Film information --->
  <CFPROCRESULT NAME="GetFilm" RESULTSET="1">
  <!--- Expense information --->
  <CFPROCRESULT NAME="GetExpenses" RESULTSET="2">
  <!--- Actor information --->
  <CFPROCRESULT NAME="GetActors" RESULTSET="3">
  <!--- Director information --->
  <CFPROCRESULT NAME="GetDirectors" RESULTSET="4">
  <!--- Director information --->
  <CFPROCRESULT NAME="GetMerch" RESULTSET="5">
</CFSTOREDPROC>

<!--- Get subtotals from the recordsets returned by stored procedure --->
<CFSET ExpenseSum  = ArraySum(ListToArray(ValueList(GetExpenses.ExpenseAmount)))>
<CFSET ActorSum    = ArraySum(ListToArray(ValueList(GetActors.Salary)))>
<CFSET DirectorSum = ArraySum(ListToArray(ValueList(GetDirectors.Salary)))>
<CFSET MerchSum    = ArraySum(ListToArray(ValueList(GetMerch.TotalSales)))>

<!--- Add up all expenses --->
<CFSET TotalExpenses = ExpenseSum + ActorSum + DirectorSum - MerchSum>
<!--- Determine how much money is left in the budget --->
<CFSET LeftInBudget = GetFilm.AmountBudgeted - TotalExpenses>

<HTML>
<HEAD><TITLE>Film Expenses</TITLE></HEAD>
<BODY>

<!--- Company logo and page title --->
<IMG SRC="../images/logo_b.gif" WIDTH="73" HEIGHT="73" ALIGN="absmiddle">
<FONT SIZE="+2"><B>Film Expenses</B></FONT><BR CLEAR="all">

<CFOUTPUT>
  <!--- Show film title--->
  <P><B>Film:</B> #GetFilm.MovieTitle#<BR>
```

Listing 30.6 (continued)

```
<!--- Film budget, expense total, and amount left in budget --->
<P><B>Budget:</B> #LSCurrencyFormat(GetFilm.AmountBudgeted)#<BR>
<B>Expenses:</B> #LSCurrencyFormat(TotalExpenses)#<BR>
<B>Currently:</B> #LSCurrencyFormat(LeftInBudget)#
<!--- Are we currently over or under budget? --->
#IIF(LeftInBudget LT 0, "'over budget'", "'under budget'")#<BR>

<!--- Output information about actors --->
<P><B>Actors:</B>
<CFLOOP QUERY="GetActors">
  <LI>#NameFirst# #NameLast# (Salary: #LSCurrencyFormat(Salary)#)
</CFLOOP>

<!--- Output information about directors --->
<P><B>Directors:</B>
<CFLOOP QUERY="GetDirectors">
  <LI>#FirstName# #LastName# (Salary: #LSCurrencyFormat(Salary)#)
</CFLOOP>

<!--- Output information about expenses --->
<P><B>Other Expenses:</B>
<CFLOOP QUERY="GetExpenses">
  <LI>#Description# (#LSCurrencyFormat(ExpenseAmount)#)
</CFLOOP>

<!--- Output information about merchandise --->
<P><B>Income from merchandise:</B>
<CFLOOP QUERY="GetMerch">
  <LI>#MerchName# (Sales: #LSCurrencyFormat(TotalSales)#)
</CFLOOP>
</CFOUTPUT>

</BODY>
</HTML>
```

NOTE

This code assumes that the procedure parameter named `pFilmsCur` is a reference cursor that exposes records from the `Films` table, `pExpensesCur` exposes data from the `Expenses` table, and so on.

Calling Procedures with <CFQUERY> Instead of <CFSTOREDPROC>

The `<CFSTOREDPROC>` and related tags were added back in ColdFusion 4. Before that, the only way to call a stored procedure from a ColdFusion template was to use special procedure-calling syntax in a normal `CFQUERY` tag, where you would normally provide a SQL statement. You can still call stored procedures this way, and you actually might prefer to do so in some situations.

When you execute a stored procedure with `<CFQUERY>`, ColdFusion treats the response from the database server similar to how it would treat the response from an ordinary `SELECT` query. In fact, ColdFusion doesn't even realize that it's executing a stored procedure exactly; it's just passing on what it assumes to be a valid SQL statement and hopes to get some rows of table-style data in return.

The fact that ColdFusion is treating the stored procedure in the same way it treats a <SELECT> statement brings with it several important limitations:

- ColdFusion cannot directly access the return code generated by the stored procedure.
- ColdFusion cannot directly access any output parameters generated by the stored procedure.
- If the stored procedure returns more than one recordset, only one of the recordsets will be captured by ColdFusion and available for your use in your templates. Details about this limitation might vary between types of database servers. With some database servers, only the first recordset will be available for your use; with others, only the last will be available. The point is that stored procedures that return multiple recordsets are not fully supported when using CFQUERY.

However, using <CFQUERY> has some important advantages over <CFSTOREDPROC>, as well:

- A recordset returned by a stored procedure via <CFQUERY> can be cached using the CACHEDWITHIN attribute. Recordsets captured via <CFSTOREDPROC> and <CFPROCRESULT> can't be cached in ColdFusion MX.
- With <CFQUERY>, you can to a stored procedure's parameters by name, instead of only by position. For details, see "Using Your Database's Native Syntax" in this section.

Using the ODBC/JDBC CALL Command

With most types of database systems (including SQLServer and Oracle), you can use the CALL command defined by the ODBC standard to execute a stored procedure. Listing 30.7 demonstrates how the FetchRatingsList stored procedure can be called using this method.

Note that this template is almost exactly the same as the FilmEntry1.cfm template shown in Listing 30.2. The only change is that <CFQUERY> is being used instead of <CFSTOREDPROC>. When this template is brought up in a browser, it should display its results exactly the same way.

Listing 30.7 FilmEntry1a.cfm—Calling a Stored Procedure

```
<!---
  Filename: FilmEntry1a.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of stored procedures
--->


<!--- Get list of ratings from database --->
<CFQUERY NAME="GetRatings" DATASOURCE="owsSqlServer">
  { CALL FetchRatingsList }
</CFQUERY>
```

Listing 30.7 (continued)

```
<HTML>
<HEAD><TITLE>Film Entry Form</TITLE></HEAD>
<BODY>
<H2>Film Entry Form</H2>

<!--- Data entry form --->
<CFFORM
  ACTION="#CGI.SCRIPT_NAME#"
  METHOD="POST"
  PRESERVEDATA="Yes">

  <!--- Text entry field for film title --->
  <P><B>Title for New Film:</B><BR>
  <CFINPUT
    NAME="MovieTitle"
    SIZE="50"
    MAXLENGTH="50"
    REQUIRED="Yes"
    MESSAGE="Please don't leave the film's title blank."><BR>

  <!--- Text entry field for pitch text --->
  <P><B>Short Description / One-Liner:</B><BR>
  <CFINPUT
    NAME="PitchText"
    SIZE="50"
    MAXLENGTH="100"
    REQUIRED="Yes"
    MESSAGE="Please don't leave the one-liner blank."><BR>

  <!--- Text entry field for expense description --->
  <P><B>New Film Budget:</B><BR>
  <CFINPUT
    NAME="AmountBudgeted"
    SIZE="15"
    REQUIRED="Yes"
    MESSAGE="Please enter a valid number for the film's budget."
    VALIDATE="float"><BR>

  <!--- Drop-down list of ratings --->
  <P><B>Rating:</B><BR>
  <CFSELECT
    NAME="RatingID"
    QUERY="GetRatings"
    VALUE="RatingID"
    DISPLAY="Rating"/>

  <!--- Text areas for summary --->
  <P><B>Summary:</B><BR>
  <TEXTAREA NAME="Summary" COLS="40" ROWS="3" WRAP="soft"></TEXTAREA>

  <!--- Submit button for form --->
  <P><INPUT TYPE="Submit" VALUE="Submit New Film">
```

Listing 30.7 (continued)

```
</CFFORM>


</BODY>
</HTML>
```

As you can see in Listing 30.7, the syntax for using the `CALL` command is simply the word `CALL` and then the procedure name. The entire command is surrounded by a set of curly braces, which indicate that the command must be interpreted by JDBC before it gets sent on to the database server.

Input parameters can be supplied in parentheses after the procedure name, separated by commas. If no input parameters exist for the procedure, leave out the parentheses. Because you are not referring to them by name, input parameters must be supplied in the proper order (as defined by whomever wrote the stored procedure). If an input parameter is of a character type (such as `char`, `varchar`, or `text`), enclose the parameter's value in single quotation marks. For instance, to call the `InsertFilm` stored procedure from Listing 30.4, you could replace the `<CFSTOREDPROC>` block in that template with the following snippet (see `FilmEntry2a.cfm` on the CD-ROM to see this snippet in a complete template):

```
<CFQUERY DATASOURCE="owsSqlServer">
  { CALL InsertFilm (
    '#Form.MovieTitle#',
    '#Form.PitchText#',
     #Form.AmountBudgeted#,
     #Form.RatingID#,
    '#Form.Summary#') }
</CFQUERY>
```

Remember, however, that ColdFusion is not aware of the return code or output parameters returned by the procedure, so the code in the rest of the listing would fail. You would need to have the stored procedure rewritten so that the information provided by the return code or output parameters instead get returned as a recordset.

Using Your Database's Native Syntax

In addition to using the `CALL` command, most database drivers also enable you to use whatever native syntax you would use normally with that database system. All the same limitations (regarding return codes, output parameters, and so on) listed at the beginning of this section apply.

The native syntax to use varies according to the database server you're using. You need to consult your database server documentation for the details. Just as an example, if you were using Microsoft SQL Server, you could replace the `<CFQUERY>` shown in Listing 30.7 with the following code; the results would be the same:

```
<CFQUERY NAME="GetRatings" DATASOURCE="owsSqlServer">
   EXEC FetchRatingsList
</CFQUERY>
```

One advantage of using the native syntax over the CALL syntax is that you may be able to refer to the input parameters by name, which leads to cleaner and more readable code. So, if you were using Microsoft SQL Server, the InsertFilm procedure could be called with the following. Consult your database server documentation for specific details:

```
<CFQUERY DATASOURCE="owsSqlServer">
  EXEC InsertFilm
    @MovieTitle = '#Form.MovieTitle#',
    @PitchText = '#Form.PitchText#',
    @AmountBudgeted = #Form.AmountBudgeted#,
    @RatingID = #Form.RatingID#,
    @Summary = '#Form.Summary#'
</CFQUERY>
```

Depending on your database server, you might be able to add more code to the <CFQUERY> tag to be able to capture the status code and output parameters from the stored procedure. Again, just as an example, if you were using Microsoft SQL Server, you would be able to use something similar to the following:

```
<CFQUERY DATASOURCE="owsSqlServer" NAME="ExecProc">
  -- Declare T-SQL variables to hold values returned by stored procedure
  DECLARE @StatusCode INT, @NewFilmID INT
  -- Execute the stored procedure, assigning values to T-SQL variables
  EXEC @StatusCode = InsertFilm
    @MovieTitle = '#Form.MovieTitle#',
    @PitchText = '#Form.PitchText#',
    @AmountBudgeted = #Form.AmountBudgeted#,
    @RatingID = #Form.RatingID#,
    @Summary = '#Form.Summary#',
    @NewFilmID = @NewFilmID OUTPUT
  -- Select the T-SQL variables as a one-row recordset
  SELECT @StatusCode AS StatusCode, @NewFilmID AS NewFilmID
</CFQUERY>
```

You then could refer to ExecProc.StatusCode and ExecProc.NewFilmID in your CFML template code. The FilmEntry2b.cfm template on the CD-ROM for this chapter is a revised version of Listing 30.4 that uses this <CFQUERY> snippet instead of <CFSTOREDPROC> to execute the InsertFilm stored procedure.

NOTE

You will need to consult your database documentation for more information about how to use this type of syntax. In general, it is really best to use <CFSTOREDPROC> instead of <CFQUERY> if the stored procedure you want to use generates important status codes or output parameters.

Creating Stored Procedures

Now that you've seen how to use stored procedures in your ColdFusion templates, you are probably curious about how you can create some stored procedures of your own. We can't cover everything about creating stored procedures in this book, but you learn enough here to hit the ground running.

Before you begin, a short note: The examples you see in this section—and the procedure-creating code you're likely to use as you write your own stored procedures—are not standard SQL. Stored procedures are implemented slightly differently by different database servers, and you often will want to use code within the stored procedures that take advantage of various extensions proprietary to the database system you are using. In other words, when you get to the point of writing your own stored procedures, you likely will be tying yourself to the database system you're using to some extent. The procedures probably will not be portable to other database systems (like ordinary SQL generally is) without reworking the procedures to some extent.

Creating Stored Procedures with Microsoft SQL Server

To create a stored procedure with Microsoft SQL Server, you use the SQL Server Enterprise Manager tool, which helps you submit a `CREATE PROCEDURE` statement to the SQL Server itself. After it's created, the procedure is available to whoever has appropriate permissions.

NOTE

This chapter assumes that you are using SQLServer 7.0. If you are using a different version (such as SQLServer 6.5 or SQLServer 2000), the specific steps might be slightly different, but the basic concepts will be the same.

NOTE

To follow along with this chapter, you need to have a database on your SQL Server machine that has the same tables and columns as the A2Z sample database. With SQL Server 7.0, select Tools, Wizards from the Enterprise Manager's menu to start the DTS Import Wizard. In just a few steps, the wizard can import the Access version of the database (the `a2z.mdb` file), with all tables, columns, and data intact.

Creating Your First Stored Procedure

As a simple example, you can create the `FetchRatingsList` stored procedure that was used in Listing 30.2. This procedure is a good one to start with because it's very simple and does not use any input or output parameters.

To create the stored procedure, follow these simple steps:

1. Start the SQL Server Enterprise Manager. Click your server, expand the databases tree, and click your database.
2. Click the Stored Procedures folder; right-click to select New Stored Procedure from the pop-up menu (Figure 30.8).
3. Type the actual code for the procedure—provided in Listing 30.8—in the window that appears (Figure 30.9).
4. Click the OK button to create the stored procedure.

Figure 30.8

Use the Enterprise Manager to create a new stored procedure.

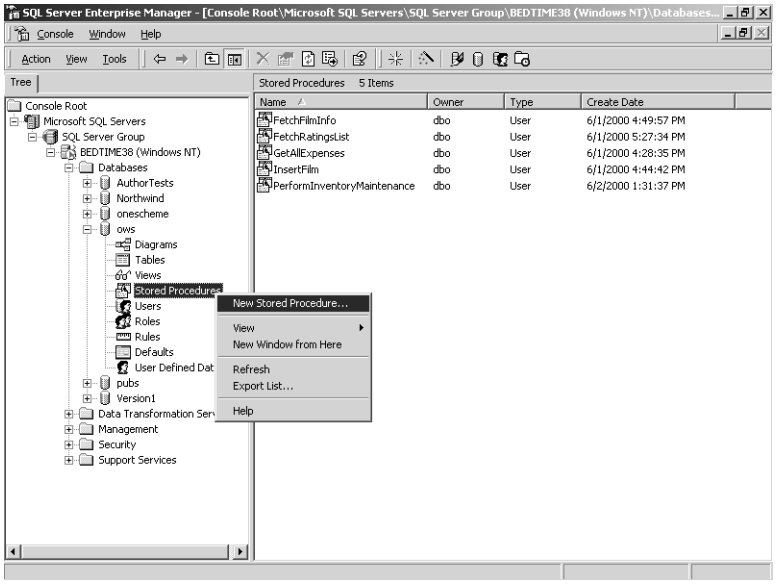
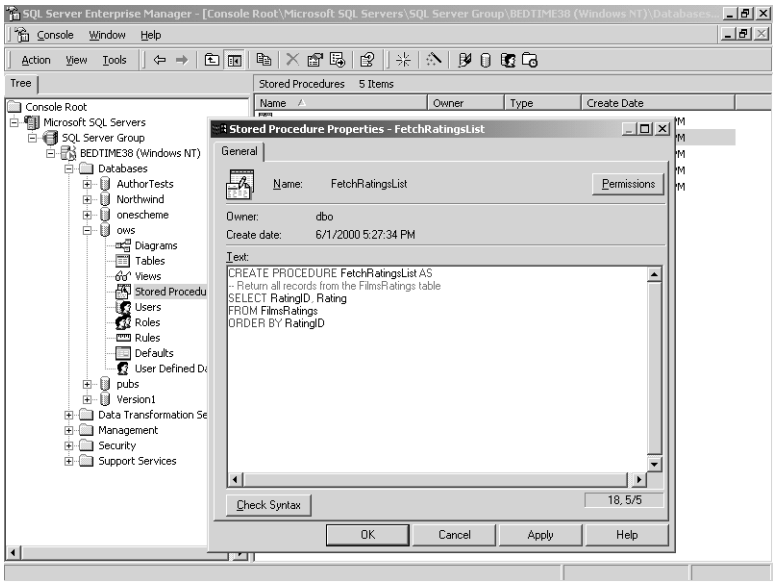


Figure 30.9

Enter the procedure's code in the New Stored Procedure window.



Listing 30.8 shows the actual code for the `FetchRatingsList` stored procedure. This is the code you should enter after selecting New Procedure from the pop-up menu (refer to Figure 30.12). As you can see, it's fairly simple. First, a `CREATE PROCEDURE` statement is used to provide a name for the new stored procedure, followed by the `AS` keyword. Everything after `AS` is the code for the stored procedure itself—the actual SQL statements you want to execute each time the procedure is called.

Listing 30.8 The `FetchRatingsList`—Stored Procedure

```
CREATE PROCEDURE FetchRatingsList AS
-- Return all records from the FilmsRatings table
SELECT RatingID, Rating
FROM FilmsRatings
ORDER BY RatingID
```

TIP

You also can execute the `CREATE PROCEDURE` statements shown in this section with the Query Analyzer (called iSQL/w in SQL Server 6.5 and earlier) or the `isql.exe` command-line utility that ships with SQL Server. You could even execute these `CREATE PROCEDURE` statements in a tag within a ColdFusion template.

This procedure's actual body is simple. It just uses a simple `SELECT` statement to return all the records from the `FilmsRatings` table. The records can be captured and accessed in your ColdFusion code via the `<CFSTOREDPROC>` tag (refer to Listing 30.2) or `<CFQUERY>` tag (refer to Listing 30.7).

Defining Status Codes, Input Parameters, and Output Parameters

As demonstrated in Listing 30.4, stored procedures can accept input parameters, return values via output parameters, and return a status code.

With SQL Server, you define input parameters in the `CREATE PROCEDURE` part of the procedure code, between the procedure name and the `AS` keyword. Each input parameter is given a name, which must begin with the `@` symbol and can't contain any spaces or other unusual characters. The SQL Server data type for each parameter is provided after the parameter's name. If more than one parameter exists, separate them with commas.

If you want the parameter to be optional, type an = sign after the data type and then type the default value. To create an output parameter, provide an initial value for the parameter (usually `NULL` is most appropriate) using the = sign and then type the word `OUTPUT`.

For instance, Listing 30.9 shows the code to create the `InsertFilm` stored procedure used in Listing 30.4. As you can see, each of the parameters provided in `<CFPROCPARAM>` tags in that template matches the corresponding parameters in the `CREATE PROCEDURE` statement here. Then, in the SQL statements that follow, the procedure is free to refer to the parameters by name—similar to how you refer to ColdFusion variables in your CFML templates. The values of the parameters are automatically plugged in with the appropriate values each time the procedure is actually used.

Listing 30.9 The `InsertFilm` Procedure Accepts Input Parameters

```
CREATE PROCEDURE InsertFilm
  @MovieTitle varchar(100),
  @PitchText varchar(100),
  @AmountBudgeted money,
  @RatingID int,
  @Summary text,
  @NewFilmID int = null OUTPUT
AS

-- Make sure there isn't a film by this name in the database already
IF EXISTS (SELECT * FROM Films WHERE MovieTitle = @MovieTitle)
```

Listing 30.9 (continued)

```
    -- If film exists already, return status of -1 to ColdFusion (or other program)
    RETURN -1

  -- Make sure the specified rating is valid
  ELSE IF NOT EXISTS (SELECT * FROM FilmsRatings WHERE RatingID = @RatingID)
    -- If specified rating code does not exist, return status of -2
    RETURN -2

  -- Assuming that the film is not i nthe database already
  ELSE
    BEGIN
      -- Insert the new film into the Films table
      INSERT INTO Films (MovieTitle, PitchText, AmountBudgeted, RatingID, Summary)
      VALUES (@MovieTitle, @PitchText, @AmountBudgeted, @RatingID, @Summary)

      -- Set output parameter called @NewFilmID to the ID of the just-inserted film
      SET @NewFilmID = @@IDENTITY

      -- Return a status of 1 to ColdFusion (or other program) to indicate success
      RETURN 1
    END
```

Note that the `InsertFilm` procedure uses SQL Server's `IF` keyword to do some conditional processing as the template executes. Similar conceptually to the `CFIF` tag in CFML, the `IF` keyword as used here enables you to execute certain chunks of SQL code depending on conditions you define. You'll find `IF` extremely helpful when you need to write a stored procedure that must perform some kind of sanity check before committing changes to the database. For instance, the first `IF` line in Listing 30.11 halts processing and sends a return code of `-1` to the calling application if the `SELECT` subquery inside the parentheses returns any rows.

Note also that the `@NewFilmID` output parameter is set to the value of SQL Server's built-in `@@IDENTITY` variable, which always contains the newly generated identity value for the last `INSERT` to a table that contained an identity-type column. (Identity columns are similar to AutoNumber columns in Access tables.) Thus, the parameter gets set to the `FilmID` that was just assigned to the row just inserted into the `Films` table.

NOTE

`IF` is not part of SQL as most people would define it, but rather it is part of SQL Server's extensions to SQL, which Microsoft calls Transact-SQL. Transact-SQL provides many other control-of-flow keywords you might want to become familiar with, such as `BEGIN`, `END`, `ELSE`, `DECLARE`, and `WHILE`. Sybase also supports most of the same Transact-SQL extensions to standard SQL. Consult your database documentation about these keywords.

Returning Multiple Recordsets to ColdFusion

As you saw in Listing 30.5, stored procedures can return multiple recordsets to ColdFusion (or whatever program is calling the procedures). Again, think of a recordset as a set of rows and columns generated by a `SELECT` statement that outputs data from a database table.

To return recordsets, simply use the appropriate SELECT statements as you would normally. Whatever would be returned by the SELECT statement outside a stored procedure is what will be returned to the calling application (in this case, your ColdFusion template) as a recordset each time the procedure executes.

Your stored procedure can output as many recordsets as it pleases. The calling ColdFusion template need only provide a <CFPROCRESULT> tag for each recordset that it wants to be aware of after the procedure runs. The second SELECT statement in the procedure that outputs rows can be captured by using RECORDSET="2" in the <CFPROCRESULT> tag, and so on.

For instance, Listing 30.10 provides the code to create the FetchFilmInfo procedure that is used by the ShowFilmExpenses.cfm template from Listing 30.5. As you can see, each <CFPROCRESULT> tag used in that template matches with a corresponding SELECT statement in the procedure code itself.

Listing 30.10 Returning Multiple Recordsets from a Stored Procedure

```
CREATE PROCEDURE FetchFilmInfo
  @FilmID int
AS

-- Recommended setting; see SET in your SQLServer documentation
SET NOCOUNT ON

-- Return information about the film itself
SELECT * FROM Films
WHERE FilmID = @FilmID

-- Return all expense records related to the film
SELECT * FROM Expenses
WHERE FilmID = @FilmID
ORDER BY ExpenseDate DESC

-- Return all actor records related to the film
SELECT a.ActorID, a.NameFirst, a.NameLast, fa.Salary
FROM Actors a, FilmsActors fa
WHERE a.ActorID = fa.ActorID
AND fa.FilmID = @FilmID
ORDER BY NameFirst, NameLast

-- Return all director records related to the film
SELECT d.DirectorID, d.FirstName, d.LastName, fd.Salary
FROM Directors d, FilmsDirectors fd
WHERE d.DirectorID = fd.DirectorID
AND fd.FilmID = @FilmID
ORDER BY LastName, FirstName

-- Return all merchandise records related to the film
SELECT m.MerchID, m.MerchName, SUM(ItemPrice) AS TotalSales
FROM Merchandise m, MerchandiseOrdersItems oi
WHERE m.MerchID = oi.ItemID
AND m.FilmID = @FilmID
GROUP BY m.MerchID, m.MerchName
ORDER BY m.MerchName
```

Creating Stored Procedures with Oracle

Stored procedures work somewhat differently with Oracle than they do with SQL Server and Sybase. The basic concepts are the same, and many of the stored procedure examples in this chapter can be implemented on an Oracle server quite easily.

NOTE

This section assumes you are using Oracle8i for Windows NT or 2000. If you're using a different version of Oracle (such as Oracle 7.3, 8, or 9i), some of the figures shown in this chapter may look a little different from what you see on your screen.

NOTE

With Oracle, procedures that send back a return value are actually called *stored functions*, rather than stored procedures. Stored functions are not supported by the `<CFSTOREDPROC>` tag (even if you set `RETURNCODE="Yes"`). Therefore, you need to write your procedures so they send back any status information as an output parameter instead of a return value.

NOTE

To follow along with this section, you must have the tables from the A2Z sample database available on your Oracle server. You can do this relatively easily by using the Oracle Migration Workbench with the Microsoft Access Plug-In, which is a simple wizard you can use to automatically migrate the Access version of the Orange Whip Studios database (the `a2z.mdb` file from this book's CD) to your Oracle server. At the time of this writing, the Migration Workbench was available as a free download from `http://www.oracle.com`.

Creating the ImportCustomers Stored Procedure

The easiest way to create a stored procedure is to use the Oracle DBA Studio, which is installed as part of the standard Oracle 8i installation. For instance, follow these steps to create the `InsertFilm` stored procedure:

1. Using the Windows Start menu, start the Oracle DBA Studio application and enter a valid username and password when prompted.
2. Navigate to the Schema item within your database, right-click the Procedure folder, and then select Create from the pop-up menu (Figure 30.10). The Create Procedure dialog box appears.
3. For the procedure's Name, type `INSERTFILM`; then select the correct Schema from the drop-down list.
4. Type the code shown in Listing 30.11 into the Source box (Figure 30.11). Do not include the first line of the listing (the `CREATE OR REPLACE PROCEDURE` line). The dialog box includes this line for you.
5. Click the Create button. Your new procedure is created and the dialog box disappears. You can now expand the Procedures folder along the left side of the screen and click the new procedure to confirm that it was created correctly.

Figure 30.10

Creating a stored procedure is simple with the Oracle DBA Studio.

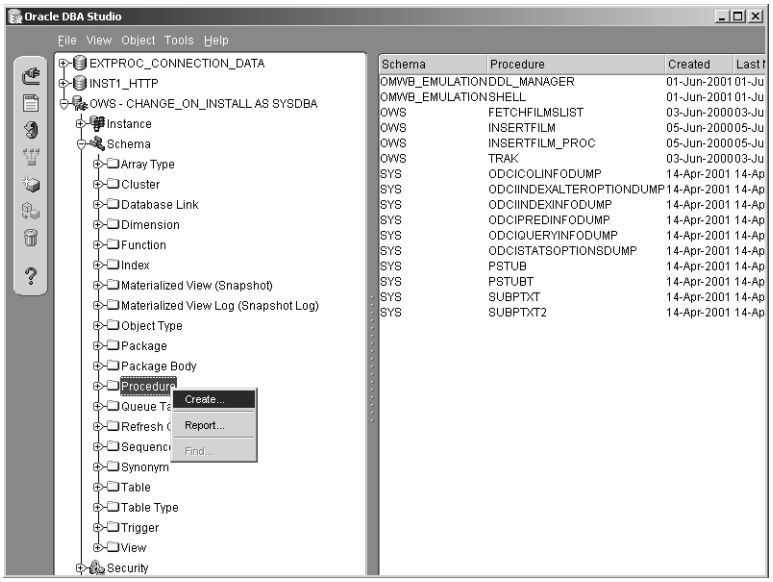
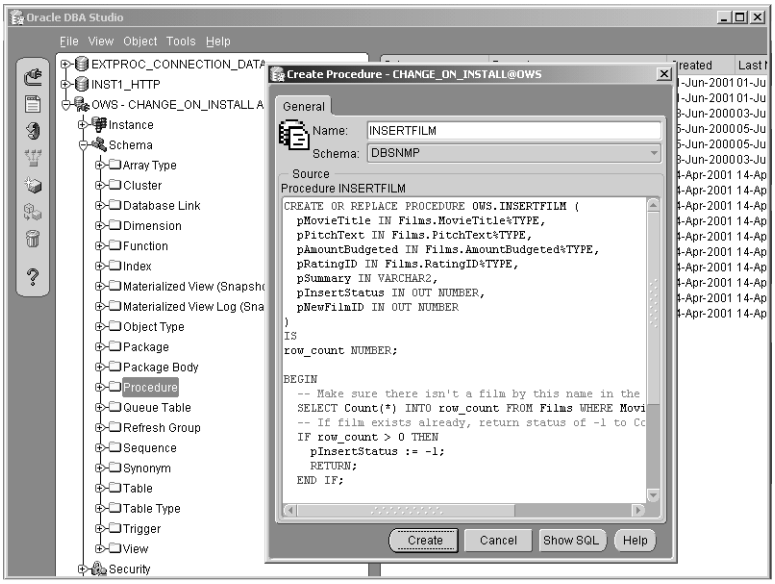


Figure 30.11

Enter the actual SQL code for the procedure into the Create Procedure dialog box.



NOTE

Instead of following the steps, you can just execute the code in Listing 30.11 directly into the Oracle SQL*Plus Worksheet utility. See your Oracle documentation for details.

Listing 30.11 Creating the `InsertFilm`—Stored Procedure for Oracle

```
CREATE OR REPLACE PROCEDURE OWS.INSERTFILM (
  pMovieTitle IN Films.MovieTitle%TYPE,
  pPitchText IN Films.PitchText%TYPE,
  pAmountBudgeted IN Films.AmountBudgeted%TYPE,
  pRatingID IN Films.RatingID%TYPE,
  pSummary IN VARCHAR2,
  pInsertStatus IN OUT NUMBER,
  pNewFilmID IN OUT NUMBER
)
IS
row_count NUMBER;

BEGIN
  -- Make sure there isn't a film by this name in the database already
  SELECT Count(*) INTO row_count FROM Films WHERE MovieTitle = pMovieTitle;
  -- If film exists already, return status of -1 to ColdFusion (or other program)
  IF row_count > 0 THEN
    pInsertStatus := -1;
    RETURN;
  END IF;

  -- Make sure the specified rating is valid
  SELECT Count(*) INTO row_count FROM FilmsRatings WHERE RatingID = pRatingID;
  -- If specified rating code does not exist, return status of -2
  IF row_count = 0 THEN
    pInsertStatus := -2;
    RETURN;
  END IF;

  -- Insert the new film into the Films table
  INSERT INTO Films (
    FilmID, MovieTitle, PitchText,
    AmountBudgeted, RatingID, Summary)
  VALUES (
    FilmsSeq.NEXTVAL, pMovieTitle, pPitchText,
    pAmountBudgeted, pRatingID, pSummary);

  -- Set output parameter called NewFilmID to the ID of the just-inserted film
  SELECT FilmsSeq.CURRVAL INTO pNewFilmID FROM DUAL;
  -- Return a status of 1 to ColdFusion (or other program) to indicate success
  pInsertStatus := 1;
END;
```

NOTE

Listing 30.11 assumes that an Oracle sequence called `FilmsSeq` is being used to autogenerate `FilmID` values. See your Oracle documentation for details about sequences.

Note that the SQL syntax for creating the stored procedure is slightly different from what it is for SQL Server, but the basic idea is the same. There's nothing here that's conceptually different from the SQL Server version of the procedure (refer to Listing 30.9).

The procedure name is provided first and follows the words `CREATE OR REPLACE PROCEDURE`. Then the procedure's parameters, if any, are provided within a pair of parentheses. The parentheses are followed by the `IS` keyword, and then the procedure's actual SQL code is provided between `BEGIN`

and END keywords. Within the code, the actual value for the output parameter is set using Oracle's assignment operator, which is a colon followed by an equal sign (:=).

NOTE

Oracle requires that you place a semicolon after each statement in your SQL code. If you've done any work with C or C++ in the past, that semicolon is a familiar friend. See your Oracle documentation for details.

Because the Oracle version of the InsertFilm stored procedure created in Listing 30.11 returns the insert status as an output parameter rather than as the status code, the ColdFusion code that uses the stored procedure must be changed slightly. Listing 30.12 is a revised version of Listing 30.4, which adds a <CFPROCPARAM> tag to capture the value of pNewFilmID. It also eliminates the RETURNCODE="Yes" attribute from the first <CFSTOREDPROC> tag.

Listing 30.12 FilmEntry2Oracle.cfm—Using the Oracle Version of the InsertFilm Stored Procedure

```
<!---
  Filename: FilmEntry2Oracle.cfm
  Author:   Nate Weiss (NMW)
  Purpose:  Demonstrates use of stored procedures
--->


<!--- Is the form being submitted? --->
<CFSET WasFormSubmitted = IsDefined("FORM.RatingID")>

<!--- Insert film into database when form is submitted --->
<CFIF WasFormSubmitted>
  <CFSTOREDPROC
    PROCEDURE="InsertFilm"
    DATASOURCE="owsOracle"
    RETURNCODE="No">

    <!--- Provide form values to the procedure's input parameters --->
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="50"
      CFSQLTYPE="CF_SQL_VARCHAR"
      VALUE="#Form.MovieTitle#">
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="100"
      CFSQLTYPE="CF_SQL_VARCHAR"
      VALUE="#Form.PitchText#">
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="100"
      CFSQLTYPE="CF_SQL_INTEGER"
      VALUE="#Form.AmountBudgeted#"
      NULL="#YesNoFormat(FORM.AmountBudgeted EQ '')#">
    <CFPROCPARAM
      TYPE="In"
      MAXLENGTH="100"
      CFSQLTYPE="CF_SQL_INTEGER"
      VALUE="#Form.RatingID#">
    <CFPROCPARAM
```

Listing 30.12 (continued)

```
          TYPE="In"
          CFSQLTYPE="CF_SQL_LONGVARCHAR"
          VALUE="#Form.Summary#">

        <!--- Capture pInsertStatus output parameter --->
        <!--- Value will be available in CFML variable named #InsertStatus# --->
        <CFPROCPARAM
          TYPE="Out"
          CFSQLTYPE="CF_SQL_INTEGER"
          VARIABLE="InsertStatus">

        <!--- Capture pNewFilmID output parameter --->
        <!--- Value will be available in CFML variable named #InsertedFilmID# --->
        <CFPROCPARAM
          TYPE="Out"
          CFSQLTYPE="CF_SQL_INTEGER"
          VARIABLE="InsertedFilmID">

      </CFSTOREDPROC>

    </CFIF>

    <!--- Get list of ratings from database --->
    <CFSTOREDPROC PROCEDURE="owsWeb.FetchRatingsList" DATASOURCE="owsOracle">
      <!--- Make the data from reference cursor available as CFML query object --->
      <CFPROCRESULT NAME="GetRatings">
    </CFSTOREDPROC>


    <HTML>
    <HEAD><TITLE>Film Entry Form</TITLE></HEAD>
    <BODY>
    <H2>Film Entry Form</H2>

    <!--- Data entry form --->
    <CFFORM ACTION="#CGI.SCRIPT_NAME#" METHOD="POST" PRESERVEDATA="Yes">

      <!--- Text entry field for film title --->
      <P><B>Title for New Film:</B><BR>
      <CFINPUT
        NAME="MovieTitle"
        SIZE="50"
        MAXLENGTH="50"
        REQUIRED="Yes"
        MESSAGE="Please don't leave the film's title blank."><BR>

      <!--- Text entry field for pitch text --->
      <P><B>Short Description / One-Liner:</B><BR>
      <CFINPUT
        NAME="PitchText"
        SIZE="50"
        MAXLENGTH="100"
        REQUIRED="Yes"
        MESSAGE="Please don't leave the one-liner blank."><BR>
```

Listing 30.12 (continued)

```
    <!--- Text entry field for expense description --->
    <P><B>New Film Budget:</B><BR>
    <CFINPUT
      NAME="AmountBudgeted" SIZE="15"
      REQUIRED="No"
      MESSAGE="Only numbers may be provided for the film's budget."
      VALIDATE="float"> (leave blank if unknown)<BR>

    <!--- Drop-down list of ratings --->
    <P><B>Rating:</B><BR>
    <CFSELECT
      NAME="RatingID"
      QUERY="GetRatings"
      VALUE="RatingID"
      DISPLAY="Rating"/>

    <!--- Text areas for summary --->
    <P><B>Summary:</B><BR>
    <TEXTAREA NAME="Summary" COLS="40" ROWS="3" WRAP="soft"></TEXTAREA>

    <!--- Submit button for form --->
    <P><INPUT TYPE="Submit" VALUE="Submit New Film">
  </CFFORM>


  <!--- If we executed the stored procedure --->
  <CFIF WasFormSubmitted>
    <!--- Display message based on status code reported by stored procedure --->
    <CFSWITCH EXPRESSION="#InsertStatus#">
      <!--- If the stored procedure returned a "success" status --->
      <CFCASE VALUE="1">
        <CFOUTPUT>
          <P>Film "#Form.MovieTitle#" was inserted as Film ID #InsertedFilmID#.<BR>
        </CFOUTPUT>
      </CFCASE>
      <!--- If the status code was -1 --->
      <CFCASE VALUE="-1">
        <CFOUTPUT>
          <P>Film "#Form.MovieTitle#" already exists in the database.<BR>
        </CFOUTPUT>
      </CFCASE>
      <!--- If the status code was -2 --->
      <CFCASE VALUE="-2">
        <P>An invalid rating was provided.<BR>
      </CFCASE>
      <!--- If any other status code was returned --->
      <CFDEFAULTCASE>
        <P>The procedure returned an unknown status code.<CFOUTPUT>#InsertStatus#
  </CFOUTPUT><BR>
      </CFDEFAULTCASE>
    </CFSWITCH>
  </CFIF>

  </BODY>
  </HTML>
```

Creating the FetchRatingsList Stored Procedure

As you learned in Listing 30.3, Oracle stored procedures can't return recordsets per se; instead, they can expose reference cursors as output parameters. Although this is an important distinction within the Oracle universe, you can treat reference cursors just like traditional recordsets in your ColdFusion template code, just by adding a `<CFPROCRESULT>` tag to capture the data in each reference cursor.

Creating a stored procedure that exposes a reference cursor requires a bit more code than the equivalent SQLServer procedure; however, the concepts are very similar. You should consult your Oracle documentation for all the details. That said, we can provide you with enough information here to help get you started.

The first step is to create a package to hold the cursor definition and your procedure. Within the package, define a cursor type that contains the columns you want to send back to ColdFusion (if you are going to select all columns from a single table, consider using the special `%ROWTYPE` attribute, as shown in the next listing). Also within the package, provide a declaration for the stored procedure itself, which is basically the first line of the procedure without the words `CREATE OR REPLACE`.

Within the package body, create the stored procedure itself, using the `OPEN`, `FOR`, and `SELECT` keywords to open the cursor and fill it with the appropriate data.

Listing 30.13 shows how the `FetchRatingsList` stored procedure can be created on an Oracle server. You could issue this code from the Oracle SQL*Plus Worksheet application. Or, you could provide the first part of the code (starting with the first `AS`) to the Oracle DBA Studio application in the Create Package dialog box, and the second part (starting with the second `AS`) to the Create Procedure dialog box.

Listing 30.13 Exposing Reference Cursors from Oracle Stored Procedures

```
CREATE OR REPLACE PACKAGE OWS.OWSWEB AS
  TYPE RatingsCurType IS REF CURSOR RETURN FilmsRatings%ROWTYPE;
  PROCEDURE FetchRatingsList(RatingsCur OUT RatingsCurType);
END owsWeb;
/

CREATE OR REPLACE PACKAGE BODY OWS.OWSWEB AS
  PROCEDURE FetchRatingsList(RatingsCur OUT RatingsCurType) IS
  BEGIN
    OPEN RatingsCur FOR
    SELECT * FROM FilmsRatings ORDER BY RatingID;
  END FetchRatingsList;
END owsWeb;
/
```

NOTE

An Oracle stored procedure that returns multiple resultsets would simply include additional `IS REF CURSOR` lines in the package declaration and then include output parameters for each cursor type between the parentheses that follow the procedure name.

Creating Stored Procedures with Sybase

Creating stored procedures with Sybase is extremely similar to creating them with Microsoft SQL Server. This has a lot to do with the fact that SQL Server originally was developed as a joint effort between Microsoft and Sybase. Both products supported almost the same functionality until relatively recently. See your Sybase documentation for details.