

CHAPTER 21

Interacting with Email

Sending Email from ColdFusion

ColdFusion's main purpose is to create dynamic, data-driven Web pages. But it also provides a set of email-related tags that let you send email messages that are just as dynamic and data-driven as your Web pages. You can also write ColdFusion templates that check and retrieve email messages, and even respond to them automatically.

NOTE

<cfmail> sends standard, Internet-style email messages using the Simple Mail Transport Protocol (SMTP). SMTP isn't explained in detail in this book; for now, all you need to know about SMTP is that it's the standard for sending email on the Internet. Virtually all email programs, such as Thunderbird, Outlook Express, Eudora, and so on send standard SMTP mail. The exceptions are proprietary messaging systems, such as Lotus Notes or older Microsoft Mail (MAPI-style) clients. If you want to learn more about the underpinnings of the SMTP protocol, visit the World Wide Web Consortium's Web site at www.w3.org.

Introducing the `<cfmail>` Tag

You can use the `<cfmail>` tag to send email messages from your ColdFusion templates. After the server is set up correctly, you can use `<cfmail>` to send email messages to anyone with a standard Internet-style email address. As far as the receiver is concerned, the email messages you send with `<cfmail>` are just like messages sent via a normal email sending program, such as Thunderbird, Outlook Express, Eudora, OS X Mail, or the like.

Table 21.1 shows the key attributes for the `<cfmail>` tag. These are the attributes you will use most often. For clarity, they are presented here in a separate table. You have almost certainly sent email messages before, so you will immediately understand what most of these attributes do.

NOTE

Some additional `<cfmail>` attributes are introduced later in this chapter (see Table 21.2 in the “Sending Data-Driven Mail” section and Table 21.4 in the “Overriding the Default Mail Server Settings” section).

Table 21.1 Key `<cfmail>` Attributes for Sending Email Messages

Attribute	Purpose
<code>subject</code>	Required. The subject of the email message.
<code>from</code>	Required. The email address that should be used to send the message. This is the address the message will be from when it's received. The address must be a standard Internet-style email address (see the section “Using Friendly Email Addresses,” later in this chapter).
<code>to</code>	Required. The address or addresses to send the message to. To specify multiple addresses, separate them with commas. Each must be a standard Internet-style email address (see the section “Using Friendly Email Addresses”).
<code>cc</code>	Optional. Address or addresses to send a carbon copy of the message to. This is the equivalent of using the CC feature when sending mail with a normal email program. To specify multiple addresses, separate them with commas. Each must be a standard Internet-style email address (see the section “Using Friendly Email Addresses”).
<code>bcc</code>	Optional. Address or addresses to send a blind carbon copy of the message to. Equivalent to using the BCC feature when sending mail with a normal email program. To specify multiple addresses, separate them with commas. Each must be a standard Internet-style email address (see the section “Using Friendly Email Addresses”).
<code>replyTo</code>	Optional. Specifies the address replies will be sent to.
<code>failTo</code>	Optional. Specifies an address where failure notifications can be sent.
<code>type</code>	Optional. <code>Text</code> or <code>HTML</code> . <code>Text</code> is the default, which means that the message will be sent as a normal, plain-text message. <code>HTML</code> means that HTML tags within the message will be interpreted as HTML, so you can specify fonts and include images in the email message. See “Sending HTML-Formatted Mail,” later in this chapter.
<code>wrapText</code>	Optional. If specified, the text in your email will automatically wrap according to the number passed in. A typical value for <code>wrapText</code> is 72. The default value is to not wrap text.
<code>mailerID</code>	Optional. Can be used to specify the X-Mailer header that is sent with the email message. The X-Mailer header is meant to identify which software program was used to send the message. This header is generally never seen by the recipient of the message but can be important to systems in between, such as firewalls. Using the <code>mailerID</code> , you can make it appear as if your message is being sent by a different piece of software. If you find that your outgoing messages are being filtered out when sent to certain users, try using a <code>mailerID</code> that matches another mail client (such as Outlook Express or some other popular, end-user mail client).
<code>mimeAttach</code>	Optional. A document on the server's drive that should be included in the mail message as an attachment. This is an older way to specify attachments, maintained for backward compatibility. It's now recommended that you use the <code><cfmailparam></code> tag to specify attachments. For details, see “Adding Attachments,” later in this chapter.
<code>charset</code>	Optional. This specifies the character encoding that will be used in the email. It defaults to the value set in the ColdFusion Administrator.
<code>spoolEnable</code>	Optional. Controls whether the email message should be sent right away, before ColdFusion begins processing the rest of the template. The default value is <code>Yes</code> , which means that the message is created and placed in a queue. The actual

	sending will take place as soon as possible, but not necessarily before the page request has been completed. If you use <code>spoolEnable="No"</code> , the message will be sent right away; ColdFusion won't proceed beyond the <code><cfmail></code> tag until the sending has been completed. In other words, <code>No</code> forces the mail sending to be a synchronous process; <code>Yes</code> (the default) lets it be an asynchronous process.
<code>priority</code>	Optional. Determines the priority of the email. This can be either a number from 1 to 5, with 1 representing the most important priority, or a string value from the following list: <code>highest</code> (or <code>urgent</code>), <code>high</code> , <code>normal</code> , <code>low</code> (or <code>non-urgent</code>).

Specifying a Mail Server in the Administrator

Before you can actually use the `<cfmail>` tag to send email messages, you need to specify a mail server in the ColdFusion Administrator. This is the mail server with which ColdFusion will interact to actually send the messages generated by your templates.

To set up ColdFusion to send email, follow these steps:

1. If you don't know it already, find out the host name or IP address for the SMTP mail server ColdFusion should use to send messages. Usually, this is the same server your normal email client program (Outlook Express, Eudora, and so on) uses to send your own mail, so you typically can find the host name or IP address somewhere in your mail client's Settings or Preferences. Often, the host name starts with something such as `mail` or `smtp`, as in `mail.orangewhipstudios.com`.
2. Open the ColdFusion Administrator, and navigate to the Mail page, as shown in Figure 21.1.
3. Provide the mail server's host name or IP address in the Mail Server field.
4. Check the Verify Mail Server Connection option.
5. If your mail server operates on a port other than the usual port number 25, provide the port number in the Server Port field. (This usually isn't necessary.)
6. Save your changes by clicking the Submit Changes button.

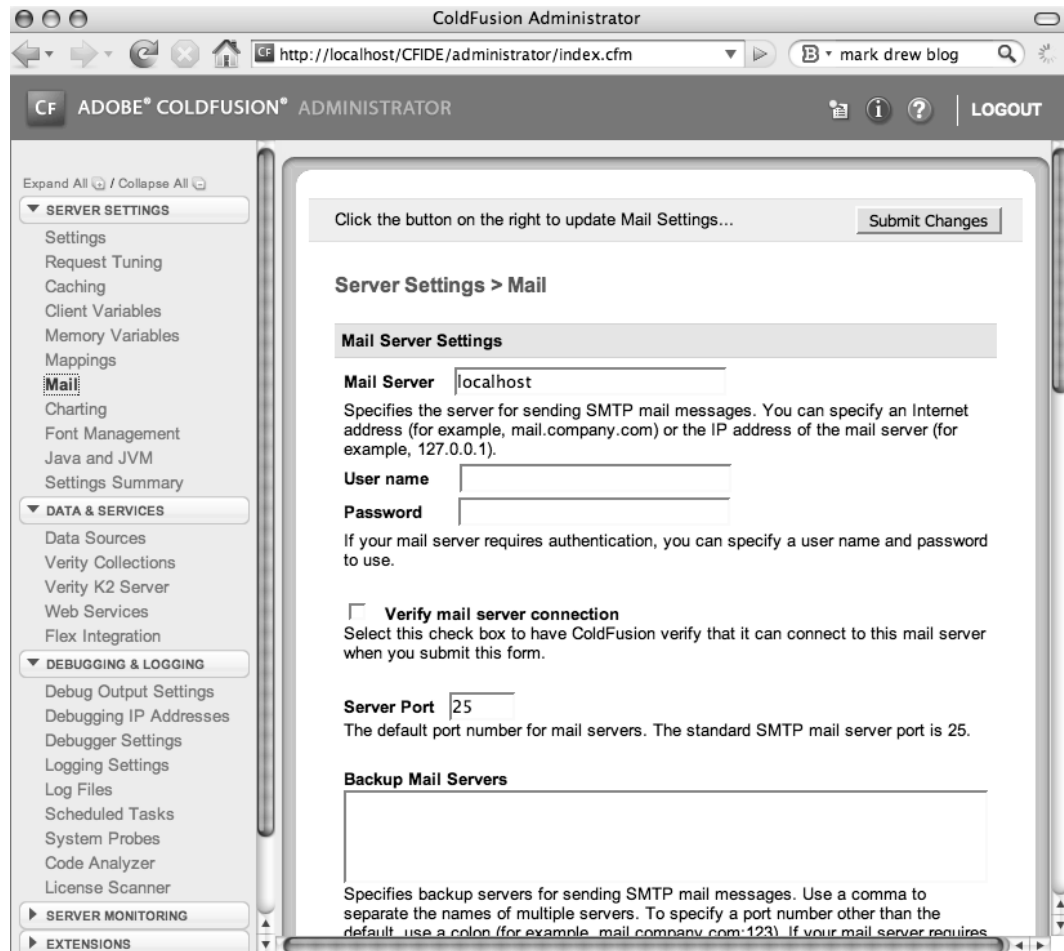


Figure 21.1

Before messages can be sent, ColdFusion needs to know which mail server to use.

TIP

These settings can be overridden in individual ColdFusion templates by the <cfmail> tag. See the “Overriding the Default Mail Server Settings” section, later in this chapter.

- ▶ *For more information about the other Mail Server options shown in Figure 21.1, see Chapter 28, “ColdFusion Server Configuration,” in ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development.*

Sending Email Messages

Sending an email message via a ColdFusion template is easy. Simply code a pair of opening and closing <cfmail> tags, and provide the to, from, and subject attributes as appropriate. Between the tags, type the actual message that should be sent to the recipient.

Of course, you can use ColdFusion variables and functions between the <cfmail> tags to build the message dynamically, using the # sign syntax you’re used to. You don’t need to place <cfoutput> tags within (or outside) the <cfmail> tags; your # variables and expressions will be evaluated as if there were a <cfoutput> tag in effect.

TIP

As you look through the examples in this chapter, you will find that the `<cfmail>` tag is basically a specially modified `<cfoutput>` tag. It has similar behavior (variables and expressions are evaluated) and attributes (`group`, `maxrows`, and so on, as listed in Table 21.2).

Sending a Simple Message

Listing 21.1 shows how easy it is to use the `<cfmail>` tag to send a message. The idea behind this template is to provide a simple form for people working in Orange Whip Studios' personnel department. Rather than having to open their normal email client programs, they can just use this Web page. It displays a simple form for the user to type a message and specify a recipient. When the form is submitted, the message is sent.

Listing 21.1 PersonnelMail1.cfm—Sending Email with ColdFusion

```
<!---
Filename: PersonnelMail1.cfm
Author: Nate Weiss (NMW)
Purpose: A simple form for sending email
-->

<html>
<head>
  <title>Personnel Office Mailer</title>
  <!--- Apply simple CSS formatting to <th> cells --->
  <style>
    th { background:blue;color:white;text-align:right}
  </style>
</head>
<body>

<h2>Personnel Office Mailer</h2>

<!--- If the user is submitting the Form... --->
<cfif isDefined("FORM.subject")>

  <!--- We do not want ColdFusion to suppress whitespace here --->
  <cfprocessingdirective suppressWhitespace="No">

<!--- Send the mail message, based on form input --->
<cfmail
  subject="#FORM.subject#"
  from="personnel@orangewhipstudios.com"
  to="#FORM.toAddress#"
  bcc="personneldirector@orangewhipstudios.com"
>This is a message from the Personnel Office:
#FORM.messageBody#

If you have any questions about this message, please
write back or call us at extension 352. Thanks!</cfmail>

</cfprocessingdirective>
```

```
<!--- Display "success" message to user --->
<p>The email message was sent.<br>
By the way, you look fabulous today.
You should be in pictures!<br>

<!--- Otherwise, display the form to user... --->
<cfelse>
<!--- Provide simple form for recipient and message --->
<cfform action="#cgi.script_name#" method="post">

<table cellpadding="2" cellspacing="2">
<!--- Table row: Input for Email Address --->
<tr>
<th>EMail Address:</th>
<td>
<cfinput type="text" name="toAddress" required="yes" size="40"
message="You must provide an email address.">
</td>
</tr>

<!--- Table row: Input for E-mail Subject --->
<tr>
<th>Subject:</th>
<td>
<cfinput type="text" name="subject" required="yes" size="40"
message="You must provide a subject for the email.">
</td>
</tr>

<!--- Table row: Input for actual Message Text --->
<tr>
<th>Your Message:</th>
<td>
<cftextarea name="messageBody" cols="30" rows="5" wrap="hard"
required="yes" message="You must provide a message body." />
</td>
</tr>

<!--- Table row: Submit button to send message --->
<tr>
<td>&nbsp;</td>
<td>
<cfinput type="submit" name="submit" value="Send Message Now">
</td>
</tr>
</table>
</cfform>
</cfif>

</body>
</html>
```

There are two parts to this listing, divided by the large `<cfif>/<cfelse>` block. When the page is first visited, the second part of the template executes, which displays the form shown in Figure 21.2.



Figure 21.2

Creating a Web-based mail-sending mechanism for your users is easy.

When the form is submitted, the first part of the template kicks in, which actually sends the email message with the `<cfmail>` tag. The message's subject line and "to" address are specified by the appropriate form values, and the content of the message itself is constructed by combining the `#FORM.messageBody#` variable with some static text. Additionally, each message sent by this template is also sent to the personnel director as a blind carbon copy, via the `bcc` attribute.

Around the `<cfmail>` tag, the `<cfprocessingdirective>` tag is used to turn off ColdFusion's default white-space-suppression behavior. This is needed in this template because the `<cfmail>` tag that follows is written to output the exact text of the email message, which includes "newlines" and other white space characters that should be included literally in the actual email message. Without the `<cfprocessingdirective>` tag, ColdFusion would see the newlines within the `<cfoutput>` tags as evil white space, deserving to be ruthlessly suppressed.

- *For more information about white-space suppression and the `<cfprocessingdirective>` tag, see the "Controlling White Space" section in Chapter 31, "Improving Performance," in Vol. 2, Application Development.*

NOTE

There is a reason why the opening and closing `<cfmail>` tags are not indented in this listing. If they were, the spaces or tabs used to do the indenting would show up in the actual email message. You will need to make some exceptions to your usual indenting practices when using `<cfmail>`. The exception would be when using `type="HTML"`, as discussed in the "Sending HTML-Formatted Mail" section, because white space isn't significant in HTML.

Using Friendly Email Addresses

The email address provided to the `to`, `from`, `cc`, and `bcc` attributes can be specified as just the email address itself (such as `rcamden@orangewhipstudios.com`), or as a combination of the address and the address's friendly name. The friendly name is usually the person's real-life first and last names.

To specify a friendly name along with an email address, place the friendly name between double quotation marks, followed by the actual email address between angle brackets. So, instead of

```
rcamden@orangewhipstudios.com
```

you would provide

```
"Raymond Camden" <rcamden@orangewhipstudios.com>
```

To provide such an address to the `from`, `to`, `cc`, or `bcc` attribute of the `<cfmail>` tag, you must double up each double quotation mark shown above, assuming that you are already using double quotation marks around the whole attribute value. So, you might end up with something such as the following:

```
<cfmail
  subject="Dinner Plans"
  from=" " "Nate Weiss" " <nweiss@orangewhipstudios.com>"
  to=" " "Belinda Foxile" " <bfoxile@orangewhipstudios.com>" ">
```

If you find the use of the doubled-up double quotation marks confusing, you could surround the `from` and `to` attributes with single quotation marks instead of double quotation marks, which would allow you to provide the double-quotation characters around the friendly name normally, like so:

```
<cfmail
  subject="Dinner Plans"
  from=' "Nate Weiss" <nweiss@orangewhipstudios.com>'
  to=' "Belinda Foxile" <bfoxile@orangewhipstudios.com>' '>
```

Now, when the message is sent, the “to” and “from” addresses shown in the recipient’s email program can be shown with each person’s real-life name along with their email address. How the friendly name and email address are actually presented to the user is up to the email client software.

The version of the `PersonnelMail.cfm` template shown in Listing 21.2 is nearly the same as the one from Listing 21.1, except that this version collects the recipient’s friendly name in addition to their email address. Additionally, this version uses a bit of JavaScript to attempt to pre-fill the email address field based on the friendly name. When the user changes the value in the `FirstName` or `LastName` field, the `ToAddress` field is filled in with the first letter of the first name, plus the whole last name.

NOTE

There isn’t space to go through the JavaScript code used in this template in detail. It’s provided to give you an idea of one place where JavaScript can be useful. Consult a JavaScript reference or online tutorial for details. One good place to look is the JavaScript section of the Reference tab of the Code panel in Dreamweaver.

Limiting Input

This version of the form makes it impossible to send messages to anyone outside of Orange Whip Studios, by simply hard-coding the `@orangewhipstudios.com` part of the email address into the `<cfmail>` tag itself. Also, it forces the user to select from a short list of Subject lines, rather than being able to type their own Subject, as shown in Figure 21.3.



Figure 21.3

Web-based forms can make sending email almost foolproof.

In a real-world application, you probably would make different choices about what exactly to allow users to do. The point is that by limiting the amount of input required, you can make it simpler for users to send consistent email messages, thus increasing the value of your application. This can be a lot of what differentiates Web pages that send mail from ordinary email programs, which can be more complex for users to learn.

Listing 21.2 PersonnelMail2.cfm—Providing Friendly Names Along with Email Addresses

```
<!---
Filename: PersonnelMail2.cfm
Author: Nate Weiss (NMW)
Purpose: A simple form for sending email
-->

<html>
<head>
<title>Personnel Office Mailer</title>
<!-- Apply simple CSS formatting to <th> cells -->
<style>
th { background:blue;color:white;
font-family:sans-serif;font-size:12px;
text-align:right;padding:5px;}
</style>

<!-- Function to guess email based on first/last name -->
<script language="JavaScript">
function guessEmail() {
var guess;

with (document.mailForm) {
guess = firstName.value.substr(0,1) + lastName.value;
toAddress.value = guess.toLowerCase();
} ;
} ;
</script>
```

```
</head>

<!-- Put cursor in FirstName field when page loads -->
<body <cfif not isDefined("FORM.subject")>
  onLoad="document.mailForm.firstName.focus()"
</cfif>>

<!-- If the user is submitting the form... -->
<cfif isDefined("FORM.subject")>
  <cfset recipEmail = listFirst(FORM.toAddress, "@") & "@orangewhipstudios.com">

  <!-- We do not want ColdFusion to suppress whitespace here -->
  <cfprocessingdirective suppressWhitespace="no">

  <!-- Send the mail message, based on form input -->
  <cfmail
  subject="#FORM.subject#"
  from="" "Personnel Office" " <personnel@orangewhipstudios.com>"
  to="" "#FORM.firstName# #FORM.lastName#" " <#recipEmail#"
  bcc="personneldirector@orangewhipstudios.com"
  >This is a message from the Personnel Office:

  #uCase(FORM.subject)#

  #FORM.messageBody#

  If you have any questions about this message, please
  write back or call us at extension 352. Thanks!</cfmail>

  </cfprocessingdirective>

  <!-- Display "success" message to user -->
  <p>The email message was sent.<br>
  By the way, you look fabulous today.
  You should be in pictures!<br>
  <!-- Otherwise, display the form to user... -->
  <cfelse>
  <!-- Provide simple form for recipient and message -->
  <cfform action="#cgi.script_name#" name="mailForm" method="post">

  <table cellPadding="2" cellSpacing="2">
  <!-- Table row: Input for Recipient's Name -->
  <tr>
  <th>Recipient's Name:</th>
  <td>
  <cfinput type="text" name="firstName" required="yes" size="15"
  message="You must provide a first name."
  onChange="guessEmail()">

  <cfinput type="text" name="lastName" required="yes" size="20"
  message="You must provide a first name."
  onChange="guessEmail()">
  </td>
  </tr>
```

```
<!-- Table row: Input for EMail Address -->
<tr>
<th>EMail Address:</th>
<td>
<cfinput type="text" name="toAddress" required="yes" size="20"
message="You must provide the recipient's email.">@orangewhipstudios.com
</td>
</tr>

<!-- Table row: Input for EMail Subject -->
<tr>
<th>Subject:</th>
<td>
<cfselect name="subject">
<option>Sorry, but you have been fired.
<option>Congratulations! You got a raise!
<option>Just FYI, you have hit the glass ceiling.
<option>The company dress code, Capri Pants, and you
<option>All your Ben Forta are belong to us.
</cfselect>
</td>
</tr>

<!-- Table row: Input for actual Message Text -->
<tr>
<th>Your Message:</th>
<td>
<cftextarea name="messageBody" cols="30" rows="5" wrap="hard"
required="yes" message="You must provide a message body." />
</td>
</tr>

<!-- Table row: Submit button to send message -->
<tr>
<td>&nbsp;</td>
<td>
<cfinput type="submit" name="submit" value="Send Message Now">
</td>
</tr>
</table>
</cfform>
</cfif>

</body>
</html>
```

Figure 21.4 shows what an email message generated by Listing 21.2 might look like when received and viewed in a typical email client (here, Thunderbird).

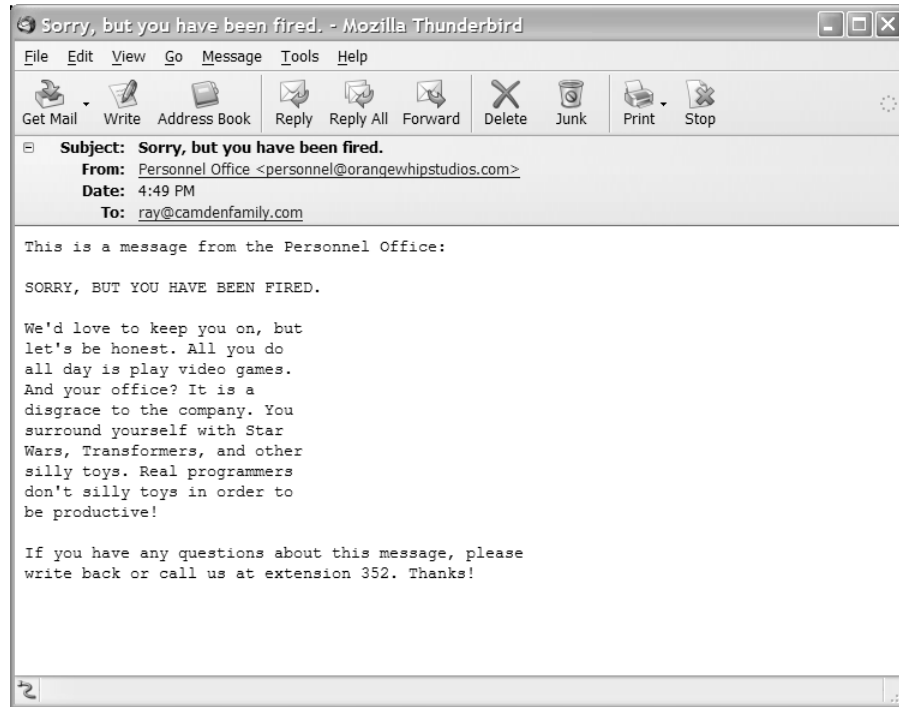


Figure 21.4

Providing a friendly name along with an email address makes for an email message that feels more personal.

Sending Data-Driven Mail

In the last section, you saw that `<cfmail>` can be thought of as an extended version of the `<cfoutput>` tag because ColdFusion variables and expressions are evaluated without the need for an explicit `<cfoutput>` within the `<cfmail>`. The similarity between the two tags doesn't end there. They also share attributes specific to the notion of looping over query records. This capability enables you to send data-driven email messages using nearly the same syntax and techniques that you use to output data-driven HTML code.

Table 21.2 shows the `<cfmail>` attributes relevant to sending data-driven mail. Each of these attributes behaves the same way as the corresponding attributes for `<cfoutput>`. Instead of causing HTML output to be repeated for each row in a query, these attributes have to do with repeating email message content for each row in a query.

Table 21.2 Additional `<cfmail>` Attributes for Sending Data-Driven Email Messages

Attribute	Purpose
query	Optional. A query to use for sending data-driven email. Very similar to the <code>QUERY</code> attribute of the <code><cfoutput></code> tag.
startRow	Optional. A number that indicates which row of the query to consider when sending data-driven email. The default is to start at the first row. Equivalent to the <code>startRow</code> attribute of the <code><cfoutput></code> tag.
maxRows	Optional. A maximum number of query rows to consider when sending data-driven email. Equivalent to the <code>maxRows</code> attribute of the <code><cfoutput></code> tag.
group	Optional. A column name from the query that indicates groups of records. Additional output or processing can occur when each new group is encountered. You can indicate nested groups by providing a comma-

	separated list of column names. Equivalent to the <code>group</code> attribute of the <code><cfoutput></code> tag.
<code>groupCaseSensitive</code>	Optional. Whether to consider text case when determining when a new group of records has been encountered in the column(s) indicated by <code>group</code> . Equivalent to the <code>groupCaseSensitive</code> attribute of the <code><cfoutput></code> tag.

NOTE

You may be thinking that a few of these additional attributes aren't really necessary. In today's ColdFusion, you usually can achieve the same results using a `<cfloop>` tag around a `<cfmail>` tag to send out multiple messages or using `<cfloop>` within `<cfmail>` to include queried information in the message itself. However, the `<cfmail>` tag appeared in CFML before the `<cfloop>` tag existed, which is one reason why these attributes exist today.

Including Query Data in Messages

By adding a `query` attribute to the `<cfmail>` tag, you can easily include query data in the email messages your application sends. Adding the `query` attribute to `<cfmail>` is similar to adding `query` to a `<cfoutput>` tag—the content inside the tags will repeat for each row of the query.

Additionally, if you add a `group` attribute, you can nest a pair of `<cfoutput>` tags within the `<cfmail>` tag. If you do, the inner `<cfoutput>` block is repeated for every record from the query, and everything outside the `<cfoutput>` block is output only when the value in the `group` column changes. This is just like the `group` behavior of the `<cfoutput>` tag itself.

- ▶ See Chapter 10, “Creating Data-Driven Pages,” for more information about grouping query results.

Listing 21.3 shows how the `<cfmail>` tag can be used with the `query` and `group` attributes to send data-driven email messages. This example creates a CFML custom tag called `<cf_SendOrderConfirmation>`, which takes one attribute called `orderID`, like this:

```
<!--- Send Confirmation E-Mail, via Custom Tag --->
<cf_SendOrderConfirmation
  orderID="3">
```

The idea is for the tag to compose an order-confirmation type of email message for the person who placed the order, detailing the items they purchased and when. If you've ever bought something online, you probably received such a confirmation email immediately after placing your order. This custom tag is used in Chapter 22, “Online Commerce,” online, after a user makes an actual purchase.

NOTE

You should save this listing as a file called `SendOrderConfirmation.cfm`, either in the special `CustomTags` folder or in the same folder as the other examples from this chapter.

- ▶ See Chapter 26, “Building Reusable Components,” in Vol. 2, Application Development, for information about the `CustomTags` folder and CFML custom tags in general.

Listing 21.3 `SendOrderConfirmation1.cfm`—Sending a Data-Driven Email Message

```
<!---
  Filename: SendOrderConfirmation1.cfm
  Author: Nate Weiss (NMW)
  Purpose: Sends an email message to the person who placed an order
-->

<!--- Tag attributes --->
<cfparam name="ATTRIBUTES.orderID" type="numeric">

<!--- Retrieve order information from database --->
<cfquery datasource="ows" name="getOrder">
  SELECT
  c.ContactID, c.FirstName, c.LastName, c.Email,
  o.OrderDate, o.ShipAddress, o.ShipCity,
  o.ShipState, o.ShipZip, o.ShipCountry,
  oi.OrderQty, oi.ItemPrice,
  m.MerchName,
  f.MovieTitle
  FROM
  Contacts c,
  MerchandiseOrders o,
  MerchandiseOrdersItems oi,
  Merchandise m,
  Films f
  WHERE
  o.OrderID = #ATTRIBUTES.orderID#
  AND c.ContactID = o.ContactID
  AND m.MerchID = oi.ItemID
  AND o.OrderID = oi.OrderID
  AND f.FilmID = m.FilmID
  ORDER BY
  m.MerchName
</cfquery>

<!--- Re-Query the GetOrders query to find total $ spent --->
<!--- The DBTYPE="Query" invokes CF's "Query Of Queries" --->
<cfquery dbtype="query" name="getTotal">
  SELECT SUM(ItemPrice * OrderQty) AS OrderTotal
  FROM GetOrder
</cfquery>

<!--- We do not want ColdFusion to suppress whitespace here --->
<cfprocessingdirective suppressWhitespace="no">

<!--- Send email to the user --->
<!--- Because of the GROUP attribute, the inner <CFOUTPUT> --->
<!--- block will be repeated for each item in the order --->
<cfmail query="getOrder" group="ContactID" groupCasesensitive="no"
  startrow="1" subject="Thanks for your order (Order number #ATTRIBUTES.orderID#)"
  to=""#FirstName# #LastName#" <#Email#>"
```

```
from="" "Orange Whip Online Store" " <orders@orangewhipstudios.com>"
>Thank you for ordering from Orange Whip Studios.
Here are the details of your order, which will ship shortly.
Please save or print this email for your records.

Order Number: #ATTRIBUTES.orderID#
Items Ordered: #recordCount#
Date of Order: #dateFormat(OrderDate, "dddd, mmmm d, yyyy")#
#timeFormat(OrderDate)#

-----
<cfoutput>
#currentRow#. #MerchName#
  (in commemoration of the film "#MovieTitle#")
  Price: #LSCurrencyFormat(ItemPrice)#
  Qty: #OrderQty#
</cfoutput>

-----
Order Total: #lsCurrencyFormat(getTotal.OrderTotal)#

This order will be shipped to:
#FirstName# #LastName#
#ShipAddress#
#ShipCity#
#ShipState# #ShipZip# #ShipCountry#

If you have any questions, please write back to us at
orders@orangewhipstudios.com, or just reply to this email.
</cfmail>

</cfprocessingdirective>
```

This listing first needs to retrieve all the relevant information about the order from the database, including the orderer's name and shipping address; the name, price, and quantity of each item ordered; and the title of the movie that goes along with each item. This is all obtained using a single query called `getOrder`, which is long but fairly straightforward.

The `getOrders` query returns one row for each item that was ordered in the specified `orderID`. Because there is, by definition, only one row for each `orderID` and only one `ContactID` for each order, the columns from the `MerchandiseOrders` and `Contacts` tables (marked with the `o` and `c` aliases in the query) will have the same values for each row. Therefore, the query can be thought of as being grouped by the `ContactID` column (or any of the other columns from the `MerchandiseOrders` or `Contacts` tables).

Next, ColdFusion's query of queries feature is used to get the grand total of the order, which is simply the sum of each price times the quantity ordered. This query returns just one row (because there is no `GROUP BY` clause) and just one column (called `OrderTotal`), which means that the total can be output at any time by referring to `getTotal.OrderTotal`.

- *For more information about `dbtype="query"`, the query of queries feature, and the `SUM` function used in this query, see Chapter 41, "More About SQL and Queries," online.*

You could forgo the `getTotal` query and just add the prices by looping over the `getOrders` query, as in the `OrderHistory2.cfm` template from Chapter 26. However, getting the total

via the query of queries feature is a quick and convenient way to obtain the total, using familiar SQL-style syntax.

NOTE

In general, you should use the `<cfprocessingdirective>` tag with `suppressWhitespace="No"` whenever you send data-driven email. The exception would be if you were using `type="HTML"` in the `<cfmail>` tag, in which case you should leave the suppression-suppression options alone. See the section "Sending HTML-Formatted Mail," later in the chapter, for details.

Now the `<cfmail>` tag is used to actually send the confirmation message. Because the `query` attribute is set to the `getOrder` query, the columns in that query can be freely referred to in the `to` attribute and the body of the email message itself. Columns specific to each item ordered are referred to within the `<cfoutput>` block. Columns specific to the order in general are referred to outside the `<cfoutput>` block, which will be repeated only once because there is only one group of records as defined by the `group` attribute (that is, all the query records have the same `ContactID` value).

NOTE

The custom tag you just created is used in Chapter 22, after a user makes an actual purchase.

Sending Bulk Messages

You can easily use ColdFusion to send messages to an entire mailing list. Simply execute a query that returns the email addresses of all the people the message should be sent to, then refer to the email column of the query in the `<cfmail>` tag's `to` attribute.

Listing 21.4 shows how easy sending a message to a mailing list is. This listing is similar to the Personnel Office Mailer templates from earlier (refer to Listings 21.1 and 21.2). It enables the user (presumably someone within Orange Whip Studios' public relations department) to type a message that will be sent to everyone on the studio's mailing list.

Listing 21.4 `SendBulkEmail.cfm`—Sending a Message to Everyone on a Mailing List

```
<!---
Filename: SendBulkEmail.cfm
Author: Nate Weiss (NMW)
Purpose: Creates form for sending email to everyone on the mailing list
-->

<html>
<head>
  <title>Mailing List</title>
  <!-- Apply simple CSS formatting to <TH> cells -->
  <style>
  th { background:blue;color:white;
font-family:sans-serif;font-size:12px;
text-align:right;padding:5px;}
  </style>
</head>

<!-- Put cursor in FirstName field when page loads -->
```

```

<body>

<!-- Page Title -->
<h2>Send Message To Mailing List</h2>

<!-- If the user is submitting the form... -->
<cfif isDefined("FORM.subject")>
  <!-- Retrieve "mailing list" records from database -->
  <cfquery datasource="ows" name="getList">
    SELECT FirstName, LastName, EMail
    FROM Contacts
    WHERE MailingList = 1
  </cfquery>

  <!-- Send the mail message, based on form input -->
  <cfmail query="getList" subject="#FORM.subject#"
    from="" "Orange Whip Studios" <mailings@orangewhipstudios.com>"
    to="" "#FirstName# #LastName#" <#EMail#>"
    bcc="personneldirector@orangewhipstudios.com"
  >#FORM.messageBody#

  -----
  We respect your privacy here at Orange Whip Studios.
  To be removed from this mailing list, reply to this
  message with the word "Remove" in the subject line.
  -----
</cfmail>

  <!-- Display "success" message to user -->
  <p>The email message was sent.<br>
  By the way, you look fabulous today.
  You should be in pictures!<br>

<!-- Otherwise, display the form to user... -->
<cfelse>
  <!-- Provide simple form for recipient and message -->
  <cfform action="#CGI.script_name#" name="mailForm" method="POST">

  <table cellPadding="2" cellSpacing="2">
    <!-- Table row: Input for email Subject -->
    <tr>
      <th>Subject:</th>
      <td>
        <cfinput type="text" name="subject" required="yes" size="40"
        message="You must provide a subject for the email.">
      </td>
    </tr>

    <!-- Table row: Input for actual Message Text -->
    <tr>
      <th>Your Message:</th>
      <td>
        <cftextarea name="messageBody" cols="30" rows="5" wrap="hard"
        required="yes" message="You must provide a message body." />
      </td>
    </tr>
  </table>

```

```
</td>
</tr>

<!-- Table row: Submit button to send message --->
<tr>
<td>&nbsp;</td>
<td>
<cfinput type="submit" name="submit" value="Send Message Now" onClick="return
confirm('Are you sure? This message will be sent to everyone on the mailing list.
This is your last chance to cancel the bulk mailing.')">
</td>
</tr>
</table>
</cfform>
</cfif>

</body>
</html>
```

Like Listings 21.1 and 21.2, this listing presents a simple form to the user, in which a subject and message can be typed. When the form is submitted, the `<cfif>` block at the top of the template is executed.

The `getList` query retrieves the name and email address for each person in the `Contacts` table who has consented to be on the mailing list (that is, where the Boolean `MailingList` column is set to 1, which represents true or yes). Then, the `<cfmail>` tag is used to send the message to each user. Because of the `query="getList"` attribute, `<cfmail>` executes once for each row in the query.

A few lines of text at the bottom of the message lets each recipient know that they can remove themselves from the mailing list by replying to the email message with the word "Remove" in the subject line. Listing 21.11—in the "Creating Automated POP Agents" section of this chapter—demonstrates how ColdFusion can respond to these remove requests.

Sending HTML-Formatted Mail

As noted in Table 21.1, you can set the optional `type` attribute of the `<cfmail>` tag to `HTML`, which enables you to use ordinary HTML tags to add formatting, images, and other media elements to your mail messages.

The following rules apply:

- The recipient's email client program must be HTML enabled. Most modern email clients, such as Outlook Express or Thunderbird, know how to display the contents of email messages as HTML. However, if the message is read in a program that isn't HTML enabled, the user will see the message literally, including the actual HTML tags.
- The mail message should be a well-formed HTML document, including . opening and closing `<html>`, `<head>`, and `<body>` tags.
- All references to external URLs must be fully qualified, absolute URLs, including the `http://` or `https://`. In particular, this includes the `href` attribute for links and the `src` attribute for images.

The version of the `<cf_SendOrderConfirmation>` tag in Listing 21.5 expands on the previous version (refer to Listing 21.3) by adding a `useHTML` attribute. If the tag is called with `useHTML="Yes"`, an HTML-formatted version of the confirmation email is sent, including small pictures of each item that was ordered (see Figure 21.5). If `useHTML` is `No` or is omitted, the email is sent as plain text (as in the previous version).

Although this approach is not covered in this chapter, you can use `<cfmailpart>` to send both HTML and plain text messages. This will send two versions of the same email to the client. The downside is that the doubling up makes each mail message that much larger.

Figure 21.5

As long as the recipient's email program supports HTML, your messages can include formatting, images, and so on.

Listing 21.5 `SendOrderConfirmation2.cfm`—Using HTML Tags to Format a Mail Message

```
<!---
  Filename: SendOrderConfirmation2.cfm
  Author: Nate Weiss (NMW)
  Purpose: Sends an email message to the person who placed an order
-->

<!--- Tag attributes --->
<cfparam name="ATTRIBUTES.orderID" type="numeric">
<cfparam name="ATTRIBUTES.useHTML" type="boolean" default="yes">

<!--- Local variables --->
<cfset imgSrcPath = "http://#CGI.HTTP_HOST#/ows/images">

<!--- Retrieve order information from database --->
<cfquery datasource="ows" name="getOrder">
  SELECT
  c.ContactID, c.FirstName, c.LastName, c.Email,
  o.OrderDate, o.ShipAddress, o.ShipCity,
  o.ShipState, o.ShipZip, o.ShipCountry,
  oi.OrderQty, oi.ItemPrice,
  m.MerchName, m.ImageNameSmall,
  f.MovieTitle
  FROM
  Contacts c,
  MerchandiseOrders o,
  MerchandiseOrdersItems oi,
  Merchandise m,
  Films f
  WHERE
  o.OrderID = #ATTRIBUTES.OrderID#
  AND c.ContactID = o.ContactID
  AND m.MerchID = oi.ItemID
  AND o.OrderID = oi.OrderID
  AND f.FilmID = m.FilmID
  ORDER BY
  m.MerchName
</cfquery>
```

```

<!--- Display an error message if query returned no records --->
<cfif getOrder.recordCount eq 0>
    <cfthrow message="Failed to obtain order information."
        detail="Either the Order ID was incorrect, or order has no detail records.">
<!--- Display an error message if email blank or not valid --->
<cfelseif isValid("email", getOwer.email)>
    <cfthrow message="Failed to obtain order information."
        detail="Email addresses need to have an @ sign and at least one 'dot'.">
</cfif>

<!--- Query the GetOrders query to find total $$ --->
<cfquery dbtype="query" name="getTotal">
    SELECT SUM(ItemPrice * OrderQty) AS OrderTotal
    FROM GetOrder
</cfquery>

<!--- *** If we are sending HTML-Formatted Email *** --->
<cfif ATTRIBUTES.useHTML>

    <!--- Send Email to the user --->
    <!--- Because of the GROUP attribute, the inner <CFOUTPUT> --->
    <!--- block will be repeated for each item in the order --->
    <cfmail query="getOrder" group="ContactID" groupCasesensitive="No"
        subject="Thanks for your order (Order number #ATTRIBUTES.orderID#)"
        to="""#FirstName# #LastName#" " <#Email#">
        from="""Orange Whip Online Store"" <orders@orangewhipstudios.com">
        type="HTML">

    <html>
    <head>
    <style type="text/css">
    body { font-family:sans-serif;font-size:12px;color:navy}
    td { font-size:12px}
    th { font-size:12px;color:white;
    background:navy;text-align:left}
    </style>
    </head>
    <body>

    <h2>Thank you for your Order</h2>

    <p><b>Thank you for ordering from
    <a href="http://www.orangewhipstudios.com">Orange Whip Studios</a>.</b><br>
    Here are the details of your order, which will ship shortly.
    Please save or print this email for your records.<br>

    <p>
    <strong>Order Number:</strong> #ATTRIBUTES.orderID#<br>
    <strong>Items Ordered:</strong> #recordCount#<br>
    <strong>Date of Order:</strong>
    #dateFormat(OrderDate, "dddd, mmmm d, yyyy")#
    #timeFormat(OrderDate)#<br>

```

```

<table>
<cfoutput>
<tr valign="top">
<th colspan="2">
#MerchName#
</th>
</tr>
<tr>
<td>
<!-- If there is an image available... -->
<cfif ImageNameSmall neq "">

</cfif>
</td>
<td>
<em>(in commemoration of the film "#MovieTitle#")</em><br>
<strong>Price:</strong> #lsCurrencyFormat(ItemPrice)#<br>
<strong>Qty:</strong> #OrderQty#<br>&nbsp;<br>
</td>
</tr>
</cfoutput>
</table>

<p>Order Total: #lsCurrencyFormat(getTotal.OrderTotal)#<br>

<p><strong>This order will be shipped to:</strong><br>
#FirstName# #LastName#<br>
#ShipAddress#<br>
#ShipCity#<br>
#ShipState# #ShipZip# #ShipCountry#<br>

<p>If you have any questions, please write back to us at
<a href="orders@orangewhipstudios.com">orders@orangewhipstudios.com</a>,
or just reply to this email.<br>
</body>
</html>
</cfmail>

<!-- *** If we are NOT sending HTML-Formatted Email *** -->
<cfelse>

<!-- We do not want ColdFusion to suppress whitespace here -->
<cfprocessingdirective suppressWhitespace="no">

<!-- Send email to the user -->
<!-- Because of the GROUP attribute, the inner <CFOUTPUT> -->
<!-- block will be repeated for each item in the order -->
<cfmail query="getOrder" group="ContactID" groupCasesensitive="No"
subject="Thanks for your order (Order number #ATTRIBUTES.OrderID#)"
to=""#FirstName# #LastName#" <#Email#>"
from=""Orange Whip Online Store" <orders@orangewhipstudios.com>"
>Thank you for ordering from Orange Whip Studios.

```

Here are the details of your order, which will ship shortly.
Please save or print this email for your records.

```
Order Number: #ATTRIBUTES.orderID#
Items Ordered: #recordCount#
Date of Order: #dateFormat(OrderDate, "dddd, mmmm d, yyyy")#
               #timeFormat(OrderDate)#
```

```
-----
<cfoutput>
#currentRow#. #MerchName#
  (in commemoration of the film "#MovieTitle#")
  Price: #lsCurrencyFormat(ItemPrice)#
  Qty: #OrderQty#
</cfoutput>
```

```
-----
Order Total: #lsCurrencyFormat(getTotal.OrderTotal)#
```

```
This order will be shipped to:
#FirstName# #LastName#
#ShipAddress#
#ShipCity#
#ShipState# #ShipZip# #ShipCountry#
```

```
If you have any questions, please write back to us at
orders@orangewhipstudios.com, or just reply to this email.
```

```
</cfmail>
```

```
</cfprocessingdirective>
```

```
</cfif>
```

In most respects, Listing 21.5 is nearly identical to the prior version (refer to Listing 21.3). A simple `<cfif>` determines whether the tag is being called with `useHTML="Yes"`. If so, `<cfmail>` is used with `type="HTML"` to send an HTML-formatted message. If not, a separate `<cfmail>` tag is used to send a plain-text message. Note that the `<cfprocessingdirective>` tag is needed only around the plain-text version of the message because HTML isn't sensitive to white space.

As already noted, a fully qualified URL must be provided for images to be correctly displayed in email messages. To make this easier, a variable called `imgSrcPath` is defined at the top of the template, which will always hold the fully qualified URL path to the `ows/images` folder. This variable can then be used in the `src` attribute of any `` tags within the message. For instance, assuming that you are visiting a copy of ColdFusion server on your local machine, this variable will evaluate to something such as `http://localhost/ows/images/`.

Note

The `CGI.HTTP_HOST` variable can be used to refer to the host name of the ColdFusion server. The `CGI.SERVER_NAME` also could be used to get the same value.

In addition, Listing 21.5 does two quick checks after the `getOrder` query to ensure that it makes sense for the rest of the template to continue. If the query fails to return any records, the `orderID` passed to the tag is assumed to be invalid, and an appropriate error

message is displayed. An error message is also displayed if the `email` column returned by the query is not a valid email.

The error messages created by the `<cfthrow>` tags in this example can be caught with the `<cfcatch>` tag, as discussed in Chapter 51, “Error Handling,” online.

NOTE

If the recipient doesn't use an HTML-enabled mail client to read the message, the message will be shown literally, including the actual HTML tags. Therefore, you should send messages of `type="HTML"` only if you know the recipient is using an HTML-enabled email client program.

Adding Custom Mail Headers

All SMTP email messages contain a number of mail headers, which give Internet mail servers the information necessary to route the message to its destination. Mail headers also provide information used by the email client program to show the message to the user, such as the message date and the sender's email address.

ColdFusion allows you to add your own mail headers to mail messages, using the `<cfmailparam>` tag.

TIP

You can see what these mail headers look like by using an ordinary email client program. For instance, in Outlook Express, highlight a message in your Inbox, select Properties from the File menu, and then click the Details tab.

Introducing the `<cfmailparam>` Tag

ColdFusion provides a tag called `<cfmailparam>` that can be used to add custom headers to your mail messages. It also can be used to add attachments to your messages, which is discussed in the next section. The `<cfmailparam>` tag is allowed only between opening and closing `<cfmail>` tags. Table 21.3 shows which attributes can be provided to `<cfmailparam>`.

Table 21.3 `<cfmailparam>` Tag Attributes

Attribute	Purpose
name	The name of the custom mail header you want to add to the message. You can provide any mail header name you want. (You must provide a <code>name</code> or <code>file</code> attribute, but not both in the same <code><cfmailparam></code> tag.)
value	The actual value for the mail header specified by <code>name</code> . The type of string you provide for <code>value</code> will depend on which mail header you are adding to the message. Required if the <code>name</code> attribute is provided.
file	The filename of the document or other file that should be sent as an attachment to the mail message. The filename must include a fully qualified, file-system-style path—for instance a drive letter if ColdFusion is running on a Windows machine. (You must provide a <code>name</code> or <code>file</code> attribute, but not both in the same <code><cfmailparam></code> tag.)
type	Describes the MIME media type of the file. This must either be a valid MIME type or one of the following simpler values: <code>text</code> (same as MIME type <code>text/plain</code>), <code>plain</code> (same as MIME type <code>text/plain</code>), or <code>html</code> (same as MIME type <code>text/html</code>).
contentID	Specifies an identifier for the attached file. This is used to identify the file that an <code>img</code> or other tag in the email uses.

disposition	This attribute describes how the file should be attached to the email. There are two possible values, <code>attachment</code> and <code>inline</code> . The default, <code>attachment</code> , means the file is added as attachment. If you specify <code>inline</code> , the file will be included in the message.
-------------	--

Adding Attachments

As noted in Table 21.3, you can also use the `<cfmailparam>` tag to add a file attachment to a mail message. Simply place a `<cfmailparam>` tag between the opening and closing `<cfmail>` tags, specifying the attachment's filename with the `file` attribute. The filename must be provided as a fully qualified file-system path, including the drive letter and volume name. It can't be expressed as a relative path or URL.

NOTE

The filename you provide for a `file` must point to a location on the ColdFusion server's drives (or a location on the local network). It can't refer to a location on the browser machine. ColdFusion has no way to grab a document from the browser's drive. If you want a user to be able to attach a file to a `<cfmail>` email, you first must have them upload the file to the server. See Chapter 70, "Interacting with the Operating System," in Vol. 3, Advanced Application Development, for details about file uploads.

TIP

The attachment doesn't have to be in your Web server's document root. In fact, you might want to ensure that it's not, if you want people to be able to access it only via email, rather than via the Web.

To add a Word document called `BusinessPlan.doc` as an attachment, you might include the following `<cfmailparam>` tag between your opening and closing `<cfmail>` tags:

```
<!-- Attach business plan document to message --->
<cfmailparam
  file="c:\OwsMailAttachments\ BusinessPlan.doc">
```

TIP

To add multiple attachments to a message, simply provide multiple `<cfmailparam>` tags, each specifying one file.

NOTE

As noted in Table 21.1, you also can use the older `mimeattach` attribute of the `<cfmail>` tag to add an attachment, instead of coding a separate `<cfmailparam>` tag. However, it's recommended that you use `<cfmailparam>` instead because it's more flexible (it allows you to add more than one attachment to a single message).

Overriding the Default Mail Server Settings

Earlier in this chapter, you learned about the settings on the Mail/Mail Logging page of the ColdFusion Administrator (refer to Figure 21.1). These settings tell ColdFusion which mail server to communicate with to send the messages that your templates generate. In most situations, you can simply provide these settings once, in the ColdFusion Administrator, and forget about them. ColdFusion will use the settings to send all messages.

However, you might encounter situations in which you want to specify the mail server settings within individual `<cfmail>` tags. For instance, your company might have two mail servers set up, one for bulk messages and another for ordinary messages. Or you might not have access to the ColdFusion Administrator for some reason, perhaps because your application is sitting on a shared ColdFusion server at an Internet service provider (ISP).

To specify the mail server for a particular `<cfmail>` tag, add the `server` attribute, as explained in Table 21.4. You also can provide the `port` and `timeout` attributes to completely override all mail server settings from the ColdFusion Administrator.

NOTE

If you need to provide these attributes for your `<cfmail>` tags, consider setting a variable called `APPLICATION.mailServer` in your `Application.cfc` file and then specifying `server="#APPLICATION.mailServer#" for each <cfmail> tag.`

Table 21.4 Additional `<cfmail>` Attributes for Overriding the Mail Server Settings

Attribute	Purpose
<code>server</code>	Optional. The host name or IP address of the mail server ColdFusion should use to actually send the message. If omitted, this defaults to the Mail Server setting on the Mail/Mail Logging page of the ColdFusion Administrator. If you are using the Enterprise edition of ColdFusion, a list of mail servers can be provided here.
<code>port</code>	Optional. The port number on which the mail server is listening. If omitted, this defaults to the Server Port setting on the Mail/Mail Logging page of the ColdFusion Administrator. The standard port number is 25. Unless your mail server has been set up in a nonstandard way, you should never need to specify the <code>port</code> .
<code>timeout</code>	Optional. The number of seconds ColdFusion should spend trying to connect to the mail server. If omitted, this defaults to the Connection Timeout setting on the Mail/Mail Logging page of the ColdFusion Administrator.
<code>username</code>	Optional. The username to use when connecting to the mail server. If omitted, this defaults to the Username setting on the Mail/Mail Logging page of the ColdFusion Administrator.
<code>password</code>	Optional. The password to use when connecting to the mail server. If omitted, this defaults to the Password setting on the Mail/Mail Logging page of the ColdFusion Administrator.

Retrieving Email with ColdFusion

You have already seen how the `<cfmail>` tag can be used to send mail messages via your ColdFusion templates. You can also create ColdFusion templates that receive and process incoming mail messages. What your templates do with the messages is up to you. You might display each message to the user, or you might have ColdFusion periodically monitor the contents of a particular mailbox, responding to each incoming message in some way.

Introducing the `<cfpop>` Tag

To check or receive email messages with ColdFusion, you use the `<cfpop>` tag, providing the username and password for the email mailbox you want ColdFusion to look in.

ColdFusion will connect to the appropriate mail server in the same way that your own email client program connects to retrieve your mail for you.

Table 21.5 lists the attributes supported by the `<cfpop>` tag.

NOTE

The `<cfpop>` tag can only be used to check email that is sitting on a mail server that uses the Post Office Protocol (POP, or POP3). POP servers are by far the most popular type of mailbox server, largely because the POP protocol is very simple. Some mail servers use the newer Internet Mail Access Protocol (IMAP, or IMAP4). The `<cfpop>` tag can't be used to retrieve messages from IMAP mailboxes. Perhaps a future version of ColdFusion will include a `<cfimap>` tag; until then, some third-party solutions are available at the Developers Exchange Web site (<http://www.macromedia.com/cfusion/exchange/index.cfm>).

Table 21.5 `<cfpop>` Tag Attributes

Attribute	Purpose
action	GetHeaderOnly, GetAll, or Delete. Use GetHeaderOnly to quickly retrieve just the basic information (the subject, who it's from, and so on) about messages, without retrieving the messages themselves. Use GetAll to retrieve actual messages, including any attachments (which might take some time). Use Delete to delete a message from the mailbox.
server	Required. The POP server to which ColdFusion should connect. You can provide either a host name, such as <code>pop.orangewhipstudios.com</code> , or an IP address.
username	Required. The username for the POP mailbox ColdFusion should access. This is likely to be case sensitive, depending on the POP server.
password	Required. The password for the POP mailbox ColdFusion should access. This is likely to be case sensitive, depending on the POP server.
name	ColdFusion places information about incoming messages into a query object. You will loop through the records in the query to perform whatever processing you need for each message. Provide a name (such as <code>GetMessages</code>) for the query object here. This attribute is required if the action is GetHeaderOnly or GetAll.
maxrows	Optional. The maximum number of messages that should be retrieved. Because you don't know how many messages might be in the mailbox you are accessing, it's usually a good idea to provide maxrows unless you are providing messageNumber (later in this table).
startRow	Optional. The first message that should be retrieved. If, for instance, you already have processed the first 10 messages currently in the mailbox, you could specify <code>startRow="11"</code> to start at the 11th message.
messageNumber	Optional. If the action is GetHeaderOnly or GetAll, you can use this attribute to specify messages to retrieve from the POP server. If the action is Delete, this is the message or messages you want to delete from the mailbox. In either case, you can provide either a single message number or a comma-separated list of message numbers.

attachmentPath	Optional. If the <code>action</code> is <code>GetAll</code> , you can specify a directory on the server's drive in which ColdFusion should save any attachments. If you don't provide this attribute, the attachments won't be saved.
generateUniqueFileNames	Optional. This attribute should be provided only if you are using the <code>attachmentPath</code> attribute. If <code>Yes</code> , ColdFusion will ensure that two attachments that happen to have the same filename will get unique filenames when they are saved on the server's drive. If <code>No</code> (the default), each attachment is saved with its original filename, regardless of whether a file with the same name already exists in the <code>attachmentPath</code> directory.
port	Optional. If the POP server specified in <code>server</code> is listening for requests on a nonstandard port, specify the port number here. The default value is <code>110</code> , which is the standard port used by most POP servers.
timeout	Optional. This attribute indicates how many seconds ColdFusion should wait for each response from the POP server. The default value is <code>60</code> .
debug	Optional. If enabled, ColdFusion will log additional information about the <code>cfpop</code> call. This information is logged to either the console or a log file.
uid	Optional. A UUID, or a list of UUIDs, that represents unique IDs for mail messages to be retrieved. This attribute is better than <code>messageNumber</code> because <code>messageNumber</code> may change between requests.

When the `<cfpop>` tag is used with `action="GetHeaderOnly"`, it will return a query object that contains one row for each message in the specified mailbox. The columns of the query object are shown in Table 21.6.

Table 21.6 Columns Returned by `<cfpop>` When `ACTION="GetHeaderOnly"`

Column	Explanation
messageNumber	A number that represents the slot the current message is occupying in the mailbox on the POP server. The first message that arrives in a user's mailbox is message number 1. The next one to arrive is message number 2. When a message is deleted, any message behind the deleted message moves into the deleted message's slot. That is, if the first message is deleted, the second message becomes message number 1. In other words, the <code>messageNumber</code> isn't a unique identifier for the message. It's just a way to refer to the messages currently in the mailbox. It is safer to use <code>uid</code> .
date	The date the message was originally sent. Unfortunately, this date value isn't returned as a native CFML <code>Date</code> object. You must use the <code>ParseDateTime()</code> function to turn the value into something you can use ColdFusion's date functions with (see Listing 21.6, later in this chapter, for an example).
subject	The subject line of the message.
from	The email address that the message is reported to be from. This address might or might not contain a friendly name for the sender, delimited by quotation marks and angle brackets (see the section "Using Friendly Email Addresses," earlier in this chapter). It's worth noting that there's no guarantee that the <code>from</code> address is a real email address that can actually receive replies.
to	The email address to which the message was sent. This address might or might not contain a friendly name for the sender, delimited by quotation marks and angle brackets (see the section "Using Friendly Email Addresses").

cc	The email address or addresses to which the message was CC'd, if any. You can use ColdFusion's list functions to get the individual email addresses. Each address might or might not contain a friendly name for the sender, delimited by quotation marks and angle brackets (see the section "Using Friendly Email Addresses").
replyTo	The address to use when replying to the message, if provided. If the message's sender didn't provide a Reply-To address, the column will contain an empty string, in which case it would be most appropriate for replies to go to the from address. This address might or might not contain a friendly name for the sender, delimited by quotation marks and angle brackets (see the section "Using Friendly Email Addresses").
header	The raw, unparsed header section of the message. This usually contains information about how the message was routed to the mail server, along with information about which program was used to send the message, the MIME content type of the message, and so on. You need to know about the header names defined by the SMTP protocol (see the section "Adding Custom Mail Headers" earlier in this chapter) to make use of header.
messageid	A unique identifier for the mail message. It is not the same value as that used in the uid column.
uid	A unique identifier for the mail message. This can be used to retrieve or delete individual messages. It should be used instead of messageNumber because messageNumber may change.

If the <cfpop> tag is used with action="GetAll", the returned query object will contain all the columns from Table 21.6, plus the columns listed in Table 21.7.

Table 21.7 Additional Columns Returned by <cfpop> When action="GetAll"

Column	Explanation
body	The actual body of the message, as a simple string. This string usually contains just plain text, but if the message was sent as an HTML-formatted message, it contains HTML tags. You can check for the presence of a Content-Type header value of text/html to determine whether the message is HTML formatted (see Listing 21.8, later in this chapter, for an example).
attachment	If you provided an attachmentPath attribute to the <cfpop> tag, this column contains a list of the attachment filenames as they were named when originally attached to the message. The list of attachments is separated by tab characters. You can use ColdFusion's list functions to process the list, but you must specify Chr(9) (which is the tab character) as the delimiter for each list function, as in listLen(ATTACHMENTS, Chr(9)).
attachmentFiles	If you provided an attachmentPath attribute to the <cfpop> tag, this column contains a list of the attachment filenames as they were saved on the ColdFusion server (in the directory specified by attachmentPath). You can use the values in this list to delete, show, or move the files after the message has been retrieved. Like the attachments column, this list is separated by tab characters.
textBody	If an email contains both a plain-text and an HTML body, this value will contain the plain-text version of the email.
htmlBody	If an email contains both a plain-text and an HTML body, this value will contain the HTML version of the email.

Retrieving the List of Messages

Most uses of the `<cfpop>` tag call for all three of the `action` values it supports. Whether you are using the tag to display messages to your users (such as a Web-based system for checking mail) or an automated agent that responds to incoming email messages on its own, the sequence of events probably involves these steps:

1. Log in to the mail server with `action="GetHeaderOnly"` to get the list of messages currently in the specified mailbox. At this point, you can display or make decisions based on who the message is from, the date, or the subject line.
2. Use `action="GetAll"` to retrieve the full text of individual messages.
3. Use `action="Delete"` to delete messages.

Listing 21.6 is the first of three templates that demonstrate how to use `<cfpop>` by creating a Web-based system for users to check their mail. This template asks the user to log in by providing the information ColdFusion needs to access their email mailbox (username, password, and mail server). It then checks the user's mailbox for messages and displays the From address, date, and subject line for each. The user can click each message's subject to read the full message.

Listing 21.6 CheckMail.cfm—The Beginnings of a Simple POP Client

```
<!---
  Filename: CheckMail.cfm
  Author: Nate Weiss (NMW)
  Purpose: Creates a very simple POP client
-->

<html>
<head><title>Check Your Mail</title></head>
<body>

<!-- Simple CSS-based formatting styles -->
<style>
  body { font-family:sans-serif;font-size:12px}
  th { font-size:12px;background:navy;color:white}
  td { font-size:12px;background:lightgrey;color:navy}
</style>

<h2>Check Your Mail</h2>

<!-- If user is logging out, -->
<!-- or if user is submitting a different username/password -->
<cfif isDefined("URL.logout") or isDefined("FORM.popServer")>
  <cfset structDelete(SESSION, "mail")>
</cfif>

<!-- If we don't have a username/password -->
<cfif not isDefined("SESSION.mail")>
  <!-- Show "mail server login" form -->
  <cfinclude template="CheckMailLogin.cfm">
</cfif>
```

```

<!-- If we need to contact server for list of messages -->
<!-- (if just logged in, or if clicked "Refresh" link) -->
<cfif not isDefined("SESSION.mail.getMessages") or isDefined("URL.refresh")>
  <!-- Flush page output buffer -->
  <cfflush>

  <!-- Contact POP Server and retrieve messages -->
  <cfpop action="GetHeaderOnly" name="SESSION.mail.getMessages"
  server="#SESSION.mail.popServer#"
  username="#SESSION.mail.username#" password="#SESSION.mail.password#"
  maxrows="50">
</cfif>

<!-- If no messages were retrieved... -->
<cfif SESSION.mail.getMessages.recordCount eq 0>
  <p>You have no mail messages at this time.<br>

<!-- If messages were retrieved... -->
<cfelse>
  <!-- Display Messages in HTML Table Format -->
  <table border="0" cellSpacing="2" cellSpacing="2" cols="3" width="550">
  <!-- Column Headings for Table -->
  <tr>
  <th width="100">Date Sent</th>
  <th width="200">From</th>
  <th width="200">Subject</th>
  </tr>
  <!-- Display info about each message in a table row -->
  <cfoutput query="SESSION.mail.getMessages">
  <!-- Parse Date from the "date" mail header -->
  <cfset msgDate = parseDateTime(date,"pop")>
  <!-- Let user click on Subject to read full message -->
  <cfset linkURL = "CheckMailMsg.cfm?uid=#urlEncodedFormat(uid)#">

  <tr valign="baseline">
  <!-- Show parsed Date and Time for message-->
  <td>
  <strong>#dateFormat(msgDate)#</strong><br>
  #timeFormat(msgDate)# #ReplyTo#
  </td>
  <!-- Show "From" address, escaping brackets -->
  <td>#htmlEditFormat(From)#</td>
  <td><strong><a href="#linkURL#">#Subject#</a></strong></td>
  </tr>
  </cfoutput>
  </table>

</cfif>

<!-- "Refresh" link to get new list of messages -->
<strong><a href="CheckMail.cfm?Refresh=Yes">Refresh Message List</a></strong><br>
<!-- "Log Out" link to discard SESSION.Mail info -->

```

```
<a href="CheckMail.cfm?Logout=Yes">Log Out</a><br>
```

```
</body>
</html>
```

NOTE

Don't forget to copy the `Application.cfc` file from the downloaded files. The code isn't described until listing 21.10.

This template maintains a structure in the `SESSION` scope called `SESSION.mail`. The `SESSION.mail` structure holds information about the current user's POP server, username, and password. It also holds a query object called `getMessages`, which is returned by the `<cfpop>` tag when the user's mailbox is first checked.

At the top of the template, a `<cfif>` test checks to see whether a URL parameter named `logout` has been provided. If so, the `SESSION.mail` structure is deleted from the server's memory, which effectively logs the user out. Later, you will see how this works. The same thing happens if a FORM parameter named `popServer` exists, which indicates that the user is trying to submit a different username and password from the login form. (I'll explain this in a moment.)

Next, a similar `<cfif>` tests checks to see whether the `SESSION.mail` structure exists. If not, the template concludes that the user hasn't logged in yet, so it displays a simple login form by including the `CheckMailLogin.cfm` template (see Listing 21.7). This is the same basic login-check technique explained in Chapter 26. In any case, all code after this `<cfif>` test is guaranteed to execute only if the user has logged in. The `SESSION.mail` structure will contain `username`, `password`, and `popServer` values, which can later be passed to all `<cfpop>` tags for the remainder of the session.

The next `<cfif>` test checks to see whether ColdFusion needs to access the user's mailbox to get a list of current messages. ColdFusion should do this whenever `SESSION.mail.getMessages` doesn't exist yet (which means that the user has just logged in), or if the page has been passed a `refresh` parameter in the URL (which means that the user has just clicked the Refresh Message List link, as shown in Figure 21.6). If so, the `<cfpop>` tag is called with `action="GetHeaderOnly"`, which means that ColdFusion should get a list of messages from the mail server (which is usually pretty fast), rather than getting the actual text of each message (which can be quite slow, especially if some of the messages have attachments). Note that the `<cfpop>` tag is provided with the username, password, and POP server name that the user provided when they first logged in (now available in the `SESSION.mail` structure).



Figure 21.6

The `<CFPOP>` tag enables email messages to be retrieved via ColdFusion templates.

NOTE

Because the `SESSION.mail.getMessages` object is a ColdFusion query, it also contains the automatic `CurrentRow` and `RecordCount` attributes returned by ordinary `<cfquery>` tags.

At this point, the template has retrieved the list of current messages from the user's mailbox, so all that's left to do is to display them to the user. The remainder of Listing 21.6 simply outputs the list of messages in a simple HTML table format, using an ordinary `<cfoutput>` block that loops over the `SESSION.mail.getMessages` query object. Within this loop, the code can refer to the `Date` column of the query object to access a message's date or to the `Subject` column to access the message's subject line. The first time through the loop, these variables refer to the first message retrieved from the user's mailbox. The second time, the variables refer to the second message, and so on.

Just inside the `<cfoutput>` block, a ColdFusion date variable called `msgDate` is created using the `parseDateTime()` function with the optional `POP` attribute. This is necessary because the `Date` column returned by `<cfpop>` doesn't contain native CFML date value, as you might expect. Instead, it contains the date in the special date format required by the mail-sending protocol (SMTP). The `parseDateTime()` function is needed to parse this special date string into a proper CFML `Date` value you can provide to ColdFusion's date functions (such as the `dateFormat()` and `dateAdd()` functions).

NOTE

Unfortunately, the format used for the date portion of mail messages varies somewhat. The `parseDateTime()` function doesn't properly parse the date string that some incoming messages have. If the function encounters a date that it can't parse correctly, an error message results. Some custom tag solutions to this problem are available at the ColdFusion Developers Exchange Web site (http://www.adobe.com/cfusion/exchange/index.cfm?event=productHome&exc=1&loc=en_us). Search for `POP` and `Date`.

Inside the `<cfoutput>` block, the basic information about the message is output as a table row. The date of the message is shown using the `dateFormat()` and `timeFormat()` functions. The Subject line of the message is presented as a link to the `CheckMailMsg.cfm` template (see Listing 21.8), passing the `UID` for the current message in the URL. Because the `UID` identifies the message, the user can click the subject to view the whole message.

At the bottom of the template, the user is provided with Refresh Message List and Log Out links, which simply reload the listing with either `refresh=Yes` or `logout=Yes` in the URL.

NOTE

Because email addresses can contain angle brackets (see the section "Using Friendly Email Addresses," earlier in this chapter), you should always use the `htmlEditFormat()` function when displaying an email address returned by `<cfpop>` in a Web page. Otherwise, the browser will think the angle brackets are meant to indicate an HTML tag, which means that the email address won't show up visibly on the page (although it will be part of the page's HTML if you view source). Here, `htmlEditFormat()` is used on the `From` column, but you should use it whenever you output the `To`, `CC`, or `ReplyTo` columns as well.

Listing 21.7 is a template that presents a login form to the user when they first visit `CheckMail.cfm`. It's included via the `<cfinclude>` tag in Listing 21.6 whenever the `SESSION.mail` structure doesn't exist (which means that the user has either logged out or has not logged in yet).

Listing 21.7 CheckMailLogin.cfm—A Simple Login Form, Which Gets Included by CheckMail.cfm

```
<!---
Filename: CheckMailLogin.cfm
Author: Nate Weiss (NMW)
Purpose: Provides a login form for the simple POP client
-->
<!--- If user is submitting username/password form --->
<cfif isDefined("FORM.popServer")>
  <!--- Retain username, password, server in SESSION --->
  <cfset SESSION.mail = structNew()>
  <cfset SESSION.mail.popServer = FORM.popServer>
  <cfset SESSION.mail.username = FORM.username>
  <cfset SESSION.mail.password = FORM.password>
  <!--- Remember server and username for next time --->
  <cfset CLIENT.mailServer = FORM.popServer>
  <cfset CLIENT.mailUsername = FORM.username>
<cfelse>
  <!--- Use server/username from last time, if available --->
  <cfparam name="CLIENT.mailServer" type="string" default="">
  <cfparam name="CLIENT.mailUsername" type="string" default="">

  <!--- Simple form for user to provide mailbox info --->
  <cfform action="#CGI.script_name#" method="post">
    <p>To access your mail, please provide the
server, username and password.<br>

    <!--- FORM field: POPServer --->
    <p>Mail Server:<br>
    <cfinput type="text" name="popServer"
value="#CLIENT.mailServer#" required="Yes"
message="Please provide your mail server.">
(example: pop.yourcompany.com)<br>

    <!--- FORM field: Username --->
    Mailbox Username:<br>
    <cfinput type="text" name="username"
value="#CLIENT.mailUsername#" required="Yes"
message="Please provide your username.">
(yourname@yourcompany.com)<br>

    <!--- FORM field: Password --->
    Mailbox Password:<br>
    <cfinput type="password" name="password"
required="yes" message="Please provide your password."><br>

    <cfinput type="submit" name="submit" value="Check Mail"><br>
  </cfform>

  </body></html>
<cfabort>
</cfif>
```

When the user first visits `CheckMail.cfm`, Listing 21.7 gets included. At first, the `FORM.popServer` variable won't exist, so the `<cfelse>` part of the code executes, which presents the login form to the user. When the form is submitted, it posts the user's entries to the `CheckMail.cfm` template, which in turn calls this template again. This time, `FORM.popServer` exists, so the first part of the `<cfif>` block executes. The `SESSION.mail` structure is created, and the `popServer`, `username`, and `password` values are copied from the user's form input into the structure so that they can be referred to during the rest of the session, or until the user logs out.

If you accidentally enter an incorrect username or password while testing this listing, you will get a rather ugly error message. You can intercept the error so that the user is kindly asked to try again, without it seeming like anything has really gone so wrong. A revised version of this listing that does just that is included in Chapter 51.

NOTE

As a convenience to the user, Listing 21.7 stores the `popServer` and `username` values (which the user provides in the login form) as variables in the `CLIENT` scope. These values are passed to the value attributes of the corresponding form fields the next time the user needs to log in. This way, the user only needs to enter their password on repeat visits.

Receiving and Deleting Messages

Listing 21.8 is the `CheckMailMsg.cfm` template the user will be directed to whenever they click the subject line in the list of messages (refer to Figure 21.6). This template requires that a URL parameter called `UID` be passed to it, which indicates the `uid` of the message the user clicked. In addition, the template can be passed a `Delete` parameter, which indicates that the user wants to delete the specified message.

Listing 21.8 `CheckMailMsg.cfm`—Retrieving the Full Text of an Individual Message

```
<!---
Filename: CheckMailMsg1.cfm
Author: Nate Weiss (NMW)
Purpose: Allows the user to view a message on their POP server
-->

<html>
<head><title>Mail Message</title></head>
<body>

<!--- Simple CSS-based formatting styles -->
<style>
body { font-family:sans-serif;font-size:12px}
th { font-size:12px;background:navy;color:white}
td { font-size:12px;background:lightgrey;color:navy}
</style>

<h2>Mail Message</h2>

<!--- A message uid must be passed in the URL -->
<cfparam name="URL.uid">
```

```

<cfparam name="URL.delete" type="boolean" default="No">

<!--- If we don't have a username/password --->
<!--- send user to main CheckMail.cfm page --->
<cfif not isDefined("SESSION.mail.getMessages")>
    <cflocation url="CheckMail.cfm">
</cfif>

<!--- find our position in the query --->
<cfset position = listFindNoCase(valueList(SESSION.mail.getMessages.uid),
URL.uid)>

<!--- If the user is trying to delete the message --->
<cfif URL.delete>
    <!--- Contact POP Server and delete the message --->
    <cfpop action="Delete" uid="#URL.uid#"
server="#SESSION.mail.popServer#"
username="#SESSION.mail.username#"
password="#SESSION.mail.password#">

    <!--- Send user back to main "Check Mail" page --->
    <cflocation url="CheckMail.cfm?refresh=Yes" addToken="false">

<!--- If not deleting, retrieve and show the message --->
<cfelse>
    <!--- Contact POP Server and retrieve the message --->
    <cfpop action="GetAll" name="GetMsg"
uid="#URL.uid#"
server="#SESSION.mail.popServer#"
username="#SESSION.mail.username#"
password="#SESSION.mail.password#">
    <cfset msgDate = parseDateTime(getMsg.Date, "pop")>

    <!--- If message was not retrieved from POP server --->
    <cfif getMsg.recordCount neq 1>
    <cfthrow message="Message could not be retrieved."
detail="Perhaps the message has already been deleted.">
    </cfif>

    <!--- We will provide a link to Delete message --->
    <cfset deleteURL = "#CGI.script_name#?uid=#uid#&delete=Yes">

    <!--- Display message in a simple table format --->
    <table border="0" cellSpacing="0" cellPadding="3">
    <cfoutput>
    <tr>
    <th bgcolor="wheat" align="left" nowrap>
Message #position# of #SESSION.mail.getMessages.recordCount#
    </th>
    <td align="right" bgcolor="beige">
    <!--- Provide "Back" button, if appropriate --->
    <cfif position gt 1>
    <cfset prevuid = SESSION.mail.getMessages.uid[decrementValue(position)]>
    <a href="CheckMailMsg.cfm?uid=#prevuid#">

```

```

</a>
</cfif>
<!-- Provide "Next" button, if appropriate -->
<cfif position lt SESSION.mail.getMessages.recordCount>
<cfset nextuid = SESSION.mail.getMessages.uid[incrementValue(position)]>
<a href="CheckMailMsg.cfm?uid=#nextuid#">
</a>
</cfif>
</td>
</tr>
<tr>
<th align="right">From:</th>
<td>#htmlEditFormat(getMsg.From)#</td>
</tr>
<cfif getMsg.CC neq "">
<tr>
<th align="right">CC:</th>
<td>#htmlEditFormat(getMsg.CC)#</td>
</tr>
</cfif>
<tr>
<th align="right">Date:</th>
<td>#dateFormat(msgDate)# #timeFormat(msgDate)#</td>
</tr>
<tr>
<th align="right">Subject:</th>
<td>#getMsg.Subject#</td>
</tr>
<tr>
<td bgcolor="beige" colspan="2">
<strong>Message:</strong><br>

<cfif getMsg.Header contains "Content-Type: text/html">
#getMsg.Body#
<cfelse>
#htmlCodeFormat(getMsg.Body)#
</cfif>
</td>
</tr>
</cfoutput>
</table>
<cfoutput>
<!-- Provide link back to list of messages -->
<strong><a href="CheckMail.cfm">Back To Message List</a></strong><br>
<!-- Provide link to Delete message -->
<a href="#deleteURL#">Delete Message</a><br>
<!-- "Log Out" link to discard SESSION.Mail info -->
<a href="CheckMail.cfm?Logout=Yes">Log Out</a><br>
</cfoutput>
</cfif>
</body>
</html>

```

NOTE

Be sure to save the previous listing as `CheckMailMsg.cfm`, not `CheckMailMsg1.cfm`.

As a sanity check, the user is first sent back to the `CheckMail.cfm` template (refer to Listing 21.6) if the `SESSION.mail.getMessages` query doesn't exist. This would happen if the user's session had timed out or if the user had somehow navigated to the page without logging in first. In any case, sending them back to `CheckMail.cfm` causes the login form to be displayed.

Because we are not using the `messageNumber` value, we need to find what position our mail is in the query. This will let us display a numerical mail number instead of the useful, but ugly, `UID` value. By using the `listFindNoCase` function on the `valueList` of the `UID` column, we can determine the position of the `UID` in the query as a whole.

Next, a `<cfif>` test is used to see whether `delete=Yes` was passed in the URL. If so, the message is deleted using the `action="Delete"` attribute of the `<cfpop>` tag, specifying the passed `URL.uid` as the `uid` to delete. The user is then sent back to `CheckMail.cfm` with `refresh=Yes` in the URL, which causes `CheckMail.cfm` to re-contact the mail server and repopulate the `SESSION.mail.getMessages` query with the revised list of messages (which should no longer include the deleted message).

If the user isn't deleting the message, the template simply retrieves and displays it in a simple HTML table format. To do so, `<cfpop>` is called again, this time with `action="GetAll"` and the `messagenumber` of the desired message. Then the columns returned by `<cfpop>` can be displayed, much as they were in Listing 21.6. Because the action was "GetAll", this template could use the `BODY` and `HEADER` columns listed previously in Table 21.7. The end result is that the user has a convenient way to view, scroll through, and delete messages, as shown in Figure 21.7.

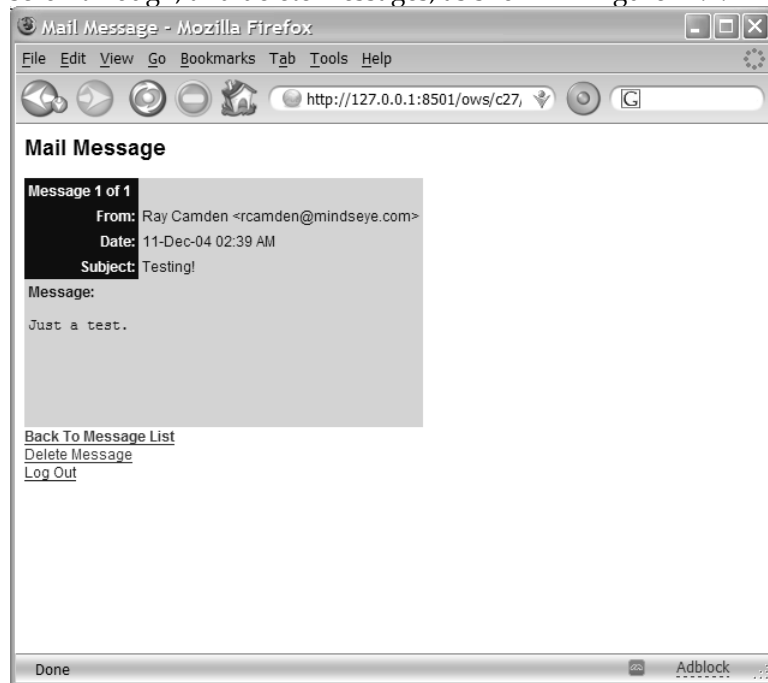


Figure 21.7

With `<CFPOP>`, retrieving and displaying the messages in a user's POP mailbox is easy.

At the bottom of the template, users are provided with the links to log out or return to the list of messages. They are also provided with a link to delete the current message,

which simply reloads the current page with `delete=Yes` in the URL, causing the Delete logic mentioned previously to execute.

Receiving Attachments

As noted previously in Table 21.5, the `<cfpop>` tag includes an `attachmentPath` attribute that, when provided, tells ColdFusion to save any attachments to a message to a folder on the server's drive. Your template can then process the attachments in whatever way is appropriate: move the files to a certain location, parse through them, display them to the user, or whatever your application needs.

Retrieving the Attachments

Listing 21.9 is a revised version of the `CheckMailMsg.cfm` template from Listing 21.8. This version enables the user to download and view any attachments that might be attached to each mail message. The most important change in this version is the addition of the `attachmentPath` attribute, which specifies that any attachments should be placed in a subfolder named `Attach` (within the folder that the template itself is in).

Listing 21.9 `CheckMailMsg2.cfm`—Allowing the User to Access Attachments

```
<!---
  Filename: CheckMailMsg2.cfm
  Author: Nate Weiss (NMW)
  Purpose: Allows the user to view a message on their POP server
-->

<html>
<head><title>Mail Message</title></head>
<body>

<!--- Simple CSS-based formatting styles --->
<style>
  body { font-family:sans-serif;font-size:12px}
  th { font-size:12px;background:navy;color:white}
  td { font-size:12px;background:lightgrey;color:navy}
</style>

<h2>Mail Message</h2>

<!--- A message uid must be passed in the URL --->
<cfparam name="URL.uid">
<cfparam name="URL.delete" type="boolean" default="no">

<!--- Store attachments in "Attach" subfolder --->
<cfset attachDir = expandPath("Attach")>
<!--- Set a variable to hold the Tab character --->
<cfset TAB = chr(9)>

<!--- Create the folder if it doesn't already exist --->
```

```
<cfif not directoryExists(attachDir)>
  <cfdirectory action="create" directory="#attachDir#">
</cfif>

<!--- If we don't have a username/password --->
<!--- send user to main CheckMail.cfm page --->
<cfif not isDefined("SESSION.mail.getMessages")>
  <cflocation url="CheckMail.cfm">
</cfif>

<!--- find our position in the query --->
<cfset position = listFindNoCase(valueList(SESSION.mail.getMessages.uid),
URL.uid)>

<!--- If the user is trying to delete the message --->
<cfif url.delete>
  <!--- Contact POP Server and delete the message --->
  <cfpop action="Delete"
uid="#URL.uid#"
server="#SESSION.mail.popServer#"
username="#SESSION.mail.username#"
password="#SESSION.mail.password#">

  <!--- Send user back to main "Check Mail" page --->
  <cflocation url="CheckMail.cfm?refresh=Yes">

<!--- If not deleting, retrieve and show the message --->
<cfelse>

  <!--- Contact POP Server and retrieve the message --->
  <cfpop action="GetAll" name="GetMsg"
uid="#URL.uid#"
server="#SESSION.mail.popServer#"
username="#SESSION.mail.username#"
password="#SESSION.mail.password#"
attachmentPath="#attachDir#"
generateUniqueFileNames="Yes">

  <!--- Parse message's date string to CF Date value --->
  <cfset msgDate = parseDateTime(getMsg.Date, "POP")>

  <!--- If message was not retrieved from POP server --->
  <cfif getMsg.recordCount neq 1>
  <cfthrow
message="Message could not be retrieved."
detail="Perhaps the message has already been deleted.">
  </cfif>

  <!--- We will provide a link to Delete message --->
  <cfset deleteURL = "#CGI.script_name#?uid=#uid#&delete=Yes">

  <!--- Display message in a simple table format --->
  <table border="0" cellSpacing="0" cellPadding="3">
  <cfoutput>
```

```

<tr>
<th bgcolor="wheat" align="left" nowrap>
Message #position# of #SESSION.mail.getMessages.recordCount#
</th>
<td align="right" bgcolor="beige">
<!-- Provide "Back" button, if appropriate -->
<cfif position gt 1>
<cfset prevuid = SESSION.mail.getMessages.uid[decrementValue(position)]>
<a href="CheckMailMsg.cfm?uid=#prevuid#">
</a>
</cfif>
<!-- Provide "Next" button, if appropriate -->
<cfif position lt SESSION.mail.getMessages.recordCount>
<cfset nextuid = SESSION.mail.getMessages.uid[incrementValue(position)]>
<a href="CheckMailMsg.cfm?uid=#nextuid#">
</a>
</cfif>
</td>
</tr>
<tr>
<th align="right">From:</th>
<td>#htmlEditFormat(getMsg.From)#</td>
</tr>
<cfif getMsg.CC neq "">
<tr>
<th align="right">CC:</th>
<td>#htmlEditFormat(getMsg.CC)#</td>
</tr>
</cfif>
<tr>
<th align="right">Date:</th>
<td>#dateFormat(msgDate)# #timeFormat(msgDate)#</td>
</tr>
<tr>
<th align="right">Subject:</th>
<td>#getMsg.Subject#</td>
</tr>
<tr>
<td bgcolor="beige" colspan="2">
<strong>Message:</strong><br>

<cfif getMsg.Header contains "Content-Type: text/html">
#getMsg.Body#
<cfelse>
#htmlCodeFormat(getMsg.Body)#
</cfif>
</td>
</tr>
<!-- If this message has any attachments -->
<cfset numAttachments = listLen(getMsg.Attachments, TAB)>
<cfif numAttachments gt 0>
<tr>

```

```

<th align="right">Attachments:</th>
<td>
<!-- For each attachment, provide a link --->
<cfloop from="1" to="#numAttachments#" index="i">
<!-- Original filename, as it was attached to message --->
<cfset thisFileOrig = listGetAt(getMsg.Attachments, i, TAB)>
<!-- Full path to file, as it was saved on this server --->
<cfset thisFilePath = listGetAt(getMsg.attachmentFiles, i, TAB)>
<!-- Relative URL to file, so user can click to get it --->
<cfset thisFileURL = "Attach/#getFileFromPath(thisFilePath)#">
<!-- Actual link --->
<a href="#thisFileURL#">#thisFileOrig#</a><br>
</cfloop>
</td>
</tr>
</cfif>
</cfoutput>
</table>

<cfoutput>
<!-- Provide link back to list of messages --->
<strong><a href="CheckMail.cfm">Back To Message List</a></strong><br>
<!-- Provide link to Delete message --->
<a href="#deleteURL#">Delete Message</a><br>
<!-- "Log Out" link to discard SESSION.Mail info --->
<a href="CheckMail.cfm?logout=Yes">Log Out</a><br>
</cfoutput>

</cfif>

</body>
</html>

```

NOTE

Be sure to save the previous listing as `CheckMailMsg.cfm`, not `CheckMailMsg2.cfm`.

The first change is the addition of a variable called `attachDir`, which is the complete path to the directory on the server that will hold attachment files. Additionally, a variable called `TAB` is created to hold a single tab character (which is character number 9 in the standard character set). This way, the rest of the code can refer to `TAB` instead of `chr(9)`, which improves code readability.

NOTE

This code uses the variable name `TAB` in all caps— instead of `tab` to indicate the notion that the variable holds a constant value. A constant value is simply a value that will never change (that is, the tab character will always be represented by ASCII code 9). Developers often write constants in all capital letters to make them stand out. You don't have to do this, but you might find it helpful as you write your ColdFusion templates.

NOTE

This code uses the `expandPath()` function to set `attachDir` to the subfolder named `Attach` within the folder that the template itself is in.

Next, a `directoryExists()` test checks to see whether the `attachDir` directory exists yet. If not, the directory is created via the `<cfdirectory>` tag. See Chapter 70 for details about creating directories on the server. After the directory is known to exist, it's safe to provide the value of `attachDir` to the `attachmentPath` attribute of the `<cfpop>` tag.

NOTE

This code also sets `generateUniqueFileNames` to `Yes` so there is no danger of two attachment files with the same name (from different messages, say) being overwritten with one another. It's generally recommended that you do this to prevent the risk of two `<cfpop>` requests interfering with one another.

Now, near the bottom of the template, the `attachments` and `attachmentFiles` columns of the `getMsg` query object are examined to present any attachments to the user. As noted previously in Table 21.7, these two columns contain tab-separated lists of the message's file attachments (if any). Unlike most ColdFusion lists, these lists are separated with tab characters, so any of ColdFusion's list functions must specify the tab character as the delimiter.

For instance, the `numAttachments` variable is set to the number of file attachments using a simple call to the `listLen()` function. If at least one attachment exists, a simple `<cfloop>` block iterates through the list of attachments. Each time through the loop, the `thisFileOrig` variable holds the original filename of the attachment (as the sender attached it), the `thisFilePath` variable holds the unique filename used to save the file in `attachDir`, and the `thisFileURL` variable holds the appropriate relative URL for the file on the server. It's then quite easy to provide a simple link the user can click to view or save the file (as shown in Figure 21.8).

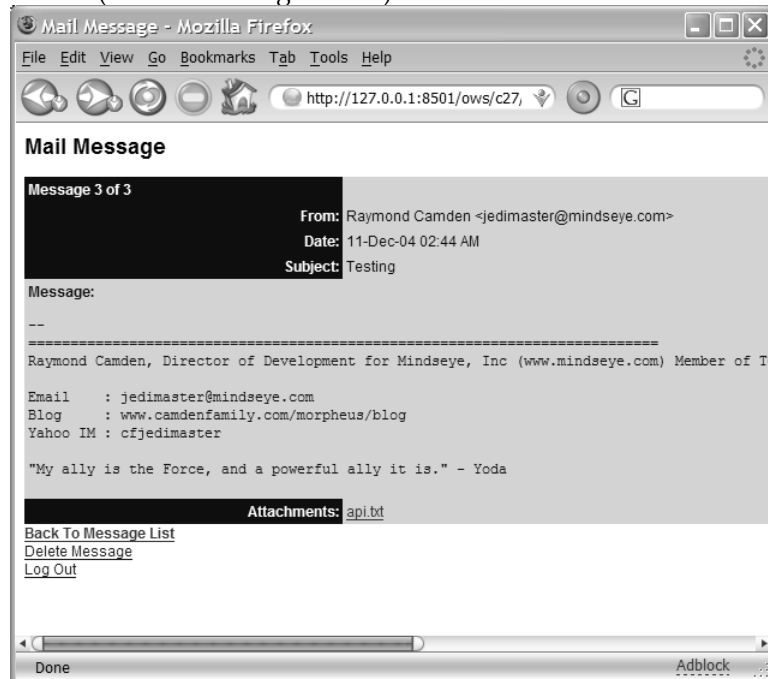


Figure 21.8

The `<CFPOP>` tag can retrieve files attached to messages in a mailbox.

Deleting Attachments After Use

One problem with Listing 21.9 is the fact that the attachment files that get placed into `attachDir` are never deleted. Over time, the directory would fill up with every attachment for every message that was ever displayed by the template. It would be nice

to delete the files when the user was finished looking at the message, but because of the stateless nature of the Web, you don't really know when that is. You could delete each user's attachment files when they log out, but the user could close their browser without logging out. Luckily, the latest version of ColdFusion allows you to run code automatically when a session ends. This is done via the `onSessionEnd()` method of the `Application.cfc` file.

Listing 21.10 lists the `Application.cfc` file for the application.

Listing 21.10 `Application.cfc`—Deleting Attachment Files Previously Saved by <CFPOP>

```
<!---
Filename: Application.cfc
Created by: Raymond Camden (ray@camdenfamily.com)
Please Note Executes for every page request
-->

<cfcomponent output="false">

    <!--- Name the application. --->
    <cfset this.name="OrangeWhipSite">
    <!--- Turn on session management. --->
    <cfset this.sessionManagement=true>
    <cfset this.clientManagement=true>

    <cffunction name="onSessionEnd" output="false" returnType="void">
        <!--- Look for attachments to delete --->

        <cfset var attachDir = expandPath("Attach")>
        <cfset var getFiles = "">
        <cfset var thisFile = "">

        <!--- Get a list of all files in the directory --->
        <cfdirectory directory="#attachDir#" name="getFiles">

        <!--- For each file in the directory --->
        <cfloop query="getFiles">
            <!--- If it's a file (rather than a directory) --->
            <cfif getFiles.type neq "Dir">
                <!--- Get full filename of this file --->
                <cfset thisFile = expandPath("Attach\ #getFiles.Name#")>

                <!--- Go ahead and delete the file --->
                <cffile action="delete" file="#thisFile#">
            </cfif>
        </cfloop>

    </cffunction>

</cfcomponent>
```

Most of this file simply handles turning on `CLIENT` and `SESSION` management. The `onSessionEnd()` method is what we are concerned with. This method will fire when the

user's session expires. It uses `<cfdirectory>` to get all the files in the attachment directory. It then loops over and deletes each file.

Creating Automated POP Agents

You can create automated agents that watch for new messages in a particular mailbox and respond to the messages in some kind of intelligent way. First, you create an agent template, which is just an ordinary ColdFusion template that checks a mailbox and performs whatever type of automatic processing is necessary. This template shouldn't contain any forms or links, because it won't be viewed by any of your users. Then, using the ColdFusion scheduler, you schedule the template to be visited automatically every ten minutes, or whatever interval you feel is appropriate.

Creating the Agent Template

Listing 21.11 creates a simple version of a typical agent template: an unsubscribe agent, which responds to user requests to be removed from mailing lists. If you look at the `SendBulkMail.cfm` template (refer to Listing 21.4), you will notice that all messages sent by the template include instructions for users who want to be removed from Orange Whip Studios' mailing list.

The instructions tell the user to send a reply to the email with the word `Remove` in the subject line. Therefore, the main job of this template is to check the mailbox to which the replies will be sent (which is `mailings@orangewhipstudios.com` in this example). The template then checks each message's subject line. If it includes the word `Remove`, and the sender's email address is found in the `Contacts` table, the user is removed from the mailing list by setting the user's `MailingList` field to 0 in the database. The next time the `SendBulkMail.cfm` is used to send a bulk message, the user will be excluded from the mailing.

Listing 21.11 `ListUnsubscriber.cfm`—Automatically Unsubscribing Users from a Mailing List

```
<!---
  Filename: ListUnsubscriber.cfm
  Author: Nate Weiss (NMW)
  Purpose: A simple automated POP agent for unsubscribing from mailing lists
-->

<!--- Mailbox info for "mailings@orangewhipstudios.com" --->
<cfset popServer = "pop.orangewhipstudios.com">
<cfset username = "mailings">
<cfset password = "ThreeOrangeWhips">

<!--- We will delete all messages in this list --->
<cfset msgDeleteList = "">

<html>
<head><title>List Unsubscriber Agent</title></head>
<body>

<h2>List Unsubscriber Agent</h2>
```

```
<p>Checking the mailings@orangewhipstudios.com mailbox for new messages...<br>
This may take a minute, depending on traffic and the number of messages.<br>

<!-- Flush output buffer so the above messages --->
<!-- are shown while <CFPOP> is doing its work --->
<cfflush>

<!-- Contact POP Server and retrieve messages --->
<cfpop action="GetHeaderOnly" name="getMessages"
  server="#popServer#" username="#username#" password="#password#"
  maxrows="20">

<!-- Short status message --->
<cfoutput>
  <p><strong>#getMessages.recordCount# messages to process.</strong><br>
</cfoutput>

<!-- For each message currently in the mailbox... --->
<cfloop query="getMessages">

  <!-- Short status message --->
  <cfoutput>
    <p><strong>Message from:</strong> #htmlEditFormat(getMessages.From)#<br>
  </cfoutput>

  <!-- If the subject line contains the word "Remove" --->
  <cfif getMessages.Subject does not contain "Remove">
    <!-- Short status message --->
    Message does not contain "Remove".<br>
  <cfelse>

    <!-- Short status message --->
    Message contains "Remove".<br>

    <!-- Which "word" in From address contains @ sign? --->
    <cfset addrPos = listFind(getMessages.From, "@", "<> ")>
    <!-- Assuming one of the "words" contains @ sign, --->
    <cfif addrPos eq 0>
      <!-- Short status message --->
      Address not found in From line.<BR>
    <cfelse>

      <!-- Email address is that word in From address --->
      <cfset fromAddress = trim(listGetAt(getMessages.From, addrPos, "<> "))>

      <!-- Who in mailing list has this email address? --->
      <cfquery name="getContact" datasource="ows" maxrows="1">
        SELECT ContactID, FirstName, LastName
        FROM Contacts
        WHERE Email = '#fromAddress#'
        AND MailingList = 1
      </cfquery>

      <!-- Assuming someone has this address... --->
```

```
<cfif getContact.recordCount eq 0>
  <!--- Short status message --->
  <cfoutput>Recipient #fromAddress# not on list.<br></cfoutput>
<cfelse>
  <!--- Short status message --->
  <cfoutput>Removing #fromAddress# from list.<br></cfoutput>

  <!--- Update the database to take them off list --->
  <cfquery datasource="ows">
    UPDATE Contacts SET
    MailingList = 0
    WHERE ContactID = #getContact.ContactID#
  </cfquery>

  <!--- Short status message --->
  Sending confirmation message via email.<br>

  <!--- Mail user a confirmation note --->
  <cfmail
    to=""#getContact.FirstName# #getContact.LastName#" <#fromAddress#"
    from=""Orange Whip Studios" <mailings@orangewhipstudios.com>"
    subject="Mailing List Request"
    >You have been removed from our mailing list.</cfmail>

  </cfif>
</cfif>
</cfif>

<!--- Add this message to the list of ones to delete. --->
<!--- If you wanted to only delete some messages, you --->
<!--- would put some kind of <CFIF> test around this. --->
<cfset msgDeleteList = listAppend(msgDeleteList, getMessages.uid)>
</cfloop>

<!--- If there are messages to delete --->
<cfif msgDeleteList neq "">
  <!--- Short status message --->
  <p>Deleting messages...

  <!--- Flush output buffer so the above messages --->
  <!--- are shown while <CFPOP> is doing its work --->
  <cfflush>
  <!--- Contact POP Server and delete messages --->
  <cfpop action="Delete" server="#popServer#"
    username="#username#"
    password="#password#"
    uid="#msgDeleteList#">

  Done.<br>
</cfif>

</body>
</html>
```

The code in Listing 21.11 is fairly simple. First, the `<cfpop>` tag is used to retrieve the list of messages currently in the appropriate mailbox. Please note that you will need to change the `popServer`, `username`, and `password` settings, because the template needs to look only at the Subject line of each message, this template only needs to perform this `GetHeaderOnly` action. It never needs to retrieve the entirety of each message via `action="GetAll"`.

Then, for each message, a series of tests are performed to determine whether the message is indeed a removal request from someone who is actually on the mailing list. First, the template checks to see if the Subject line contains the word `Remove`. If so, it now must extract the sender's email address from the string in the message's `From` line (which might contain a friendly name or just an email address). To do so, the template uses `ListFind()` to determine which word in the `From` line—if any—contains an `@` sign, where each word is separated by angle brackets or spaces. If such a word is found, that word is assumed to be the user's email address and is stored in the `fromAddress` variable via the `listGetAt()` function.

Next, the query named `getContact` is run to determine whether a user with the email address in question does indeed exist—and hasn't already been removed from the mailing list. If the query returns a row, the email is coming from a legitimate email address, so it represents a valid removal request.

NOTE

The `getContact` query uses `maxrows="1"` just in case two users exist in the database with the email address in question. If so, only one is removed from the mailing list.

The next `<cfquery>` updates the sender's record in the `Contacts` table, setting the `MailingList` column to 0, effectively removing them from the mailing list. Finally, the sender is sent a confirmation note via a `<cfmail>` tag, so they know their remove request has been received and processed.

The `<cfloop>` then moves on to the next message in the mailbox, until all messages have been processed. With each iteration, the current `UID` is appended to a simple ColdFusion list called `msgDeleteList`. After all messages have been processed, they are deleted from the mailbox using the second `<cfpop>` tag at the bottom of the template. As the template executes, messages are output for debugging purposes, so you can see what the template is doing if you visit using a browser (see Figure 21.9).

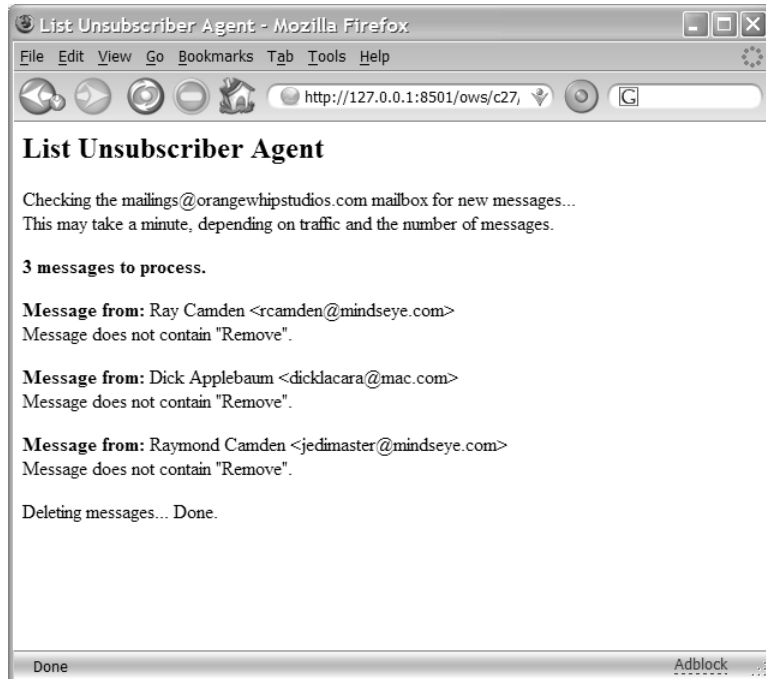


Figure 21.9

Automated POP agents can scan a mailbox for messages and act appropriately.

If you use the `<cfflush>` tag before each `<cfpop>` tag, as this template does, the messages output by the page is displayed in real time as the template executes. See Chapter 29, “Improving the User Experience,” in Vol. 2, Application Development, for details about `<cfflush>`.

Scheduling the Agent Template

Once you have your agent template working properly, you should schedule for automatic, periodic execution using the ColdFusion Administrator or the `<cfschedule>` tag. See Chapter 40, “Event Scheduling,” in Vol. 2, Application Development, for details.

Other Uses for POP Agents

This example simply created a POP-based agent template that responds to unsubscribe requests. It could be expanded to serve as a full-fledged list server, responding to both subscribe and unsubscribe requests. It could even be in charge of forwarding incoming messages back out to members of the mailing list.

That said, ColdFusion isn't designed to be a round-the-clock, high-throughput, mail-generating engine. If you will be sending out tens of thousands of email messages every hour, you should probably think about a different solution. You wouldn't want your ColdFusion server to be so busy tending to its mail delivery duties that it wasn't capable of responding to Web page requests in a timely fashion.

Other POP-based agents could be used to create auto-responder mailboxes that respond to incoming messages by sending back standard messages, perhaps with files attached. You could also create a different type of agent that examines incoming help messages for certain words and sends back messages that should solve the user's problem.

CHAPTER 22

Online Commerce

Building E-commerce Sites

For better or worse, more and more of today's World Wide Web is about selling goods and services, rather than providing freely available information for educational or other purposes. Once the realm of researchers, educators, and techies, the Net is now largely seen as a way to sell to a larger market with less overhead.

Whether this counts as progress is a debate I'll leave for the history books. What it means for you as a Web developer is that sooner or later, you will probably need to build some type of e-commerce Web application, hopefully with ColdFusion.

Common Commerce-Site Elements

No two commerce projects are exactly alike. Nearly every company will have its own idea about what its commerce application should look and feel like, complete with a wish list and feature requirements.

That said, a number of common elements appear in one shape or another on most online commerce sites. If your project is about selling goods or services to the general public, it makes sense to implement a format that's reasonably familiar to users. This section discusses some of the features nearly all shopping sites have in common.

Storefront Area

Most online shopping starts at some type of storefront page, which presents the user with a top-level view of the items or services for sale. Depending on the number of items, these are usually broken down into various categories. From the main storefront page, users generally navigate to the item they want to purchase and then add the item to a virtual shopping cart.

Depending on the company, the storefront area might be its home page and might occupy nearly all of its Web site. This is often the case with an online bookstore or software reseller, for instance. In other situations, the storefront is only a section of a larger site. For instance, Orange Whip Studios' online store is a place to buy merchandise, such as posters and movie memorabilia. It's an important part of the site, but information about upcoming releases, star news, and investor relations will probably be the primary focus.

NOTE

The `Store.cfm` template presented in this chapter is Orange Whip Studios' storefront page.

Promotions and Featured Items

Most shopping sites also ensure that certain items jump out at the user by displaying them prominently, labeled as sale items, featured products, or by some other promotional term. These items are often sprinkled in callouts throughout the company's site to make them easy to find.

NOTE

The `<cf_MerchDisplay>` Custom Tag provided in this chapter offers a simple way to display featured merchandise throughout Orange Whip Studios' Web site.

Shopping Cart

Of course, one of the most important aspects of most commerce sites is the shopping cart. Shopping carts are so ubiquitous on today's Web that users have come to expect them and navigate through them almost intuitively.

If you implement a shopping cart for your application, make sure it looks and feels like carts on other sites, especially those in similar industries. Typically, the user can see the contents of his or her cart on a designated page. From there, the user should be able to remove items from the cart, change quantities, review the total price of items in the cart, and move on to a checkout process.

NOTE

The `StoreCart.cfm` template discussed in this chapter provides a simple shopping cart for the Orange Whip Studios' online store.

Checkout Process

Most users coming to your site will have a preconceived idea of what the checkout process should be like, so you should make it as straightforward and predictable as possible. This means asking the user to fill out one or two pages of forms, on which they provide information such as shipping addresses and credit-card data. Then the user clicks some type of Purchase Now button, which generates an order number and usually charges the user's credit card in real time.

NOTE

The `StoreCheckout.cfm` and `StoreCheckoutForm.cfm` templates in this chapter provide the checkout experience for Orange Whip Studios' virtual visitors.

Order Status and Package Tracking

If you are selling physical goods that need to be shipped after a purchase, users will expect to be able to check the status of their orders online. At a minimum, you should provide an email address to which users can write. You should also consider building a page that lets them check the status of current and past orders in real time.

Many users will also expect to be able to track shipped packages online. You can usually accomplish this via a simple link to the shipping carrier's Web site—such as <http://www.ups.com> OR <http://www.fedex.com>—perhaps passing a tracking number in the

URL. Visit the Web site of your shipping carrier for details (look for some type of developer's section).

- *There isn't space in this chapter to discuss how to build such an order-tracking page, but the `OrderHistory.cfm` template discussed in Chapter 23, "Securing Your Applications," online, is a solid start that gets you most of the way there.*

Using a Secure Server

Before you deploy your commerce site on a production server, consider investing in a Secure Sockets Layer (SSL) server certificate from a company such as VeriSign (<http://www.verisign.com>). You can then use the certificate to set up a secure Web server that employs encryption when communicating with Web browsers. This secured server might or might not reside on the same machines as the company's regular Web servers.

NOTE

Many people are unwilling to place an order at a site that doesn't use SSL security, and rightly so. The decision is yours, of course, but you are strongly urged to use a secured server for collecting any kind of personal information such as credit card numbers.

The secured Web-server instance may have its own document root (perhaps `c:\inetpub\secroot` instead of `c:\inetpub\wwwroot`), or it may share the same document root that your normal Web pages are served from. This will depend on your preferences and the Web-server software you are using. According to your needs, you'll place some or all of your commerce application on the secure server. A typical scenario would put your checkout and order-history pages on the secure server and leave the storefront and cart pages on the regular Web server. If so, the URL for the checkout page would likely be something like `https://secure.orangewhipstudios.com/ows/Checkout.cfm` (instead of `http://www.orangewhipstudios.com/ows/Checkout.cfm`).

NOTE

You configure SSL encryption at the Web-server level; it doesn't relate directly to ColdFusion. Consult your Web server's documentation for details on how to enable SSL and HTTPS with the software you are using.

Creating Storefronts

Before creating code for the shopping cart and checkout process, you should create a simple framework for displaying the products and services your company will be offering for purchase. Then you can organize the items into an online storefront.

Displaying Individual Items

Listing 22.1 creates a CFML Custom Tag called `<cf_MerchDisplay>`. The tag displays a single piece of merchandise for sale, including a picture of the item and the appropriate Add To Cart link. You can use this to display a series of items in Orange Whip Studios' storefront page, as well as to feature individual items as callouts on the home page and throughout the site.

After you have this tag in place, you can use it like this (where `someMerchID` is the name of a variable that identifies the desired item from the `Merchandise` table):

```
<!-- Show item for sale, via custom tag --->
<cf_MerchDisplay
  merchID="#someMerchID#"
  showAddLink="Yes">
```

This Custom Tag is similar conceptually to the `<cf_ShowMovieCallout>` Custom Tag covered in Chapter 26, “Building Reusable Components,” in *ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development*. The two `<cfparam>` tags at the top force the tag to accept two attributes: the desired `merchID` (which is required) and `showAddLink` (which is optional). If `showAddLink` is `Yes` or is not provided, the Custom Tag displays the merchandise item with a link for the user to add the item to the shopping cart. If `showAddLink` is `No`, the same content is displayed, but without the Add To Cart link.

- To make this Custom Tag available to ColdFusion, you should save Listing 22.1 as a file called `MerchDisplay.cfm`, either within the special `CustomTags` folder or in the same folder as the templates that will call it. See Chapter 26 in Vol. 2, *Application Development*, for more information about where to save Custom Tag templates and about CFML Custom Tags in general.

Listing 22.1 `MerchDisplay.cfm`—Creating a Custom Tag to Display Individual Items for Sale

```
<!--
  Filename: MerchDisplay.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides simple online shopping interface
  Please Note Used by Store.cfm page
  --->

<!-- Tag Attributes --->
<!-- MerchID to display (from Merchandise table) --->
<cfparam name="ATTRIBUTES.merchID" type="numeric">

<!-- Controls whether to show "Add To Cart" link --->
<cfparam name="ATTRIBUTES.showAddLink" type="boolean" default="Yes">

<!-- Get information about this part from database --->
<!-- Query-Caching cuts down on database accesses. --->
<cfquery name="getMerch" datasource="#APPLICATION.dataSource#"
  cachedWithin="#createTimeSpan(0,1,0,0)#">
  SELECT
  m.MerchName, m.MerchDescription, m.MerchPrice,
  m.ImageNameSmall, m.ImageNameLarge,
  f.FilmID, f.MovieTitle
  FROM
  Merchandise m INNER JOIN Films f
  ON m.FilmID = f.FilmID
  WHERE
  m.MerchID = #ATTRIBUTES.merchID#
</cfquery>

<!-- Exit tag silently (no error) if item not found --->
<cfif getMerch.recordCount neq 1>
```

```

    <cfexit>
</cfif>

<!--- URL for "Add To Cart" link/button --->
<cfset addLinkURL = "StoreCart.cfm?addMerchID=#ATTRIBUTES.merchID#">

<!--- Now display information about the merchandise --->
<cfoutput>
  <table width="300" cellspacing="0" border="0">
    <tr>
      <!--- Pictures go on left --->
      <td align="center">
        <!--- If there is an image available for item --->
        <!--- (allow user to click for bigger picture) --->
        <cfif getMerch.imageNameLarge neq "">
          <a href=" ../images/#getMerch.ImageNameLarge#">
            </a>
          </cfif>
        </td>
        <!--- Item description, price, etc., go on right --->
        <td style="font-family:arial;font-size:12px">
          <!--- Name of item, associated movie title, etc --->
          <strong>#getMerch.MerchName#</strong><br>
          <font size="1">From the film: #getMerch.MovieTitle#</font><br>
          #GetMerch.MerchDescription#<br>
          <!--- Display Price --->
          <b>Price: #lsCurrencyFormat(getMerch.MerchPrice)#</b><br>

          <!--- If we are supposed to show an "AddToCart" link --->
          <cfif ATTRIBUTES.showAddLink>
            
            <a href="#addLinkURL#">Add To Cart</a><br>
          </cfif>
        </td>
      </tr>
    </table>
  </cfoutput>

```

After the two `<cfparam>` tags, a simple query named `getMerch` gets the relevant information about the piece of merchandise, based on the `merchID` passed to the tag. If for some reason the `merchID` no longer exists, the tag simply stops its processing via the `<cfexit>` tag. No error message is displayed and processing in the calling template continues normally. Next, a variable called `addLinkURL` is constructed, which is the URL to which the user will be sent if he or she decides to add the item to the shopping cart.

- ▶ *The `cachedWithin` attribute to cache the `GetMerch` query in the server's RAM keeps database interaction to a minimum, which improves performance. See the section "Improving Query Performance with Caching" in Chapter 31, "Improving Performance," in Vol. 2, Application Development, for details.*

The rest of the template is straightforward. An HTML table displays a picture of the part (if available), based on the value of the `ImageNameSmall` column in the `Merchandise` table. The user can see a larger version of the image by clicking it.

This makes it easy to display various items for sale throughout a site, based on whatever logic your application calls for. For instance, assuming you've already run a query called `getMerch` that includes a `MerchID` column, you could select a random `MerchID` from one of the query's rows, like so:

```
<!--- Pick an item at random to display as a "Feature" --->
<cfset randNum = randRange(1, getMerch.recordCount)>
<cfset randMerchID = getMerch.MerchID[randNum]>
```

The following could then display the randomly selected merchandise:

```
<!--- Display featured item --->
<cf_MerchDisplay
    merchID="#randMerchID#">
```

Collecting Items into a Store

Depending on the nature of the company, the actual store part of a Web site can be a sprawling, category-driven affair, or something quite simple. Because Orange Whip Studios has relatively few products for sale (less than 20 rows exist in the `Merchandise` table), the best thing might be to create a one-page store that displays all the items.

Listing 22.2 outputs all the items currently available for sale in a two-column display, using ordinary HTML table tags. Because the job of actually displaying the product's name, image, and associated links is encapsulated within the `<cf_MerchDisplay>` Custom Tag, this simple storefront template turns out to be quite short.

Figure 22.1 shows what this storefront looks like in a user's browser.

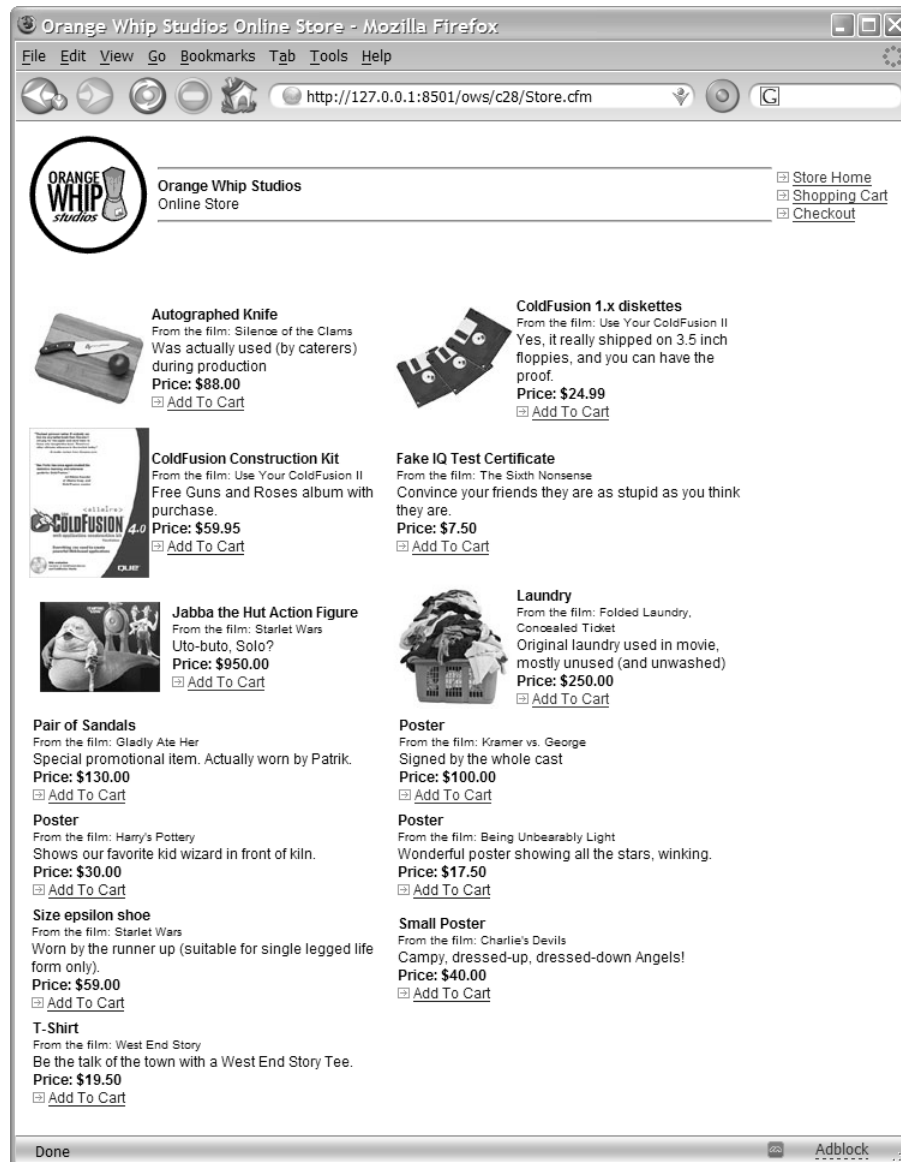


Figure 22.1

Orange Whip Studios' online store lets users peruse the merchandise available for sale.

Listing 22.2 store.cfm—Displaying All Items for Sale

```

<!--
Filename: Store.cfm
Created by: Nate Weiss (NMW)
Purpose: Provides simple online shopping interface
Please Note Relies upon CF_MerchDisplay custom tag
-->

<!-- Get list of merchandise from database -->
<cfquery name="getMerch" datasource="#APPLICATION.dataSource#"
cachedwithin="#createTimeSpan(0,1,0,0)#">
SELECT MerchID, MerchPrice
FROM Merchandise
ORDER BY MerchName
    
```

```

</cfquery>

<!-- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!-- Show merchandise in a HTML table --->
<p>
<table>
  <tr>
    <!-- For each piece of merchandise --->
    <cfloop query="getMerch">
      <td>
        <!-- Show this piece of merchandise --->
        <cf_MerchDisplay
        merchID="#MerchID#">
      </td>

      <!-- Alternate left and right columns --->
      <cfif currentRow mod 2 eq 0></tr><tr></cfif>
    </cfloop>
  </tr>
</table>

</body>
</html>

```

TIP

By altering the ORDER BY part of the query, you could display the items in terms of popularity, price, or other measure.

The `Store.cfm` template in Listing 22.2 displays a common storefront page header at the top of the page by including a file called `StoreHeader.cfm` via a `<cfinclude>` tag. Listing 22.3 creates that header template, which displays Orange Whip Studios' logo, plus links marked Store Home, Shopping Cart, and Checkout. It also establishes a few default font settings via a `<style>` block.

NOTE

This template, and others that follow, require that the `Application.cfc` template be copied from online to your Web server. This file sets up the application variables in use for these templates. Later in the chapter, you will overwrite this basic `Application.cfc` file with a more advanced version.

Listing 22.3 `storeHeader.cfm`—Common Header for All of Orange Whip's Shopping Pages

```

<!--
  Filename: StoreHeader.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides consistent navigation within store
  --->

<!-- "Online Store" page title and header --->
<cfoutput>
  <html>
  <head><title>#APPLICATION.companyName# Online Store</title></head>

```

```

<body>
<style type="text/css">
BODY { font-family:arial,Helvetica,sans-serif;font-size:12px}
TD { font-size:12px}
TH { font-size:12px}
</style>

<table border="0" width="100%">
<tr>
<td width="101">
<!-- Company logo, with link to company home page --->
<a href="http://www.orangewhipstudios.com">
</a>
</td>
<td>
<hr>
<strong>#APPLICATION.companyName#</strong><br>
Online Store<br clear="all">
<hr>
</td>
<td width="100" align="left">
<!-- Link to "Shopping Cart" page --->

<a href="Store.cfm">Store Home</a><br>
<!-- Link to "Shopping Cart" page --->

<a href="StoreCart.cfm">Shopping Cart</a><br>
<!-- Link to "Checkout" page --->

<a href="StoreCheckout.cfm">Checkout</a><br>
</td>
</tr>
</table>
&nbsp;  <br>
</cfoutput>

```

- ▶ *This listing displays all the items for sale on the same page. If you will be selling many items, you might want to create a next-n interface for browsing through the merchandise in groups of 10 or 20 items per page. See Chapter 29, "Improving the User Experience," in Vol. 2, Application Development, for information about how to build next-n interfaces.*

Creating Shopping Carts

Not all shopping-cart experiences are alike, but most are reasonably similar. After you have built one cart application, others will come naturally and quickly. This section presents one way of implementing a shopping cart, which you can adapt for your own applications. First I'll discuss several approaches for remembering shopping-cart contents. Then you'll assemble a simple cart, using just a few ColdFusion templates.

- ▶ *This section discusses a number of concepts introduced in Chapter 20, “Working with Sessions,” including the definition of a Web-based session, as well as ColdFusion’s special `CLIENT` and `SESSION` scopes. I suggest you read or at least skim Chapter 20 before you continue here.*

Storing Cart Information

One of the first things to consider when building a shopping cart is how to store the shopping-cart information. Most users expect to be able to add items to a cart, go somewhere else (perhaps back to your storefront page or to another site for comparison shopping), and then return later to check out. This means you need to maintain the contents of each user’s cart somewhere on your site.

No matter how you decide to store the information, you usually have at least two pieces of information to maintain:

- **Items Added to the Cart.** In most situations, each item will have its own unique identifier (in the sample database, this is the `MerchID` column in the Merchandise table), so remembering which items the user has added to the cart is usually just a matter of remembering one or more ID numbers.
- **Desired Quantity.** Generally, when the user first adds an item to the cart, you should assume they want to purchase just one of that item. The user can usually increase the quantity for each item by adding the same item to the cart multiple times or by going to a View Cart page and entering the desired quantity in a text field.

As far as these examples are concerned, these two pieces of information, considered together, comprise the user’s shopping cart. In many situations, this is all you need to track. Sometimes, though, you also need to track some kind of option for each item, such as a color or discounted price. Typically you can deal with these extra modifiers in the same way that you’ll deal with the quantity in this chapter.

In any case, you can store this information in a number of ways. The most common approaches are summarized here.

CLIENT-Scoped Lists

Perhaps the simplest approach is to simply store the item IDs and quantities as variables in the `CLIENT` scope. As you learned in Chapter 20, the `CLIENT` scope can only store simple values, rather than arrays, structures, and so on. So the simplest option is probably to maintain two variables in the `CLIENT` scope, a ColdFusion-style list of `MerchIDs` and a list of associated quantities.

Aside from its simplicity, one nice thing about this approach is that the user’s cart will persist between visits, without your having to write any additional code. Also, client variables can be set up so that they work within all servers in a cluster.

- ▶ *See Chapter 20 for details about how long `CLIENT` variables are maintained and how they can be stored in the server’s registry, in a database, or as a cookie on the user’s machine.*

This chapter includes example code for a Custom Tag called `<cf_ShoppingCart>`, which uses `CLIENT`-scoped lists to provide shopping-cart experiences for Orange Whip’s visitors.

While it’s true that the `CLIENT` scope can only hold simple values, you can use the `<cfwddx>` tag as a way to store a complex value like an array or structure as a client variable.

SESSION-Scoped Structures and Arrays

Another approach would be to maintain each user's shopping-cart data in the `SESSION` scope. Unlike the `CLIENT` scope, the `SESSION` scope can contain structured data, such as structures and arrays. This means your code can be more elegant and flexible, especially if you have to track more information about each item than just the desired quantity. However, `SESSION` variables are RAM resident and not cluster aware as ColdFusion ships, so you might want to stay away from this approach if you plan to run a number of ColdFusion servers together in a cluster. See Chapter 20 for more pros, cons, and techniques regarding session variables.

This chapter provides sample code for a ColdFusion Component (CFC) called `ShoppingCart`, which uses a `SESSION`-scoped array of structures to provide a shopping-cart experience.

Cart Data in a Database

Another approach is to create additional tables in your database to hold cart information. If you require your users to register before they add items to their carts, you could use their `ContactID` (or whatever unique identifiers you are using for users) to associate cart contents with users. Therefore, you might have a table called `CartContents`, with columns such as `ContactID`, `MerchID`, `Quant`, `DateAdded`, and whatever additional columns you might need, such as `Color` or `Size`. If you don't require users to register before using the cart, you could use the automatic `CLIENT.CFID` variable as a reasonably unique identifier for tracking cart contents.

This approach gives you more control than the others, in particular the capability to maintain easily queried historical information about which items users have added to carts most often and so on (as opposed to items actually purchased). It would also work in a clustered environment. You would probably need to come up with some type of mechanism for flushing very old cart records from the database, however, because they wouldn't automatically expire in the way that `SESSION` and `CLIENT` variables do.

- ▶ *You could handle this periodic table flushing via a scheduled template, as explained in Chapter 40, "Event Scheduling," in Vol. 2, Application Development.*

Building a Shopping Cart

Now that a storefront has been constructed with Add To Cart links for each product, it's time to build the actual shopping cart. This section creates two versions of a ColdFusion template called `StoreCart.cfm`.

If the `StoreCart.cfm` template is visited without any URL parameters, it displays the items in the cart and gives the user the opportunity to either change the quantity of each item or check out, as shown in Figure 22.2. If a `MerchID` parameter is passed in the URL, that item is added to the user's cart before the cart is actually displayed. You will notice that the Add To Cart links generated by the `<cf_MerchDisplay>` Custom Tag (refer to Listing 22.1) do exactly that.

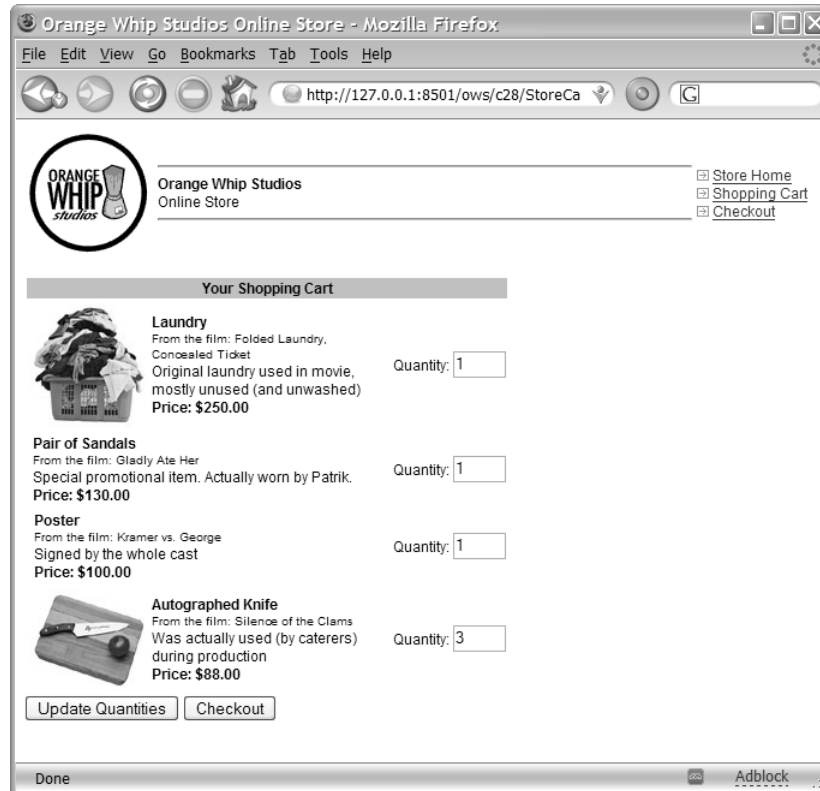


Figure 22.2

From the Shopping Cart page, users can update quantities or proceed to the Checkout phase.

The Simplest Approach

The version of the `StoreCart.cfm` template in Listing 22.4 is probably one of the simplest shopping-cart templates possible. As suggested in the “Storing Cart Information” section earlier in this chapter, each user’s cart data is stored using two comma-separated lists in the `CLIENT` scope. The `CLIENT.cartMerchList` variable holds a comma-separated list of merchandise IDs, and `CLIENT.cartQuantList` holds a comma-separated list of corresponding quantities.

TIP

The next version of the template improves on this one by moving the task of remembering the user’s cart into a CFML Custom Tag. It is recommended that you model your code after the next version of this template, rather than this one. Just use this listing as a study guide to familiarize yourself familiar with the basic concepts.

NOTE

To make the links to the shopping-cart page work correctly, save Listing 22.4 as `StoreCart.cfm`, not `StoreCart1.cfm`.

NOTE

This listing uses client variables, which means you need to enable the `CLIENT` scope in `Application.cfc`. The `Application.cfc` file for this chapter (Listing 22.8) does this. It also creates some additional code used by the CFC version of this shopping cart, discussed in the “A ColdFusion Component Version of the Shopping Cart” section, later in this chapter.

Listing 22.4 storeCart1.cfm—A Simple Shopping Cart

```

<!---
  Filename: StoreCart.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Provides a simple shopping cart interface
-->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- URL parameter for MerchID --->
<cfparam name="URL.addMerchID" type="string" default="">
<!--- These two variables track MerchIDs / Quantities --->
<!--- for items in user's cart (start with empty cart) --->
<cfparam name="CLIENT.cartMerchList" type="string" default="">
<cfparam name="CLIENT.cartQuantList" type="string" default="">

<!--- If MerchID was passed in URL --->
<cfif isNumeric(URL.addMerchID)>
  <!--- Get position, if any, of MerchID in cart list --->
  <cfset currentListPos=listFind(cartMerchList, URL.addMerchID)>
  <!--- If this item *is not* already in cart, add it --->
  <cfif currentListPos eq 0>
    <cfset CLIENT.cartMerchList=listAppend(CLIENT.cartMerchList,
      URL.addMerchID)>
    <cfset CLIENT.cartQuantList=listAppend(CLIENT.cartQuantList, 1)>
  <!--- If item *is* already in cart, change its qty --->
  <cfelse>
    <cfset currentQuant=listGetAt(CLIENT.cartQuantList, currentListPos)>
    <cfset updatedQuant=currentQuant + 1>
    <cfset CLIENT.cartQuantList=listSetAt(CLIENT.cartQuantList, currentListPos,
      updatedQuant)>
  </cfif>
</cfif>

<!--- If no MerchID passed in URL --->
<cfelse>
  <!--- For each item currently in user's cart --->
  <cfloop from="1" to="#listLen(CLIENT.cartMerchList)#" index="i">
    <cfset thisMerchID=listGetAt(CLIENT.cartMerchList, i)>

    <!--- If FORM field exists for this item's Quant --->
    <cfif isDefined("FORM.quant_#thisMerchID#")>
      <!--- The FORM field value is the new quantity --->
      <cfset newQuant=FORM["quant_#thisMerchID#"]>
      <!--- If new quant is 0, remove item from cart --->
      <cfif newQuant eq 0>
        <cfset CLIENT.cartMerchList=listDeleteAt(CLIENT.cartMerchList, i)>
        <cfset CLIENT.cartQuantList=listDeleteAt(CLIENT.cartQuantList, i)>
      <!--- Otherwise, Update cart with new quantity --->
      <cfelse>
        <cfset CLIENT.cartQuantList=listSetAt(CLIENT.cartQuantList, i,
          newQuant)>
      </cfif>
    </cfif>
  </cfloop>
</cfelse>

```

```

        </cfif>
    </cfloop>

    <!--- If user submitted form via "Checkout" button --->
    <cfif isDefined("FORM.isCheckingOut")>
        <cflocation URL="StoreCheckout.cfm">
    </cfif>
</cfif>

<!--- Stop here if user's cart is empty --->
<cfif CLIENT.cartMerchList eq "">
    There is nothing in your cart.
    <cfabort>
</cfif>

<!--- Create form that submits to this template --->
<cfform action="#CGI.script_name#">
<table>
<tr>
    <th colspan="2" bgcolor="Silver">Your Shopping Cart</th>
</tr>
<!--- For each piece of merchandise --->
<cfloop from="1" to="#listLen(CLIENT.cartMerchList)#" index="i">
    <cfset thisMerchID=listGetAt(CLIENT.cartMerchList, i)>
    <cfset thisQuant=listGetAt(CLIENT.cartQuantList, i)>
    <tr>
        <td>
            <!--- Show this piece of merchandise --->
            <cf_MerchDisplay merchID="#thisMerchID#" showAddLink="No">
        </td>
        <td>
            <!--- Display Quantity in Text entry field --->
            <cfoutput>
                Quantity:
                <cfinput type="Text" name="quant_#thisMerchID#" size="3" value="#thisQuant#">
            </cfoutput>
        </td>
    </tr>
</cfloop>
</table>

    <!--- Submit button to update quantities --->
    <cfinput type="submit" name="submit" value="Update Quantities">

    <!--- Submit button to Check out --->
    <cfinput type="Submit" value="Checkout" name="IsCheckingOut">
</cfform>

```

The `<cfform>` section at the bottom of this template is what displays the contents of the user's cart, based on the contents of the `CLIENT.CartMerchList` and `CLIENT.CartQuantList` variables. Suppose for the moment that the current value of `CartMerchList` is 5,8 (meaning the user has added items number 5 and 8 to the cart) and that `CartQuantList` is 1,2 (meaning the user wants to buy one of item number 5 and two of item number 8). If so, the `<cfloop>` near the bottom of this template will execute twice. The first time

through the loop, `thisMerchID` will be 5 and `thisQuant` will be 1. Item number 5 is displayed with the `<cf_MerchDisplay>` tag, and then a text field called `quant_5` is displayed, prefilled with a value of 1. This text field enables the user to adjust the quantities for each item.

At the very bottom of the template, two submit buttons are provided, labeled Update Quantities and Checkout. Both submit the form, but the Checkout button sends the user on to the Checkout phase after the cart quantities have been updated.

Updating Cart Quantities

Three `<cfparam>` tags are at the top of Listing 22.4. The first makes it clear that the template can take an optional `addMerchID` parameter. The next two ensure that the `CLIENT.cartMerchList` and `CLIENT.cartQuantList` variables are guaranteed to exist. If not, they are initialized to empty strings, which represent an empty shopping cart.

If a numeric `addMerchID` is passed to the page, the first `<cfif>` block executes. The job of this block of code is to add the item indicated by `URL.addMerchID` to the user's cart. First, the `listFind()` function sets the `currentListPos` variable. This variable is 0 if the `addMerchID` value is not in `CLIENT.cartMerchList` (in other words, if the item is not in the user's cart). Therefore, this function places the `addMerchID` value in the user's cart by appending it to the current `cartMerchList` value, and by appending a quantity of 1 to the current `merchQuantList` value.

If, on the other hand, the item is already in the user's cart, `currentListPos` is the position of the item in the comma-separated lists that represent the cart. Therefore, the current quantity for the passed `addMerchID` value can be obtained with the `listGetAt()` function and stored in `currentQuant`. The current quantity is incremented by 1, and the updated quantity is placed in the appropriate spot in `CLIENT.cartQuantList`, via the `listSetAt()` function.

The large `<cfelse>` block executes when the user submits the form, using the Update Quantities or Checkout button (see Figure 22.2). The `<cfloop>` loops through the list of items in the user's cart. Again, supposing that `CLIENT.cartMerchList` is currently 5,8, then `thisMerchID` is set to 5 the first time through the loop. If a form variable named `FORM.quant_5` exists, that form value represents the user's updated quantity for the item. If the user has specified an updated quantity of 0, it is assumed that the user wants to remove the item from the cart, so the appropriate values in `cartMerchList` and `cartQuantList` are removed using the `listDeleteAt()` function. If the user has specified some other quantity, the quantity in `cartQuantList` is updated, using the `listSetAt()` function.

Finally, if the user submitted the form using the Checkout button, the browser is directed to the `CartCheckout.cfm` template via the `<cflocation>` tag.

At this point, the shopping cart is quite usable. The user can go to the `Store.cfm` template (refer to Figure 22.1) and add items to the shopping cart. Once at the shopping cart (see Figure 22.2), the user can update quantities or remove items by setting the quantity to 0.

Encapsulating the Shopping Cart in a Custom Tag

Although the version of `StoreCart.cfm` in Listing 22.4 works just fine, the code itself is a bit messy. It contains quite a few list functions, which don't necessarily have to do with

the conceptual problem at hand (the user’s cart). Worse, other templates that need to refer to the user’s cart (such as the Checkout template) must use nearly all the same list functions over again, resulting in quite a bit of code for you to maintain.

You will learn about Custom Tags in Chapter 25, “Creating Custom Tags,” in Vol. 2, *Application Development*. This early example is rather simple, but if you find it confusing, read ahead to Chapter 25.

Building <cf_ShoppingCart>

The code in Listing 22.5 creates a Custom Tag called <cf_ShoppingCart>, which encapsulates all the list-manipulation details necessary to maintain the user’s shopping cart. After you save Listing 22.5 as `ShoppingCart.cfm` in the special `CustomTags` folder (or just in the same folder where you’ll be using it), it will be capable of accomplishing any of the tasks shown in Table 22.1.

Table 22.1 Syntax Supported by the <cf_ShoppingCart> Custom Tag Example

Desired Action	Sample Code
Add an item to the user’s cart	<cf_ShoppingCart action="Add" merchID="5">
Update the quantity of an item	<cf_ShoppingCart action="Update" merchID="5" quantity="10">
Remove an item from the cart	<cf_ShoppingCart action="Remove" merchID="5">
Remove all items from cart	<cf_ShoppingCart action="Empty">
Retrieve all items in cart	<cf_ShoppingCart action="List" returnVariable="GetCart">

`action="List"` returns the cart’s contents as a ColdFusion query object; the query object will contain two columns, `MerchID` and `Quantity`.)

Because the various `action` tasks provided by this Custom Tag all relate to a single concept (a shopping cart), you can think of the Custom Tag as an object-based representation of the cart. That makes it an ideal candidate for turning into a ColdFusion Component (CFC). See Chapter 23, online, for further discussion of Custom Tags and CFCs as objects.

Listing 22.5 `ShoppingCart.cfm`—Constructing the <cf_ShoppingCart> Custom Tag

```
<!---
  Filename: ShoppingCart.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Creates the <CF_ShoppingCart> Custom Tag
  --->

<!--- Tag Parameters --->
<cfparam name="ATTRIBUTES.action" type="string">

<!--- These two variables track MerchIDs / Quantities --->
<!--- for items in user’s cart (start with empty cart) --->
<cfparam name="CLIENT.cartMerchList" type="string" default="">
<cfparam name="CLIENT.cartQuantList" type="string" default="">
```

```

<!-- This tag is being called with what ACTION? -->
<cfswitch expression="#ATTRIBUTES.action#">

    <!-- *** ACTION="Add" or ACTION="Update" *** -->
    <cfcase value="Add,Update">
        <!-- Tag attributes specific to this ACTION -->
        <cfparam name="ATTRIBUTES.merchID" type="numeric">
        <cfparam name="ATTRIBUTES.quantity" type="numeric" default="1">

        <!-- Get position, if any, of MerchID in cart list -->
        <cfset currentListPos = listFind(CLIENT.cartMerchList, ATTRIBUTES.merchID)>
        <!-- If this item *is not* already in cart, add it -->
        <cfif currentListPos eq 0>
            <cfset CLIENT.cartMerchList =
                listAppend(CLIENT.cartMerchList, ATTRIBUTES.merchID)>
            <cfset CLIENT.cartQuantList =
                listAppend(CLIENT.cartQuantList, ATTRIBUTES.quantity)>
            <!-- If item *is* already in cart, change its qty -->
        <cfelse>
            <!-- If Action="Add", add new Qty to existing -->
            <cfif ATTRIBUTES.action eq "Add">
                <cfset ATTRIBUTES.quantity =
                    ATTRIBUTES.quantity + listGetAt(CLIENT.cartQuantList, currentListPos)>
            </cfif>

            <!-- If new quantity is zero, remove item from cart -->
            <cfif ATTRIBUTES.quantity eq 0>
                <cfset CLIENT.cartMerchList =
                    listDeleteAt(CLIENT.cartMerchList, currentListPos)>
                <cfset CLIENT.cartQuantList =
                    listDeleteAt(CLIENT.cartQuantList, currentListPos)>
            <!-- If new quantity not zero, update cart quantity -->
            <cfelse>
                <cfset CLIENT.cartQuantList =
                    listSetAt(CLIENT.cartQuantList, currentListPos, ATTRIBUTES.quantity)>
            </cfif>
        </cfif>
    </cfcase>

    <!-- *** ACTION="Remove" *** -->
    <cfcase value="Remove">
        <!-- Tag attributes specific to this ACTION -->
        <cfparam name="ATTRIBUTES.merchID" type="numeric">
        <!-- Treat "Remove" action same as "Update" with Quant=0 -->
        <cf_ShoppingCart
            aCTION="Update"
            merchID="#ATTRIBUTES.MerchID#"
            quantity="0">
    </cfcase>

    <!-- *** ACTION="Empty" *** -->
    <cfcase value="Empty">

```

```

        <cfset CLIENT.cartMerchList = "">
        <cfset CLIENT.cartQuantList = "">
    </cfcase>

    <!--- *** ACTION="List" *** --->
    <cfcase value="List">
        <!--- Tag attributes specific to this ACTION --->
        <cfparam name="ATTRIBUTES.returnVariable" type="variableName">

        <!--- Create a query, to return to calling template --->
        <cfset q = queryNew("MerchID,Quantity")>

        <!--- For each item in CLIENT lists, add row to query --->
        <cfloop from="1" to="#listLen(CLIENT.cartMerchList)#" index="i">
            <cfset queryAddRow(q)>
            <cfset querySetCell(q, "MerchID", listGetAt(CLIENT.cartMerchList, i))>
            <cfset querySetCell(q, "Quantity", listGetAt(CLIENT.cartQuantList, i))>
        </cfloop>

        <!--- Return query to calling template --->
        <cfset "Caller.#ATTRIBUTES.returnVariable#" = q>
    </cfcase>

    <!--- If an unknown ACTION was provided, display error --->
    <cfdefaultcase>
        <cfthrow
            message="Unknown ACTION passed to &lt;CF_ShoppingCart&gt;"
            detail="Recognized ACTION values are <B>List</B>, <B>Add</B>,
                <B>Update</B>, <B>Remove</B>, and <B>Empty</B>."
    </cfdefaultcase>

</cfswitch>

```

NOTE

Some of the <cfset> tags in this template are broken somewhat unusually across two lines (the left side of the expression on one line and the right side on the next line) to make them easier to read in this book. In your actual code templates, you would probably have the whole <cfset> statement on one line, but this listing does show that ColdFusion can deal uncomplainingly with expressions spanning multiple lines.

This Custom Tag supports its various tasks by requiring an `action` attribute (required by the `<cfparam>` tag at the top of Listing 22.5), and then handling each of the supported actions in separate `<cfcase>` tags within a large `<cfswitch>` block. If the action is Add or Update, the first `<cfcase>` tag executes. If the action is Remove, the second one executes, and so on.

If `action="Add"` or `action="Update"`, the tag accepts two additional parameters—`merchID` (required) and `quantity` (optional, defaulting to 1). The `<cfcase>` block for these actions is similar to the top portion of Listing 22.4, using `listFind()` to determine whether the item is already in the user's cart and then adding it to the cart with `listAppend()` or updating the quantity using `listSetAt()`. Also, if `action="Update"` and `Quantity="0"`, the item is removed from the user's cart.

If the tag is called `action="Remove"`, the tag just calls itself again, using `action="Update"` and `quantity="0"` to remove the item from the user's cart. So `Remove` is just a synonym for an `Update` that sets the quantity to 0.

If `action="Empty"`, the `CLIENT.cartMerchList` and `CLIENT.cartQuantList` are emptied by setting them both to empty strings. This has the effect of removing all items from the user's cart.

Finally, if `action="List"`, the tag creates a new ColdFusion query object using the `queryNew()` function, which the calling template will be capable of using as if it were generated by an ordinary `<cfquery>` tag. The new query has two columns, `merchID` and `quantity`. For each item in the `cartMerchList` and `cartQuantList` lists, a row is added to the query using `queryAddRow()`; then the `MerchID` and `Quantity` columns of the just-inserted row are set using the `querySetCell()` function. The end result is a simple two-column query that contains a row for each item in the user's cart. The query object is returned to the calling template with the name specified in the tag's `returnVariable` attribute.

- ▶ *One of the goals for this Custom Tag is to ensure that no other template will need to refer to the `CLIENT.cartMerchList` and `CLIENT.cartQuantList` variables. The Custom Tag will therefore be a clean abstraction of the concept of a user's cart, including the storage method (currently the two lists in the `CLIENT` scope). If you later decide to use `SESSION` variables or a database table to hold each user's cart data, you only have to change the code in the Custom Tag template. See Chapter 25 in Vol. 2, Application Development, for more discussion about attaining the holy grail of abstraction via Custom Tags.*

Putting `<cf_shoppingCart>` to Work

The version of `StoreCart.cfm` in Listing 22.6 is a revision of the one in Listing 22.4. As far as the user is concerned, it behaves the same way. However, it removes all references to the internal storage mechanisms (the `CLIENT` variables, list functions, and so on). As a result, the code reads well, and it will be clearer to future coders and easier for you to reuse and maintain.

NOTE

To make the links to the shopping-cart page work correctly, you should save Listing 22.6 as `StoreCart.cfm`, not `StoreCart2.cfm`.

Listing 22.6 `storeCart2.cfm`—Using `<cf_shoppingCart>` to Rebuild `storeCart.cfm` Template

```
<!---
Filename: StoreCart.cfm
Created by: Nate Weiss (NMW)
Purpose: Provides a simple shopping cart interface
-->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- If MerchID was passed in URL --->
<cfif isDefined("URL.addMerchID")>
    <!--- Add item to user's cart data, via custom tag --->
    <cf_shoppingCart
```

```

    action="Add"
    merchID="#URL.addMerchID#"

<!-- If user is submitting cart form -->
<cfelseif isDefined("FORM.merchID")>
    <!-- For each MerchID on Form, Update Quantity -->
    <cfloop list="#FORM.merchID#" INDEX="thisMerchID">
        <!-- Update Quantity, via Custom Tag -->
        <cf_ShoppingCart
            action="Update"
            merchID="#ThisMerchID#"
            quantity="#FORM['quant_#thisMerchID#']#">
        </cfloop>

    <!-- If user submitted form via "Checkout" button, -->
    <!-- send on to Checkout page after updating cart. -->
    <cfif isDefined("FORM.isCheckingOut")>
        <cflocation url="StoreCheckout.cfm">
    </cfif>
</cfif>

<!-- Get current cart contents, as a query object -->
<cf_ShoppingCart action="List" returnVariable="getCart">

<!-- Stop here if user's cart is empty -->
<cfif getCart.recordCount eq 0>
    There is nothing in your cart.
    <cfabort>
</cfif>

<!-- Create form that submits to this template -->
<cfform action="#CGI.script_name#">
<table>
<tr>
    <th colspan="2" bgcolor="Silver">Your Shopping Cart</th>
</tr>
<!-- For each piece of merchandise -->
<cfloop query="getCart">
    <tr>
        <td>
            <!-- Show this piece of merchandise -->
            <cf_MerchDisplay
                merchID="#getCart.MerchID#"
                showAddLink="No">
            </td>
            <td>
                <!-- Display Quantity in Text entry field -->
                <cfoutput>
                    Quantity:
                    <cfinput type="hidden" name="merchID" value="#getCart.MerchID#">
                    <cfinput type="text" size="3" name="quant_#getCart.MerchID#"
                        value="#getCart.Quantity#">
                </cfoutput>
            </td>
        </tr>
    </cfloop>
</table>
</cfform>

```

```
        </td>
    </tr>
</cfloop>
</table>

<!-- Submit button to update quantities --->
<cfinput type="submit" name="submit" value="Update Quantities">

<!-- Submit button to Check out --->
<cfinput type="submit" value="Checkout" name="IsCheckingOut">
</cfform>
```

If the template receives an `addMerchID` parameter in the URL, the `<cf_ShoppingCart>` tag is called with `action="Add"` to add the item to the user's cart. If the user submits the shopping cart form with the Update Quantities or Checkout button, the template loops through the merchandise elements on the form, calling `<cf_ShoppingCart>` with `action="Update"` for each one.

Then, to display the items in the user's cart, the `<cf_ShoppingCart>` tag is called again, this time with `action="List"`. Because `getCart` is specified for the `returnVariable` attribute, the display portion of the code just needs to use a `<cfloop>` over the `getCart` query, calling `<cf_MerchDisplay>` for each row to get the merchandise displayed to the user.

A ColdFusion Component Version of the Shopping Cart

Later in this book, you will learn about ColdFusion Components (see Chapter 26 in Vol. 2, *Application Development*). Components provide a powerful way to abstract and encapsulate logic. We are going to introduce you to a ColdFusion Component a bit before Chapter 26. If at any point you get confused, don't worry. A full and complete discussion of CFCs will come later.

At a basic level, a component is a collection of functions (things to do) and data. A shopping cart, which we have already seen brought to life by `<cf_ShoppingCart>`, is such a collection. Think of the cart as a type of object. The object can execute different actions, like `Add`, `Update`, and `List`. We can think of these actions as the object's methods. Most important, each instance of the shopping-cart object holds its own data. That is, each individual user's shopping cart is structurally the same (all spawned from the same object type, with the same methods and so on), but holds a different set of items.

It seems, then, that a shopping cart would be an ideal candidate for turning into a ColdFusion component. As you will see in this section, it is quite easy to turn the code for the `<cf_ShoppingCart>` Custom Tag into a ColdFusion Component called `ShoppingCart`. That will give Orange Whip Studios the option of exposing the shopping cart as a Web Service in the future. It will also open up the possibility of creating a slick Flash version of the shopping experience for visitors, because of Flash's ability to interact easily with ColdFusion Components via the Flash Remoting service.

Building the `shoppingCart` Component

The first step in converting the shopping-cart code from the Custom Tag implementation to a CFC implementation is largely a matter of changing the large `<cfswitch>` block from the Custom Tag into a `<cfcomponent>` tag, and changing each of the `<cfcase>` tags to

<cffunction> tags. Obviously, that's not all you must do, but you'll see that the CFC code within each <cffunction> is strongly related to the corresponding <cfcase> from the Custom Tag (see Listing 22.5).

Listing 22.7 provides code for a simple component-based shopping cart. The most obvious change is the introduction of CFC framework tags like <cfcomponent> and <cffunction> to establish the CFC's methods. Table 22.2 shows the methods exposed by the CFC.

Remember that once you create this CFC, you can view automatically generated documentation for it by visiting the URL for the .cfc file with your browser, or by choosing Get Description from the right-click menu for the component in the Components tab of the Application panel in Dreamweaver. The automatic documentation page for the ShoppingCart component is shown in Figure 22.3.

ows.22.ShoppingCart

Component ShoppingCart

hierarchy:	WEB-INF.cftags.component ows.22.ShoppingCart
path:	/Library/WebServer/Documents/ows/22/ShoppingCart.cfc
properties:	
methods:	Add, Empty, List, Remove, Update

* - private method

Add

```
public void Add ( required numeric merchID, numeric quantity="1" )
```

Adds an item to the shopping cart

Output: suppressed

Parameters:
merchID: numeric, required, merchID
quantity: numeric, optional, quantity

Empty

```
public void Empty ( )
```

Removes all items from the shopping cart

Output: suppressed

List

```
public query List ( )
```

Returns a query object containing all items in shopping cart. The query object has two columns: MerchID and Quantity.

Output: suppressed

Remove

```
public void Remove ( required numeric merchID )
```

Figure 22.3

ColdFusion generates documentation for CFCs.

Table 22.2 Methods Exposed by the ShoppingCart CFC

Method	What It Does
add(merchID, quantity)	Adds an item to the shopping cart. The quantity argument is optional (defaults to 1).
update(merchID, quantity)	Updates the quantity of a particular item in the shopping cart.
remove(merchID)	Removes an item from the shopping cart.
empty()	Removes all items from the shopping cart.
list()	Returns a query that contains all items currently in the shopping cart.

Listing 22.7 ShoppingCart.cfc—Creating the ShoppingCart Component

```
<!---
Filename: ShoppingCart.cfc
Created by: Nate Weiss (NMW)
Purpose: Creates a CFC called ShoppingCart
-->

<cfcomponent output="false">
    <!--- Initialize the cart's contents --->
    <!--- Because this is outside of any <CFFUNCTION> tag, --->
    <!--- it only occurs when the CFC is first created --->
    <cfset VARIABLES.cart = structNew()

    <!--- *** ADD Method *** --->
    <cffunction name="Add" access="public" returnType="void" output="false"
        hint="Adds an item to the shopping cart">
        <!--- Two Arguments: MerchID and Quantity --->
        <cfargument name="merchID" type="numeric" required="Yes">
        <cfargument name="quantity" type="numeric" required="no" default="1">

        <!--- Is this item in the cart already? --->
        <cfif structKeyExists(VARIABLES.cart, arguments.merchID)>
            <cfset VARIABLES.cart[arguments.merchID] =
                VARIABLES.cart[arguments.merchID] + arguments.quantity>
        <cfelse>
            <cfset VARIABLES.cart[arguments.merchID] = arguments.quantity>
        </cfif>

    </cffunction>

    <!--- *** UPDATE Method *** --->
    <cffunction name="Update" access="public" returnType="void" output="false"
        hint="Updates an item's quantity in the shopping cart">
        <!--- Two Arguments: MerchID and Quantity --->
        <cfargument name="merchID" type="numeric" required="Yes">
        <cfargument name="quantity" type="numeric" required="Yes">

        <!--- If the new quantity is greater than zero --->
        <cfif arguments.quantity gt 0>
            <cfset VARIABLES.cart[arguments.merchID] = arguments.quantity>
            <!--- If new quantity is zero, remove the item from cart --->
        <cfelse>
            <cfset remove(arguments.merchID)>
        </cfif>

    </cffunction>
```

```

<!--- *** REMOVE Method *** --->
<cffunction name="Remove" access="public" returnType="void" output="false"
    hint="Removes an item from the shopping cart">
    <!--- One Argument: MerchID --->
    <cfargument name="merchID" type="numeric" required="Yes">

    <cfset structDelete(VARIABLES.cart, arguments.merchID)>
</cffunction>

<!--- *** EMPTY Method *** --->
<cffunction name="Empty" access="public" returnType="void" output="false"
    hint="Removes all items from the shopping cart">

    <!--- Empty the cart by clearing the This.CartArray array --->
    <cfset structClear(VARIABLES.cart)>
</cffunction>

<!--- *** LIST Method *** --->
<cffunction name="List" access="public" returnType="query" output="false"
    hint="Returns a query object containing all items in shopping
    cart. The query object has two columns: MerchID and Quantity.">

    <!--- Create a query, to return to calling process --->
    <cfset var q = queryNew("MerchID,Quantity")>
    <cfset var key = "">

    <!--- For each item in cart, add row to query --->
    <cfloop collection="#VARIABLES.cart#" item="key">
        <cfset queryAddRow(q)>
        <cfset querySetCell(q, "MerchID", key)>
        <cfset querySetCell(q, "Quantity", VARIABLES.cart[key])>
    </cfloop>

    <!--- Return completed query --->
    <cfreturn q>
</cffunction>

</cfcomponent>

```

Aside from the structural makeup of the code, you've made another important change here. While the Custom Tag version used comma-separated lists stored in the `CLIENT` scope to remember the items in each user's cart, this new component version uses a struct called `cart` to hold the cart's contents. Each time the user selects merchandise to purchase, a key will be added to the structure to hold the quantity of that particular item. So if the user has selected three items for purchase, the structure will have three keys. Each key represents the `MerchID` value, with the value of that key being the `Quantity`.

The `cart` structure is stored in the CFC's `VARIABLES` scope. This allows all of the methods of the CFC to manipulate the values.

The `Add()` method at the top of Listing 22.7 does its work with just a few lines of code. It first checks to see if the `merchID` value already exists in the struct. If it does, the passed in quantity is added to the existing quantity. If the value doesn't exist, a new key is created

with the quantity. The `Update()` method is even simpler. Because we are resetting the quantity and not adding, we can simply set the value in the struct. It doesn't matter if it existed already. If the quantity provided is 0, the CFC's `Remove()` method is called to remove the item from the cart. That `Remove()` function is also really simple. It just uses the `structDelete()` function to remove the key from the structure. This method could be modified to check to see if the key actually existed, but it really isn't necessary.

The `Empty()` method is the simplest of all; it simply discards all items from the `VARIABLES.cart` variable with the `structClear()` function. And the code for the `List()` function is very similar to the corresponding code in Listing 22.5.

Using the ShoppingCart Component

Now that the `ShoppingCart` component has been built, it's easy to put it to work. The first thing to do to is to make sure each user gets their own `ShoppingCart` instance. Take a look at Listing 22.8, a simple `Application.cfc` file. The `<cfset>` lines are nothing new; they have been used in most `Application.cfc` files since Chapter 19, "Introducing the Web Application Framework."

The new thing here is the addition of the `onSessionStart()` method and the `<cfobject>` tag.

Listing 22.8 `Application.cfc`—Creating a New CFC Instance for Each User

```
<!---
Filename:   Application.cfc
Created by: Raymond Camden (ray@camdenfamily.com)
Please Note Executes for every page request!
--->

<cfcomponent output="false">

    <!--- Name the application. --->
    <cfset this.name = "c22">
    <!--- Turn on session management. --->
    <cfset this.sessionManagement = true>
    <cfset this.clientManagement = true>

    <cffunction name="onApplicationStart" returnType="void" output="false">
        <cfset APPLICATION.dataSource="ows">
        <cfset APPLICATION.companyName="Orange Whip Studios">
    </cffunction>

    <cffunction name="onSessionStart" returnType="void" output="false">

        <cfobject name="SESSION.myShoppingCart" component="ShoppingCart">

    </cffunction>

</cfcomponent>
```

When a user first visits the application, the code inside the `onSessionStart()` block executes, which means that a new instance of the `ShoppingCart` CFC is created and stored

in the `SESSION` scope. The CFC instance remains in ColdFusion's memory for the remainder of the user's visit, or until the server is restarted or the user's session expires. That's all you need to do to give each user a shopping cart.

NOTE

Note that the component code itself (in Listing 22.7) did not refer to the `SESSION` scope at all. Instead, it referred only to the `VARIABLES` scope provided by the CFC framework. It is only now, when the CFC is instantiated, that it becomes attached to the notion of a session.

Now all that remains is to go back to the `StoreCart2.cfm` template from Listing 22.6 and replace the calls to the `<cf_ShoppingCart>` tag with calls to the CFC's methods. Listing 22.9 is the resulting template.

NOTE

To keep the various links between pages intact, remember to save this template as `StoreCart.cfm`, not `StoreCart3.cfm`.

Listing 22.9 `storeCart3.cfm`—Putting the `ShoppingCart` CFC to Work

```
<!---
Filename: StoreCart.cfm
Created by: Nate Weiss (NMW)
Purpose: Provides a simple shopping cart interface
-->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- If MerchID was passed in URL --->
<cfif isDefined("URL.AddMerchID")>
    <!--- Add item to user's cart data --->
    <cfinvoke component="#SESSION.myShoppingCart#" method="Add"
    merchid="#URL.addMerchID#">

<!--- If user is submitting cart form --->
<cfelseif isDefined("FORM.merchID")>
    <!--- For each MerchID on Form, Update Quantity --->
    <cfloop list="#FORM.merchID#" index="thisMerchID">
        <!--- Add item to user's cart data --->
        <cfinvoke component="#SESSION.myShoppingCart#" method="Update"
        merchid="#thisMerchID#" quantity="#FORM['quant_#thisMerchID#']#">
    </cfloop>

    <!--- If user submitted form via "Checkout" button, --->
    <!--- send on to Checkout page after updating cart. --->
    <cfif isDefined("FORM.isCheckingOut")>
        <cflocation url="StoreCheckout.cfm">
    </cfif>
</cfif>

<!--- Get current cart contents, as a query object --->
<cfset getCart = SESSION.myShoppingCart.List()>
```

```

<!-- Stop here if user's cart is empty --->
<cfif getCart.recordCount eq 0>
    There is nothing in your cart.
    <cfabort>
</cfif>

<!-- Create form that submits to this template --->
<cfform action="#CGI.SCRIPT_NAME#">
<table>
<tr>
    <th colspan="2" bgcolor="Silver">Your Shopping Cart</th>
</tr>
<!-- For each piece of merchandise --->
<cfloop query="getCart">
    <tr>
        <td>
            <!-- Show this piece of merchandise --->
            <cf_MerchDisplay
            merchID="#getCart.MerchID#"
            showAddLink="No">
        </td>
        <td>
            <!-- Display Quantity in Text entry field --->
            <cfoutput>
            Quantity:
            <cfinput type="hidden" name="merchID" value="#getCart.MerchID#">
            <cfinput type="text" size="3" name="quant_#getCart.MerchID#"
                value="#getCart.Quantity#">
            </cfoutput>
        </td>
    </tr>
</cfloop>
</table>

<!-- Submit button to update quantities --->
<cfinput type="submit" name="submit" value="Update Quantities">

<!-- Submit button to Check out --->
<cfinput type="submit" value="Checkout" name="IsCheckingOut">
</cfform>

```

As you will learn in Chapter 26 in Vol. 2, *Application Development*, there are two basic ways to invoke a CFC's methods from a ColdFusion template: with the `<cfinvoke>` tag, or by using script-style method syntax in a `<cfset>` or other expression. This listing shows both.

For instance, in Listing 22.6, the following code retrieved the contents of the user's cart:

```

<!-- Get current cart contents, via Custom Tag --->
<cf_ShoppingCart
    action="List"
    returnVariable="GetCart">

```

In Listing 22.9, the equivalent line is:

```

<!-- Get current cart contents, as a query object --->

```

```
<cfset getCart = SESSION.myShoppingCart.List()>
```

The following `<cfinvoke>` syntax would do the same thing:

```
<!-- Add item to user's cart data --->
<cfinvoke
component="#SESSION.myShoppingCart#"
method="List"
returnVariable="getCart">
```

Similarly, this line calls the CFC's `Add` method:

```
<!-- Add item to user's cart data --->
<cfinvoke
component="#SESSION.myShoppingCart#"
method="Add"
merchID="#URL.addMerchID#">
```

You could change it to the following, which would do the same thing:

```
<cfset SESSION.myShoppingCart.add(URL.addMerchID)>
```

As you can see, the `<cfinvoke>` syntax is a bit more self-explanatory because the arguments are explicitly named. However, the script-style syntax is usually much more concise and perhaps easier to follow logically. Use whatever style you prefer.

Payment Processing

Now that Orange Whip Studios' storefront and shopping-cart mechanisms are in place, it's time to tackle the checkout process. While by no means difficult, this part generally takes the most time to get into place because you must make some decisions about how to accept and process the actual payments from your users.

Depending on the nature of your application, you might not need real-time payment processing. For instance, if your company bills its customers at the end of each month, you probably just need to perform some type of query to determine the status of the user's account, rather than worrying about collecting a credit-card number and charging the card when the user checks out.

However, most online shopping applications call for getting a credit-card number from a user at checkout time and charging the user's credit-card account in real time. That is the focus of this section.

Payment-Processing Solutions

Assuming you'll be collecting credit-card information from your users, you must first decide how you will process the credit-card charges that come through your application. ColdFusion doesn't ship with any specific functionality for processing credit-card transactions. However, a number of third-party packages enable you to accept payments via credit cards and checks.

NOTE

Because it is quite popular, the examples in this chapter use the Payflow Pro payment-processing service from VeriSign (www.verisign.com). But VeriSign's service is just one of several solutions available. You are encouraged to investigate other payment-processing software to find the service or package that makes the most sense for your project.

Processing a Payment

The exact ColdFusion code you use to process payments will vary according to the payment-processing package you decide to use. Although multiple options are available to you, most require either setting up an account with a bank or providing a credit card to access the service. Because of this, the code use for the Orange Whip site will be fake. What do we mean by that? We simply mean that the code will accept common payment options (amount, credit card number, and so on), but will always return `true`. This approach is necessary because of the difficulty of setting up a “real” e-commerce connection. If you are interested in setting up a real payment process account, consider one of the following options:

- Merchant Services by PayPal (http://www.paypal.com/cgi-bin/webscr?cmd=_merchant-outside)
- Google Checkout (<http://checkout.google.com/sell>)

Writing a Custom Tag Wrapper to Accept Payments

Listing 22.10 creates a CFML Custom Tag called `<cf_ProcessPayment>`. It will act as our fake “accept anything” payment processor.

The idea here is similar to the idea behind the `<cf_ShoppingCart>` tag created earlier in this chapter: Keep all the mechanics in the Custom Tag template, so each individual page can use simpler, more goal-oriented syntax. In addition to the `Processor` attribute, this sample version of the `<cf_ProcessPayment>` tag accepts the following attributes:

- `orderId`. This is passed along to the credit-card company as a reference number.
- `orderAmount`, `creditCard`, `creditExpM`, `creditExpY`, and `creditName`. These describe the actual payment to be processed. `orderAmount` is the total of the order. `creditCard` is the credit card number. `creditExpM` and `creditExpY` are the month and year when the credit card expires. `creditName` is the name on the credit card.
- `returnVariable`. This indicates a variable name the Custom Tag should use to report the status of the attempted payment transaction. The returned value is a ColdFusion structure that contains a number of status values.

Listing 22.10 `ProcessPayment.cfm`—Creating the `<cf_ProcessPayment>` Custom Tag

```
<!---
Filename: ProcessPayment.cfm
Created by: Nate Weiss (NMW)
Please Note Creates the <CF_ProcessPayment> Custom Tag
Purpose: Handles credit card and other transactions
-->
```

```

<!--- Tag Parameters --->
<cfparam name="ATTRIBUTES.orderID" type="numeric">
<cfparam name="ATTRIBUTES.orderAmount" type="numeric">
<cfparam name="ATTRIBUTES.creditCard" type="string">
<cfparam name="ATTRIBUTES.creditExpM" type="string">
<cfparam name="ATTRIBUTES.creditExpY" type="string">
<cfparam name="ATTRIBUTES.returnVariable" type="variableName">
<cfparam name="ATTRIBUTES.creditName" type="string">

<!--- Values to return to calling template --->
<cfset s = structNew()>
<!--- Always return IsSuccessful (Boolean) --->
<cfset s.isSuccessful = True>
<!--- Always return status of transaction --->
<cfset s.status = "success">
<!--- Return other data, as if transaction succeeded --->
<cfset s.authCode = "DummyAuthCode">
<cfset s.orderID = ATTRIBUTES.orderID>
<cfset s.orderAmount = ATTRIBUTES.orderAmount>

<!--- Return values to calling template --->
<cfset "Caller.#ATTRIBUTES.returnVariable#" = s>

```

At the top of Listing 22.10, eight `<cfparam>` tags establish the tag's attributes (all of the attributes are required). Because our tag is not doing any processing, we move immediately to the result portion of the operation. A new structure named `s` is created using `structNew()`. Next, a Boolean value called `isSuccessful` is added to the structure. Its value will always be `true`. The calling template can look at `isSuccessful` to tell whether the payment was completed successfully. Additionally, `AuthCode`, `OrderID`, and `OrderAmount` values are added to the structure. Then the whole structure is passed back to the calling template using the quoted `<cfset>` return variable syntax (explained in Chapter 23, online).

NOTE

If you adapt this Custom Tag to handle real payment processors, try to return the same value names (`IsSuccessful`, `AuthCode`, and so on) to a structure. In this way, you will build a common API to deal with payment processing in an application-agnostic way.

Processing a Complete Order

In addition to explaining how to build commerce applications, this chapter emphasizes the benefits of hiding the mechanics of complex operations within goal-oriented Custom Tag wrappers that can accomplish whole tasks on their own. Actual page templates that use these Custom Tags look very clean, since they deal only with the larger concepts at hand rather than including a lot of low-level code. That's the difference, for instance, between the two versions of the `StoreCart.cfm` template (Listings 22.4 and 22.6).

In keeping with that notion, Listing 22.11 creates another Custom Tag, called `<cf_PlaceOrder>`. This tag is in charge of handling all aspects of accepting a new order from a customer, including:

- Inserting a new record into the `MerchandiseOrders` table
- Inserting one or more detail records into the `MerchandiseOrdersItems` table (one detail record for each item ordered)
- Attempting to charge the user's credit card, using the `<cf_ProcessPayment>` Custom Tag created in the previous section (refer to Listing 22.10)
- For a successful charge, sending an order-confirmation message to the user via email, using the `<cf_SendOrderConfirmation>` Custom Tag created in Chapter 21, "Interacting with Email," online
- For an unsuccessful charge (because of an incorrect credit-card number, expiration date, or the like), ensuring that the just-inserted records from `MerchandiseOrders` and `MerchandiseOrdersItems` are not actually permanently committed to the database

Listing 22.11 `PlaceOrder.cfm`—Creating the `<cf_PlaceOrder>` Custom Tag

```
<!---
Filename: PlaceOrder.cfm (creates <CF_PlaceOrder> Custom Tag)
Created by: Nate Weiss (NMW)
Please Note Depends on <CF_ProcessPayment> and <CF_SendOrderConfirmation>
Purpose: Handles all operations related to placing a customer's order
--->

<!--- Tag Parameters --->
<cfparam name="ATTRIBUTES.processor" type="string" default="PayflowPro">
<cfparam name="ATTRIBUTES.merchList" type="string">
<cfparam name="ATTRIBUTES.quantList" type="string">
<cfparam name="ATTRIBUTES.contactID" type="numeric">
<cfparam name="ATTRIBUTES.creditCard" type="string">
<cfparam name="ATTRIBUTES.creditExpM" type="string">
<cfparam name="ATTRIBUTES.creditExpY" type="string">
<cfparam name="ATTRIBUTES.creditName" type="string">
<cfparam name="ATTRIBUTES.shipAddress" type="string">
<cfparam name="ATTRIBUTES.shipCity" type="string">
<cfparam name="ATTRIBUTES.shipCity" type="string">
<cfparam name="ATTRIBUTES.shipState" type="string">
<cfparam name="ATTRIBUTES.shipZIP" type="string">
<cfparam name="ATTRIBUTES.shipCountry" type="string">
<cfparam name="ATTRIBUTES.htmlMail" type="boolean">
<cfparam name="ATTRIBUTES.returnVariable" type="variableName">

<!--- Begin "order" database transaction here --->
<!--- Can be rolled back or committed later --->
<cftransaction action="begin">
    <!--- Insert new record into Orders table --->
    <cfquery datasource="#APPLICATION.dataSource#">
        INSERT INTO MerchandiseOrders (
            ContactID,
            OrderDate,
            ShipAddress, ShipCity,
            ShipState, ShipZip,
```

```

ShipCountry)
VALUES (
#ATTRIBUTES.contactID#,
<cfqueryparam cfsqltype="CF_SQL_TIMESTAMP"
VALUE="#dateFormat(now())# #timeFormat(now())#">,
'#ATTRIBUTES.shipAddress#', '#ATTRIBUTES.shipCity#',
'#ATTRIBUTES.shipState#', '#ATTRIBUTES.shipZip#',
'#ATTRIBUTES.shipCountry#'
)
</cfquery>

<!--- Get just-inserted OrderID from database --->
<cfquery datasource="#APPLICATION.dataSource#" name="getNew">
SELECT MAX(OrderID) AS NewID
FROM MerchandiseOrders
</cfquery>

<!--- For each item in user's shopping cart --->
<cfloop from="1" to="#listLen(ATTRIBUTES.merchList)#" index="i">
<cfset thisMerchID = listGetAt(ATTRIBUTES.merchList, i)>
<cfset thisQuant = listGetAt(ATTRIBUTES.quantList, i)>

<!--- Add the item to "OrdersItems" table --->
<cfquery datasource="#APPLICATION.dataSource#">
INSERT INTO MerchandiseOrdersItems
(OrderID, ItemID, OrderQty, ItemPrice)
SELECT
#getNew.NewID#, MerchID, #thisQuant#, MerchPrice
FROM Merchandise
WHERE MerchID = #thisMerchID#
</cfquery>
</cfloop>

<!--- Get the total of all items in user's cart --->
<cfquery datasource="#APPLICATION.dataSource#" name="getTotal">
SELECT SUM(ItemPrice * OrderQty) AS OrderTotal
FROM MerchandiseOrdersItems
WHERE OrderID = #getNew.NewID#
</cfquery>

<!--- Attempt to process the transaction --->
<cf_ProcessPayment
processor="#ATTRIBUTES.processor#"
orderID="#getNew.NewID#"
orderAmount="#getTotal.OrderTotal#"
creditCard="#ATTRIBUTES.creditCard#"
creditExpM="#ATTRIBUTES.creditExpM#"
creditExpY="#ATTRIBUTES.creditExpY#"
creditName="#ATTRIBUTES.creditName#"
returnVariable="chargeInfo">

<!--- If the order was processed successfully --->
<cfif chargeInfo.IsSuccessful>
<!--- Commit the transaction to database --->

```

```

        <cftransaction action="Commit"/>
    <cfelse>
        <!--- Rollback the Order from the Database --->
        <cftransaction action="RollBack"/>
    </cfif>
</cftransaction>

<!--- If the order was processed successfully --->
<cfif ChargeInfo.isSuccessful>
    <!--- Send Confirmation E-Mail, via Custom Tag --->
    <cf_SendOrderConfirmation
        orderID="#getNew.NewID#"
        useHTML="#ATTRIBUTES.htmlMail#"
    </cfif>

<!--- Return status values to calling template --->
<cfset "Caller.#ATTRIBUTES.returnVariable#" = chargeInfo>

```

At the top of the template is a rather large number of `<cfparam>` tags that define the various attributes for the `<cf_PlaceOrder>` Custom Tag. The `Processor`, `ReturnVariable`, and four `Credit` attributes are passed directly to `<cf_ProcessPayment>`. The `MerchList` and `QuantList` attributes specify which item is actually being ordered, in the same comma-separated format that the `CLIENT` variables use in Listing 22.4. The `contactID` and the six `ship` attributes are needed for the `MerchandiseOrders` table. The `htmlMail` attribute sends an email confirmation to the user if the payment is successful.

After the tag attributes have been defined, a large `<cftransaction>` block starts. The `<cftransaction>` tag is ColdFusion's representation of a database transaction. You saw it in action in Chapter 14, "Using Forms to Add or Change Data," and you will learn about it more formally in Chapter 41, "More About SQL and Queries," online, and Chapter 51, "Error Handling," online. The `<cftransaction>` tag tells the database to consider all queries and other database operations within the block as a single transaction, which other operations cannot interrupt.

The use of `<cftransaction>` in this template accomplishes two things. First, it makes sure that no other records are inserted between the first `INSERT` query and the `getNew` query that comes right after it. This in turn ensures that the ID number retrieved by the `getNew` query is indeed the correct one, rather than a record some other process inserted. Second, the `<cftransaction>` tag allows any database changes (inserts, deletes, or updates) to be rolled back if some kind of problem occurs. A rollback is basically the database equivalent of the Undo function in a word processor – it undoes all changes, leaving the database in the same state as at the start of the transaction. Here, the transaction is rolled back if the credit-card transaction fails (perhaps because of an incorrect credit-card number), which means that all traces of the new order will be removed from the database.

After the opening `<cftransaction>` tag, the following actions are taken:

1. A new order record is inserted into the `MerchandiseOrders` table.
2. The `getNew` query obtains the `OrderID` number for the just-inserted order record.
3. A simple `<cfloop>` tag inserts one record into the `MerchandiseOrdersItems` table for each item supplied to the `MerchList` attribute. Each record includes the new `OrderID`, the appropriate quantity from the `QuantList` attribute, and the current price for the

item (as listed in the Merchandise table). (The `INSERT/SELECT` syntax used here is explained in Chapter 30. “Managing Threads,” in Vol. 2, *Application Development*.)

4. The `getTotal` query obtains the total price of the items purchased by adding up the price of each item times its quantity.
5. The `<cf_ProcessPayment>` tag (refer to Listing 22.10) attempts to process the credit-card transaction. The structure of payment-status information is returned as a structure named `ChargeInfo`.
6. If the charge was successful, the database transaction is committed by using the `<cftransaction>` tag with `action="Commit"`. This permanently saves the inserted records in the database and ends the transaction.
7. If the charge was not successful, the database transaction is rolled back with a `<cftransaction>` tag of `action="RollBack"`. This permanently removes the inserted records from the database and ends the transaction.
8. If the charge was successful, a confirmation email message is sent to the user, using the `<cf_SendOrderConfirmation>` Custom Tag from Chapter 21, online. Because this tag performs database interactions of its own, it should sit outside the `<cftransaction>` block.
9. Finally, the `chargeInfo` structure returned by `<cf_PlaceOrder>` is passed back to the calling template so it can understand whether the order was placed successfully.

Creating the Checkout Page

Now that you’ve created the `<cf_PlaceOrder>` Custom Tag, actually creating the Checkout page for Orange Whip Studios’ visitors is simple. Listing 22.12 provides the code for the `StoreCheckout.cfm` page, which users can access via the Checkout link at the top of each page in the online store or by clicking the Checkout button on the `StoreCart.cfm` page (refer to Figure 22.2).

Listing 22.12 `StoreCheckout.cfm`—Allowing the User to Complete the Online Transaction

```
<!---
Filename: StoreCheckout.cfm
Created by: Nate Weiss (NMW)
Purpose: Provides final Checkout/Payment page
Please Note Depends on <CF_PlaceOrder> and StoreCheckoutForm.cfm
-->

<!--- Show header images, etc., for Online Store --->
<cfinclude template="StoreHeader.cfm">

<!--- Get current cart contents, as a query object --->
<cfset getCart = SESSION.myShoppingCart.List()>

<!--- Stop here if user's cart is empty --->
<cfif getCart.recordCount eq 0>
    There is nothing in your cart.
    <cfabort>
</cfif>
```

```

<!-- If user is not logged in, force them to now -->
<!--
<cfif not isDefined("SESSION.auth.isLoggedIn")>
  <cfinclude template="LoginForm.cfm">
  <cfabort>
</cfif>
-->
<!-- If user is attempting to place order -->
<cfif isDefined("FORM.isPlacingOrder")>

  <cftry>
    <!-- Attempt to process the transaction -->
    <!-- Change to PayflowPro to use VeriSign -->
    <cf_PlaceOrder
      Processor="JustTesting"
      contactID="1"
      merchList="#valueList(getCart.MerchID)#"
      quantList="#valueList(getCart.Quantity)#"
      creditCard="#FORM.creditCard#"
      creditExpM="#FORM.creditExpM#"
      creditExpY="#FORM.creditExpY#"
      creditName="#FORM.creditName#"
      shipAddress="#FORM.shipAddress#"
      shipState="#FORM.shipState#"
      shipCity="#FORM.shipCity#"
      shipZIP="#FORM.shipZIP#"
      shipCountry="#FORM.shipCountry#"
      htmlMail="#FORM.htmlMail#"
      returnVariable="orderInfo">

    <!-- If any exceptions in the "ows.MerchOrder" family are thrown... -->
    <cfcatch type="ows.MerchOrder">
      <p>Unfortunately, we are not able to process your order at the moment.<br>
      Please try again later. We apologize for the inconvenience.<br>
    <cfabort>
    </cfcatch>
  </cftry>

  <!-- If the order was processed successfully -->
  <cfif orderInfo.isSuccessful>

    <!-- Empty user's shopping cart -->
    <cfset SESSION.myShoppingCart.Empty()>

    <!-- Display Success Message -->
    <cfoutput>
      <h2>Thanks For Your Order</h2>
      <p><b>Your Order Has Been Placed.</b><br>
      Your order number is: #orderInfo.orderID#<br>
      Your credit card has been charged:
      #lsCurrencyFormat(orderInfo.OrderAmount)#<br>
      <p>A confirmation is being Emailed to you.<br>

```

```
</cfoutput>

<!-- Stop here. -->
<cfabort>
<cfelse>
  <!-- Display "Error" message -->
  <font color="red">
    <strong>Your credit card could not be processed.</strong><br>
    Please verify the credit card number, expiration date, and
    name on the card.<br>
  </font>

  <!-- Show debug info if viewing page on server -->
  <cftrace inline="True" var="OrderInfo">
</cfif>
</cfif>

<!-- Show Checkout Form (Ship Address/Credit Card) -->
<cfinclude template="StoreCheckoutForm.cfm">
```

First, the standard page header for the online store is displayed with the `<cfinclude>` tag at the top of Listing 22.12. Next, the `List` method of the `ShoppingCart` CFC gets the current contents of the user's cart. If `getCart.RecordCount` is 0, the user's cart must be empty, so the template displays a short "Your cart is empty" message and stops further processing. Of course, if you want to use the client variable-based `<cf_ShoppingCart>` Custom Tag instead of the session variable-based `ShoppingCart` CFC, you can easily do so by changing the first `<cfset>` line (near the top of Listing 22.12) to this:

```
<!-- Get current cart contents, as a query object -->
<cf_ShoppingCart
  action="List"
  returnVariable="GetCart">
```

At the bottom of the template, a `<cfinclude>` tag includes the form shown in Figure 22.4, which asks the user for shipping and credit-card information. The form is self-submitting, so when the user clicks the Place Order Now button, the code in Listing 22.12 is executed again. This is when the large `<cfif>` block kicks in.

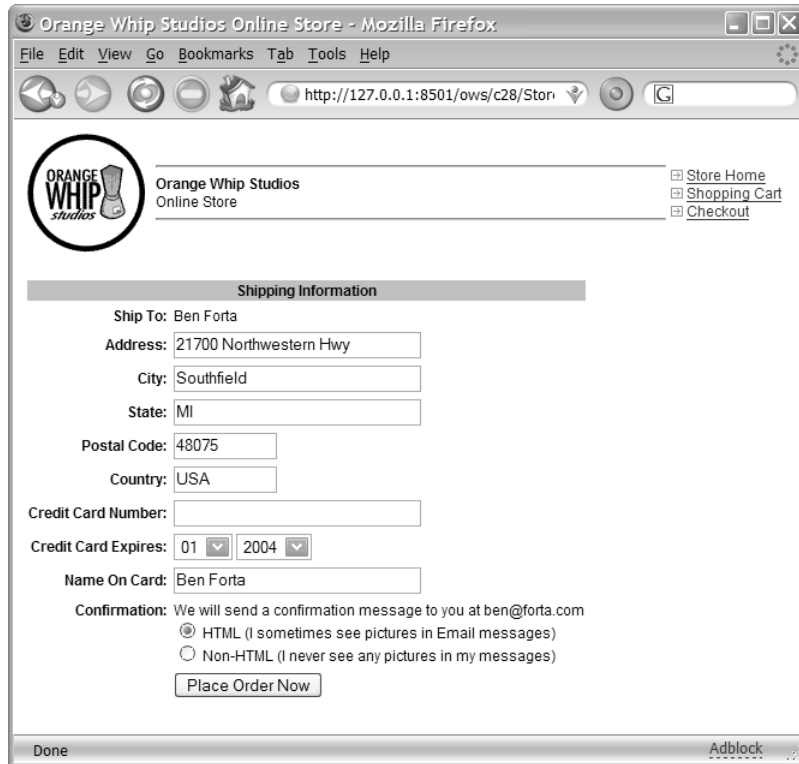


Figure 22.4

Users provide credit-card information on the Checkout page.

Within the `<cfif>` block, the `<cf_PlaceOrder>` tag attempts to complete the user's order. If all goes well, the order will be committed to the database, the confirmation email message will be sent, the `orderInfo.isSuccessful` value will be `True`, and the new order's ID number will be returned as `orderInfo.orderID`.

If the order is actually successful, the user's cart is emptied using a final call to `<cf_ShoppingCart>`, and a "Thanks for your order" message appears. If not, an error message appears and the checkout form (see Listing 22.13) is displayed again. Also, the `<cftrace>` tag displays additional diagnostic information if the debugging options are on in the ColdFusion Administrator.

Listing 22.13 is the `StoreCheckoutForm.cfm` template included via the `<cfinclude>` tag at the bottom of Listing 22.12. This template uses `<cfform>` and `<cfinput>` to display a Web-based form with some simple data validation (such as the `validate="creditcard"` attribute for the `creditCard` field). As a convenience to the user, it prefills the shipping-address fields based on the address information currently in the `Contacts` table. The resulting form was shown in Figure 22.4.

Listing 22.13 `storeCheckoutForm.cfm`—Collecting Shipping and Card Information from the User

```
<!---
Filename: StoreCheckoutForm.cfm
Created by: Nate Weiss (NMW)
Please Note Included by StoreCheckout.cfm
Purpose: Displays a simple checkout form
-->
```

```
<!-- Used to pre-fill user's choice of HTML or Plain email -->
<cfparam name="FORM.htmlMail" type="string" default="Yes">

<cfoutput>
<cfform action="#CGI.script_name#" method="post" preservedata="Yes">
<cfinput type="hidden" name="isPlacingOrder" value="Yes">

<table border="0" cellspacing="4">
<tr>
  <th colspan="2" bgcolor="silver">Shipping Information</th>
</tr>

<tr>
  <th align="right">Ship To:</th>
  <td>
    Your Name
  </td>
</tr>
<tr>
  <th align="right">Address:</th>
  <td>
    <cfinput name="shipAddress" size="30"
      required="yes" value=""
      message="Please don't leave the Address blank!">
  </td>
</tr>
<tr>
  <th align="right">City:</th>
  <td>
    <cfinput name="shipCity" size="30" required="yes" value=""
      message="Please don't leave the City blank!">
  </td>
</tr>
<tr>
  <th align="right">State:</th>
  <td>
    <cfinput name="shipState" size="30" required="yes" value=""
      message="Please don't leave the State blank!">
  </td>
</tr>
<tr>
  <th align="right">Postal Code:</th>
  <td>
    <cfinput name="shipZIP" size="10" required="yes" value=""
      message="Please don't leave the ZIP blank!">
  </td>
</tr>
<tr>
  <th align="right">Country:</th>
  <td>
    <cfinput name="shipCountry" size="10" required="Yes"
      value=""
      message="Please don't leave the Country blank!">
  </td>
</tr>
</table>
</cfoutput>
```

```

        </td>
    </tr>
    <tr>
        <th align="right">Credit Card Number:</th>
        <td>
            <cfinput name="creditCard" size="30" required="yes" validate="creditcard"
                message="You must provide a credit card number.">
        </td>
    </tr>
    <tr>
        <th align="right">Credit Card Expires:</th>
        <td>
            <cfselect name="creditExpM">
                <cfloop from="1" to="12" index="i">
                    <option value="#i#">#numberFormat(i, "00")#
                </cfloop>
            </cfselect>
            <cfselect name="creditExpY">
                <cfloop from="#year(now())#" to="#val(year(now())+10)#" index="i">
                    <option value="#i#">#i#
                </cfloop>
            </cfselect>
        </td>
    </tr>
    <tr>
        <th align="right">Name On Card:</th>
        <td>
            <cfinput name="creditName" size="30" required="Yes"
                value=""
                message="You must provide the Name on the Credit Card.">
        </td>
    </tr>
    <tr valign="baseline">
        <th align="right">Confirmation:</th>
        <td>
            We will send a confirmation message to you at foo@foo.com<br>
            <cfinput type="radio" name="htmlMail" value="Yes"
                checked="#form.htmlMail#">
            HTML (I sometimes see pictures in Email messages)<br>
            <cfinput type="radio" name="htmlMail" value="No"
                checked="#not form.htmlmail#">
            Non-HTML (I never see any pictures in my messages)<br>
        </td>
    </tr>
    <tr>
        <td>&nbsp;</td>
        <td>
            <cfinput type="submit" name="submit" value="Place Order Now">
        </td>
    </tr>
</table>

</cfform>
</cfoutput>

```

The online store for Orange Whip Studios is now complete. Users can add items to their shopping carts, adjust quantities, remove items, and check out. And all the code has been abstracted in a reasonable and maintainable fashion, thanks to ColdFusion's wonderful Custom Tag feature.

Other Commerce-Related Tasks

For a real-world online commerce site, your application pages will need to take care of some other tasks. This book doesn't cover these tasks explicitly, but they should be well within your reach now that you have been introduced to the basic concepts and have walked through the construction of the main shopping experience for your users.

Order Tracking

Most users who place orders online will expect some mechanism to allow them to check the status of their orders, usually online. Depending on what your site sells, simply making an email address available for status inquiries might be enough. However, some type of secure order-tracking page will satisfy more people and cut down on support costs.

The `OrderHistory.cfm` templates from Chapter 21, online, are a great start. You would just need to ensure that the user has a way to see the `ShipDate` as well as the `OrderDate`, and you might even add a button the user could use to cancel an order.

Order Fulfillment

This chapter hasn't even touched on actually fulfilling an order after it has been placed. How this works will depend entirely on the company for which you are building your commerce application. For instance, you might provide an Order Queue page for employees in Orange Whip Studios' shipping department. This Order Queue page could be similar to the `OrderHistory.cfm` templates (see the previous section, "Order Tracking"), except that it would show all pending orders, not just the ones for a particular user.

Or you might decide that an email message should be sent to the shipping department using the `<cfmail>` tag. If your company's needs are simple, it might be sufficient to simply `bcc` the shipping department on the email message sent by the `<cf_SendOrderConfirmation>` Custom Tag from Chapter 27, "Creating Advanced ColdFusion Components," in Vol. 2, *Application Development*.

Cancellations, Returns, and Refunds

If you are taking money from your visitors, there is always the possibility that you will need to give some of it back at some point. You must ensure that your company has some means for dealing with returns and refunds, including the ability to credit back any charges made to the user's credit card.

You might build your own Web-based interface for refunds and cancellations, using the `<cf_VerisignPayflowPro>` tag (which does support items such as refunds), if that is the

payment-processing mechanism you are using. Many payment-processing services provide their own Web-based systems, which your company's accounting or customer service departments can use to handle such special cases.

Inventory Tracking

The examples in this chapter assume that all items in the `Merchandise` table are always available for sale. Orange Whip Studios might not need to be concerned about its merchandise ever selling out, but your company probably does. At a minimum, you should probably add an `InStock` Boolean field to the `Merchandise` table and ensure that users can't add items to their carts unless the `InStock` field is 1. More sophisticated applications might call for maintaining a `NumberOnHand` field for each item, which gets decremented each time an item is ordered and incremented each time new shipments come in from the supplier.

Reporting

Once a company is doing business online, it will need to know how much business its online store is generating. You should supply your company's executives with some type of reporting functionality that shows purchase trends over time, which products are profitable, and so on. To build such reports, you could use Crystal Reports with the `<cfreport>` tag or build your own reporting templates, perhaps illustrating the data visually using ColdFusion's dynamic graphing and charting capabilities (see Chapter 16, "Graphing, Printing, and Reporting").

CHAPTER 23

Securing Your Applications

At this point, you have learned how to create interactive, data-driven pages for your users and have started to see how your applications can really come alive using the various persistent scopes (particularly client and session variables) provided by Macromedia ColdFusion's Web application framework. Now is a good time to learn how to lock down your application pages so they require a user name and password and show only the right information to the right people.

Options for Securing Your Application

This section briefly outlines the topics to consider if you need to secure access to your ColdFusion templates. You can use more than one of these options at the same time if you want.

- SSL encryption
- HTTP basic authentication
- Application-based security
- ColdFusion's `<cflogin>` framework
- ColdFusion Sandbox Security
- Operating System Security

SSL Encryption

Most of today's Web servers allow you to make a connection between the browser and the server more secure by using encryption. After encryption has been enabled, your ColdFusion templates and related files become available at URLs that begin with `https://` instead of `http://`. The HTML code your templates generate is scrambled on its way out of the Web server. Provided that everything has been set up correctly, browsers

can unscramble the HTML and use it normally. The framework that makes all of this possible is called the Secure Sockets Layer (SSL).

Browsers generally display a small key or lock icon in their status bar to indicate that a page is encrypted. You probably have encountered many such sites yourself, especially on pages where you are asked to provide a credit card number.

This topic isn't discussed in detail here because encryption is enabled at the Web-server level and doesn't affect the ColdFusion Application Server directly. You don't need to do anything special in your ColdFusion templates for it to work properly. The encryption and decryption are taken care of by your Web server and each user's browser.

You might want to look into turning on your Web server's encryption options for sections of your applications that need to display or collect valuable pieces of information. For instance, most users hesitate to enter a credit card number on a page that isn't secure, so you should think about using encryption during any type of checkout process in your applications.

TIP

If you are working on a company intranet project, you might consider enabling SSL for the entire application, especially if employees will access it from outside your local network.

The steps you take to enable encryption differ depending on which Web server software you are using (Apache, Netscape/iPlanet, Microsoft IIS, and so on). You will need to consult your Web server's documentation for details. Along the way, you will learn a bit about public and private keys, and you will probably need to buy an annual SSL certificate from a company such as VeriSign. VeriSign's Web site is also a good place to look if you want to find out more about SSL and HTTPS technology in general. Visit the company at <http://www.verisign.com>.

TIP

If you want your code to be capable of detecting whether a page is being accessed with an `https://` URL, you can use one of the variables in the CGI scope to make this determination. The variables might have slightly different names from server to server, but they generally start with `HTTPS`. For instance, on a Microsoft IIS server, the value of `CGI.HTTPS` is `on` or `off`, depending on whether the page is being accessed in an encrypted context. Another way to perform the test is by looking at the value of `CGI.SERVER_PORT`; under most circumstances, it will hold a value of `443` if encryption is being used, and a value of `80` if not. We recommend that you turn on the Show Variables debugging option in the ColdFusion Administrator to see which HTTPS-related variables are made available by your Web server software.

HTTP Basic Authentication

Nearly all Web servers provide support for something called HTTP basic authentication. *Basic authentication* is a method for password-protecting your Web documents and images and usually is used to protect static files, such as straight HTML files. However, you can certainly use basic authentication to password-protect your ColdFusion templates. Users will be prompted for their user names and passwords via a dialog box presented by the browser, as shown in Figure 23.1. You won't have control over the look or wording of the dialog box, which varies from browser to browser.



Figure 23.1

Basic authentication prompts the user to log in using a standard dialog box.

Basic authentication isn't the focus of this chapter. However, it is a quick, easy way to put a password on a particular folder, individual files, or an entire Web site. It is usually best for situations in which you want to give the same type of access to everyone who has a password. With basic authentication, you don't need to write any ColdFusion code to control which users are allowed to see what. Depending on the Web server software you are using, the user names and passwords for each user might be kept in a text file, an LDAP server, an ODBC database, or some type of proprietary format.

To find out how to enable basic authentication, see your Web server's documentation.

NOTE

One of the shortcomings of HTTP Basic Authentication is that the user's entries for username and password are sent to the server with every page request, and the password isn't scrambled strongly. Therefore you may want to consider enabling SSL Encryption (discussed in the previous section) when using HTTP Basic Authentication, which will cause all communications between server and browser to be scrambled.

TIP

When basic authentication is used, you should be able to find out which user name was provided by examining either the `#CGI.AUTH_USER#` variable or the `#CGI.REMOTE_USER#` variable. The variable name depends on the Web server software you are using.

NOTE

Microsoft's Web servers and browsers extend the idea of basic authentication by providing a proprietary option called Integrated Windows Authentication (also referred to as NTLM or Challenge/Response Authentication), which enables people to access a Web server using their Windows user names and passwords. For purposes of this section, consider Windows Authentication to be in the same general category as basic authentication. That is, it isn't covered specifically in this book and is enabled at the Web-server level.

Application-Based Security

The term *application-based security* is used here to cover any situation in which you give users an ordinary Web-based form with which to log in. Most often, this means using the same HTML form techniques you already know to present that form to the user, then using a database query to verify that the user name and password they typed was valid.

This method of security gives you the most control over the user experience, such as what the login page looks like, when it is presented, how long users remain logged in, and what they have access to. In other words, by creating a homegrown security or login process, you get to make it work however you need it to. The downside, of course, is that

you must do a bit of extra work to figure out exactly what you need and how to get it done. That's what a large portion of this chapter is all about.

ColdFusion's `<cflogin>` Framework

ColdFusion provides a set of tags and functions for creating login pages and generally enforcing rules about which of your application's pages can be used by whom. The tags are `<cflogin>`, `<cfloginuser>`, and `<cflogout>`. For basic web applications, the framework provides the same kind of user experience as Session-Based security. The main purpose of the framework is to make it easier to secure more advanced applications that make use of ColdFusion Components and Flash Remoting.

NOTE

For details and examples, see the "Using ColdFusion's `<cflogin>` Framework," below.

ColdFusion Sandbox Security

ColdFusion 8 provides a set of features called Sandbox Security (also called Resource Security), which takes the place of the Advanced Security system present in previous versions of ColdFusion. Advanced Security attempted to provide a unified system that could be used internally by CFML developers in their applications, and also by hosting companies needing to be able to turn off certain parts of ColdFusion for individual developers. It included the `<cfauthenticate>` and `<cfimpersonate>` tags, and a number of specialized CFML functions. Many people complained that this system, while very powerful, was too complex.

Aimed mostly at Internet Service Providers and hosting companies, the Sandbox Security system in ColdFusion is simpler, while still remaining flexible and powerful. It is now possible to use the ColdFusion Administrator to designate which data sources, CFML tags, and other server resources can be used by which applications. For instance, if a single ColdFusion server is being used by several different developers, they can each feel confident that the others won't be able to access the data in their data sources. Also, if an ISP doesn't want developers to be able to create or read files on the server itself via the `<cffile>` tag (discussed in Chapter 70, "Interacting with the Operating System," in *ColdFusion 8 Web Application Construction Kit, Volume 3: Advanced Application Development*), it's easy for the ISP to disallow the use of `<cffile>` while still allowing developers to use the rest of CFML's functionality.

NOTE

*Because it is designed primarily for ISPs and hosting companies that administer ColdFusion servers, Sandbox Security isn't covered in detail in this book. If you have an interest in what the Sandbox Security system can allow and disallow, please refer to the ColdFusion documentation, the online help for the Sandbox Security page in the ColdFusion Administrator, or our companion volume, *ColdFusion 8 Web Application Construction Kit, Volume 2: Application Development* (Macromedia Press, ISBN 0-321-29269-3).*

Operating System Security

Included in ColdFusion is a `<cfntauthenticate>` tag. This allows you to integrate your ColdFusion code directly with the operating system's security (as long as your operating

system is Windows). The tag, `<cfntauthenticate>`, allows you to not only check a username and password against a Windows domain. You can optionally also return the list of groups a user belongs to.

Using ColdFusion to Control Access

The rest of this chapter discusses how to build your own form-based security mechanism. In general, putting such a mechanism in place requires three basic steps:

- Deciding which pages or information should be password-protected
- Creating a login page and verifying the user name and password
- Restricting access to pages or information based on who the user is, either using a homegrown Session-Based mechanism, or with the `<cflogin>` framework

Deciding What to Protect

First, you have to decide exactly what it is you are trying to protect with your security measures. Of course, this step doesn't involve writing any code, but we strongly recommend that you think about this as thoroughly as possible. You should spend some time just working through what type of security measures your applications need and how users will gain access.

Be sure you have answers to these questions:

- Does the whole application need to be secured, or just a portion of it? For company intranets, you usually want to secure the whole application. For Internet sites available to the general public, you usually want to secure only certain sections (Members Only or Registered Users areas, for instance).
- What granularity of access do you need? Some applications need to lock only certain people out of particular folders or pages. Others need to lock people out at a more precise, data-aware level. For instance, if you are creating some type of Manage Your Account page, you aren't trying to keep a registered user out of the page. Instead, you need to ensure that the users see and change only their own account information.
- When should the user be asked for their user name and password? When they first enter your application, or only when they try to get something that requires it? The former might make the security seem more cohesive to the user, whereas the latter might be more user friendly.

We also recommend that you think about how passwords will be maintained, rather than what they will protect:

- Should user names and passwords become invalid after a period of time? For instance, if a user has purchased a 30-day membership to your site, what happens on the 31st day?

- Does the user need the option of voluntarily changing their password? What about their user name?
- Should some users be able to log in only from certain IP addresses? Or during certain times of the day, or days of the week?
- How will user names and passwords be managed? Do you need to implement some form of user groups, such as users in an operating system? Do you need to be able to grant rights to view certain items on a group level? What about on an individual user level?

The answers to these questions will help you create whatever database tables or other validation mechanics will be necessary to implement the security policies you have envisioned. You will learn where and when to refer to any such custom tables as you work through the code examples in this chapter.

Using Session Variables for Authentication

An effective and straightforward method for handling the mechanics of user logins is outlined in the following section. Basically, the strategy is to turn on ColdFusion's session-management features, which you learned about in Chapter 20, "Working with Sessions," and use session variables to track whether each user has logged in. There are many ways to go about this, but it can be as simple as setting a single variable in the `SESSION` scope after a user logs in.

NOTE

Before you can use the `SESSION` scope in your applications, you need to enable it using the `Application.cfc` file. See Chapter 20 for details.

Checking and Maintaining Login Status

For instance, assume for the moment that the user has just filled out a user name/password form (more on that later), and you have verified that the user name and password are correct. You could then use a line such as the following to remember that the user is logged in:

```
<cfset SESSION.isLoggedIn = "Yes">
```

As you learned in the last chapter, the `isLoggedIn` variable is tracked for the rest of the user's visit (until their session times out). From this point forward, if you wanted to ensure that the user was logged in before they were shown something, all you would need to do would be to check for the presence of the variable:

```
<cfif not isDefined("SESSION.isLoggedIn")>  
    Sorry, you don't have permission to look at that.  
</cfif>
```

And with that, you have modest security. Clearly, this isn't final code yet, but that really is the basic idea. A user won't be able to get past the second snippet unless their session

has already encountered the first. The rest of the examples in this chapter are just expanded variations on these two code snippets.

So, all you have to do is put these two lines in the correct places. The first line must be wrapped within whatever code validates a user's password (probably by checking in some type of database table), and the second line must be put on whatever pages you need to protect.

Restricting Access to Your Application

Assume for the moment that you want to require your users to log in as soon as they enter your application. You could put a login form on your application's front page or home page, but what if a user doesn't go through that page for whatever reason? For instance, if they use a bookmark or type the URL for some other page, they would bypass your login screen. You must figure out a way to ensure that the user gets prompted for a password on the first page request for each session, regardless of which page they are actually asking for.

A great solution is to use the special `Application.cfc` file set aside by ColdFusion's Web application framework, which you learned about in Chapter 19, "Introducing the Web Application Framework." You will recall that if you create a template called `Application.cfc`, it automatically is included before each page request. This means you could put some code in `Application.cfc` to see whether the `SESSION` scope is holding an `isLoggedIn` value, as discussed previously. If it's not holding a value, the user must be presented with a login form. If it is holding a value, the user has already logged in during the current session.

With that in mind, take a look at the `Application.cfc` file shown in Listing 23.1. Make sure to save this listing as `Application.cfc`, not `Application1.cfc`.

Listing 23.1 `Application1.cfc`—Sending a User to a Login Page

```
<!---
Filename: Application.cfc
Created by: Raymond Camden (ray@camdenfamily.com)
Please Note Executes for every page request
-->

<cfcomponent output="false">

    <!--- Name the application. --->
    <cfset this.name="OrangeWhipSite">
    <!--- Turn on session management. --->
    <cfset this.sessionManagement=true>

    <cffunction name="onApplicationStart" output="false" returnType="void">

        <!--- Any variables set here can be used by all our pages --->
        <cfset APPLICATION.dataSource = "ows">
        <cfset APPLICATION.companyName = "Orange Whip Studios">

    </cffunction>

    <cffunction name="onRequestStart" output="false" returnType="void">
```

```
<!-- If user isn't logged in, force them to now -->
<cfif not isDefined("SESSION.auth.isLoggedIn")>
  <!-- If the user is now submitting "Login" form, -->
  <!-- Include "Login Check" code to validate user -->
  <cfif isDefined("FORM.UserLogin")>
    <cfinclude template="loginCheck.cfm">
  </cfif>

  <cfinclude template="loginForm.cfm">
  <cfabort>
</cfif>

</cffunction>

</cfcomponent>
```

First, the application is named using the `This` scope. Then `sessionManagement` is turned on. Don't forget that sessions are *not* enabled by default. The first method in the `Application.cfc` file, `onApplicationStart`, will run when the application starts up, or when the first user hits the site. Two application variables are set. These will be used later on in other code listings. The `onRequestStart` method will run before every request. An `isDefined()` test is used to check whether the `isLoggedIn` value is present. If it's not, a `<cfinclude>` tag is used to include the template called `LoginForm.cfm`, which presents a login screen to the user. Note that a `<cfabort>` tag is placed directly after the `<cfinclude>` so that nothing further is presented.

The net effect is that all pages in your application have now been locked down and will never appear until you create code that sets the `SESSION.auth.isLoggedIn` value.

NOTE

Soon, you will see how the `Auth` structure can be used to hold other values relevant to the user's login status. If you don't need to track any additional information along with the login status, you could use a variable named `SESSION.IsLoggedIn` instead of `SESSION.Auth.IsLoggedIn`. However, it's not much extra work to add the `Auth` structure, and it gives you some extra flexibility.

Creating a Login Page

The next step is to create a login page, where the user can enter their user name and password. The code in Listing 23.1 is a simple example. This code still doesn't actually do anything when submitted, but it's helpful to see that most login pages are built with ordinary `<form>` or `<cfform>` code. Nearly all login pages are some variation of this skeleton.

Figure 23.2 shows what the form will look like to a user.

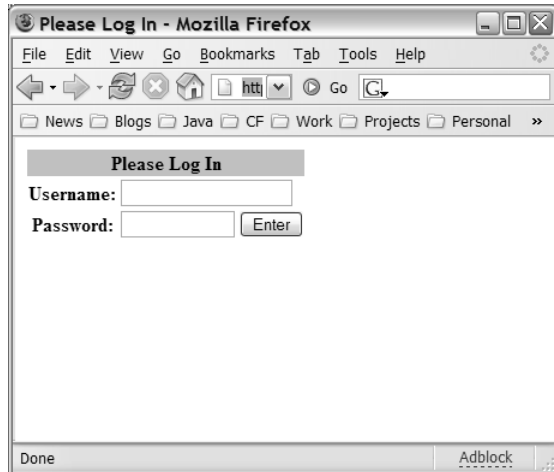


Figure 23.2

Users are forced to log in before they can access sensitive information in this application.

NOTE

Use `type="password"` wherever you ask your users to type a password, as shown in Listing 23.2. That way, as the user types, their password will be masked so that someone looking over their shoulder can't see their password.

Listing 23.2 `LoginForm.cfm`—A Basic Login Page

```
<!---
Filename: LoginForm.cfm
Created by: Nate Weiss (NMW)
Purpose: Presented whenever a user has not logged in yet
Please Note Included by Application.cfc
--->

<!--- If the user is now submitting "Login" form, --->
<!--- Include "Login Check" code to validate user --->
<cfif isDefined("FORM.UserLogin")>
  <cfinclude template="LoginCheck.cfm">
</cfif>

<html>
<head>
  <title>Please Log In</title>
</head>

<!--- Place cursor in "User Name" field when page loads--->
<body onLoad="document.LoginForm.userLogin.focus();">

<!--- Start our Login Form --->
<cfform action="#CGI.script_name#?#CGI.query_string#" name="LoginForm"
method="post">
  <!--- Make the UserLogin and UserPassword fields required --->
  <input type="hidden" name="userLogin_required">
  <input type="hidden" name="userPassword_required">
```

```
<!-- Use an HTML table for simple formatting -->
<table border="0">
<tr><th colspan="2" bgcolor="silver">Please Log In</th></tr>
<tr>
<th>Username:</th>
<td>

<!-- Text field for "User Name" -->
<cfinput
type="text"
name="userLogin"
size="20"
value=""
maxlength="100"
required="Yes"
message="Please type your Username first.">

</td>
</tr><tr>
<th>Password:</th>
<td>

<!-- Text field for Password -->
<cfinput
type="password"
name="userPassword"
size="12"
value=""
maxlength="100"
required="Yes"
message="Please type your Password first.">

<!-- Submit Button that reads "Enter" -->
<input type="Submit" value="Enter">

</td>
</tr>
</table>

</cfform>

</body>
</html>
```

NOTE

In general, users won't be visiting `LoginForm.cfm` directly. Instead, the code in Listing 23.2 is included by the `<cfif>` test performed in the `Application.cfc` page (Listing 23.1) the first time the user accesses some other page in the application (such as the `OrderHistory.cfm` template shown later in Listing 23.4).

Please note that this form's `action` attribute is set to `#CGI.script_name#`. The special `CGI.script_name` variable always holds the relative URL to the currently executing ColdFusion template. So, for example, if the user is being presented with the login form after requesting a template called `HomePage.cfm`, this form will rerequest that same page

when submitted. In other words, this form always submits back to the URL of the page on which it is appearing. Along with the current script, we also append the current query string, using `CGI.query_string`. This ensures that if the person requested `HomePage.cfm?id=5`, the portion after the question mark, known as the query string, will also be included when we submit the form.

TIP

Using `CGI.script_name` and `CGI.query_string` can come in handy any time your code needs to be capable of reloading or resubmitting the currently executing template.

When the form is actually submitted, the `FORM.userLogin` value will exist, indicating that the user has typed a user name and password that should be checked for accuracy. As a result, the `<cfinclude>` tag fires, and includes the password-validation code in the `LoginCheck.cfm` template (see Listing 23.3).

The Text and Password fields on this form use the `required` and `message` client-side validation attributes provided by `<cfinput>` and `<cfform>`. The two hidden fields add server-side validation. See Chapter 13, “Form Data Validation,” if you need to review these form field validation techniques.

NOTE

This template’s `<body>` tag has JavaScript code in its `onLoad` attribute, which causes the cursor to be placed in the `userLogin` field when the page loads. You must consult a different reference for a full discussion of JavaScript, but you can use this same basic technique to cause any form element to have focus when a page first loads.

TIP

JavaScript is case sensitive, so the `onLoad` code must be capitalized correctly; otherwise, scripting-error messages will pop up in the browser. Of course, you can just leave out the `onLoad` code altogether if you want.

Verifying the Login Name and Password

Listing 23.3 provides simple code for your `LoginCheck.cfm` template. This is the template that will be included when the user attempts to gain access by submitting the login form from Listing 23.2.

The most important line in this template is the `<cfset>` line that sets the `SESSION.auth.isLoggedIn` variable to `Yes`. After this value is set for the session, the `isDefined()` test in the `Application.cfc` file (refer to Listing 23.1) will succeed and the user will be able to view pages normally.

Listing 23.3 `LoginCheck.cfm`—Granting Access

```
<!---
  Filename: LoginCheck.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Validates a user's password entries
  Please Note Included by LoginForm.cfm
-->

<!--- Make sure we have Login name and Password --->
<cfparam name="FORM.userLogin" type="string">
```

```
<cfparam name="FORM.userPassword" type="string">

<!--- Find record with this Username/Password --->
<!--- If no rows returned, password not valid --->
<cfquery name="getUser" datasource="#APPLICATION.dataSource#">
    SELECT ContactID, FirstName
    FROM Contacts
    WHERE UserLogin = '#FORM.UserLogin#'
    AND UserPassword = '#FORM.UserPassword#'
</cfquery>

<!--- If the username and password are correct --->
<cfif getUser.recordCount eq 1>
    <!--- Remember user's logged-in status, plus --->
    <!--- ContactID and First Name, in structure --->
    <cfset SESSION.auth = structNew()>
    <cfset SESSION.auth.isLoggedIn = "Yes">
    <cfset SESSION.auth.contactID = getUser.contactID>
    <cfset SESSION.auth.firstName = getUser.firstName>

    <!--- Now that user is logged in, send them --->
    <!--- to whatever page makes sense to start --->
    <cflocation url="#CGI.script_name#?#CGI.query_string#">
</cfif>
```

TIP

The query in this template can be adapted or replaced with any type of database or lookup procedure you need. For instance, rather than looking in a database table, you could query an LDAP server to get the user's first name.

NOTE

For more information about LDAP, consult ColdFusion 8 Web Application Construction Kit: Advanced Application Development (ISBN 0-321-51547-3), in this series.

First, the two `<cfparam>` tags ensure that the login name and password are indeed available as form fields, which they should be unless a user has somehow been directed to this page in error. Next, a simple `<cfquery>` tag attempts to retrieve a record from the `Contacts` table where the `UserLogin` and `UserPassword` columns match the user name and password that were entered in the login form. If this query returns a record, the user has, by definition, entered a valid user name and password and thus should be considered logged in.

Assume for the moment that the user name and password are correct. The value of `getUser.recordCount` is therefore 1, so the code inside the `<cfif>` block executes. A new structure called `auth` is created in the `SESSION` scope, and three values are placed within the new structure. The most important of the three is the `isLoggedIn` value, which is used here basically in the same way that was outlined in the original code snippets near the beginning of this chapter.

The user's unique ID number (their `contactID`) is also placed in the `SESSION.auth` structure, as is their first name. The idea here is to populate the `SESSION.auth` structure with whatever information is pertinent to the fact that the user has indeed been authenticated. Therefore, any little bits of information that might be helpful to have later in the user's session can be saved in the `auth` structure now.

TIP

By keeping the `SESSION.auth.FirstName` value, for instance, you will be able to display the user's first name on any page, which will give your application a friendly, personalized feel. And, by keeping the `SESSION.auth.contactID` value, you will be able to run queries against the database based on the user's authenticated ID number.

Finally, the `<cflocation>` tag is used to redirect the user to the current value of `CGI.script_name` and `CGI.query_string`. Because `CGI.script_name` and `CGI.query_string` were also used for the action of the login form, this value will still reflect the page the user was originally looking for, before the login form appeared. The browser will respond by rerequesting the original page with the same query string. This time, the `SESSION.auth.isLoggedIn` test in `Application.cfc` (refer to Listing 23.1) won't `<cfinclude>` the login form, and the user will thus be allowed to see the content they originally were looking for.

NOTE

The underlying assumption here is that no two users can have the same `UserLogin` and `UserPassword`. You must ensure that this rule is enforced in your application. For instance, when a user first chooses (or is assigned) their user name and password, there needs to be a check in place to ensure that nobody else already has them.

Personalizing Based on Login

After Listings 23.1–23.3 are in place, the `SESSION.auth` structure is guaranteed to exist for all your application's pages. What's more, the user's unique ID and first name will be available as `SESSION.auth.contactID` and `SESSION.auth.firstName`, respectively. This makes providing users with personalized pages, such as Manage My Account or My Order History, easy.

Listing 23.4 shows a template called `OrderHistory.cfm`, which lets a user review the merchandise orders they have placed in the past. Because the authenticated `contactID` is readily available, doing this in a reasonably secure fashion is easy. In most respects, this is just a data-display template, the likes of which you learned about in Chapter 10, "Creating Data-Driven Pages." The only new concept here is the notion of using authenticated identification information from the `SESSION` scope (in this case, the `contactID`).

Listing 23.4 `OrderHistory.cfm`—Personalizing Based on Login

```
<!---
  Filename: OrderHistory.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Displays a user's order history
-->

<!--- Retrieve user's orders, based on ContactID --->
<cfquery name="getOrders" datasource="#APPLICATION.dataSource#">
  SELECT OrderID, OrderDate,
  (SELECT Count(*)
  FROM MerchandiseOrdersItems oi
  WHERE oi.OrderID = o.OrderID) AS ItemCount
  FROM MerchandiseOrders o
```

```

WHERE ContactID = #SESSION.auth.contactID#
ORDER BY OrderDate DESC
</cfquery>

<html>
<head>
  <title>Your Order History</title>
</head>
<body>
<!-- Personalized message at top of page-->
<cfoutput>
  <h2>Your Order History</h2>
  <p><strong>Welcome back, #SESSION.auth.firstName#!</strong><br>
  You have placed <strong>#getOrders.recordCount#</strong>
  orders with us to date.</p>
</cfoutput>

<!-- Display orders in a simple HTML table -->
<table border="1" width="300" cellpadding="5" cellspacing="2">
  <!-- Column headers -->
  <tr>
    <th>Date Ordered</th>
    <th>Items</th>
  </tr>

  <!-- Display each order as a table row -->
  <cfoutput query="getOrders">
    <tr>
      <td>
        <a href="OrderHistory.cfm?OrderID=#OrderID#">
          #dateFormat(orderDate, "mmmm d, yyyy")#
        </a>
      </td>
      <td>
        <strong>#itemCount#</strong>
      </td>
    </tr>
  </cfoutput>
</table>

</body>
</html>

```

First, a fairly ordinary `<cfquery>` tag is used to retrieve information about the orders the user has placed. Because the user's authenticated `contactID` is being used in the `WHERE` clause, you can be certain that you will be retrieving the order information appropriate only for this user.

Next, a personalized message is displayed to the user, including their first name. Then the order records are displayed using an ordinary `<cfoutput query>` block. The order records are displayed in a simple tabular format using simple HTML table formatting. Figure 23.3 shows what the results will look like for the end user.

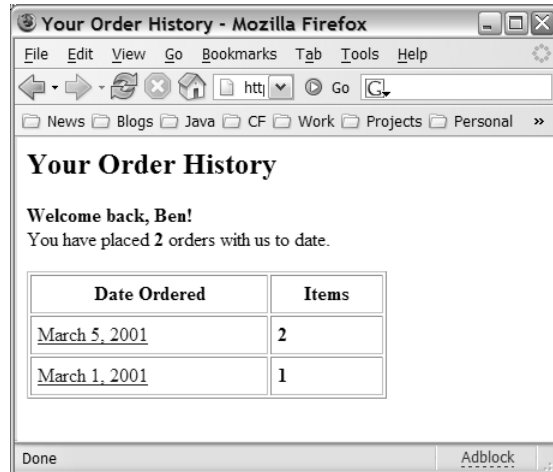


Figure 23.3

After a user's identification information is authenticated, providing a personalized experience is easy.

Being Careful with Passed Parameters

When you are dealing with sensitive information, such as account or purchase histories, you need to be more careful when passing parameters from page to page. It's easy to let yourself feel that your work is done after you force your users to log in. Of course, forcing them to log in is an important step, but your code still needs to check things internally before it exposes sensitive data.

Recognizing the Problem

Here's a scenario that illustrates a potential vulnerability. After putting together the `OrderHistory.cfm` template shown in Listing 23.4, you realize that people will need to be able to see the details of each order, such as the individual items purchased. You decide to allow the user to click in each order's Order Date column to see the details of that order. You decide, sensibly, to turn each order's date into a link that passes the desired order's ID number as a URL parameter.

So, you decide to change this:

```
#dateFormat(orderDate, "mmm d, yyyy")#
```

to this:

```
<a href="OrderHistory.cfm?OrderID=#OrderID#">
  #dateFormat(OrderDate, "mmm d, yyyy")#
</a>
```

This is fine. When the user clicks the link, the same template is executed – this time with the desired order number available as `URL.OrderID`. You just need to add an `isDefined()` check to see whether the URL parameter exists, and if so, run a second query to obtain the detail records (item name, price, and quantity) for the desired order. After a bit of thought, you come up with the following:

```
<cfif isDefined("URL.orderID")>
  <cfquery name="getDetail" datasource="#APPLICATION.dataSource#">
    SELECT m.MerchName, oi.ItemPrice, oi.OrderQty
    FROM Merchandise m, MerchandiseOrdersItems oi
    WHERE m.MerchID = oi.ItemID
    AND oi.OrderID = #URL.orderID#
```

```
</cfquery>
</cfif>
```

The problem with this code is that it doesn't ensure that the order number passed in the URL indeed belongs to the user. After the user notices that the order number is being passed in the URL, they might try to play around with the passed parameters just to, ahem, see what happens. And, indeed, if the user changes the `?orderID=5` part of the URL to, say, `?orderID=10`, they will be able to see the details of some other person's order. Depending on what type of application you are building, this kind of vulnerability could be a huge problem.

Checking Passed Parameters

The problem is relatively easy to solve. You just need to ensure that, whenever you retrieve sensitive information based on a URL or FORM parameter, you somehow verify that the parameter is one the user has the right to request. In this case, you must ensure that the `URL.orderID` value is associated with the user's ID number, `SESSION.auth.contactID`.

In this application, the easiest policy to enforce is probably ensuring that each query involves the `SESSION.auth.contactID` value somewhere in its `WHERE` clause. Therefore, to turn the unsafe query shown previously into a safe one, you would add another subquery or inner join to the query, so the `Orders` table is directly involved. After the `Orders` table is involved, the query can include a check against its `ContactID` column.

A safe version of the snippet shown previously would be the following, which adds a subquery at the end to ensure the `OrderID` is a legitimate one for the current user:

```
<cfif isDefined("URL.orderID")>
  <cfquery name="getDetail" datasource="#APPLICATION.dataSource#">
    SELECT m.MerchName, oi.ItemPrice, oi.OrderQty
    FROM Merchandise m, MerchandiseOrdersItems oi
    WHERE m.MerchID = oi.ItemID
    AND oi.OrderID = #URL.orderID#
    AND oi.OrderID IN
      (SELECT o.OrderID FROM MerchandiseOrders o
      WHERE o.ContactID = #SESSION.auth.contactID#)
  </cfquery>
</cfif>
```

Another way to phrase the query, using an additional join, would be

```
<cfif isDefined("URL.orderID")>
  <cfquery name="getDetail" datasource="#APPLICATION.dataSource#">
    SELECT
      m.MerchName, m.MerchPrice,
      oi.ItemPrice, oi.OrderQty
    FROM
      (Merchandise m INNER JOIN
      MerchandiseOrdersItems oi
      ON m.MerchID = oi.ItemID) INNER JOIN
      MerchandiseOrders o
      ON o.OrderID = oi.OrderID
    WHERE o.ContactID = #SESSION.auth.contactID#
    AND oi.OrderID = #URL.orderID#
  </cfquery>
</cfif>
```

With either of these snippets, it doesn't matter if the user alters the `orderID` in the URL. Because the `contactID` is now part of the query's `WHERE` criteria, it will return zero records if the requested `orderID` isn't consistent with the session's authenticated `ContactID`. Thus, the user will never be able to view any orders but their own.

Putting It Together and Getting Interactive

The `OrderHistory2.cfm` template shown in Listing 23.5 builds on the previous version (refer to Listing 23.4) by adding the ability for the user to view details about each order. The code is a bit longer, but there really aren't any big surprises here. The main additions display the detail information. Some formatting has also been applied to make the template look nicer when displayed to the user.

Listing 23.5 `OrderHistory2.cfm`—Providing Details About Orders

```
<!---
  Filename: OrderHistory2.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Displays a user's order history
-->

<!--- Retrieve user's orders, based on ContactID --->
<cfquery name="getOrders" datasource="#APPLICATION.dataSource#">
  SELECT OrderID, OrderDate,
  (SELECT Count(*)
  FROM MerchandiseOrdersItems oi
  WHERE oi.OrderID = o.OrderID) AS itemCount
  FROM MerchandiseOrders o
  WHERE ContactID = #SESSION.auth.contactID#
  ORDER BY OrderDate DESC
</cfquery>

<!--- Determine if a numeric OrderID was passed in URL --->
<cfset showDetail = isDefined("URL.orderID") and isNumeric(URL.orderID)>

<!--- If an OrderID was passed, get details for the order --->
<!--- Query must check against ContactID for security --->
<cfif showDetail>
  <cfquery name="getDetail" datasource="#APPLICATION.dataSource#">
    SELECT m.MerchName, oi.ItemPrice, oi.OrderQty
    FROM Merchandise m, MerchandiseOrdersItems oi
    WHERE m.MerchID = oi.ItemID
    AND oi.OrderID = #URL.orderID#
    AND oi.OrderID IN
    (SELECT o.OrderID FROM MerchandiseOrders o
    WHERE o.ContactID = #SESSION.auth.contactID#)
  </cfquery>

  <!--- If no Detail records, don't show detail --->
  <!--- User may be trying to "hack" URL parameters --->
  <cfif getDetail.recordCount eq 0>
    <cfset showDetail = False>
  </cfif>
</cfif>
```

```

<html>
<head>
  <title>Your Order History</title>

  <!-- Apply some simple CSS style formatting -->
  <style type="text/css">
  BODY { font-family:sans-serif;font-size:12px;color:navy}
  H2 { font-size:20px}
  TH { font-family:sans-serif;font-size:12px;color:white;
  background:MediumBlue;text-align:left}
  TD { font-family:sans-serif;font-size:12px}
  </style>
</head>
<body>

<!-- Personalized message at top of page-->
<cfoutput>
  <h2>Your Order History</h2>
  <p><strong>Welcome back, #SESSION.auth.firstName#!</strong><br>
  You have placed <strong>#getOrders.recordCount#</strong>
  orders with us to date.</p>
</cfoutput>

<!-- Display orders in a simple HTML table -->
<table border="1" width="300" cellpadding="5" cellspacing="2">
  <!-- Column headers -->
  <tr>
    <th>Date Ordered</th>
    <th>Items</th>
  </tr>

  <!-- Display each order as a table row -->
  <cfoutput query="getOrders">
    <!-- Determine whether to show details for this order -->
    <!-- Show Down arrow if expanded, otherwise Right -->
    <cfset isExpanded = showDetail and (getOrders.OrderID eq URL.orderID)>
    <cfset arrowIcon = iif(isExpanded, "'ArrowDown.gif'", "'ArrowRight.gif'")>

    <tr>
      <td>
        <!-- Link to show order details, with arrow icon -->
        <a href="OrderHistory2.cfm?OrderID=#orderID#">
          
          #dateFormat(orderDate, "mmmm d, yyyy")#
        </a>
      </td>
      <td>
        <strong>#itemCount#</strong>
      </td>
    </tr>
  </cfoutput>

```

```

<!-- Show details for this order, if appropriate -->
<cfif isExpanded>
<cfset orderTotal = 0>
<tr>
<td colspan="2">

<!-- Show details within nested table -->
<table width="100%" cellpadding="0" border="0">
<!-- Nested table's column headers -->
<tr>
<th>Item</th><th>Qty</th><th>Price</th>
</tr>

<!-- Show each ordered item as a table row -->
<cfloop query="getDetail">
<cfset orderTotal = orderTotal + itemPrice>
<tr>
<td>#merchName#</td>
<td>#orderQty#</td>
<td>#dollarFormat(itemPrice)#</td>
</tr>
</cfloop>

<!-- Last row in nested table for total -->
<tr>
<td colspan="2"><b>Total:</b></td>
<td><strong>#dollarFormat(orderTotal)#</strong></td>
</tr>
</table>
</td>
</tr>
</cfif>
</cfoutput>
</table>

</body>
</html>

```

The first `<cfquery>` is unchanged from Listing 23.4. Next, a Boolean variable called `showDetail` is created. Its value is `true` if a number is passed as `URL.orderID`. In that case, the second `<cfquery>` (which was shown in the code snippet before the listing) executes and returns only detail records for the session's `contactID`. The `<cfif>` test after the query resets `showDetail` to `false` if the second query fails to return any records. The remainder of the code can rely on `showDetail` being `true` only if a legitimate `orderID` was passed in the URL.

Two `<cfset>` tags have been added at the top of the main `<cfoutput>` block to determine whether the `orderID` of the order currently being output is the same as the `orderID` passed in the URL. If so, the `isExpanded` variable is set to `true`. Additionally, an `arrowIcon` variable is created, which is used to display an open or closed icon to indicate whether each order record is expanded. If the current order is the one the user has asked for details about, `isExpanded` is `true` and `arrowIcon` is set to show the `ArrowDown.gif` image. If not, the `ArrowRight.gif` image is shown instead. The appropriate arrow is displayed using an `` tag a few lines later.

At the end of the template is a large `<cfif>` block, which causes order details to be shown if `isExpanded` is `true`. If so, an additional row is added to the main HTML table, with a `colspan` of 2 so that the new row has just one cell spanning the `Date Ordered` and `Items` columns. Within the new cell, another, nested `<table>` is added, which shows one row for each record in the `getDetail` query via a `<cfloop>` block. As each detail row is output, the `orderTotal` variable is incremented by the price of each item. Therefore, by the time the `<cfloop>` is finished, `orderTotal` will indeed contain the total amount the customer paid. The total is displayed as the last row of the nested table.

The result is a pleasant-looking interface in which the user can quickly see the details for each order. At first, only the order dates and item counts are displayed (as shown previously in Figure 23.3), with an arrow pointing to the right to indicate that the order isn't expanded. If the user clicks the arrow or the order date, the page is reloaded, now with the arrow pointing down and the order details nested under the date. Figure 23.4 shows the results.

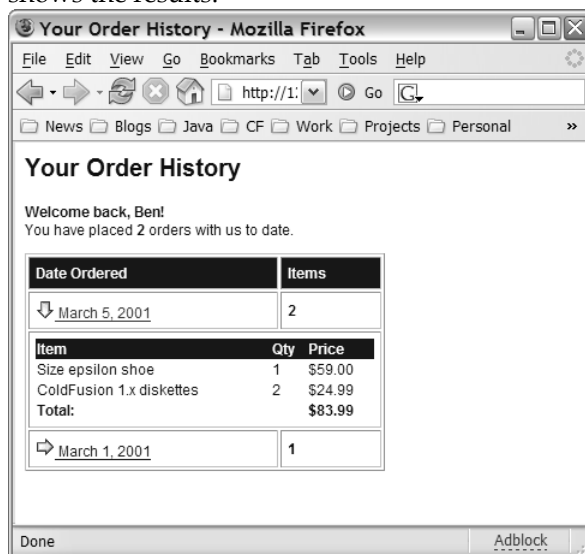


Figure 23.4

With a little bit of caution, you can safely provide an interactive interface for sensitive information.

NOTE

A `<style>` block is included in the `<head>` section of this listing to apply some CSS-based formatting. All type on the page will be shown in a blue, sans serif font (usually Arial or Helvetica, depending on the operating system), except for `<th>` cells, which will be shown with white type on a blue background. Consult a CSS reference for details.

Other Scenarios

This chapter has outlined a practical way to force a user to log in to your application and to show them only the appropriate information. Of course, your actual needs are likely to vary somewhat from what has been discussed here. Here are a couple of other scenarios that are commonly encountered, with suggestions about how to tackle them.

Delaying the Login Until Necessary

The examples in this chapter assume that the entire application needs to be secured and that each user should be forced to log in when they first visit any of your application's pages. If, however, only a few pages need to be secured here and there, you might want

to delay the login step until the user actually requests something of a sensitive nature. For instance, it might be that the user doesn't need to log in unless they try to visit pages such as Manage My Account or My Order History. For all other pages, no security measures are necessary.

To get this effect, you could move the `isDefined("SESSION.auth.isLoggedIn")` check from `Application.cfc` (refer to Listing 23.1) to a new template called `ForceUserLogin.cfm`. Then, at the top of any page that requires a password, you could put a `<cfinclude>` tag with a `template="ForceLogin.cfm"` attribute. This is a simple but effective way to enforce application security only where it's needed.

Implementing Different Access Levels

This chapter has focused on the problems of forcing users to log in and using the login information to safely provide sensitive information. Once logged in, each user is treated equally in this chapter's examples. Each user simply has the right to see their own data.

If you are building a complex application that needs to allow certain users to do more than others, you might need to create some type of access right or permission or user level. This need is most commonly encountered in intranet applications, in which executives need to be able to view report pages that most employees cannot see, or in which only certain high-level managers can review the performance files of other employees.

Maybe all you need is to add another column somewhere to tell you which type of user each person is. For the Orange Whip Studios example, this might mean adding a new Yes/No column called `IsPrivileged` to the `Contacts` table. The idea is that if this column is set to `true`, the user should get access to certain special things that others don't have. Then, in `LoginCheck.cfm` (refer to Listing 23.3), select this new column along with the `ContactID` and `FirstName` columns in the `<cfquery>`, and add a line that saves the `isPrivileged` value in the `SESSION.auth` structure, such as this:

```
<cfset SESSION.auth.isPrivileged = getUser.IsPrivileged>
```

TIP

For an intranet application, you might use a column called `IsSupervisor` or `IsAdministrator` instead of `IsPrivileged`.

Now, whenever you need to determine whether something that requires special privileges should be shown, you could use a simple `<cfif>` test, such as

```
<cfif SESSION.auth.isPrivileged>
  <a href="SalesData.cfm">Sacred Sales Data</a>
</cfif>
```

Or, instead of a simple Yes/No column, you might have a numeric column named `UserLevel` and save it in the `SESSION.auth` structure in `LoginCheck.cfm`. This would give you an easy way to set up various access levels, where 1 might be used for normal employees, 2 for supervisors, 3 for managers, 4 for executives, and 100 for developers. So, if only security level 3 and above should be able to view a page, you could use something similar to this:

```
<cfif SESSION.auth.userLevel lt 3>
  Access denied!
<cfabort>
</cfif>
```

Access Rights, Users, and Groups

Depending on the application, you may need something more sophisticated than what is suggested in the previous code snippets. If so, you might consider creating database tables to represent some notion of access rights, users, and groups. A typical implementation would establish a many-to-many relationship between users and groups, so that a user can be in more than one group, and each group have any number of users. In addition, a one-to-many relationship generally would exist between groups and access rights. Tables with names such as `GroupsUsers` and `GroupsRights` would maintain the relationships.

After the tables were in place, you could adapt the code examples in this chapter to enforce the rules established by the tables. For instance, assuming that you had a table called `Rights`, which had columns named `RightID` and `RightName`, you might put a query similar to the following after the `GetUser` query in `LoginCheck.cfm` (refer to Listing 23.3):

```
<!-- Find what rights user has from group membership -->
<cfquery name="getRights" datasource="#APPLICATION.dataSource#">
  SELECT r.RightName
  FROM Rights r, GroupsContacts gu, GroupsRights gr
  WHERE r.RightID = gr.RightID
  AND gu.GroupID = gr.GroupID
  AND gu.ContactID = #SESSION.auth.contactID#
</cfquery>

<!-- Save comma-separated list of rights in SESSION -->
<cfset SESSION.auth.rightsList = valueList(getRights.RightName)>
```

Now, `SESSION.auth.rightsList` would be a list of string values that represented the rights the user should be granted. The user is being granted these rights because the rights have been granted to the groups they are in.

After the previous code is in place, code such as the following could be used to find out whether a particular user is allowed to do something, based on the rights they have actually been granted:

```
<cfif listFind(SESSION.auth.rightsList, "SalesAdmin">
  <a href="SalesData.cfm">Sacred Sales Data</a>
</cfif>

or

<cfif not listFind(SESSION.auth.rightsList, "SellCompany">
  Access denied.
  <cfabort>
</cfif>
```

NOTE

The `<cflogin>` framework discussed in the next section provides the `IsUserInRole()` function, which is a similar way to implement security based on groups or rights. In particular, the `OrderHistory4.cfm` template shown later in Listing 23.9 uses this function to provide different levels of access to different users.

Using ColdFusion's `<cflogin>` Framework

So far, this chapter has presented a simple session-based method for securing and personalizing an application, built on user names and passwords. Using the preceding

examples as a foundation, you can easily create your own custom security framework. For the purposes of this section, let's call this type of security a *homegrown* security framework.

With ColdFusion, you also have the option of using a security framework that ships as part of the CFML language itself. This new framework includes a few tags; most important is the `<cflogin>` tag, which you will learn about in this section. The ColdFusion documentation refers to the new framework as *user security*. For clarity, let's call it the `<cflogin>` framework.

Because the `<cflogin>` framework boasts tight integration with the rest of the CFML language, you may want to use it to provide security for some applications. That said, you may want to stick to a homegrown approach for flexibility. In either case, you will probably end up writing approximately the same amount of code.

Table 23.1 shows some of the advantages and disadvantages of the `<cflogin>` framework versus a homegrown framework.

Table 23.1 Comparing the `<cflogin>` framework with homegrown approaches

Strategy	Advantages	Disadvantages
Homegrown Framework		Not immediately recognizable by other developers. Not automatically recognized and enforced by CFCs.
<code><cflogin></code> Framework		Tightly integrated with CFCs: You just tell the component which user groups (roles) may access each method. While open and flexible, it may not suit your particular needs. Still requires approximately the same amount of careful coding as the homegrown approach.

NOTE

One key advantage of the `<cflogin>` framework is its integration with the ColdFusion Components (CFC) system, which you will learn about soon.

- ▶ *CFCs are covered in Chapter 26, "Building Reusable Components," in Vol. 2, Application Development.*

Tags and Functions Provided by the `<cflogin>` Framework

The `<cflogin>` framework currently includes eight CFML tags and functions, as shown in Table 23.2. You will see how these tags work together in a moment. For now, all you need is a quick sense of the tags and functions involved.

Table 23.2 <cflogin> and Related Tags and Functions

Tag or Function	Purpose
<cflogin>	This tag is always used in a pair. Between the opening and closing tags, you place whatever code is needed to determine whether the user should be able to proceed. In most situations, this means checking the validity of the user name and password being presented. You will often place this tag in <code>Application.cfc</code> . Code between the opening and closing tags won't be run if the user is logged in.
<cfloginuser>	Once the user has provided a valid user name and password, you use the <cfloginuser> tag to tell ColdFusion that the user should now be considered logged in. This tag always appears within a pair of <cflogin> tags. Like <cflogin>, this tag will typically get called by <code>Application.cfc</code> .
<code>getAuthUser()</code>	Once the user has logged in, you can use this function to retrieve or display the user's name, ID, or other identifying information.
<code>isUserInRole()</code>	If you want different users to have different rights or privileges, you can use this function to determine whether they should be allowed to access a particular page or piece of information.
<code>isUserInAnyRole()</code>	Allows you to check users to see if they belong to any of a set of roles.
<code>getUserRoles()</code>	Returns a list of all users roles.
<code>isUserLoggedIn()</code>	Returns <code>true</code> if the user is logged in.
<cflogout>	If you want to provide a way for users to explicitly log out, use the <cflogout> tag. Otherwise, users will be logged out after 30 minutes of inactivity. This value can be modified using the <code>idleTimeout</code> value of the <cflogin> tag. The <cflogin> framework can be tied to a user's <code>SESSION</code> scope so that both expire at the same time. This option is enabled by using the <code>loginStorage</code> setting in the <code>This</code> scope in your <code>Application.cfc</code> file. This is the preferred option when using the <cflogin> framework.

Using <cflogin> and <cfloginuser>

The first thing you need to do is add the <cflogin> and <cfloginuser> tags to whatever parts of your application you want to protect. To keep things nice and clean, the examples in this chapter keep the <cflogin> and <cfloginuser> code in a separate ColdFusion template called `ForceUserLogin.cfm` (Listing 23.6).

Once a template like this is in place, you just need to include it via <cfinclude> from any ColdFusion page that you want to password-protect. To protect an entire application, just place the <cfinclude> into your `Application.cfc` template. Since it automatically executes for every page request, the <cflogin> and related tags will be automatically protecting all of your application's pages.

Listing 23.6 `ForceUserLogin.cfm`—Using the <cflogin> Framework

```

<!---
Filename: ForceUserLogin.cfm
Created by: Nate Weiss (NMW)
Purpose: Requires each user to log in
Please Note Included by Application.cfc
-->

```

```

<!-- Force the user to log in -->
<!-- *** This code only executes if the user has not logged in yet! *** -->
<!-- Once the user is logged in via <cfloginuser>, this code is skipped -->
<cflogin>

<!-- If the user hasn't gotten the login form yet, display it -->
<cfif not (isDefined("FORM.userLogin") and isDefined("FORM.userPassword"))>
    <cfinclude template="UserLoginForm.cfm">
    <cfabort>

<!-- Otherwise, the user is submitting the login form -->
<!-- This code decides whether the username and password are valid -->
<cfelse>

    <!-- Find record with this Username/Password -->
    <!-- If no rows returned, password not valid -->
    <cfquery name="getUser" datasource="#APPLICATION.dataSource#">
    SELECT ContactID, FirstName, rTrim(UserRoleName) as UserRoleName
    FROM Contacts LEFT OUTER JOIN UserRoles
    ON Contacts.UserRoleID = UserRoles.UserRoleID
    WHERE UserLogin = '#FORM.UserLogin#'
    AND UserPassword = '#FORM.UserPassword#'
    </cfquery>

    <!-- If the username and password are correct... -->
    <cfif getUser.recordCount eq 1>
        <!-- Tell ColdFusion to consider the user "logged in" -->
        <!-- For the NAME attribute, we will provide the user's -->
        <!-- ContactID number and first name, separated by commas -->
        <!-- Later, we can access the NAME value via GetAuthUser() -->
        <cfloginuser
        name="#getUser.ContactID#, #getUser.FirstName#"
        password="#FORM.userPassword#"
        roles="#getUser.userRoleName#">

    <!-- Otherwise, re-prompt for a valid username and password -->
    <cfelse>
        Sorry, that username and password are not recognized.
        Please try again.
        <cfinclude template="UserLoginForm.cfm">
        <cfabort>
    </cfif>

</cfif>

</cflogin>

```

NOTE

This template is very similar conceptually to the homegrown `LoginCheck.cfm` template discussed earlier (see Listing 23.3). The logic still centers around the `getUser` query, which checks the user name and password that have been provided. It just remembers each user's login status using `<cflogin>` and `<cfloginuser>`, rather than the homegrown `SESSION.auth` structure.

NOTE

Whenever Listing 23.6 needs to display a login form to the user, it does so by including `UserLoginForm.cfm` with a `<cfinclude>` tag. This login form is nearly identical to the one shown earlier in Listing 23.2; the main difference is that it takes advantage of two user-defined functions to preserve any `URL` and `FORM` variables that might be provided before the login form is encountered. The actual code for this version of the login form is included on the CD-ROM for this book; it is also discussed in Chapter 24, "Building User-Defined Functions," in Vol. 2, Application Development.

The first thing to note is the pair of `<cflogin>` tags. In most cases, the `<cflogin>` tag can be used with no attributes to simply declare that a login is necessary (see the notes before Table 23.3 for information about `<cflogin>`'s optional attributes). It is up to the code inside the `<cflogin>` tag to do the work of collecting a user name and password, or forcing the user to log in. If they have already logged in, the code within the `<cflogin>` block is skipped completely. The `<cflogin>` code executes only if the user has yet to log in.

At the top of the `<cflogin>` block, a simple `<cfif>` test sees whether form fields named `userLogin` and `userPassword` have been provided. In other words, has the user been presented with a login form? If not, the login form is presented via a `<cfinclude>` tag. Note that the `<cfabort>` tag is needed to make sure that execution stops once the form is displayed.

Therefore, the code beginning with `<cfelse>` executes only if the user has not yet successfully logged in and is currently attempting to log in with the login form. First, a simple `<cfquery>` tag is used to validate the user name and password. This is almost the same as the query used in Listing 23.3. The only difference is that this query also retrieves the name of the user's security role from the `UserRoles` table.

NOTE

For this example application, the security role is conceptually similar to that of a user group in an operating system; you can use the role to determine which users have access to what. For instance, those in the `Admin` security role might be able to do things other users cannot. You'll see how this works in the `OrderHistory4.cfm` template, later in this chapter.

TIP

You don't have to use database queries to validate users' credentials and retrieve their information. For instance, if you are storing this type of user information in an LDAP data store (such as one of the iPlanet server products, or Microsoft's Windows 2000, XP, or .NET systems), you could use the `<cfldap>` tag instead of `<cfquery>` to validate the user's security data. The ColdFusion documentation includes an example of using `<cfldap>` together with `<cflogin>` and `<cfloginuser>` in such a way.

If the user name and password are valid, the `<cfloginuser>` tag tells ColdFusion that the user should now be considered logged in. If not, the login form is redisplayed. Table 23.3 shows the attributes `<cfloginuser>` supports.

Take a look at how `<cfloginuser>` is used in Listing 23.6. The purpose of the `name` attribute is to pass a value to ColdFusion that identifies the user in some way. The actual value of `name` can be whatever you want; ColdFusion retains the value for as long as the user is logged in. At any time, you can use the `getAuthUser()` function to retrieve the value you passed to `name`. Because the various pages in the application need to have the user's `ContactID` and first name handy, they are passed to `name` as a simple comma-separated list.

NOTE

Behind the scenes, the `<cflogin>` framework sets a cookie on the browser machine to remember that a user has been logged in. The cookie's name will start with `CFAUTHORIZATION`. The `<cflogin>` tag supports two optional attributes that control how that cookie is set. The `cookieDomain` attribute allows you to share the authorization cookie between servers in the same domain; it works like the `domain` attribute of the `<cfcookie>` tag (as discussed in Chapter 20). The `applicationToken` attribute can be used to share a user's login state among several applications; normally, this attribute defaults to the current application's name (which means that all pages that use the same name in their `This` scope from the `Application.cfc` file will share login information), but if you provide a different value, all `<cflogin>` blocks that use the same `applicationToken` will share the login information (creating a "single sign on" effect).

TIP

The `<cflogin>` tag supports an optional `idleTimeout` attribute, which you can use to control how long a user remains logged in between page requests. The default value is 1800 seconds (30 minutes). If you want users to be considered logged out after just 5 minutes of inactivity, use `idleTimeout="300"`.

Table 23.3 `<cfloginuser>` Tag Attributes

Attribute	Purpose
name	Required. Some kind of identifying string that should be remembered for as long as the user remains logged in. Whatever value you provide here will be available later via the <code>getAuthUser()</code> function. It is important to note that the value you provide here does not actually need to be the user's name. It can contain any simple string information that you would like to be able to refer to later in the user's session, like an ID number.
password	Required. The password that the user logs in with. This is used internally by ColdFusion to track the user's login status.
roles	Optional. The role or roles that you want the user to be considered a part of. Later, you will be able to use the <code>isUserInRole()</code> function to find out whether the user is a member of a particular role. You can use <code>getUserRoles()</code> to return all of the user's roles.

The other values passed to `<cfloginuser>` are straightforward. The password that the user entered is supplied to the `password` attribute, and the name of the role to which the user is assigned is passed as the `roles`.

NOTE

It is up to you to ensure that each possible combination of `name` and `password` is unique (this is almost always the case anyway; an application should never allow two users to have the same user name and password). The best practice would be to make sure that some kind of unique identifier (such as the `ContactID`) be used as part of the `name`, just to make sure that ColdFusion understands how to distinguish your users from one another.

NOTE

The way the Orange Whip Studios example database is designed, each user will always have only one role (or associated group). That is, they can be assigned to the `Admin` role or the `Marketing` role, but not both. If your application needed to let users be in multiple roles or groups, you would likely have an additional database table with a row for each combination of `ContactID` and `UserRoleID`. Then, before your `<cfloginuser>` tag, you might have a query called `getUserRoles` that retrieved the appropriate list of role names from the database. You would then use the `valueList()` function to supply this query's records to the `roles` attribute as a comma-separated list; for instance:

```
roles="#valueList(getUserRoles.userRoleName)#".
```

Now that the code to force the user to log in is in place, it just needs to be pressed into service via `<cfinclude>`. You could either place the `<cfinclude>` at the top of each ColdFusion page that you wanted to protect, or you can just place it in `Application.cfc` to protect all your application's templates. Listing 23.7 shows such an `Application.cfc` file.

Listing 23.7 `Application2.cfc`—Logging with `<cflogin>` Framework

```
<!---
Filename: Application.cfc
Created by: Raymond Camden (ray@camdenfamily.com)
Please Note Executes for every page request
-->

<cfcomponent output="false">

    <!--- Name the application. --->
    <cfset this.name="OrangeWhipSite">
    <!--- Turn on session management. --->
    <cfset this.sessionManagement=true>

    <cffunction name="onApplicationStart" output="false" returnType="void">

        <!--- Any variables set here can be used by all our pages --->
        <cfset APPLICATION.dataSource = "ows">
        <cfset APPLICATION.companyName = "Orange Whip Studios">

    </cffunction>

    <cffunction name="onRequestStart" output="false" returnType="void">

        <cfinclude template="ForceUserLogin.cfm">

    </cffunction>

</cfcomponent>
```

Using `getAuthUser()` in Your Application Pages

Once you save Listing 23.7 as a file named `Application.cfc`, users will be forced to log in whenever they visit any of the pages in that folder (or its subfolders). However, the order history pages that were created above will no longer work, since they rely on the `SESSION.auth` variables populated by the homegrown login framework. A few changes must be made to allow the order history pages with the `<cflogin>` framework. Basically, this just means referring to the value returned by `getAuthUser()` to get the user's ID and first name, rather than using `SESSION.auth.contactID` and `SESSION.auth.firstName`. Listing 23.8 shows the new version of the order history template.

Listing 23.8 `OrderHistory3.cfm`—Using `getAuthUser()`

```
<!---
  Filename: OrderHistory3.cfm
  Created by: Nate Weiss (NMW)
  Purpose: Displays a user's order history
-->

<html>
<head>
  <title>Order History</title>

  <!--- Apply some simple CSS style formatting --->
  <style type="text/css">
  BODY { font-family:sans-serif;font-size:12px;color:navy}
  H2 { font-size:20px}
  TH { font-family:sans-serif;font-size:12px;color:white;
  background:MediumBlue;text-align:left}
  TD { font-family:sans-serif;font-size:12px}
  </style>
</head>
<body>

  <!--- getAuthUser() returns whatever was supplied to the name --->
  <!--- attribute of the <cflogin> tag when the user logged in. --->
  <!--- We provided user's ID and first name, separated by --->
  <!--- commas; we can use list functions to get them back. --->
  <cfset contactID = listFirst(getAuthUser())>
  <cfset contactName = listRest(getAuthUser())>

  <!--- Personalized message at top of page--->
  <cfoutput>
    <h2>YourOrder History</h2>
    <p><strong>Welcome back, #contactName#!</strong><br>
  </cfoutput>

  <!--- Retrieve user's orders, based on ContactID --->
  <cfquery name="getOrders" datasource="#APPLICATION.dataSource#">
```

```

SELECT OrderID, OrderDate,
(SELECT Count(*)
FROM MerchandiseOrdersItems oi
WHERE oi.OrderID = o.OrderID) AS ItemCount
FROM MerchandiseOrders o
WHERE ContactID = #contactID#
ORDER BY OrderDate DESC
</cfquery>

<!-- Determine if a numeric OrderID was passed in URL -->
<cfset showDetail = isDefined("URL.orderID") and isNumeric(URL.orderID)>

<!-- If an OrderID was passed, get details for the order -->
<!-- Query must check against ContactID for security -->
<cfif showDetail>
<cfquery name="getDetail" datasource="#APPLICATION.dataSource#">
SELECT m.MerchName, oi.ItemPrice, oi.OrderQty
FROM Merchandise m, MerchandiseOrdersItems oi
WHERE m.MerchID = oi.ItemID
AND oi.OrderID = #URL.orderID#
AND oi.OrderID IN
(SELECT o.OrderID FROM MerchandiseOrders o
WHERE o.ContactID = #contactID#)
</cfquery>

<!-- If no Detail records, don't show detail -->
<!-- User may be trying to "hack" URL parameters -->
<cfif getDetail.recordCount eq 0>
<cfset showDetail = False>
</cfif>
</cfif>
<cfif getOrders.recordCount eq 0>
<p>No orders placed to date.<br>
<cfelse>
<cfoutput>
<p>Orders placed to date:
<strong>#getOrders.recordCount#</strong><br>
</cfoutput>

<!-- Display orders in a simple HTML table -->
<table border="1" width="300" cellpadding="5" cellspacing="2">
<!-- Column headers -->
<tr>
<th>Date Ordered</th>
<th>Items</th>
</tr>

<!-- Display each order as a table row -->
<cfoutput query="getOrders">
<!-- Determine whether to show details for this order -->
<!-- Show Down arrow if expanded, otherwise Right -->
<cfset isExpanded = showDetail and (getOrders.OrderID eq URL.orderID)>
<cfset arrowIcon = iif(isExpanded, "\ArrowDown.gif", "\ArrowRight.gif")>

```

```

<tr>
<td>
<!-- Link to show order details, with arrow icon -->
<a href="OrderHistory3.cfm?OrderID=#orderID#">

#dateFormat(orderDate, "mmmm d, yyyy")#
</a>
</td>
<td>
<strong>#ItemCount#</strong>
</td>
</tr>

<!-- Show details for this order, if appropriate -->
<cfif isExpanded>
<cfset orderTotal = 0>
<tr>
<td colspan="2">

<!-- Show details within nested table -->
<table width="100%" cellpadding="0" border="0">
<!-- Nested table's column headers -->
<tr>
<th>Item</th><th>Qty</th><th>Price</th>
</tr>

<!-- Show each ordered item as a table row -->
<cfloop query="getDetail">
<cfset orderTotal = orderTotal + itemPrice>
<tr>
<td>#merchName#</td>
<td>#orderQty#</td>
<td>#dollarFormat(itemPrice)#</td>
</tr>
</cfloop>

<!-- Last row in nested table for total -->
<tr>
<td colspan="2"><strong>Total:</strong></td>
<td><strong>#dollarFormat(orderTotal)#</strong></td>
</tr>
</table>
</td>
</tr>
</cfif>
</cfoutput>
</table>
</cfif>

</body>
</html>

```

As noted earlier, `getAuthUser()` always returns whatever value was provided to the `name` attribute of the `<cfloginuser>` tag at the time of login. The examples in this chapter provide the user's ID and first name to `name` as a comma-separated list. Therefore, the current user's ID and name can easily be retrieved with the `listFirst()` and `listRest()` functions, respectively. Two `<cfset>` tags near the top of Listing 23.8 use these functions to set two simple variables called `contactID` and `contactName`. The rest of the code is essentially identical to the previous version of the template (refer to Listing 23.5). The only change is the fact that `contactID` is used instead of `SESSION.auth.contactID`, and `contactName` is used instead of `SESSION.auth.firstName`.

Using Roles to Dynamically Restrict Functionality

So far, the examples in this chapter provide the same level of access for each person. That is, each user is allowed access to the same type of information. ColdFusion's `<cflogin>` framework also provides a simple way for you to create applications in which different people have different levels of access. The idea is for your code to make decisions about what to show each user based on the person's role (or roles) in the application.

NOTE

For the purposes of this discussion, consider the word "role" to be synonymous with group, right, or privilege. The example application for this book thinks of roles as groups. That is, each contact is a member of a role called `Admin` or `User` or the like. Those role names sound a lot like group names. Other ColdFusion applications might have role names that sound more like privileges; for instance, `MayReviewAccountHistory` or `MayCancelOrders`. Use the concept of a role in whatever way makes sense for your application. How exactly you view a role is up to you.

Back in Listing 23.6, the name of the role assigned to the current user was supplied to the `roles` attribute of the `<cfloginuser>` tag. As a result, ColdFusion always knows which users belong to which roles. Elsewhere, the `isUserInRole()` function can be used to determine whether the user is a member of a particular role.

For instance, if you want to display some kind of link, option, or information for members of the `Admin` role but not for other users, the following `<cfif>` test would do the trick:

```
<cfif isUserInRole("Admin")>
  <!-- special information or options here -->
</cfif>
```

Listing 23.9 is one more version of the order history template. This version uses the `isUserInRole()` function to determine whether the user is a member of the `Admin` role. If so, the user is given the ability to view any customer's order history, via a drop-down list. If the user is an ordinary visitor (not a member of `Admin`), then they only have access to their own order history.

Listing 23.9 `OrderHistory4.cfm`—Using `isUserInRole()` to Restrict Access

```
<!---
Filename: OrderHistory4.cfm
Created by: Nate Weiss (NMW)
Purpose: Displays a user's order history
```

```

---->

<html>
<head>
  <title>Order History</title>

  <!-- Apply some simple CSS style formatting -->
  <style type="text/css">
    BODY { font-family:sans-serif;font-size:12px;color:navy}
    H2 { font-size:20px}
    TH { font-family:sans-serif;font-size:12px;color:white;
        background:MediumBlue;text-align:left}
    TD { font-family:sans-serif;font-size:12px}
  </style>
</head>
<body>

<!-- getAuthUser() returns whatever was supplied to the NAME -->
<!-- attribute of the <cflogin> tag when the user logged in. -->
<!-- We provided user's ID and first name, separated by -->
<!-- commas; we can use list functions to get them back. -->
<cfset contactID = listFirst(getAuthUser())>
<cfset contactName = listRest(getAuthUser())>

<!-- If current user is an administrator, allow user -->
<!-- to choose which contact to show order history for -->
<cfif isUserInRole("Admin")>
  <!-- This session variable tracks which contact to show history for -->
  <!-- By default, assume the user should be viewing their own records -->
  <cfparam name="SESSION.orderHistorySelectedUser" default="#contactID#">

  <!-- If user is currently choosing a different contact from list -->
  <cfif isDefined("FORM.selectedUser")>
    <cfset SESSION.orderHistorySelectedUser = FORM.selectedUser>
  </cfif>
  <!-- For rest of template, use selected contact's ID in queries -->
  <cfset showHistoryForContactID = SESSION.orderHistorySelectedUser>

  <!-- Simple HTML form, to allow user to choose -->
  <!-- which contact to show order history for -->
  <cfform
    action="#CGI.SCRIPT_NAME#"
    method="POST">

    <h2>Order History</h2>
    Customer:

    <!-- Get a list of all contacts, for display in drop-down list -->
    <cfquery datasource="#APPLICATION.dataSource#" name="getUsers">
      SELECT ContactID, LastName || ', ' || FirstName AS FullName
      FROM Contacts
      ORDER BY LastName, FirstName
  </cfquery>
  </cfform>
  </cfif>
</cfif>

```

```
</cfquery>

<!-- Drop-down list of contacts -->
<cfselect
name="selectedUser"
selected="#SESSION.orderHistorySelectedUser#"
query="getUsers"
display="FullName"
value="ContactID"></cfselect>

<!-- Submit button, for user to choose a different contact -->
<input type="Submit" value="Go">

</cfform>

<!-- Normal users can view only their own order history -->
<cfelse>
<cfset showHistoryForContactID = contactID>

<!-- Personalized message at top of page-->
<cfoutput>
<h2>YourOrder History</h2>
<p><strong>Welcome back, #contactName#!</strong><br>
</cfoutput>

</cfif>

<!-- Retrieve user's orders, based on ContactID -->
<cfquery name="getOrders" datasource="#APPLICATION.dataSource#">
SELECT OrderID, OrderDate,
(SELECT Count(*)
FROM MerchandiseOrdersItems oi
WHERE oi.OrderID = o.OrderID) AS ItemCount
FROM MerchandiseOrders o
WHERE ContactID = #showHistoryForContactID#
ORDER BY OrderDate DESC
</cfquery>

<!-- Determine if a numeric OrderID was passed in URL -->
<cfset showDetail = isDefined("URL.orderID") and isNumeric(URL.orderID)>

<!-- If an OrderID was passed, get details for the order -->
<!-- Query must check against ContactID for security -->
<cfif showDetail>
<cfquery name="getDetail" datasource="#APPLICATION.dataSource#">
SELECT m.MerchName, oi.ItemPrice, oi.OrderQty
FROM Merchandise m, MerchandiseOrdersItems oi
WHERE m.MerchID = oi.ItemID
AND oi.OrderID = #URL.orderID#
AND oi.OrderID IN
(SELECT o.OrderID FROM MerchandiseOrders o
WHERE o.ContactID = #showHistoryForContactID#)
```

```

</cfquery>

<!--- If no Detail records, don't show detail --->
<!--- User may be trying to "hack" URL parameters --->
<cfif getDetail.recordCount eq 0>
<cfset showDetail = False>
</cfif>
</cfif>

<cfif getOrders.recordCount eq 0>
<p>No orders placed to date.<br>
<cfelse>
<cfoutput>
<p>Orders placed to date:
<strong>#getOrders.recordCount#</strong><br>
</cfoutput>

<!--- Display orders in a simple HTML table --->
<table border="1" width="300" cellpadding="5" cellspacing="2">
<!--- Column headers --->
<tr>
<th>Date Ordered</th>
<th>Items</th>
</tr>

<!--- Display each order as a table row --->
<cfoutput query="getOrders">
<!--- Determine whether to show details for this order --->
<!--- Show Down arrow if expanded, otherwise Right --->
<cfset isExpanded = showDetail and (getOrders.OrderID eq URL.orderID)>
<cfset arrowIcon = iif(isExpanded, "`ArrowDown.gif'", "`ArrowRight.gif'")>

<tr>
<td>
<!--- Link to show order details, with arrow icon --->
<a href="OrderHistory4.cfm?orderID=#OrderID#">

#dateFormat(OrderDate, "mmmm d, yyyy")#
</a>
</td>
<td>
<strong>#ItemCount#</strong>
</td>
</tr>

<!--- Show details for this order, if appropriate --->
<cfif isExpanded>
<cfset orderTotal = 0>
<tr>
<td colspan="2">

<!--- Show details within nested table --->
<table width="100%" cellspacing="0" border="0">

```

```

<!--- Nested table's column headers --->
<tr>
<th>Item</th><th>Qty</th><th>Price</th>
</tr>

<!--- Show each ordered item as a table row --->
<cfloop query="getDetail">
<cfset orderTotal = orderTotal + itemPrice>
<tr>
<td>#MerchName#</td>
<td>#OrderQty#</td>
<td>#dollarFormat(ItemPrice)#</td>
</tr>
</cfloop>

<!--- Last row in nested table for total --->
<tr>
<td colspan="2"><strong>Total:</strong></td>
<td><strong>#dollarFormat(orderTotal)#</strong></td>
</tr>
</table>
</td>
</tr>
</cfif>
</cfoutput>
</table>
</cfif>

</body>
</html>

```

As you can see, the `isUserInRole()` function is used to determine whether the user is an administrator. If so, the `<cfselect>` tag is used to provide the user with a drop-down list of everyone from the `Contacts` table. The `SESSION.orderHistorySelectedUser` variable is used to track the user's current drop-down selection; this is very similar conceptually to the way the `CLIENT.lastSearch` variable was used in the `SearchForm.cfm` examples in Chapter 20. Another variable, called `showHistoryForContactID`, is created to hold the current value of `SESSION.orderHistorySelectedUser`.

If, on the other hand, the user isn't an administrator, the value of `showHistoryForContactID` is simply set to their own contact ID number. In other words, after the large `<cfif>` block at the top of this listing is finished, `showHistoryForContactID` always holds the appropriate ID number with which to retrieve the order history. The rest of the code is very similar to that in the earlier versions of the template in this chapter; it just uses `showHistoryForContactID` in the `WHERE` parts of its queries to make sure the user sees the appropriate order history records.

Figure 23.5 shows the results for users who log in as administrators (you can log in with username `Ben` and password `Forta` to see these results). All other users will continue to see the interface shown in Figure 23.4.



Figure 23.5

The concept of user roles can be used to expose whatever functionality is appropriate for each one.

You may also want to know what roles a user belongs to. ColdFusion 8 adds the `getUserRoles()` function, which returns all the roles that were assigned to a user at login. Another new feature is `isUserInAnyRole()`. Before discussing this function, let's consider how `isUserInRole()` handles multiple roles. If you pass a list of roles to `isUserInRole()`, the user must be in all of the roles for the function to return a `true` value. If you want to see whether a user is any role, you would use the `isUserInAnyRole()` function. A common use for this functionality is in situations where you want to restrict access to a certain group *and* also enable a special admin group to always be allowed. Previous versions of ColdFusion required code like this:

```
<cfif isUserInRole("test") or isUserInRole("admin")>
```

Now you can simply test for either role, like this:

```
<cfif isUserInAnyRole("test,admin")>
```

You also may wonder how you can check to see whether a user is currently logged in. Our example application forced all users to log in, but you may build a Web site where login is optional. ColdFusion 8 adds the `isUserLoggedIn()` function, which returns `true` or `false` based on whether or not the user is logged in.

Using Operating System Security

We've seen how you can roll your own security system so that authentication can be performed in multiple fashions. You can use a database lookup, LDAP, or any other method. One that may be particularly useful is the operating system itself. If your ColdFusion server runs on a Windows machine using domains, ColdFusion allows you to authenticate against any domain. You can not only authenticate a user, you can get a list of groups the user is a member of. This is all possible with the `<cfNTAuthenticate>` tag. Table 23.4 lists the attributes for this tag.

Table 23.4 `<cfNTAuthenticate>` Tag Attributes

Attribute	Purpose
username	Required. Username to authenticate.

password	Required. Password to authenticate.
domain	Required. The domain that the user belongs to. ColdFusion must be running on a box that has access to this domain.
result	Optional. Specifies the name of a variable that will contain the result of the authentication attempt. This structure will contain an auth key that indicates if the user was authenticated, a groups key that lists the groups the user is a member of (if the listGroups attribute is used), and a status value. Status will be success, UserNotInDirFailure (the user isn't a member of the domain), or AuthenticationFailure (password failure).
listGroups	Optional. If true, the user's groups will be returned in the structure specified by the result attribute. The default value is false.
throwOnError	Optional. Specifies if the tag should throw an exception if the authentication fails. This defaults to false.

Listing 23.10 demonstrates a simple example of using `<cfNTAuthenticate>`. I'm keeping this example very simple since it will only run on Windows machines, and only those machines that are part of a domain. Obviously you will need to modify the username and password values.

Listing 23.10 DomainAuth.cfm—Using `<cfNTAuthenticate>`

```

<!---
  Filename: DomainAuth.cfm
  Created by: Raymond Camden (ray@camdenfamily.com)
  Purpose: Uses <cfNTAuthenticate>
-->

<!--- Change this username! --->
<cfset username="changeme">

<!--- Change this password! --->
<cfset password="changeme">

<!--- Change this domain! --->
<cfset domain="changeme">

<!--- Attempt to logon --->
<cfNTAuthenticate username="#username#" password="#password#" result="result"
  domain="#domain#" listGroups="yes">

<cfdump var="#result#" label="Result of NT authentication.">

```

The script begins by creating variables for the three main pieces needed for authentication, username, password, and domain. As it obviously states in the code, you will need to change these values. However, if you want to see a failed authentication result, you can leave these alone. Finally, we run the `<cfNTAuthenticate>` tag, passing in the values and telling it to return the result in a struct called result and enumerating the groups the user belongs to. Lastly we dump the result structure. Again, you will have to modify the values in order to get a valid authentication result.

Defending Against Cross-Site Scripting

One way your web application can be harmed is by cross-site scripting. This is simply the use of HTML and other codes within web based forms. As a simple example, imagine a forums application that lets people write their own entries. Someone could write an entry that contained JavaScript code. When someone else views that page, the JavaScript code is executed just as if you had written it yourself. This could be very dangerous. Luckily, ColdFusion provides a simple solution. In Chapter 19, you learned about the `Application.cfc` file and how you can configure ColdFusion applications via the `THIS` scope. You can simply add one more attribute to the `THIS` scope:

```
<cfset THIS.scriptProtect="all">
```

This one line will clean all `FORM`, `URL`, `CGI`, and `COOKIE` variables. So for example, this line of text:

```
<script>alert('hi');</script>
```

becomes

```
<InvalidTag>alert('hi');</script>
```

You can specify just one of the above scopes instead of "ALL" if you want to be more specific. You can also turn this feature on automatically for all files on the server. You do this by enabling Enable Global Script Protection in the ColdFusion Administrator.