Programmer to Programmer ™

ASP.NET Distributed Data Applications

Written and tested for final release of .NET v1.0

Alex Homer and Dave Sussman

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com

What You Need to Use This Book

The following list is the recommended system requirements for running the code in this book:

- Windows 2000 Professional or higher with IIS installed
- Windows XP Professional with IIS installed
- ASP.NET Version 1.0
- SQL Server 2000 or MSDE
- Visual Studio .NET Professional or higher (optional)

In addition, this book assumes the following knowledge:

- A good understanding of the .NET Framework and ASP.NET
- An understanding of the VB.NET language and JavaScript

Summary of Contents

Introduction		**1**
Chapter 1:	The Distributed Application	**9**
Chapter 2:	Components and Data Access	**49**
Chapter 3:	Accessing XML Documents	**91**
Chapter 4:	The Application Plumbing	**119**
Chapter 5:	Working with Down-Level Clients	**159**
Chapter 6:	Working with Rich Clients	**209**
Chapter 7:	Remoting to .NET Clients	**281**
Chapter 8:	Updating Data in Down-Level Clients	**321**
Chapter 9:	Updating Remote Cached Data	**389**
Chapter 10:	Components for Updating Data	**431**
Chapter 11:	Rich Client Update Applications	**475**
Chapter 12:	Reconciling Update Errors	**525**
Chapter 13:	Updating Data from Remote .NET Applications	**579**
Chapter 14:	Furthermore	**621**
Index		**625**

Working with Down-Level Clients

In previous chapters we've seen how we can access data sources and create a data tier for our applications. We also discussed techniques for remoting data to various types of client, and how we can detect the client type when they first hit our application. It's now time to see an application that puts all these techniques into practice.

The application we've built is quite compact and tries not to obscure the processes by being too complicated. We also tried to make it look attractive and easy to understand and use. You'll have to judge how well we succeeded for yourself, of course, but it does neatly demonstrate some useful techniques for combining ASP.NET, server-side .NET components and client-side programming to create distributed data applications.

Many of the techniques in this book are primarily aimed at **rich clients**, such as Internet Explorer and .NET applications, and we'll be looking at these in the following chapters. However, unless we can be certain that these are the only types of client that will use the application, we should also provide a version that works on other **down-level clients**. We described how and why we categorize down-level clients in Chapter 1. The term *down-level* is the one Microsoft prefer to use, and is possibly less offensive than some other choices!

Anyway, the topics we'll be concentrating on in this chapter are:

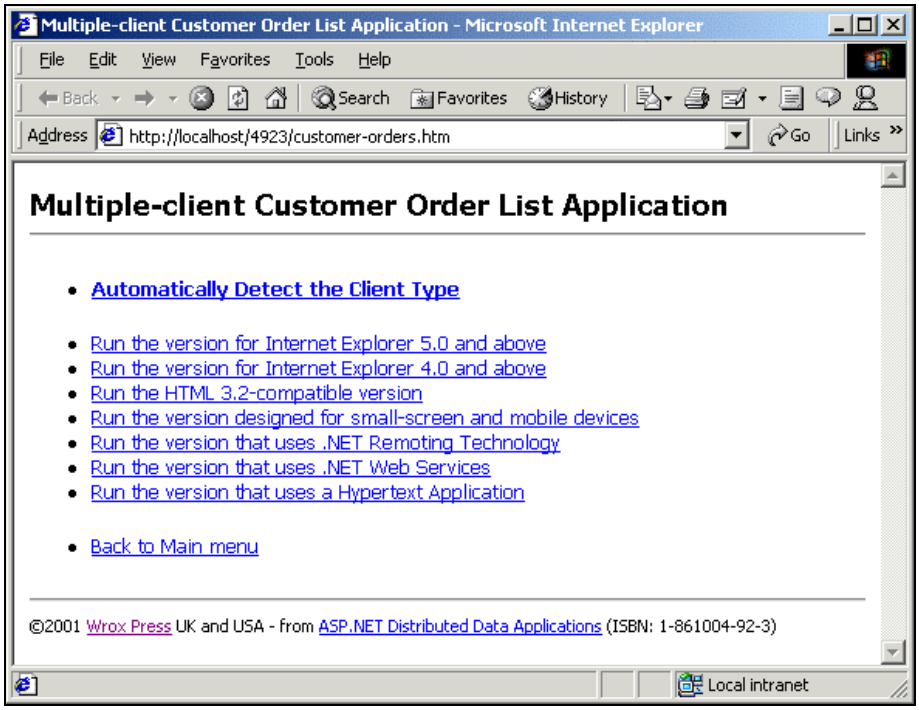
- What the application looks like
- Some of the design considerations involved
- Specific client detection techniques for the example application
- The version of the application aimed at HTML 3.2 clients
- The version aimed at small screen and mobile devices

We start with a brief look at the appearance and functionality of the application.

A Multiple-Client Order List Application

In this chapter, we'll build an application that can be used to view and edit the orders placed by customers of a fictional food distribution corporation. We'll use the Northwind database that comes as a sample with SQL Server versions 7.0 onwards. We plan to support several different types of client, at the same time taking advantage of several different technological approaches.

We'll build a series of pages that take full advantage of several distinct and different types of down-level and rich client variations. We'll incorporate some client-detection code, like the code we developed in the previous chapter, and we'll use the associated techniques such as remoting data and using ASP.NET Sessions. To give you an idea of the clients and technologies we're supporting, the next screenshot shows the menu page for the application:



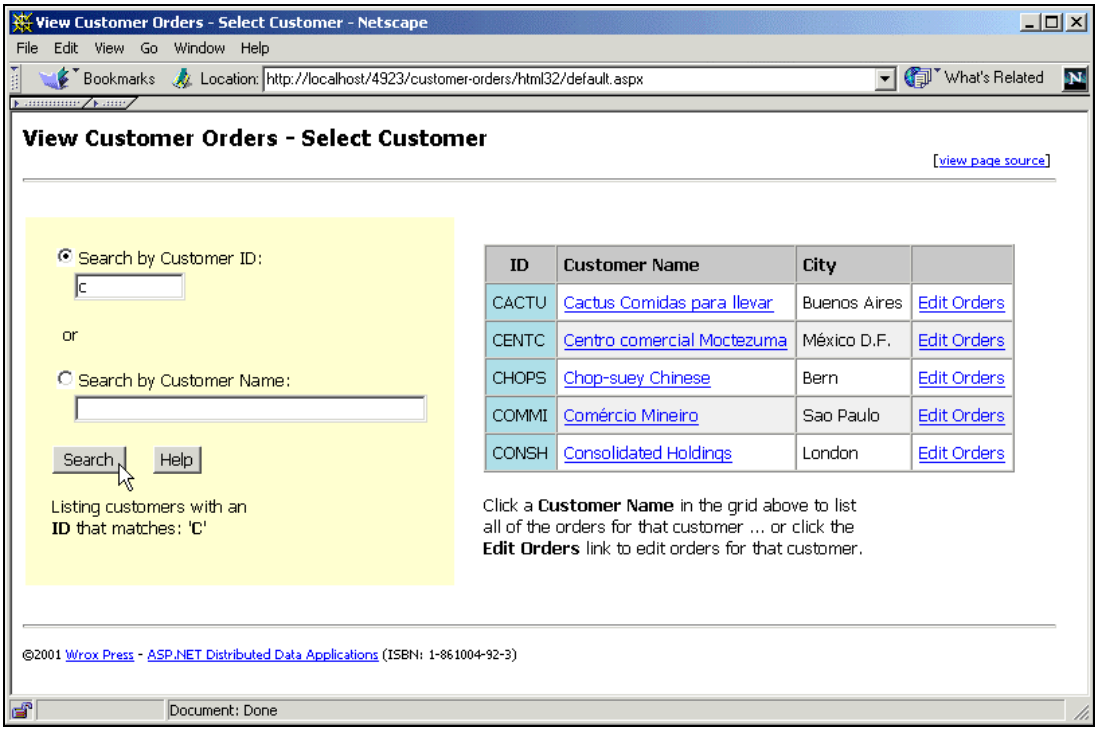
You can obtain the example source code from http://www.wrox.com *or from* http://www.daveandal.com/books/4923/. *For details of how to install and configure them, and the database we use to drive the examples, see the section* Setting Up the Examples *in Chapter 2. If you just want to view the results, we've provided the application online at* http://www.daveandal.com/books/4923/.

As well as an option that automatically detects the client type you are using, there are links inviting the user to view the different versions irrespective of the client type. Just bear in mind that some may not work if your client doesn't support the features used by that version of the application. For example, the Internet Explorer 5.0 and above version requires the MSXML parser to be installed, and so it will most likely not work with other versions of Internet Explorer or other makes of browser.

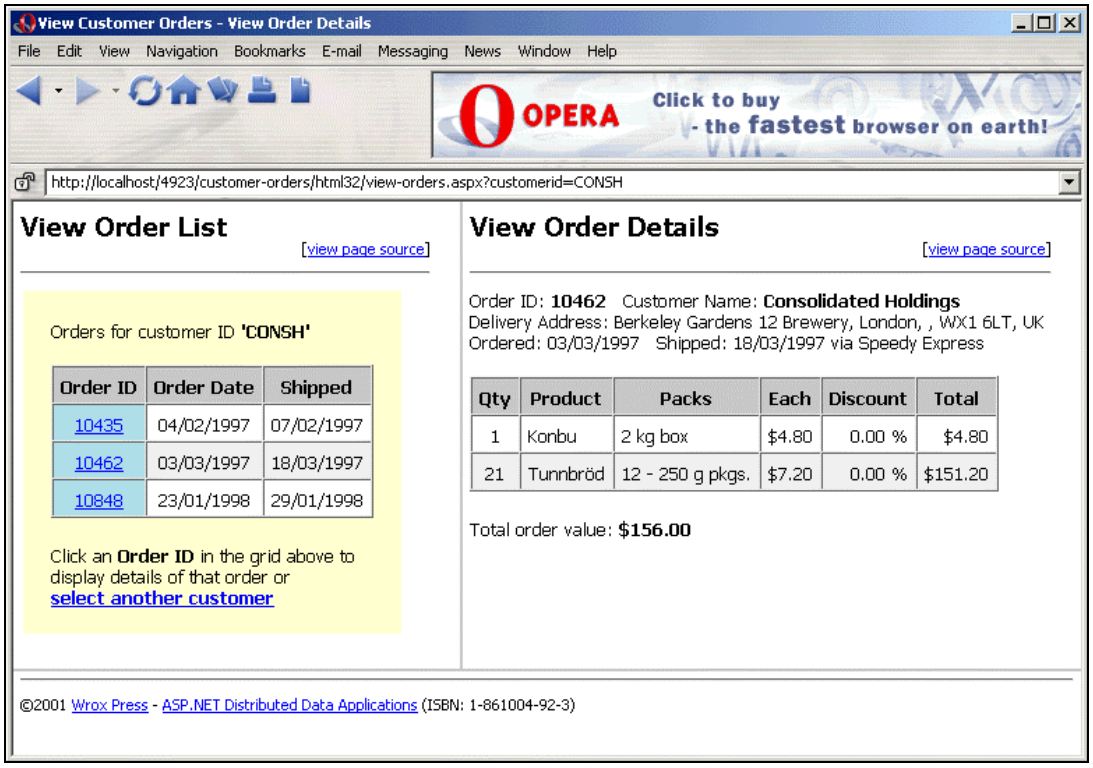
The User Interface

With the exception of the pages designed for small-screen and mobile devices, and those that use .NET remoting techniques, the application is designed to give a very similar user interface for all clients. The following two screenshots show the version designed for use with any Web browser that supports HTML 3.2 – you'll see several different browsers in the screenshots for this version of the application. The two Internet Explorer-specific versions, and the Hypertext Application (which will only work in IE 5 and above) look almost identical.

When the user opens the application they first select the customer whose orders they want to view. After entering all or part of the customer ID or name and clicking the Search button, the right-hand section of the page displays a list of matching customers:



Clicking on a customer name in the table opens the second page. Here the user can select an order in the left-hand list for this customer, and the details of that order are then displayed in the right-hand part of the page:



The application also allows orders to be edited, though we will not be covering this feature just yet. We'll concentrate on how we extract, manipulate and display the order information in this and the next couple of chapters. We'll come back to look at how we can update the order data later in the book.

Detecting the Client Type

In the menu page we saw earlier, the top link (Automatically Detect the Client Type) uses the techniques we discussed in Chapter 4 to redirect the client to the appropriate version of the application. It checks for client-side scripting support on the way if this is a requirement in the version designed for the current client. We use the `client-detect.ascx` user control that we developed in Chapter 4 to detect which "group" our client falls into.

In the `Page_Load` event handler of our `default.aspx` page (http://localhost/4923/customer-orders/default.aspx), we can then make decisions on where to redirect the client depending on this group. Notice that we check for unsupported clients, and send them a simple text message. We don't use client-side scripting for small-screen and mobile devices, and so we can transfer them directly to the appropriate version of the application in the subfolder named `mobile`. All other clients are redirected to the page `client-script-check.aspx`, with the client type appended to the query string.

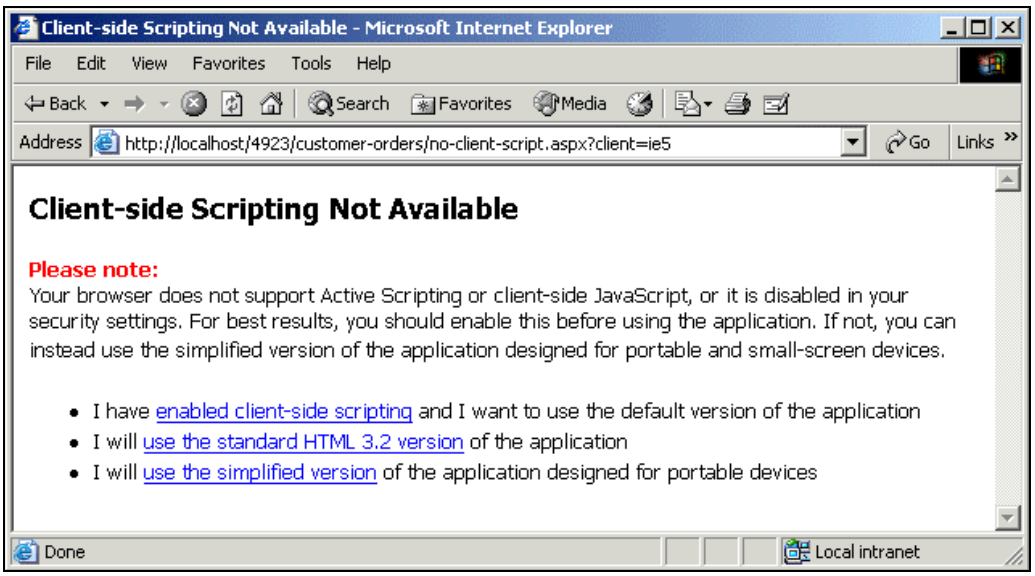
Overall, this is very similar to the code we saw in http://localhost/4923/detect-client/default.aspx, in Chapter 4:

```
Sub Page_Load()
  Select Case ClientDetect.ClientType
    Case 0  'not supported
      Response.Clear
      Response.ContentType = "text/text"
      Response.Write("Sorry, this application does not support " _
                  & "your client type: " & Request.UserAgent)
      Response.End
    Case 2  'IE 4.x
      Response.Clear
      Response.Redirect("client-script-check.aspx?client=ie4")
      Response.End
    Case 3  'IE 5.x and above
      Response.Clear
      Response.Redirect("client-script-check.aspx?client=ie5")
      Response.End
    Case 4, 5  'small-screen HTML device or WML client
      Response.Clear
      Server.Transfer("mobile/default.aspx")
    Case Else  'assume HTML 3.2 client
      Response.Clear
      Response.Redirect("client-script-check.aspx?client=html32")
      Response.End
  End Select
End Sub
```

Each client-specific version of the application is stored in a subfolder that is named in the `client` name/value pair of the query string we show in this listing. So, for example, Internet Explorer 5 browsers will be redirected to the version in a folder named `ie5`. And, as we're not checking for session support in our application, we can use `Server.Transfer` in the case of the version for small-screen and mobile devices.

Checking for Client-Side Scripting Support

Our application uses a page named `client-script-check.aspx`, which works in just the same way as the example we saw in Chapter 4. If the client has scripting available (*and enabled*), they are redirected to the appropriate version of the application. However, if client-side scripting is not available (or is disabled), the client is redirected to a page named `no-client-script.aspx`, which displays a message telling the user what options are available to them in this case:



This page is simple, using ASP.NET code only to set the value in the query string for the first option. If the user enables scripting and then clicks this link, we can take them to the version of the application that our client type detection code originally suggested. Alternatively, they can go directly to the HTML 3.2 or the small-screen and mobile devices version:

`no-client-script.aspx`

```
<% Dim strClientType As String = Request.QueryString("client") %>
...
<ul>
  <li>I have <a href="<% = strClientType %>/default.aspx">
      enabled client-side scripting</a> and I want to use ... </li>
  <li>I will <a href="html32/default.aspx">
      use the standard HTML 3.2 version</a> of the application</li>
  <li>I will <a href="mobile/default.aspx">
      use the simplified version</a> of the application ...</li>
</ul>
...
```

Session Support

In this application, we aren't checking for session support for each client. As we hope to do most of the processing on the client, we don't need to use ASP.NET Sessions. However, two versions of the application (the HTML 3.2 and the small screen and mobile devices versions covered in this chapter) will benefit from session support. Nevertheless it is not an absolute requirement, even for these. They will automatically take advantage of sessions to give improved performance and reduce the number of data access operations required when supported, or work fine without them otherwise.

Of course, if an application depends on session support, we can use the techniques we examined in Chapter 4 to check that session support exists, and redirect to a version of the application in a different subfolder where we have a `web.config` file that specifies cookie-less sessions (by munging the session ID into the URL itself). We'll look at session issues for our application in more depth shortly.

The current release of the Microsoft Mobile Internet Toolkit stores the ViewState data in the current ASP.NET Session. If the mobile client you use does not support cookies (and hence does not support sessions) you may get an error. See the section that describes the mobile version of the application for more details of how to configure the samples in this case.

The Data Access Tier

We use the same data access component as we did in the last couple of examples in Chapter 2 (`Wrox4923Orders.OrderList`), for all versions of our application. This is intentional, and demonstrates how the separation of data access, business logic and presentation tiers in our applications allows us to use the *same* data access tier for several versions of the application. If the data store we are using changes in type or structure, or different data access techniques are implemented in the future, we only have to update the single component. As long as it continues to expose the same interface (the same methods with the same parameters and returning the same data types), then all the versions of our application will continue to work with the new component.

As we saw in Chapter 2, our `Wrox4923Orders.OrderList` component exposes two methods:

- The `GetCustomerByIdOrName` method accepts two `String` parameters and returns a `DataSet`. The parameters are all or part of the customer ID, and all or part of the customer's name. The returned `DataSet` contains a list of customers that match the ID (if this is provided) or name (if an empty string is used for the ID parameter).
- The `GetOrdersByCustomerDataSet` method accepts a single `String` parameter and returns a `DataSet`. The parameter is the full customer ID, and the returned `DataSet` has two tables that contain all the orders for this customer, and all the order lines (details) for these orders.

In some cases this is a little wasteful, because we don't always end up displaying all the information in the `DataSet`. For example, if the user only views one order, then the "detail" rows we extracted from the database for all the others are never used. In this case, it would have been more efficient to fetch the "detail" rows only when an order is selected, minimizing the volume of data that is extracted from the database. However, if we used this alternative technique when the user wanted to browse order details, it would mean repeated trips to the database to collect the relevant "detail" rows.

Maximizing Efficiency through Data Caching

It's generally accepted that the two major bottlenecks in any Web-based application are the data access process and the network bandwidth. As far as server loading goes, data access is usually the worst culprit. So, if we can extract all the data we need in one go, we can minimize server loading and reduce the number of data access hits we get.

OK, so the volume of data that we extract is greater, but we are only moving it from the database to the Web server. Hopefully, this will be a LAN connection where there is plenty of bandwidth; and in the "small site" case where the database and Web server are on the same machine, using Named Pipes for the data transfer, bandwidth is extremely unlikely to be an issue.

What we must do, however, is make sure that we **cache** the data so that we don't need to keep going back to the database each time. In the examples in this chapter, we cache the data in the client's session. Most clients these days do support "per-session" cookies such as those required for ASP.NET Sessions to function correctly. In fact, many modern on-line applications, such as banking and e-commerce sites, demand per-session cookie support and users have generally become accustomed to accepting them.

As you'll see later, lack of session support will not break our applications. If the client does not support sessions, we simply hit the data access layer again to fetch a fresh copy of the data. This is not an ideal situation, but we hope that this problem applies to just a small number of clients, and therefore it is not worth building special versions of the application. Of course, if your situation is different, and you do have to support a large proportion of clients that do not support cookie-based sessions, you might prefer to build special versions that use cookie-less sessions (as mentioned earlier in this chapter and in Chapter 4). Alternatively, you could build a version that uses a `DataReader` to access the database – thereby reducing the loading that repeated data access hits will produce.

In the versions of the application we discuss in later chapters, the data is cached on the client rather than on the server; in most cases this gives the best of both worlds. It limits server loading by avoiding repeated hits on the data access tier, and avoids the memory and processing overhead required for managing ASP.NET Sessions for each user.

The Order List Application – the HTML Version

The screenshots we looked at earlier are from the HTML version of the application, and this is designed to run in any HTML 3.2 compatible browser. The appearance will differ between browsers from different manufacturers and (to some extent) between different versions, because we're using Cascading Style Sheets (CSS) and other features to control the display. These might not be supported in full or in exactly the same way in all browsers, but the overall layout and usability should be unaffected. And while we do use some client-side script in this version of the application, it is only minimal and the pages will still work when client-side scripting is not supported.

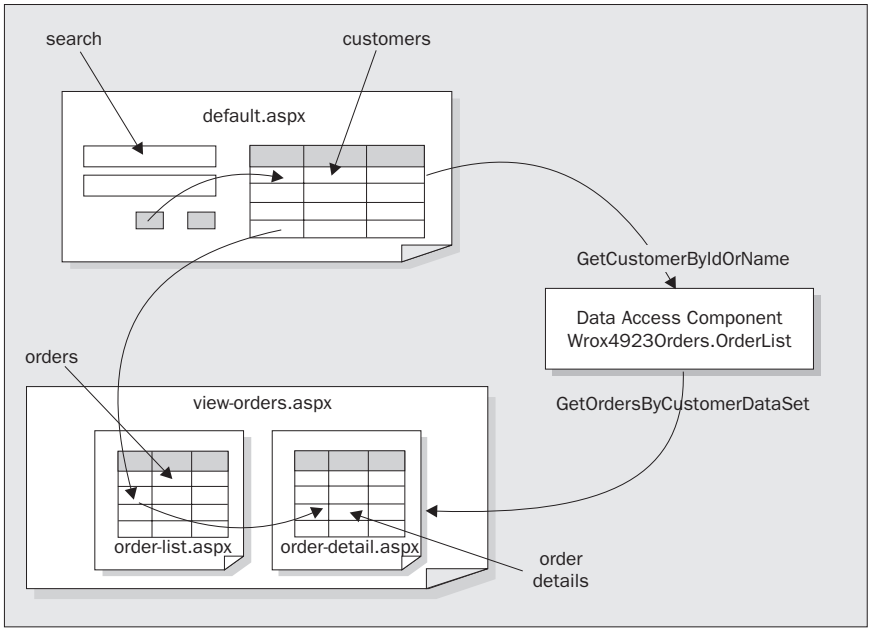
The Outline Process

The outline process for the HTML version of the application is shown in the next schematic. The `default.aspx` page (in the `customer-orders\html32` folder) allows users to search for and display a list of customers based on their ID or name. It uses the `GetCustomerByIdOrName` method of our data access component. Selecting a customer opens the `view-orders.aspx` page. This is an HTML frameset, into which the two pages `order-list.aspx` and `order-detail.aspx` are loaded.

> *It's worth noting that the HTML 3.2 specification doesn't include frames – they were introduced into HTML in version 4.0. However, the majority of HTML 3.2-compatible browser clients* do *support frames – IE 3.0 and Netscape Navigator 2.0 among them – so we'll assume support for framesets here, and provide a link to the simplified small-screen version for the small number of HTML 3.2-compliant clients that don't support frames.*

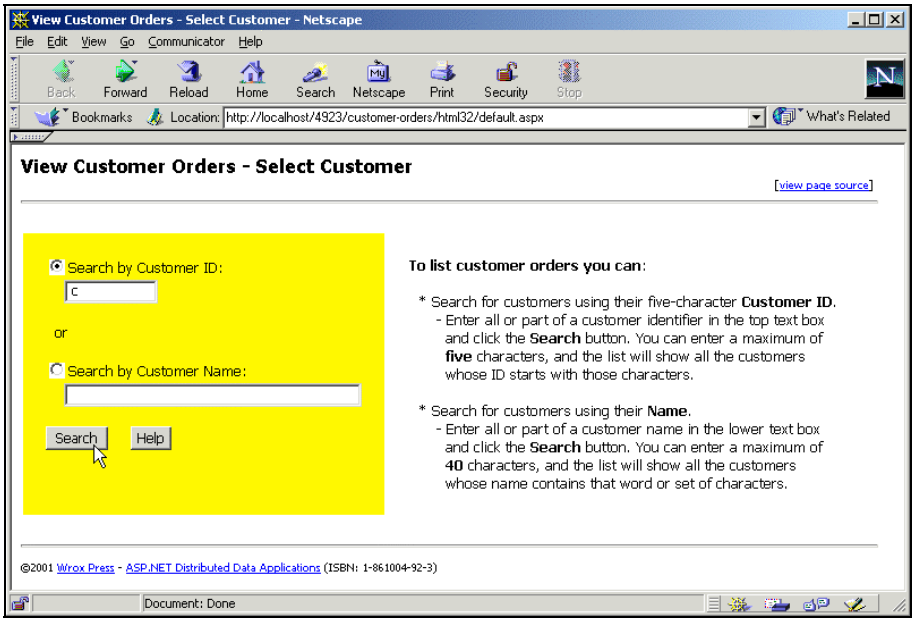
The left-hand page, `order-list.aspx`, uses the `GetOrdersByCustomerDataSet` method of our data access component to get the order data, and displays a list of all the orders for this customer as soon as this page loads. It also caches the complete `DataSet` in the user's session. If the page is refreshed for any reason, it uses this cached data automatically. However, if the user elects to go back to the previous page and select another customer, the cached order data for this customer is removed from the session by code in `default.aspx`. We'll see all this as we step through this code in the coming pages.

Selecting an order in the list in the left-hand page reloads the page `order-details.aspx` in the right-hand frame, and it then shows a list of all the "detail" rows for the selected order. It uses the `DataSet` that was cached in the user's session by the page `order-list.aspx`. If this is not available, it calls the same `GetOrdersByCustomerDataSet` method to get the data directly from the data store instead:



Searching for Customers

When this version of the application starts, it displays the "Select Customer" page shown in the next screenshot. As well as some hints on how to search for customers by ID or name, it contains the relevant text boxes and a Search button – plus a Help button that can be used at any stage to redisplay the search hints. Here, we're searching for all customers whose ID starts with "C":



This page contains the `Import` directives for our custom data access component `OrderListData`, and for the `System.Data` namespace. We'll need this to be able to create a `DataSet` object in our code. We also register and insert the user control that exposes our server's database connection string, as we demonstrated in the examples in previous chapters:

default.aspx

```
<%@Page Language="VB" %>
<%@Import Namespace="OrderListData" %>
<%@Import Namespace="System.Data" %>

<%@ Register TagPrefix="wrox" TagName="connect"
             Src="..\..\global\connect-strings.ascx" %>

<%'-- insert connection string script --%>
<wrox:connect id="ctlConnectStrings" runat="server" />
```

The HTML Form Controls

This is followed by the HTML that creates the visible portion of the page. We use a server-side include statement to insert a stylesheet that is common to most of our application versions, and define the client-side JavaScript function that makes the interface a little more useable by setting the option buttons to the relevant value as the user types in the text boxes. This function is used for the (client-side) `onkeypress` events of the two text boxes:

```
<html>
  <head>
    <title>View Customer Orders - Select Customer</title>
    <!-- #include file="../../global/style.inc" -->
    <script language="JavaScript">
      <!--
        // client-side script section used to set radio
        // buttons to correct option as text is typed.
        function setCheck(strName) {
          document.forms(0).elements(strName).checked = true;
        }
      //-->
    </script>
  </head>

  <body link="#0000ff" alink="#0000ff" vlink="#0000ff">
  ...
  <form runat="server">
    <table border="0" cellpadding="20">
      <tr><td valign="top" bgcolor="#ffffacd">

        <!-- controls for specifying the required customer ID or name -->
        <asp:RadioButton id="optByID" groupname="SearchBy" Align="right"
                         text="Search by Customer ID: " runat="server"
                         checked="true" /><br />
        <asp:TextBox id="txtCustID" columns="5" maxlength="5"
                     onkeypress="setCheck('optByID');" runat="server" /><p />
```

```
        or<p />
        <asp:RadioButton id="optByName" groupname="SearchBy" Align="right"
                         text="Search by Customer Name:" runat="server" /><br />
        <asp:TextBox id="txtCustName" columns="20" maxlength="40"
                     onkeypress="setCheck('optByName');" runat="server" /><p />
        <asp:Button id="btnSearch" text="Search" onclick="DoSearch"
                    runat="server" />
        <asp:Button id="btnHelp" text="Help" onclick="ShowHelp"
                    runat="server" /><p />
        <asp:Label id="lblStatus" runat="server" />

      </td>
      ...
```

Displaying a List of Customers

Before we look at the remainder of the HTML to see how we display a list of matching customers in the right-hand part of the page, we'll examine the code that searches for them to see how it actually works. The server-side code section of the `default.aspx` page declares two `Page`-level (global) variables that we need to store the values that are selected in the two text boxes. In the `Page_Load` event handler that comes next, we just display the "Help" text in the right-hand part of the screen by calling the `ShowHelp` function:

```
<script language="VB" runat="server">
  'page-level variables accessed from more than one routine
  Dim strCustID As String = ""
  Dim strCustName As String = ""

  Sub Page_Load()
    'show Help when page first loads
    If Not Page.IsPostBack Then ShowHelp(Nothing, Nothing)
  End Sub
```

Showing the Help Text

The `ShowHelp` function simply displays some text in an ASP.NET `Label` control located in the right-hand part of the page – you'll see this control in the next part of the HTML listing, when we come to look at the `DataGrid` control:

```
Sub ShowHelp(ByVal objSender As Object, ByVal objArgs As EventArgs)
'shows help on using page in the right-hand part of the window

  lblMessage.Text = "<b>To list customer orders you can</b>:<p />" _
     & "  * Search for customers using their five-character " _
     & "<b>Customer ID</b>.<br />" _
     ...
     ... etc ...
     ...
End Sub
```

Calling the Data Access Component

Getting the `DataSet` that contains matching customers from the data store is relatively easy. We create a separate function that takes the two parameters required by the data access component, instantiates the component with the connection string obtained from our user control, and returns the `DataSet` created by the data access component. We enclose it all in a `Try...Catch` construct so that we can trap and display details of any error that might occur:

```
Function GetDataSetFromServer(strCustID As String, _
                              strCustName As String) As DataSet
'uses data access component to get DataSet of matching customers

  'get connection string from connect-strings.ascx user control
  Dim strConnect As String
  strConnect = ctlConnectStrings.OLEDBConnectionString

  Try
    'create an instance of the data access component
    Dim objOrderList As New Wrox4923Orders.OrderList(strConnect)

    'call the method to return the data as a DataSet
    Return objOrderList.GetCustomerByIdOrName(strCustID, strCustName)

  Catch objErr As Exception

    'there was an error and no data will be returned
    lblMessage.Text = "ERROR: No data returned. " & objErr.Message

  End Try
End Function
```

Performing the Customer Search

The `DoSearch` event handler, which is executed in response to a click on the Search button, is shown next. It starts off by removing any existing `DataSet` of order details from the user's session. This is required if the user comes back from examining orders for one customer to search for a different customer. We have to destroy the cached `DataSet` containing the order details for the previous customer or they will not see any orders for the new one:

```
Sub DoSearch(ByVal objSender As Object, ByVal objArgs As EventArgs)
'display the list of matching customers in the DataGrid control

  'remove any existing "Orders" DataSet from the user's Session
  'as we're searching now for a different customer
  Session("4923HTMLOrdersDataSet") = Nothing
```

Next we can collect the values from the appropriate one of the two text boxes, depending on which option button (Search by Customer ID or Search by Customer Name) is checked. We also convert any ID value that is entered into uppercase (which is how they are stored in the database):

```
    'get one or other value from text boxes depending on selection
    If optByID.Checked Then
      strCustID = txtCustID.Text.ToUpper()
      lblStatus.Text = "Listing customers with ID ..." & strCustID
    Else
      strCustName = txtCustName.Text
      lblStatus.Text = "Listing customers with Name..." & strCustName
    End If
```

Fetching and Displaying the Results

Now we can use the `GetDataSetFromServer` function we described earlier to collect our `DataSet`, and we can see how many matching customers were actually found by checking the number of rows in the single table within the `DataSet`:

```
    'get DataSet using function elsewhere in this page
    Dim objDataSet As DataSet = GetDataSetFromServer(strCustID, strCustName)

    'check how many matching customers were found
    Dim intRowsFound As Integer = objDataSet.Tables(0).Rows.Count
```

If we found any rows at all, we set the `CurrentPageIndex` of the `DataGrid` to zero (though this is the default anyway, so we could get away with omitting this), and assign the single table in the `DataSet` to the `DataSource` property of the grid. Before we actually bind it, however, we use the number of rows found to see if we need to display the paging controls. Finally, we can bind the grid and display some informative text to the user:

```
    If intRowsFound > 0 Then
      'reset DataGrid page index to zero for new rowset
      'and set DataSource of DataGrid control
      dgrCustomers.CurrentPageIndex = 0
      dgrCustomers.DataSource = objDataSet.Tables(0)

      'display the "paging" controls (Previous/Next) only when required
      dgrCustomers.PagerStyle.Visible = (intRowsFound > dgrCustomers.PageSize)

      'bind the DataGrid and display status message
      dgrCustomers.DataBind()
      lblMessage.Text = "Click a <b>Customer Name</b> in the grid above " _
                        ... etc.

    Else
      lblMessage.Text = "No matching customers found in database ..."

    End If
  End Sub
```

If there were no matching customers found, the `Else` part of the `If...Then` construct displays a message to this effect.

The DataGrid to Display the Customer List

The controls shown in the earlier HTML listing are in a table cell on the left of the page. The *right-hand* cell contains an ASP.NET `DataGrid` control that we use to display a list of matching customers, and an ASP.NET `Label` control for status messages and other information. As the `DataGrid` is not bound to a data source when the page loads the first time, it is not actually visible in the page:

```
      ...
      <td valign="top">
        <!-- DataGrid control to display matching customers -->
        <asp:DataGrid id="dgrCustomers" runat="server"
             AutoGenerateColumns="False"
             CellPadding="5"
             GridLines="Vertical"
             HeaderStyle-BackColor="silver"
             PagerStyle-BackColor="silver"
             AlternatingItemStyle-BackColor="#eeeeee"
             AllowPaging="true"
             PageSize="8"
             PagerStyle-Mode="NextPrev"
             PagerStyle-NextPageText="Next"
             PagerStyle-PrevPageText="Previous"
             PagerStyle-HorizontalAlign="Right"
             PagerStyle-Visible="false"
             DataKeyField="CustomerID"
             OnPageIndexChanged="ShowGridPage">

          <Columns>
            <asp:BoundColumn HeaderText="<b>ID</b>"
                 HeaderStyle-HorizontalAlign="center"
                 DataField="CustomerID" ItemStyle-BackColor="#add8e6"
            />
            <asp:HyperlinkColumn HeaderText="<b>Customer Name</b>"
                 DataTextField="CompanyName"
                 DataNavigateUrlField="CustomerID"
                 DataNavigateUrlFormatString="view-orders.aspx?customerid={0}"
            />
            <asp:BoundColumn HeaderText="<b>City</b>" DataField="City" />
            <asp:HyperlinkColumn Text="Edit Orders"
                 DataNavigateUrlField="CustomerID"
                 DataNavigateUrlFormatString=                                    ↴
                    "../../update-orders/html32/edit-orders.aspx?customerid={0}"
            />
          </Columns>

        </asp:DataGrid><p />

        <!-- label to display interactive messages -->
        <asp:Label id="lblMessage" runat="server" />

      </td></tr>
    </table>
  </form>
```

You can see from the listing that we've set several properties of the `DataGrid` to control its appearance. We also set the `AllowPaging` property to `true` so that a list of more than eight customers (the `PageSize` property) will be broken up into "pages" when displayed. The properties that follow the `PageSize` define the appearance of the paging controls, and we also make them invisible by default. As you saw in the previous code section, we show them by dynamically changing this property value to `true` if the number of matching customers found exceeds the value of `PageSize`.

We also specify that the `CustomerID` column in our `DataSet` should be used as the `DataKeyField`, giving us an easy way to extract the row key for a selected row if we need to do so later on. Finally, we specify an event handler named `ShowGridPage`, located within the code section of our `default.aspx` page, which will be executed when the grid paging controls are clicked (the event handler is specified by the `OnPageIndexChanged` attribute).

Defining the Columns in the DataGrid

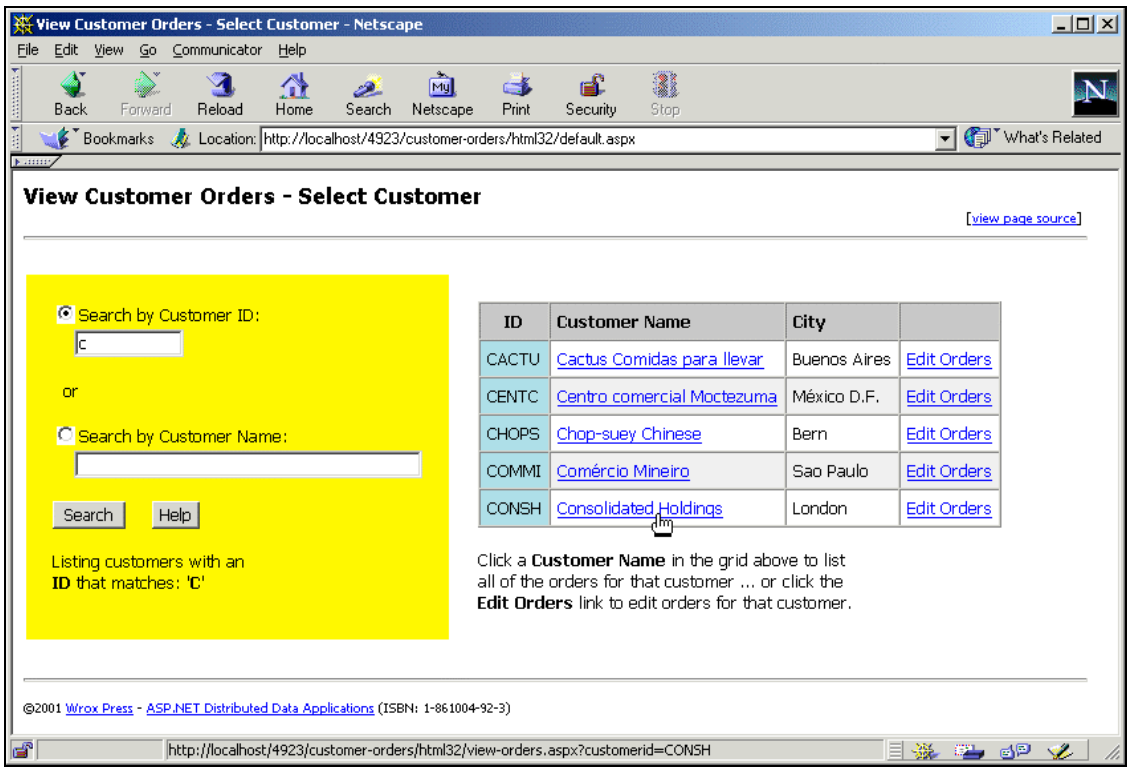
When we declared the `DataGrid` element (as shown in the previous listing), we included the attribute `AutoGenerateColumns="False"` so that the grid will not automatically generate the columns based on the contents of the data source we bind to it. Instead, we define the columns we want ourselves within a `<Columns>` element. We have a column bound to the `CustomerID` column in our data source, followed by a column that will display the customer name as a hyperlink. The combination of the `DataNavigateUrlField` and `DataNavigateUrlFormatString` attributes we use mean that this hyperlink's `href` attribute will contain the value `view-orders.aspx` with a name/value pair indicating the customer ID for that row appended to the query string:

```
<asp:HyperlinkColumn HeaderText="<b>Customer Name</b>"
     DataTextField="CompanyName"
     DataNavigateUrlField="CustomerID"
     DataNavigateUrlFormatString="view-orders.aspx?customerid={0}"
/>
```

Next comes a simple bound column that displays the `City` value from the source `DataSet`, followed by another hyperlink column that displays the text Edit Orders for every row (we'll see more about this in Chapter 8). We use the `DataNavigateUrlField` and `DataNavigateUrlFormatString` attributes for this column to specify that the `href` for the hyperlink will be the page `edit-orders.aspx` (in a separate folder of our application). Again, a name/value pair indicating the customer ID for that row is appended to the query string:

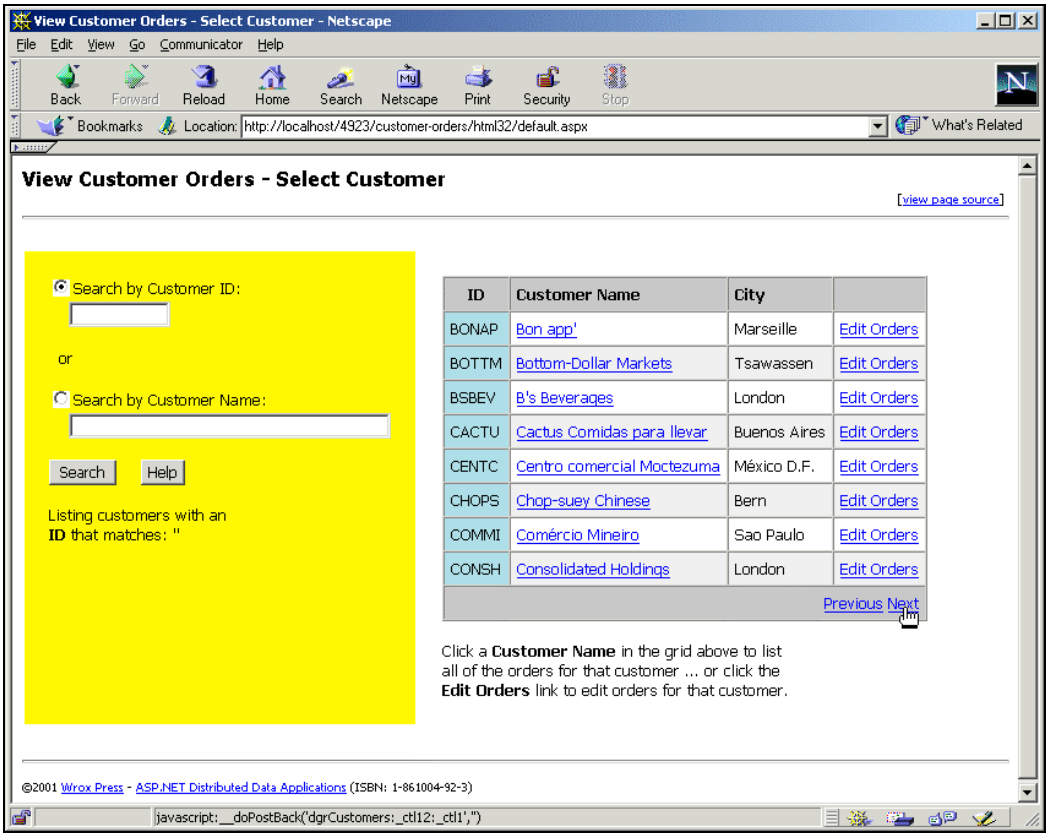
```
<asp:HyperlinkColumn Text="Edit Orders"
     DataNavigateUrlField="CustomerID"
     DataNavigateUrlFormatString=                                    ↴
        "../../update-orders/html32/edit-orders.aspx?customerid={0}"
/>
```

In the next screenshot, you can see the output that our `DataGrid` creates after searching for customers whose ID starts with the letter `c`. There are only five that match, so the paging controls do not appear (recall that we set the `Visible` property for the "pager" row within our code every time we fetch the list of matching customers):



Handling Paging in the DataGrid

However, the paging controls *do* appear when we get more than eight matching customers in our list. This is shown in the next screenshot. To get this, click Search with a blank string for both the ID and name, so that our data access component returns all the customers in the database. (In a production environment, you might prefer to limit the number of rows returned by the data access tier to some specific maximum, like 100 rows, to protect against excessive use of bandwidth and server resources if you have a large number of customers.) We've used the paging controls to go to the second page of results:



When we defined our `DataGrid` control, we specified that the event handler named `ShowGridPage` should be executed automatically when the "pager" controls are clicked:

```
OnPageIndexChanged="ShowGridPage"
```

The `ShowGridPage` routine is remarkably simple. All it has to do is extract the index of the new page (the one that has just been requested) from the `NewPageIndex` property of the arguments to the event handler, set the `CurrentPageIndex` property of the `DataGrid` to this value, collect the customer ID or name from the text boxes on the left-hand side of the page, and re–bind the grid control to the original data source:

```
Sub ShowGridPage(ByVal objSender As Object, _
                 ByVal objArgs As DataGridPageChangedEventArgs)
'runs when the paging controls are clicked to display different page

  'set page index of DataGrid control to new value
  dgrCustomers.CurrentPageIndex = objArgs.NewPageIndex

  'get one or other value from text boxes depending on selection
  If optByID.Checked Then
```

```
      strCustID = txtCustID.Text.ToUpper()
    Else
      strCustName = txtCustName.Text
    End If

    'get the DataSet to populate the control and bind it
    dgrCustomers.DataSource = GetDataSetFromServer(strCustID, strCustName)
    dgrCustomers.DataBind()

  End Sub
```

Notice that this involves another hit on our data access component. We could get round this by caching the original `DataSet` in the user's session (as we do with the order details). However, this may not actually improve performance or reduce server loading. It will only be of benefit if there are usually more than eight matching customers, and the user has to page through the list to find the one they want. If not, we are taking up server resources by caching data in the session that will only rarely be re-used. Like so many design decisions, the best choice depends mostly on the nature of the data and how the application will actually be used.

Displaying a List of Orders

In the list of customers in the page we've just seen, each customer name is a hyperlink pointing to the page `view-orders.aspx`, with a name/value pair that contains the customer ID appended to the query string – for example, `view-orders.aspx?customerid=CONSH`.

The Order List Frameset Page

As we saw in the schematic earlier on, the page `view-orders.aspx` is an HTML frameset page that contains other pages. It also includes some ASP.NET code that collects the customer ID from the query string and passes it on to the page in the left-hand frame. This is the code we use:

`view-orders.aspx`

```
<% Dim strCustID As String = Request.QueryString("customerid") %>
```

The next listing shows the HTML that creates the frameset. You can see that we use the customer ID in the query string of the URL for the page `order-list.aspx` that we load into the left-hand frame:

```
<frameset rows="*,60" frameborder="0">
  <frameset cols="320,*" frameborder="0">
    <frame name="left" src="order-list.aspx?customerid=<% = strCustID %>"
           frameborder="0" />
    <frame name="right" src="order-detail.aspx"
           frameborder="0" />
  </frameset>
  <frame src="footer.htm" frameborder="0" scrolling="no"/>
</frameset>
```

The right-hand frame page, `order-detail.aspx`, doesn't require any query string, as it will not display any data until the user makes a selection in the list in the left-hand frame page. At the end of the frameset you can also see the narrow strip where we display our standard page footer.

As we mentioned earlier, we're assuming that the majority of the HTML 3.2-compliant browsers using this version of the application *also* have support for HTML frames. For those few that don't, we include this escape hatch which will take them to a more simple version of the application:

```
<noframes>
  <font size="2" face="Tahoma,Arial,sans-serif">
    This page requires HTML Frames support.
    As your browser does not support<br />
    frames, you should use the
    <b><a href="../mobile/default.aspx">simplified version</a></b>
    of the application instead.
  </font>
</noframes>
```

Getting the Data from the Session or Server

When the `view-orders.aspx` frameset page is loaded in response to a click in the list of customers on the previous page, it will load the page `order-list.aspx` into the left-hand frame – passing it the ID of the selected customer.

In the page `order-list.aspx`, we just need to collect the customer ID, use it to look up a list of orders for this customer, and display the list in the page. However, before we build that code, we need to think a little more about how we are going to *use* the data we extract from our database.

We're using a data access component that returns a `DataSet` containing the two tables required for displaying both the list of orders and the details of each one (the order lines). We've chosen to present this information in two `DataGrid` controls on two separate pages, though there is no reason why we couldn't use it in other ways as well. For example, we could use the relationship between the tables in the `DataSet` to display the data hierarchically if required.

But we're straying from the point. What we need to consider is how we cache and use the `DataSet` that our data access tier exposes in our user's session (something we already decided was a good idea and would usually offer a considerable performance boost). Getting data into the session is easy – we just specify the key we want to refer to it by later:

```
Session("key-name") = MyDataSetObject
```

To get it back out, we use the key and cast (convert) the returned object to the correct data type:

```
MyDataSetObject = CType(Session("key-name"), DataSet)
```

However, in our application we are using this data in two pages concurrently, so we need to be sure that it is appropriate for both of these pages. We only want to fetch and store one copy of it. In fact, the second page (`order-detail.aspx`, which we'll look at shortly) has an extra requirement for the data. Each "detail" row contains columns for the product, quantity, price and discount, but we would *also* like to display the "line total" for each row as well.

We *could* calculate this "line total" as we display the data, but it makes a lot more sense to make it part of the data itself. This way we only calculate the values once, and it also makes binding the data to the `DataGrid` control for display much easier. Even better, we can actually do it without having to calculate the totals ourselves – we can let the `DataSet` do it for us.

The GetDataSetFromSessionOrServer Function

In the page `order-list.aspx`, we have a function `GetDataSetFromSessionOrServer` that is responsible for returning the order details as a `DataSet` for a specified customer. It first attempts to fetch the `DataSet` from the session using a key that is unique for this version of the application. If the `DataSet` is not stored in the session at this point, the variable `objDataSet` will be empty, so we need to go off to our data access tier and fetch it from the database in this case:

`order-list.aspx`

```
Function GetDataSetFromSessionOrServer(strCustID As String) As DataSet
'gets a DataSet containing all orders for this customer

  Try
    Dim objDataSet As DataSet

    'try and get DataSet from user's Session
    objDataSet = CType(Session("4923HTMLOrdersDataSet"), DataSet)

    If objDataSet Is Nothing Then   'not in Session
      'get connection string from connect-strings.ascx user control
      Dim strConnect As String
      strConnect = ctlConnectStrings.OLEDBConnectionString

      'create an instance of the data access component
      Dim objOrderList As New Wrox4923Orders.OrderList(strConnect)

      'call the method to return the data as a DataSet
      objDataSet = objOrderList.GetOrdersByCustomerDataSet(strCustID)
      ...
```

Now we can manipulate the `DataSet` to make sure it is compatible with the `order-detail.aspx` page. We simply add a new column to the `OrderLines` table in the `DataSet`, and specify an expression for this column that will calculate the line total for each row automatically:

```
      ...
      'add a column containing the total value of each line
      Dim objLinesTable As DataTable = objDataSet.Tables("OrderLines")
      Dim objColumn As DataColumn
      objColumn = objLinesTable.Columns.Add("LineTotal", _
                                 System.Type.GetType("System.Double"))
      objColumn.Expression = "[Quantity] * ([UnitPrice] - ([UnitPrice]" _
                           & " * [Discount]))"
      ...
```

Having done that, we can store the `DataSet` in our user's session, and then return it to the calling routine. If there is an error, we display a message in the page and return `Nothing`:

```
        ...
        'save DataSet in Session for next order inquiry
        Session("4923HTMLOrdersDataSet") = objDataSet

      End If

      Return objDataSet

    Catch objErr As Exception
      'there was an error and no data will be returned
      lblMessage.Text = "* ERROR: No data returned. " & objErr.Message

      Return Nothing
    End Try
  End Function
```

The next time the user submits a request that calls this function the `DataSet` will be extracted not from the database, but from the session. (Of course, this assumes the session has not timed out.) The `DataSet` will already have the extra `LineTotal` column, so we can just return it "as is".

The DataGrid to Display the Order List

Now we've got our `DataSet`, we can think about how we'll display it in the page. We've used a `DataGrid` server control for this, specifying our own custom column layout as in the previous example page. However, in this page we have also turned off **ViewState** support, so that the values in the `DataGrid` are not persisted across postbacks:

```
<%@Page Language="VB" EnableViewState="False" %>
```

This means that there will be no `hidden`-type control in the page to store the values of the `DataGrid` (this is how ViewState is persisted by default), so the page will be smaller. We don't need ViewState to be supported, as we aren't planning to do postbacks directly from this page.

Note that we could not do this in the previous page. There, we had enabled **paging** in our `DataGrid`, and this requires ViewState support in order to work. We aren't using paging in either of the pages that display the order details. Of course, if you could potentially have a great many orders per customer you may decide to use paging, in which case you will need to enable ViewState support for the page.

Our code also defines a single `Page`-level (global) variable that we'll use to hold the current customer ID:

```
'page-level variable accessed from more than one routine
Dim strCustID As String = ""
```

So, back to the HTML; the following is the declaration of our `DataGrid` control and the two `Label` controls that provide ancillary information for the user:

```
<form runat="server">
  <!-- label to display customer ID -->
  <asp:Label id="lblStatus" runat="server" /><p />

  <!-- DataGrid control to display matching orders -->
  <asp:DataGrid id="dgrOrders" runat="server"
       AutoGenerateColumns="False"
       CellPadding="5"
       GridLines="Vertical"
```

```
        HeaderStyle-BackColor="#c0c0c0"
        ItemStyle-BackColor="#ffffff"
        AlternatingItemStyle-BackColor="#eeeeee">

    <Columns>

      <asp:TemplateColumn HeaderText="<b>Order ID</b>"
                          ItemStyle-BackColor="#add8e6"
                          ItemStyle-HorizontalAlign="center">
        <ItemTemplate>
          <asp:Hyperlink Text='<%# Container.DataItem("OrderID")%>'
               NavigateUrl='<%# DataBinder.Eval(Container.DataItem, "OrderID", _
                 "order-detail.aspx?customerid=" & strCustID & "&orderid={0}") %>'
               Target="right" runat="server" />
        </ItemTemplate>
      </asp:TemplateColumn>

      <asp:BoundColumn HeaderText="<b>Order Date</b>"
                       HeaderStyle-HorizontalAlign="center"
                       ItemStyle-HorizontalAlign="center"
                       DataField="OrderDate" DataFormatString="{0:d}" />

      <asp:BoundColumn HeaderText="<b>Shipped</b>"
                       HeaderStyle-HorizontalAlign="center"
                       ItemStyle-HorizontalAlign="center"
                       DataField="ShippedDate" DataFormatString="{0:d}" />
    </Columns>
  </asp:DataGrid><p />

  <!-- label to display interactive messages -->
  <asp:Label id="lblMessage" runat="server" />
</form>
```

Again we are generating the columns in the grid ourselves (we set the `AutoGenerateColumns` property to `False`). The only complicated column is the first one – an `asp:TemplateColumn`. This is used because it allows us to implement the content using a **template**, and a template allows a more flexible approach to setting the content than any other technique that we've used in the previous example.

Using a "Template" Column

The reason we need the extra flexibility is because we want to use a hyperlink in the column, but we also want to specify the `target` attribute of this hyperlink so that the target page is loaded into the other frame of our frameset rather than the current frame. If we use a standard hyperlink column (as in the previous example), we can't set the `target` attribute in our declaration of the `DataGrid`.

The template content we're using is shown again below. It defines an `asp:Hyperlink`, which will output an `<a>` element in the table, and you can see that we've specified the right-hand frame (rather unimaginatively named `right` in the frameset page) as the `Target` attribute. The text of the hyperlink is the value of the `OrderID` column in the `DataSet` table that is bound to the `DataGrid` control. The syntax used is that of server-side data binding, and indicates we want this specific column (`DataItem`) from the data source that is bound to the control (the `Container`):

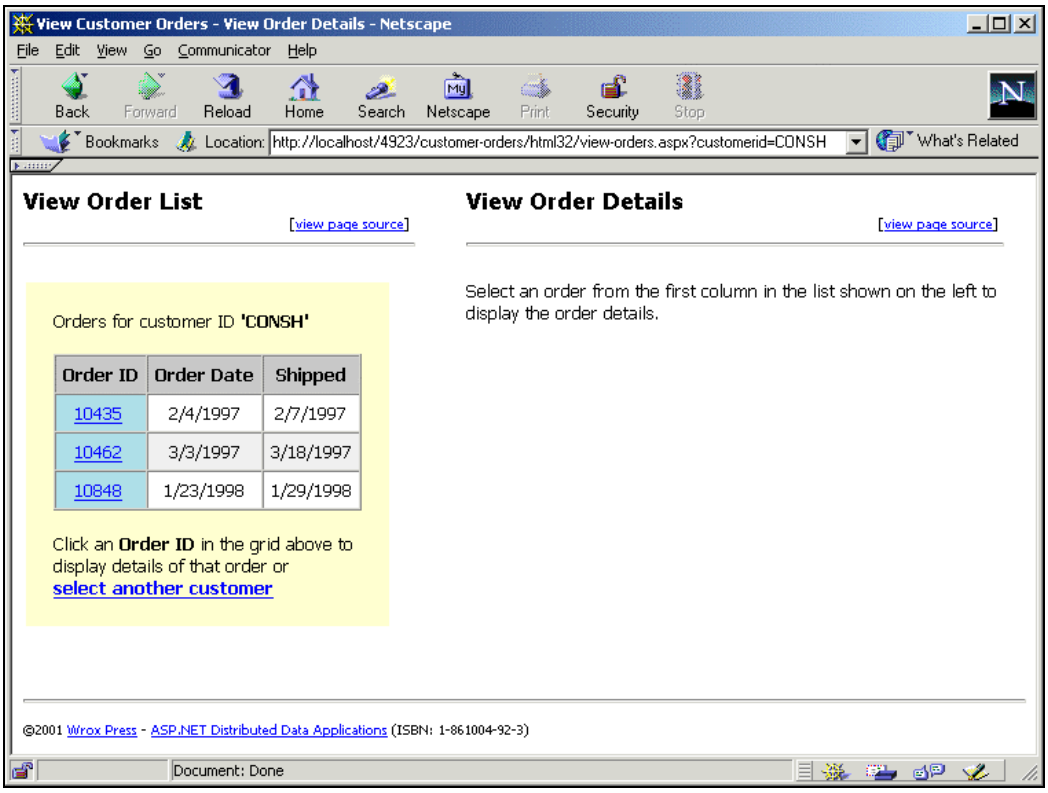
```
        <ItemTemplate>
          <asp:Hyperlink Text='<%# Container.DataItem("OrderID")%>'
               NavigateUrl='<%# DataBinder.Eval(Container.DataItem, "OrderID", _
                  "order-detail.aspx?customerid=" & strCustID & "&orderid={0}") %>'
               Target="right" runat="server" />
        </ItemTemplate>
```

The `NavigateUrl` attribute of the hyperlink specifies the `href` attribute that will be added to the output `<a>` element. In this we are using the `Eval` method of the `DataBinder` object (the object that carries out the binding) to specify the format of the string we want to be used for the `href`. You can see that the method takes the data source (`Container.DataItem`) as it's first parameter, the column name within the data source as it's second parameter, and the format string as the third parameter.

The great thing is that this is a "proper" method call (notice the underscore at the end of the line where the third parameter has wrapped to the next line), and it means we can use code to build the format string. We use the `Page`-level variable named `strCustID`, and the value of the order ID for the current row, so the `href` will be something like:

```
order-detail.aspx?customerid=CONSH&orderid=10462
```

So, in the `view-orders.aspx` page we can extract the customer ID from the query string and use it in our code. The next screenshot shows what the `view-orders.aspx` frameset page looks like when it first loads:



The Page_Load Event Handler

To display the list of orders shown in the previous screenshot, we execute some code in response to the `Page_Load` event. If it is not a postback (that is, if this is the first time the page has been loaded), we collect the customer ID from the query string and store it in the `Page`-level variable named `strCustID`. If a customer ID was not specified (perhaps because the page was loaded directly rather than from the previous `default.aspx` page), we display an error message – we need the customer ID to be able to look up their orders:

```
Sub Page_Load()

  If Not Page.IsPostBack Then
    strCustID = Request.QueryString("customerid")
    If (strCustID Is Nothing) Or (strCustID = "") Then
      'display error message
      lblMessage.Text = "* ERROR: no Customer ID provided.<br />" _
        & "You must <a href='default.aspx' target='_top'><b>select" _
        & " a customer</b></a> first."

    Else
      'display all orders for this customer
      ShowOrders()

    End If
  End If
End Sub
```

As long as we got a customer ID, we can call the `ShowOrders` routine located elsewhere in this page to display the list of orders for the specified customer.

Displaying the Order List

The `ShowOrders` routine is relatively simple. It just has to fetch the `DataSet` from the custom `GetDataSetFromSessionOrServer` function we looked at earlier and bind it to the `DataGrid` control. However, there are a couple of other issues to contend with. Recall that we are "sharing" this `DataSet` with the page in the other frame of the `<frameset>` (`order-detail.aspx`). As you'll see when we look at that page, it applies a filter to the `Orders` table in the `DataSet` in order to extract and display details of the shipping address and the carrier that is delivering it.

If the user refreshes this page, the row filter will prevent all the other orders for this customer from being listed, so we take the precaution of removing it by setting the `RowFilter` property of the `DefaultView` of the table to an empty string:

```
Sub ShowOrders()
'display list of all orders for this customer in DataGrid control

  'get DataSet using function elsewhere in this page
  Dim objDataSet As DataSet = GetDataSetFromSessionOrServer(strCustID)

  'remove any existing filter from table DefaultView
  'otherwise refreshing page in browser shows only one row
  objDataSet.Tables("Orders").DefaultView.RowFilter = ""
  ...
```

The only other issue is to check whether we actually have any orders for this customer to display, and show an appropriate message in the `Label` control on this page:

```
    ...
    'check if any orders were found for this customer
    If objDataSet.Tables("Orders").Rows.Count > 0 Then
      'diplay heading above DataGrid
      lblStatus.Text = "Orders for customer ID <b>'" & strCustID & "'</b>"

      'set DataSource, bind the DataGrid and display status message
      dgrOrders.DataSource = objDataSet.Tables("Orders")
      dgrOrders.DataBind()
      lblMessage.Text = "Click an <b>Order ID</b> in the grid above to" _
                      & "<br /> display details of that order or "

    Else
      lblMessage.Text = "No orders found for this customer ..."

    End If

    lblMessage.Text &= "<br /><a href='default.aspx' target='_top'>" _
                     & "<b>select another customer</b></a>"
  End Sub
```

Displaying the Order Details

The final page in our HTML version of the application (`order-detail.aspx`) is displayed in the right-hand frame of the `view-orders.aspx` page. Most of the techniques are similar to the previous two pages we've looked at. The page contains an ASP.NET `DataGrid` control to display the list of order lines for the selected order, plus a couple of `Label` controls to display the shipping details for the order and the total value.

The Declaration of the DataGrid and Labels

The next listing shows the HTML for this page. You can see the `Label` controls and the `DataGrid`. As in previous pages, we have "turned off" the `AutoGenerateColumns` property of the `DataGrid`. We just have a series of standard `asp:BoundColumn` elements in this page – there is one for each column in the `OrderLines` table of our `DataSet` (including the calculated column that we added to the `DataSet` when we stored it in the session):

`order-detail.aspx`

```
  <form runat="server">

    <!-- label to display order details -->
    <asp:Label id="lblMessage" runat="server" /><p />

    <!-- DataGrid control to display order lines -->
    <asp:DataGrid id="dgrOrders" runat="server"
         AutoGenerateColumns="False"
         CellPadding="5"
         GridLines="Vertical"
```

```
        HeaderStyle-BackColor="#c0c0c0"
        AlternatingItemStyle-BackColor="#eeeeee">

      <Columns>
        <asp:BoundColumn HeaderText="<b>Qty</b>"
                         HeaderStyle-HorizontalAlign="center"
                         ItemStyle-HorizontalAlign="center"
                         DataField="Quantity" />
        <asp:BoundColumn HeaderText="<b>Product</b>"
                         HeaderStyle-HorizontalAlign="center"
                         DataField="ProductName" />
        <asp:BoundColumn HeaderText="<b>Packs</b>"
                         HeaderStyle-HorizontalAlign="center"
                         DataField="QuantityPerUnit" />
        <asp:BoundColumn HeaderText="<b>Each</b>"
                         HeaderStyle-HorizontalAlign="center"
                         ItemStyle-HorizontalAlign="right"
                         DataField="UnitPrice" DataFormatString="${0:N2}" />
        <asp:BoundColumn HeaderText="<b>Discount</b>"
                         HeaderStyle-HorizontalAlign="center"
                         ItemStyle-HorizontalAlign="right"
                         DataField="Discount" DataFormatString="{0:P}" />
        <asp:BoundColumn HeaderText="<b>Total</b>"
                         HeaderStyle-HorizontalAlign="center"
                         ItemStyle-HorizontalAlign="right"
                         DataField="LineTotal" DataFormatString="${0:N2}" />
      </Columns>

    </asp:DataGrid><p />

    <!-- label to display order total -->
    <asp:Label id="lblTotal" runat="server" /><p />

  </form>
```

You'll notice that we specify values for the `DataFormatString` property of the last three columns to display the content as currency or as a percentage format. We used the standard format specifier `P` to format the percentage column. For the other two we used the currency character "$" followed by the numeric value formatted to two decimal places (`N2`). You might be tempted to use the standard currency format specifier, `C`, here – but this would cause the page to display a currency character that depends on the locale settings of the server. If your database holds the price in US dollars, you probably don't want to display that number with any other currency symbol!

The ASP.NET Code for the Page

Our code declares two `Page`-level variables that will contain the customer ID and the order ID for the selected order:

```
  <script language="VB" runat="server">

    'page-level variables accessed from more than one routine
    Dim strCustID As String = ""
    Dim strOrderID As String = ""
```

This page will probably be loaded several times as the user browses through the list of orders displayed in the left-hand frame of the `view-orders.aspx` page, and each time they select an order the current page (`order-detail.aspx`) is reloaded with the customer ID and the order ID in the query string. So, in the `Page_Load` event we can extract the customer ID and the order ID from the query string as the page loads each time:

```
Sub Page_Load()

  If Not Page.IsPostBack Then
    strCustID = Request.QueryString("customerid")
    strOrderID = Request.QueryString("orderid")
    If (strOrderID Is Nothing) Or (strOrderID = "") _
    Or (strCustID Is Nothing) Or (strCustID = "") Then
      'display help message
      lblMessage.Text = "Select an order from the first column " _
        & "in the list shown on the left to display the order details."

    Else
      'display order details for this order
      ShowOrderLines()

    End If
  End If
End Sub
```

Notice that we display a simple "help" message if there is no customer ID or order ID. When the frameset containing this page is loaded the first time, there will be no "current order" (we saw this in the previous screenshot).

When the page is loaded *with* customer ID or order ID values in the query string, we call the `ShowOrderLines` routine elsewhere in our page to display the details of the selected order.

Getting the DataSet from the Server or Session

We'll need to be able to extract the data for this page from the `DataSet` created by the data access tier of our application. If the client supports sessions, this will already be in the user's session – placed there by the `order-list.aspx` page that we loaded into the left-hand frame. This `DataSet` also has the calculated `LineTotal` column we need in this page.

However, if the client does not support sessions, we have to hit the data access component again to extract it from the database, and add the calculated column. In fact, the function we need is exactly the same as the one we used in the `order-list.aspx` page, so we just include the same function (`GetDataSetFromSessionOrServer`) in this page as well:

```
Function GetDataSetFromSessionOrServer(strCustID As String) As DataSet
'gets a DataSet containing all orders for this customer
... exactly the same as in order-list.aspx page ...
... and it uses the same Session-cached DataSet ...
End Function
```

The ShowOrderLines Routine

The `ShowOrderLines` routine is a little more complex than the equivalent function we used in previous pages, as it has to display data from *both* of the tables in the `DataSet`. The shipping details come from the `Orders` table, and the "detail line" rows come from the `OrderLines` table.

We start by calling the routine to get the `DataSet` from the session or direct from the data tier:

```
Sub ShowOrderLines()
'display all the order line details for this order in DataGrid control

  'get DataSet using function elsewhere in this page
  Dim objDataSet As DataSet = GetDataSetFromSessionOrServer(strCustID)
  ...
```

Next we get a reference to the `DefaultView` of the `Orders` table, and apply a filter so that the *only* row exposed is the one for the current order. (This is the same filter that we have to remove in `order-list.aspx`, so that *all* order rows are shown whenever that page is refreshed. We discussed that part earlier in the chapter.) We also get a reference to the `DefaultView` of the `OrderLines` table, and apply the same filter to that `DataView`, too:

```
  ...
  'create filtered DataView from Orders table in DataSet
  Dim objOrderView As DataView = objDataSet.Tables("Orders").DefaultView
  objOrderView.RowFilter = "OrderID = " & strOrderID

  'create filtered DataView from OrderLines table
  Dim objLinesView As DataView = objDataSet.Tables("OrderLines").DefaultView
  objLinesView.RowFilter = "OrderID = " & strOrderID
  ...
```

Calculating the Order Total and Displaying the Details

Now we can calculate the total value of the order by summing the values in the `LineTotal` calculated column that we've already added to the `DataSet`:

```
  ...
  'calculate total value of order
  Dim dblTotal As Double = 0
  Dim objDataRowView As DataRowView
  For Each objDataRowView In objLinesView
    dblTotal += objDataRowView("LineTotal")
  Next
  ...
```

Next we check that there is at least one order line for this order (this should always be the case), and display the shipping details – selecting them from the appropriate table as we go. Notice the use of the VB.NET `IsDbNull` function to test for a `null` value for the shipping date. As well as providing a more informative output, this prevents an error arising from trying to format a `null` value:

```
    ...
    'check that there are some matching order lines
    If objLinesView.Count > 0 Then

      'display the shipping details in Message Label
      lblMessage.Text = "Order ID:" & strOrderID _
        & "Customer Name:" & objOrderView.Item(0)("ShipName") & "<br />" _
        & "Delivery Address:" & objOrderView.Item(0)("OrderAddress") _
        & "Ordered: " & FormatDateTime(objOrderView.Item(0)("OrderDate"))
      If IsDbNull(objOrderView.Item(0)("ShippedDate")) Then
        lblMessage.Text &= "Awaiting shipping"
      Else
        lblMessage.Text &= "Shipped: " _
          & FormatDateTime(objOrderView.Item(0)("ShippedDate"))
      End If
      lblMessage.Text &= " via " & objOrderView.Item(0)("CompanyName")
      ...
```

The final steps are to bind the `DefaultView` of the `OrderLines` table to our `DataGrid` control and display the total order value. Again, we have specified the currency symbol explicitly rather than depending on the local settings of the server. If there happen to be no order lines for this order, we just display a message to this effect instead:

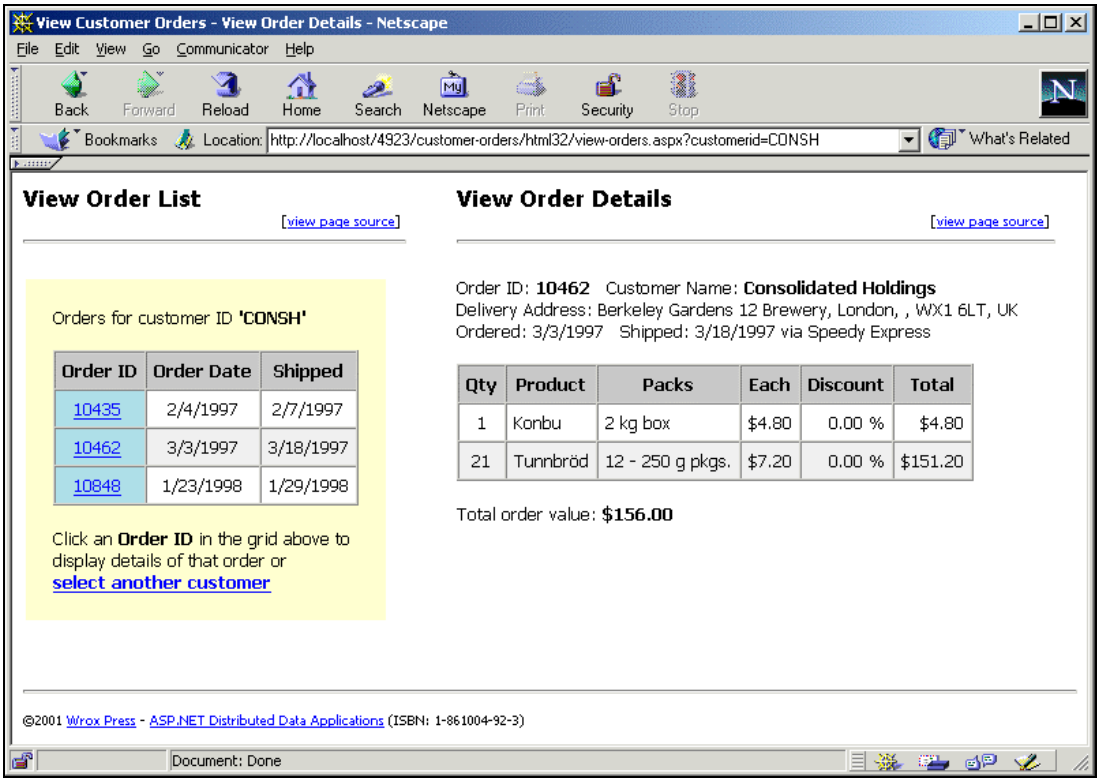
```
      ...
      'set DataSource and bind the DataGrid
      dgrOrders.DataSource = objLinesView
      dgrOrders.DataBind()

      'display the total value of the order
      lblTotal.Text = "Total order value:" & dblTotal.ToString("$#,##0.00")

    Else
      lblMessage.Text = "No order lines found for this order..."

    End If
  End Sub
```

The next screenshot shows the results for order number 10462 for customer Consolidated Holdings:

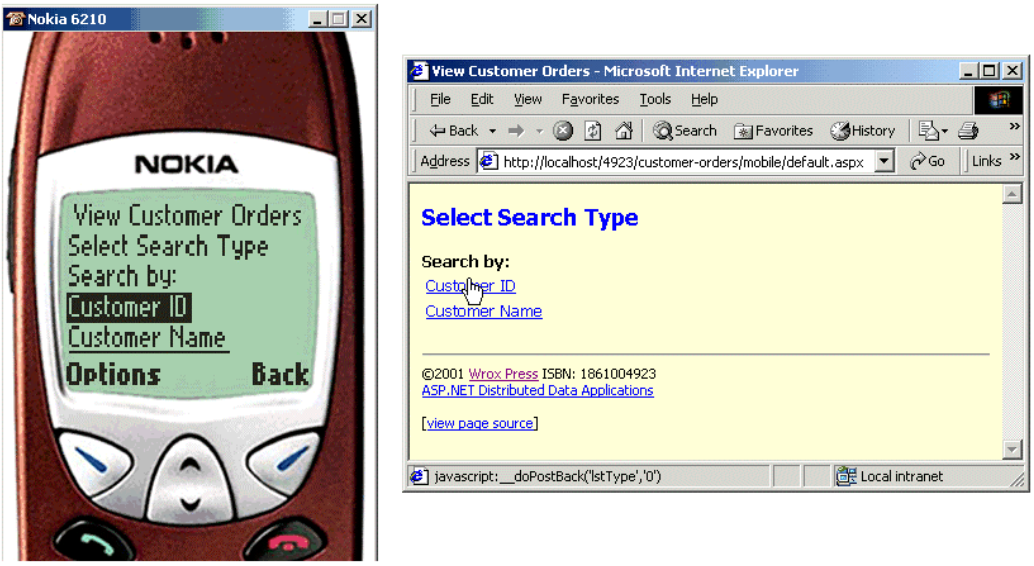


The Order List Application – the "Mobile" Version

The other "down-level" client we are looking at in this chapter is the small-screen or mobile device, such as a PDA or a cellular phone. While we can't hope to offer the same kind of usability for these devices as we can in a standard Web browser, their increasing use for mobile Internet access means that we can benefit from supporting them as well as we possibly can.

Our version of the application for mobile devices gives the same functionality as the browser-based equivalents, but tries to minimize the effects of the small screen and lack of a keyboard and mouse for input. We'll build it with controls from the Microsoft Mobile Internet Toolkit (MIT), so it will also work fine in a PDA or hand-held device that supports HTML rather than the more usual WML of the cellular phone.

The next two screenshots show the opening page of the application in a cellular phone emulator, and in an HTML browser (actually a resized Internet Explorer window). To get this, browse to http://localhost/4923/customer-orders/default.aspx, and the application should redirect you to the http://localhost/4923/customer-orders/mobile/default.aspx page automatically:



The emulator we're using here is part of the ***Nokia Mobile Internet Toolkit****. Various versions are available, and you can read about these and download them from:* http://forum.nokia.com/.

You can obtain the ***Microsoft Mobile Internet Toolkit*** *from:*
http://msdn.microsoft.com/vstudio/nextgen/device/mitdefault.asp

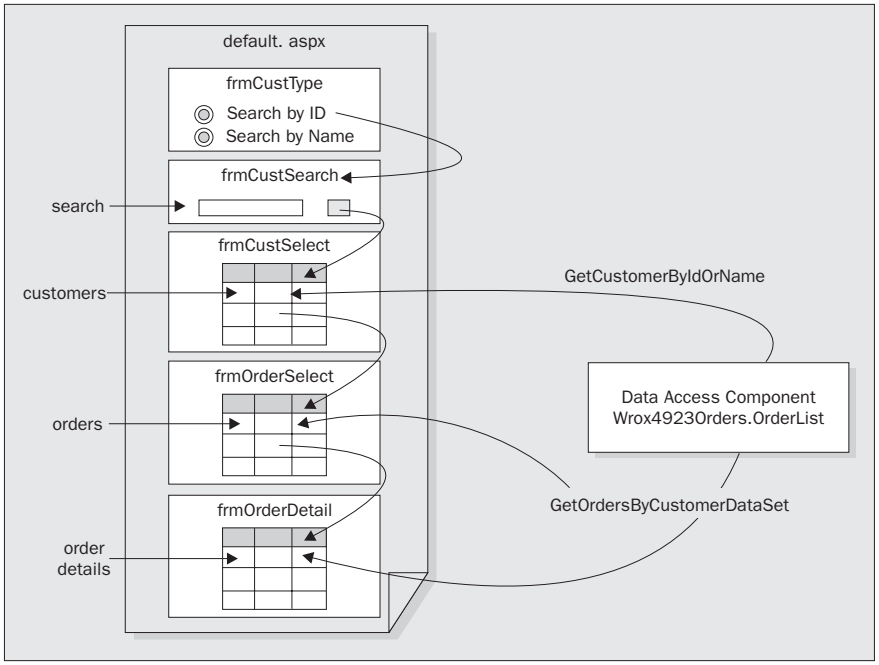
The Outline Process

The way we structure pages for the MIT is very different from the way this is done in a normal HTML-targeted ASP.NET page. Devices that support WML rather than HTML have the concept of multiple "screens", often called **cards** in WML-speak, for each page. And, not surprisingly, the page itself is often referred to as a **deck** (there is a huge opportunity for bad puns when developing for mobile devices).

The theory is that devices like cellular phones have very narrow bandwidth and high latency connections (at least at the moment), and so sending several screens in one go provides a more interactive and responsive user interface. However, if each screen is dynamically generated on the server, it means that a server connection and postback is still required for each screen – but the principle is there and is useful if you decide to use client-side WMLScript to manipulate the pages.

To implement the requirement for multiple screens in one page, the MIT provides a sub-classed version of the `Page` object that accepts multiple server-side `<form>` elements (the standard `Page` object will not). Each `<form>` becomes a single screen or card, and we can activate (or display) the appropriate ones within our code as the user interacts with the page.

So, to implement our application requirements we have a single page, `default.aspx` (in the `mobile` subfolder of the application), which contains five `<form>` elements that implement the five screens:



The first two screens collect the search details from the user, and the third screen displays a list of matching customers using the `GetCustomerByIdOrName` method of our application's standard data access component. Selecting a customer then displays a list of all their orders in the next screen, using the `GetOrdersByCustomerDataSet` method of the same component. Finally, the fifth screen displays the details of the order that the user selected in the previous screen – again using the `GetOrdersByCustomerDataSet` method.

So the data access processes are very similar to those we used in the HTML version of the example. We also take advantage of sessions in exactly the same way, and the application still works if sessions are not supported.

Every screen (except the first) allows the user to jump backwards through the process using either the hyperlinks at the bottom of each screen, or with the "soft keys" that are implemented on most mobile devices. In the final screen, they can go back to the previous screen (the list of orders for the selected customer) and select a different order.

Fetching Data from the Server

Before we look at the server controls and the page structure for this example, we'll examine the code that interacts with our data access tier. To provide the kind of compact display we need for small screen devices, we have to "massage" the data a little in our application's middle tier. In our case, the code for this is actually part of this page, though there is no reason why we couldn't separate it out into a component if we needed the same functionality for other versions of the application as well. In later chapters we actually do this with the middle tier so that we can reuse it in more than one version.

Fetching the Customer List

Getting the list of customers is a simple process. The `GetCustomerDataSetFromServer` function takes care of all this – it's almost identical to the `GetDataSetFromServer` function found in the HTML version.

As in the HTML version of the application, we first remove any existing `DataSet` containing order details from the user's session before we fetch the data for the customer they selected this time. The partial or full customer ID or name are passed to this function in the two parameters in the same way as we saw in earlier examples:

default.aspx

```
Function GetCustomerDataSetFromServer(strCustID As String, _
                                      strCustName As String) As DataSet

  'remove any existing "Orders" DataSet from the user's session
  'as we're searching now for a different customer
  Session("4923MobileShowOrdersDataSet") = Nothing
  ...
```

Now we can collect the connection string from our custom user control, instantiate the data access component, and fetch and return the `DataSet` it exposes:

```
  ...
  'get connection string from connect-strings.ascx user control
  Dim strConnect As String
  strConnect = ctlConnectStrings.OLEDBConnectionString

  Try
    'create an instance of the data access component
    Dim objOrderList As New Wrox4923Orders.OrderList(strConnect)

    'call the method to return the data as a DataSet
    Return objOrderList.GetCustomerByIdOrName(strCustID, strCustName)

  Catch objErr As Exception
    'there was an error and no data will be returned
    Return Nothing

  End Try
End Function
```

Fetching the Orders Data

Fetching the order data for a specific customer is rather more complex. It's handled by the `GetOrderDataSetFromSessionOrServer` method (which is similar to the `GetDataSetFromSessionOrServer` function in both order pages of the HTML version, but as we'll see here, has some important additional coding).

Here, as in the HTML version, we'll get back a `DataSet` containing two tables – an `Orders` table containing the details of every order for this customer, and an `OrderLines` table containing the order rows for all these orders. We also cache this `DataSet` in the user's session when we first fetch it from the data access component, so we can reuse this on subsequent calls to the function:

```
Function GetOrderDataSetFromSessionOrServer(strCustID As String) As DataSet

  Dim objDataSet As DataSet

  Try
    'try and get DataSet from user's session
    objDataSet = CType(Session("4923MobileShowOrdersDataSet"), DataSet)

    If objDataSet Is Nothing Then   'not in session

      'get connection string from connect-strings.ascx user control
      Dim strConnect As String
      strConnect = ctlConnectStrings.OLEDBConnectionString

      'create an instance of the data access component
      Dim objOrderList As New Wrox4923Orders.OrderList(strConnect)

      'call the method to return the data as a DataSet
      objDataSet = objOrderList.GetOrdersByCustomerDataSet(strCustID)
      ...
```

Massaging the DataSet

At this point in the code, we know that the `DataSet` was not in the session, and we have extracted it from our data access tier. Before we cache it in the session, we need (as in the HTML version) to makes some changes to it. We aren't using fancy UI controls like the `DataGrid` in this version of the application. Instead, we want to provide columns that contain *all* the information in a compact format ready for display.

For the `Orders` table, we add a column named `DisplayCol` that contains the concatenated values for the order ID and the order date. This will be used for the list of orders from which the user can choose:

```
      ...
      'get a reference to the "Orders" table in the DataSet
      Dim objTable As DataTable = objDataSet.Tables("Orders")

      'add column containing order ID and date as one string value
      Dim objColumn As DataColumn
      objColumn = objTable.Columns.Add("DisplayCol", _
                           System.Type.GetType("System.String"))

      'fill column for each row with "#{order number} - {order date}"
      Dim objRow As DataRow
      Dim objDate As DateTime
      For Each objRow In objTable.Rows
        objDate = objRow("OrderDate")
        objRow("DisplayCol") = "#" & objRow("OrderID") & " - " _
                             & objDate.ToString("d")
      Next
      ...
```

For the `OrderLines` table, we want to provide a column that concatenates the quantity, product name, pack details, unit price, discount (if any) and the line total. We go about this by first adding a calculated column to the table in the same way as we did for the HTML version of the application. Then we can use this and the other columns values to build the content for the new `DisplayCol` column in this table:

```
        ...
        'get a reference to the "OrderLines" table in the DataSet
        objTable = objDataSet.Tables("OrderLines")

        'add column containing total value of each line
        objColumn = objTable.Columns.Add("LineTotal", _
                                System.Type.GetType("System.Double"))
        objColumn.Expression = "[Quantity] * ([UnitPrice] - ([UnitPrice]" _
                                & " * [Discount]))"

        'add column containing the order details as one string value
        objColumn = objTable.Columns.Add("DisplayCol", _
                                System.Type.GetType("System.String"))

        'fill with "qty x product (pack) @ price less discount = total"
        Dim strColValue As String
        For Each objRow In objTable.Rows
          Dim dblThisPrice as Double = objRow("UnitPrice")
          strColValue = objRow("Quantity").ToString & " x " _
                      & objRow("ProductName") & " (" _
                      & objRow("QuantityPerUnit") & ") @ " _
                      & dblThisPrice.ToString("$#0.00")
          If objRow("Discount") > 0 Then
            dblThisPrice = objRow("Discount")
            strColValue &= " Less " & dblThisPrice.ToString("P")
          End If
          dblThisPrice = objRow("LineTotal")
          objRow("DisplayCol") = strColValue & " = " _
                               & dblThisPrice.ToString("$#0.00")
        Next
        ...
```

Now we can save this "massaged" `DataSet` in the user's session ready for use in the "Order List" and "Order Details" screens of the application. We finish up by return it to the calling routine. Of course, if there is a `DataSet` already in the session, we simply return this instead of re-creating it:

```
        ...
        'save DataSet in session for next order inquiry
        Session("4923MobileShowOrdersDataSet") = objDataSet

      End If

      Return objDataSet   'return DataSet to calling routine

    Catch objErr As Exception

      'there was an error and no data will be returned
      Return Nothing

    End Try

  End Function
```

The Mobile Page Content

Next we'll look at the structure of the page itself, and see how each screen is built up. The page starts with the special `Page` directive for the MIT `MobilePage` object, and a `Register` directive to register the MIT controls. This is followed by the `Import` directives for our custom data access component, and the `System.Data` namespace we need to be able to use objects such as the `DataSet`, `DataColumn` and `DataRow` you saw in the previous code listings:

```
<%@Page Inherits="System.Web.UI.MobileControls.MobilePage" Language="VB"%>
<%@Register TagPrefix="Mobile" Namespace="System.Web.UI.MobileControls"
            Assembly="System.Web.Mobile"%>
<%@Import Namespace="OrderListData" %>
<%@Import Namespace="System.Data" %>

<%@ Register TagPrefix="wrox" TagName="connect"
             Src="..\..\global\connect-strings.ascx" %>

<%'-- insert connection string script --%>
<wrox:connect id="ctlConnectStrings" runat="server" />
```

At the end of this code, you can also see that we register and insert into the page our custom user control that exposes the connections strings for our database.

> *For more information on the syntax involved in writing Mobile Forms applications, have a look at* Beginning ASP.NET Mobile Controls *(Wrox, ISBN 1-861005-22-9).*

Adding Style to the Output

When we serve content to devices like cellular phones, we don't have much control over the style or appearance of the output. However, the MIT controls can also be used to create output for devices that support HTML, as you saw in the earlier screenshot. It's nice to be able to add at least some basic styling to these, and it's done through the `mobile:Stylesheet` control which is implemented by the MIT.

This requires us to set up **choices** using a `web.config` file, which specifies the devices that belong to a specific `<Choice>` filter category in a stylesheet (such as the one we'll examine in a moment). In our sample application, we have the following `web.config` file in the `mobile` folder (where this version of the application resides):

```
<configuration>
  <system.web>
    <deviceFilters>
      <filter name="IsIE" compare="browser" argument="IE" />
      <filter name="IsHTML32" compare="PreferredRenderingMIME"
              argument="text/html" />
    </deviceFilters>
  </system.web>
</configuration>
```

This sets up two filters. One is named `IsIE` and will only include Internet Explorer browsers. The other filter, named `IsHTML32`, will only include client devices that support HTML. The `compare` attribute is an entry from the list of properties exposed by either the `MobileCapabilities` or `BrowserCapabilities` objects, while the `argument` attribute is (obviously) the value for that property that we want to match.

The Mobile Stylesheet for our Application

The next listing shows the `mobile:Stylesheet` we use in this version of our application. You'll find it in `default.aspx`. You can see that it implements three styles in separate `<Style>` elements. The first is the style we'll apply to every screen in the application, and it specifies the content that we want to include at the top and bottom of the screen where the current client supports HTML (using the `IsHTML32` filter we defined in `web.config`):

```
<mobile:Stylesheet runat="server">

  <Style name="styPage">
    <DeviceSpecific>
      <Choice Filter="IsHTML32">
        <HeaderTemplate>
          <body bgcolor="#ffffacd"
                style="font-family:Tahoma, Arial, sans-serif; font-size:10pt">
          <font face="Tahoma,Arial,sans-serif" size="2">
        </HeaderTemplate>
        <FooterTemplate>
          <br /><hr /></font>
          <font face="Tahoma,Arial,sans-serif" size="1">
          <div style="font-family:Tahoma, Arial, sans-serif; font-size:8pt">
          &copy;2001 <a href="http://www.wrox.com/">
          ... etc ...
          </font></body>
        </FooterTemplate>
      </Choice>
    </DeviceSpecific>
  </Style>
  ...
```

This useful feature means that we can easily tailor the output for HTML devices to make the screens attractive, and to include extra content. None of this content will be sent to other clients such as cellular phones that expect WML – if this was not the case the client would report an error when it encountered a non-supported element.

The other two styles we include in the page are also specific to HTML clients. They simply define the style for page headings (large blue text) and lists (small black text):

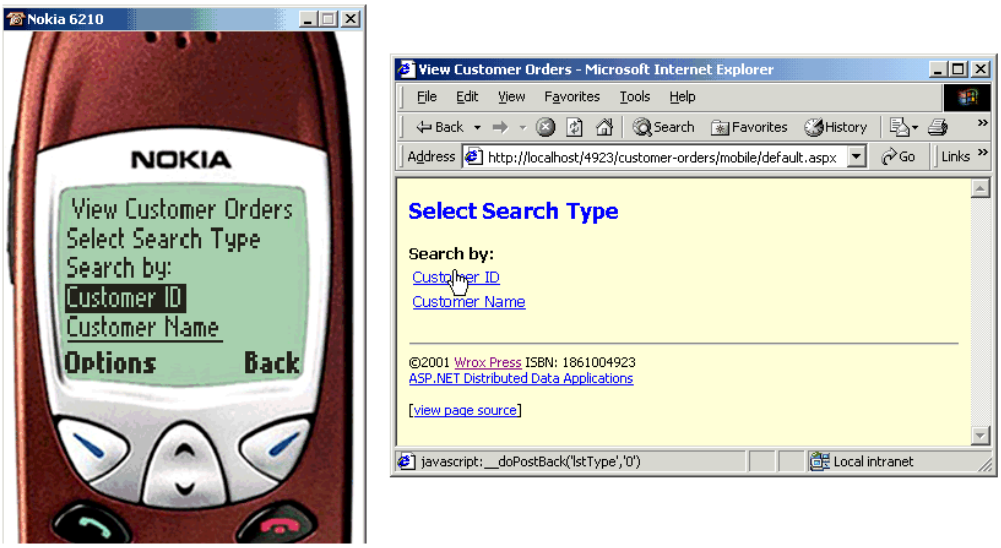
```
  ...
  <Style name="styHeading">
    <DeviceSpecific>
      <Choice Filter="IsHTML32" Font-Name="Tahoma,Arial,sans-serif"
              ForeColor="blue" Font-Size="large" Font-Bold="true" />
    </DeviceSpecific>
  </Style>

  <Style name="styListAndLink">
    <DeviceSpecific>
      <Choice Filter="IsHTML32" Font-Name="Tahoma,Arial,sans-serif"
              ForeColor="black" Font-Size="small" />
    </DeviceSpecific>
  </Style>

</mobile:Stylesheet>
```

Setting the Search Type

At last we're in a position to see some of the controls that create the visible parts of our application. The next screenshot shows the opening screen again, in both the Nokia emulator and an HTML client:



This output is created by the first `<form>` in our page. It is quite basic, using the MIT controls to create the text in two `Label` controls, and the two options within a `List` control. The form itself has the ID value `frmCustType`, and caption View Customer Orders. Notice how we specify the style to apply to each control – some use a reference to one of the styles in the stylesheet we showed earlier, and some use attributes of the control to set the appearance:

```
<mobile:Form id="frmCustType" title="View Customer Orders"
             styleReference="styPage" runat="server">
  <mobile:Label id="lblMsg1" styleReference="styHeading" runat="server">
    Select Search Type
  </mobile:Label><br /><br />
  <mobile:Label id="lblType" Font-Bold="true" runat="server">
    Search by:
  </mobile:Label>
  <mobile:List id="lstType" styleReference="styListAndLink"
               OnItemCommand="SetSearchType" runat="server">
    <Item Text="Customer ID" Value="CustID" />
    <Item Text="Customer Name" Value="CustName" />
  </mobile:List>
</mobile:Form>
```

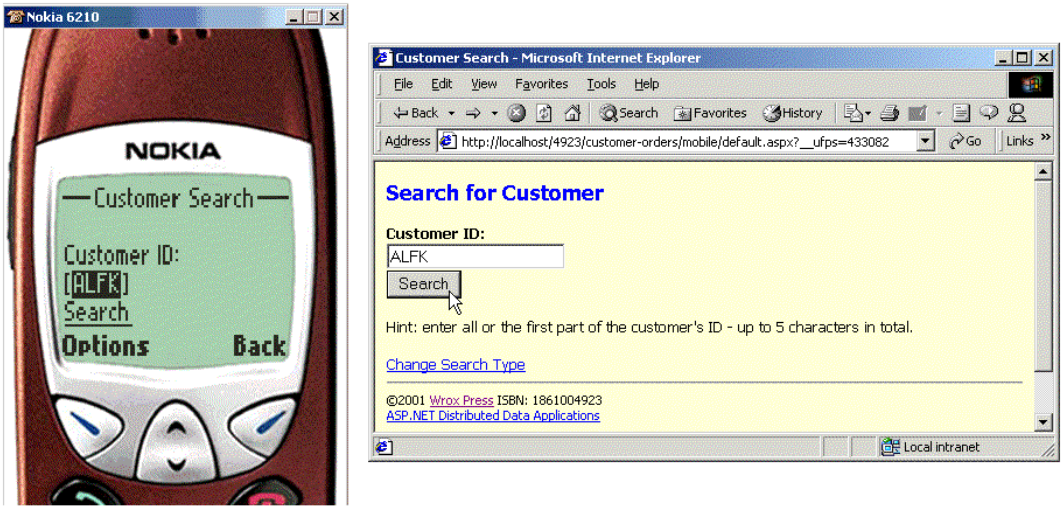
The `List` control has the `OnItemCommand` attribute set to point to an event handler named `SetSearchType`, which is located elsewhere in our page, and will be executed when the user selects one of the options. As in an HTML `<select>` list, we can specify values for the visible text of each item (using the `Text` attribute), and the value that is returned when the item is selected (using the `Value` attribute).

Specifying the Search String

To understand what the code that runs in response to a selection in the list does, we first need to look at the declaration for the next screen. The following listing implements the `frmCustSearch` form, which contains three `Label` controls, a `TextBox` control, a `Command` control and a `Link` control:

```
<mobile:Form id="frmCustSearch" title="Customer Search"
             styleReference="styPage" runat="server">
  <mobile:Label id="lblMsg2" styleReference="styHeading" runat="server">
    Search for Customer
  </mobile:Label><br /><br />
  <mobile:Label id="lblSearchType" Font-Bold="true" runat="server" />
  <mobile:TextBox id="txtSearchString" runat="server" />
  <mobile:Command id="cmdSearch" CommandName="Search" runat="server"
                  OnClick="GetCustomers" SoftKeyLabel="Search" Text="Search" />
  <mobile:Label id="lblSearchTips" runat="server" /><br />
  <mobile:Link NavigateUrl="#frmCustType" SoftKeyLabel="Back"
               Text="Change Search Type"
               styleReference="styListAndLink" runat="server" />
</mobile:Form>
```

Notice that the `Label` controls with `id` attributes `lblSearchType` and `lblSearchTips` have no text content at the moment. We'll be setting this text within our code by assigning it to the `Text` property of these controls. This means that we can change the caption for the text box and the Hint text displayed below it, depending on the selection the user made in the previous screen. For example, the next screenshot shows what this screen looks like when the user selected Search by Customer ID in the previous screen:



In the mobile device emulator, you can enter the search term through the Edit or Options soft key or similar.

You can also see from this that the `Command` control automatically creates a device-specific Search button – a soft key command in a phone, and an HTML button in the browser. You can also see (at least in the HTML output) the effect of the `Link` control we added at the end of the `<form>`. In the phone it creates a second soft key command (Back), while in the browser it creates a hyperlink. The declaration we used for this control was:

```
<mobile:Link NavigateUrl="#frmCustType" SoftKeyLabel="Back"
             Text="Change Search Type"
             styleReference="styListAndLink" runat="server" />
```

The `SoftKeyLabel` attribute sets the soft key in a phone, and the `NavigateUrl` attribute we use is the ID of the `<form>` we want to activate when the link is clicked.

The Code in the SetSearchType Event Handler

We haven't quite finished with this screen yet. We still need to look at how we set the text of the label controls, and how we activate the next screen. When the user makes a selection for the search type in the first screen, as we saw earlier, our event handler named `SetSearchType` is executed. This is listed next:

```
Sub SetSearchType(objSender As Object, objArgs As ListCommandEventArgs)
'set the correct text on page 2 (frmCustSearch) and display it

  'fill in Label text
  If objArgs.ListItem.Value = "CustName" Then
    lblSearchType.Text = "Customer Name:"
    lblSearchTips.Text = "Hint: enter any part of the customer... etc."
  Else
    lblSearchType.Text = "Customer ID:"
    lblSearchTips.Text = "Hint: enter all or the first part...etc."
  End If

  'display screen 2
  ActiveForm = frmCustSearch

End Sub
```

All this code has to do is set the text of the two `Label` controls in the second screen, then activate it so that it is displayed. Depending on the user's selection in the first screen, they will see different text in the two `Label` controls.

Showing the Customer List

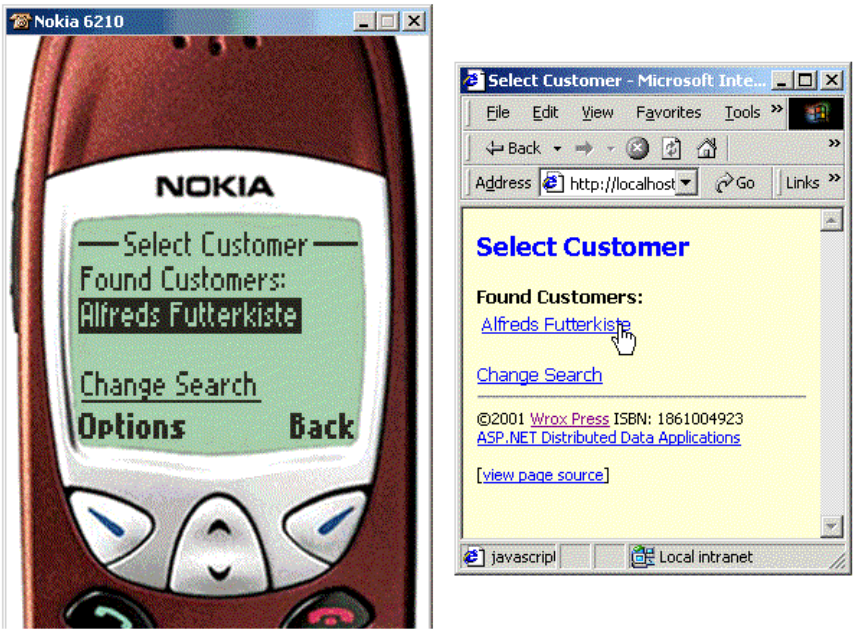
Once the user enters the search string into the text box on the second screen, they use the `Command` control in that screen (a soft key or a button, depending on the device) to get a list of customers that match their criteria. The declaration of that `Command` control looked like this:

```
<mobile:Command id="cmdSearch" CommandName="Search" runat="server"
                OnClick="GetCustomers" SoftKeyLabel="Search" Text="Search" />
```

So our event handler named `GetCustomers` will be executed when this `Command` is activated, and the user will see the list of customers in the next screen – which has the ID `frmCustSelect`. The declaration of this screen's `<form>` contains a heading `Label` control, an empty `Label` control with the `ID` property value `lblStatus1`, a `List` control and a `Link` control:

```
<mobile:Form id="frmCustSelect" title="Select Customer"
             styleReference="styPage" runat="server">
  <mobile:Label id="lblMsg3" styleReference="styHeading" runat="server">
    Select Customer
  </mobile:Label><br /><br />
  <mobile:Label id="lblStatus1" Font-Bold="true" runat="server" />
  <mobile:List id="lstCustomers" OnItemCommand="GetOrders"
               styleReference="styListAndLink" runat="server" /><br />
  <mobile:Link NavigateUrl="#frmCustType" SoftKeyLabel="Customer"
               Text="Change Search"
               styleReference="styListAndLink" runat="server" />
</mobile:Form>
```

The empty `Label` control is where we'll insert status information, and the `List` control will display the matching customers we found. The `Link` we use here is similar to that in the previous screen, allowing the user to go back and enter different search criteria. The next screenshot shows what this screen looks like after searching for customers with the ID "ALFK":



This example, like many others in this book, uses ViewState to retain the state of the forms. Because of bandwidth restrictions, mobile pages (unlike other ASP.NET pages) save their ViewState in the session.

If you get an error here (which declares that the page "requires session state that is no longer available"), it's related to this. You'll get the error if both (a) your client doesn't support cookies, and (b) your server is configured to disallow cookieless sessions (`cookieless="false"` in machine.config or web.config, as described in Chapter 4).

A quick workaround, in order to get the example working, is to find the `configuration/system.web/sessionState` *node in the* web.config *file in your application root, and set the* `cookieless` *attribute to* `true` *(don't forget to change it back again afterwards). If you're planning to use this technique in a real application, you may need to implement the mobile version as a separate application with its own* `web.config`*.*

The Code in the GetCustomers Event Handler

We can get a list of customers that match the search criteria by calling the function named `GetCustomerDataSetFromServer`, which is located elsewhere in this page. We described this function when we started looking at this version of the application. However, our event handler (named `GetCustomers`) has a few other things to do as well.

We hide the `List` control by setting its `Visible` property to `False` while we fetch the data, and only show it again if we actually find one or more matching customers. We also have to decide what type of search the user specified, by collecting the value from the text box on the previous screen. Notice how we can reference the values for controls on any `<form>` in our page directly. All the values are held in the ViewState of the page, as in an ordinary ASP.NET HTML page:

```
Sub GetCustomers(objSender As Object, objArgs As EventArgs)
'create and display list of matching customers in screen 3

  Dim strCustID As String = ""
  Dim strCustName As String = ""

  'hide the list control until we see if we get a result
  lstCustomers.Visible = "False"

  'check customer search type selection
  If Instr(lblSearchType.Text, "Name") > 0 Then
    strCustName = txtSearchString.Text
  Else
    strCustID = txtSearchString.Text.ToUpper()
  End If
  ...
```

Note that some cellular phones limit the size of a page, the volume of data that can be posted back from a `<form>`*, or require special encoding for the ViewState. You can use the* `MobileCapabilities` *object we saw in Chapter 4 to check these features for different devices – the relevant properties are* `RequiresSpecialViewStateEncoding` *and* `MaximumRenderedPageSize`*.*

You may instead decide to switch off ViewState altogether and manage the values you need to pass between screens with another technique such as a session variable or the query string. This will also circumvent the issue we mentioned a moment ago, caused by the tendency of mobile pages to store ViewState in the ASP.NET session.

Now we can go and fetch the data we need, and check that there was no error. If we don't get a `DataSet` back (because an error occurred), we set the text of the status label to show this. Otherwise we can see how many rows were returned by querying the `Count` property of the `Rows` collection for the table in the `DataSet`. If it's greater than zero, we can bind the `DataSet` table to the `List` control in this screen, make it visible again, and display a suitable status message:

```
    ...
    'get DataSet using function elsewhere in this page
    Dim objDataSet As DataSet = _
                      GetCustomerDataSetFromServer(strCustID, strCustName)

    'if there was an error display message
    If objDataSet Is Nothing Then
      lblStatus1.Text = "Error accessing database"

    Else
      'check how many matching customers were found
      Dim intRowsFound As Integer = objDataSet.Tables(0).Rows.Count

      If intRowsFound > 0 Then

        'bind the DataGrid and display status message
        lstCustomers.DataSource = objDataSet.Tables(0)
        lstCustomers.DataTextField = "CompanyName"
        lstCustomers.DataValueField = "CustomerID"
        lstCustomers.DataBind()
        lstCustomers.Visible = True
        lblStatus1.Text = "Found Customers:"
        ...
```

As you can see, the MIT `List` control (and a few other MIT controls) supports server-side data binding to the same data types as the ordinary ASP.NET list controls. Here we have specified the column to be used for the text displayed in the list (the customer name), and a different column (the customer ID) to be used for the value of each list item.

If no matching customers were found we set the text of the status label to inform the user. Finally we activate this screen to display the list of any matching customers:

```
        ...
      Else
        lblStatus1.Text = "No matching customers found in database ..."

      End If

    End If

    'display screen 3 containing the list
    ActiveForm = frmCustSelect

  End Sub
```

Listing the Orders for the Selected Customer

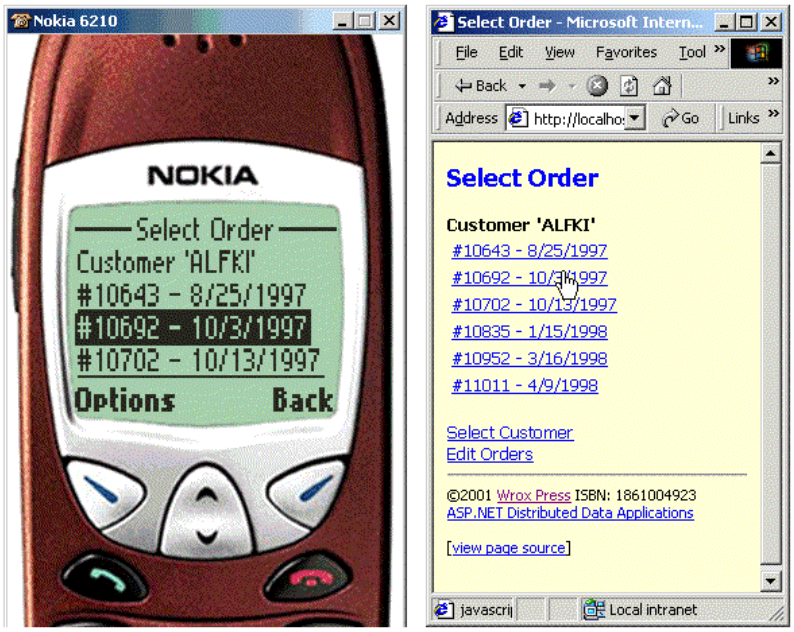
In the third screen (the one captioned Select Customer), we now (hopefully) have a list of customers that match the search criteria the user specified. When we declared the `List` control, we specified that selecting an item should execute an event handler named `GetOrders`:

```
<mobile:List id="lstCustomers" OnItemCommand="GetOrders"
             styleReference="styListAndLink" runat="server" /><br />
```

The `GetOrders` routine will fetch a list of orders for the selected customer and display them, much as the `GetCustomer` event handler code did to display a list of customers. We'll look at the code shortly. The following listing shows the declaration of the screen that will display this list of orders:

```
<mobile:Form id="frmOrderSelect" title="Select Order"
             styleReference="styPage" runat="server">
  <mobile:Label id="lblMsg4" styleReference="styHeading" runat="server">
    Select Order
  </mobile:Label><br /><br />
  <mobile:Label id="lblStatus2" Font-Bold="true" runat="server" />
  <mobile:List id="lstOrders" OnItemCommand="ShowOrderDetail"
               styleReference="styListAndLink" runat="server" /><br />
  <mobile:Link NavigateUrl="#frmCustType" SoftKeyLabel="Customer"
               Text="Select Customer"
               styleReference="styListAndLink" runat="server" />
  <mobile:Link id="lnkEditOrders"
               SoftKeyLabel="Edit Orders" Text="Edit Orders"
               styleReference="styListAndLink" runat="server" />
</mobile:Form>
```

The output generated by this `<form>` is shown in the next screenshot. The empty `Label` control with ID value `lblStatus2` has its text set by code in the page as it runs, and the `List` control is bound to the `DataSet` of orders returned by our data access tier. The two `Link` controls provide soft keys or hyperlinks (depending on the client type) so that the user can go back and select a different customer, or edit the orders for this customer. We'll be looking at the editing feature in a later chapter:



The Code in the GetOrders Event Handler

The `GetOrders` event handler itself is comparable in outline to the `ShowOrders` function in the `html32` version of the application. When the user selects a customer in the previous screen, we retrieve the order ID from the list control in that screen and use it in a call to the `GetOrderDataSetFromSessionOrServer` we looked at earlier. If we get back a `DataSet` (meaning that there was no error) we can then check to see if any orders were found, and display them by binding the `Orders` table in the `DataSet` to the `List` control in this screen. Notice that we bind to the new column named `DisplayCol` that we added to the `Orders` table within our `GetOrderDataSetFromSessionOrServer` function:

```
Sub GetOrders(objSender As Object, objArgs As ListCommandEventArgs)
'create a list of all orders for this customer on page 4

  'get CustomerID from selection made in List control on page 3
  Dim strCustID As String = objArgs.ListItem.Value

  'get DataSet using function elsewhere in this page
  Dim objDataSet As DataSet = GetOrderDataSetFromSessionOrServer(strCustID)

  'if there was an error display message
  If objDataSet Is Nothing Then
    lblStatus1.Text = "Error accessing database"

  Else
    'check if any orders were found for this customer
    If objDataSet.Tables("Orders").Rows.Count > 0 Then
      'diplay heading above List control
      lblStatus2.Text = "Customer '" & strCustID & "'"

      'set DataSource and bind the List control
      lstOrders.DataSource = objDataSet.Tables("Orders")
      lstOrders.DataTextField = "DisplayCol"
      lstOrders.DataValueField = "OrderID"
      lstOrders.DataBind()
      ...
```

We also have to set the `href` of the Edit Orders `Link` control at the bottom of the screen so that it contains the selected customer ID in the query string. This link opens a separate page, which allows the user to edit the orders of this customer (we'll look at this in Chapter 8). Then we can activate this screen to display it:

```
      ...
      'set URL for editing orders in Link control
      lnkEditOrders.NavigateUrl = _
                "../../update-orders/mobile/edit-orders.aspx" _
                & "?customerid=" & strCustID

    Else
      lblStatus2.Text = "No orders found for this customer ..."

    End If

  End If

  'display page 4 containing the list
  ActiveForm = frmOrderSelect

End Sub
```

Displaying the Order Details

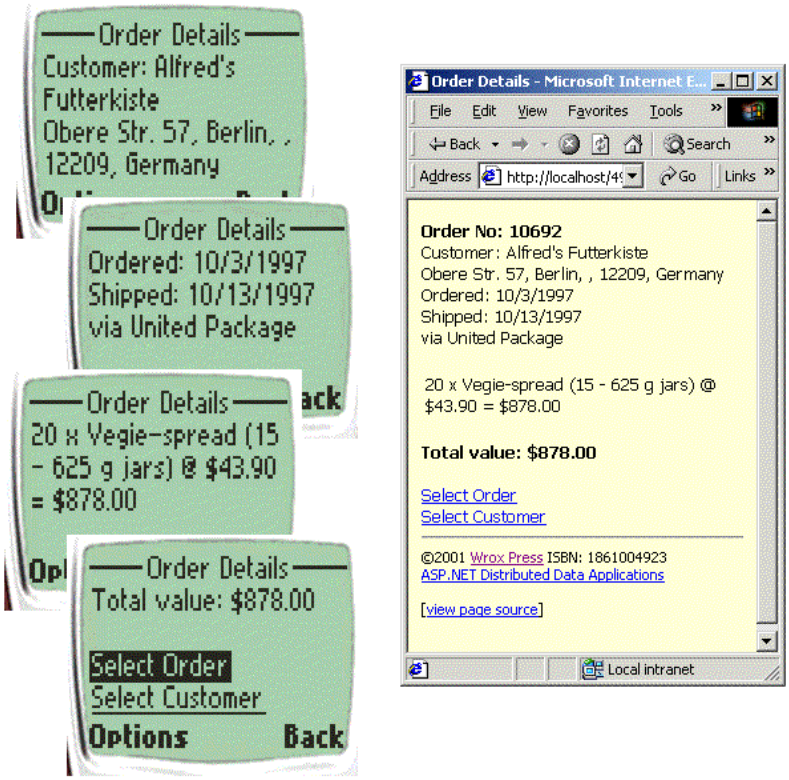
The final screen in this version of the application displays details of the order selected in the previous screen. This screen is activated when the user selects an order in the previous screen, where we have a `List` control that specifies the event handler named `ShowOrderDetail`:

```
<mobile:List id="lstOrders" OnItemCommand="ShowOrderDetail"
             styleReference="styListAndLink" runat="server" /><br />
```

The declaration of the Order Details screen is shown in the next listing. This time we have several `Label` controls that will contain details of the order itself, and a `List` control that we'll bind the list of order lines to. There are also two `Link` controls at the bottom of the screen that allow the user to select a different customer or a different order:

```
<mobile:Form id="frmOrderDetail" title="Order Details"
             styleReference="styPage" runat="server">
  <mobile:Label id="lblOrderNo" Font-Bold="true" runat="server" />
  <mobile:Label id="lblCustName" runat="server" />
  <mobile:Label id="lblAddress" runat="server" />
  <mobile:Label id="lblOrdered" runat="server" />
  <mobile:Label id="lblShipped" runat="server" />
  <mobile:Label id="lblVia" runat="server" /><br /><br />
  <mobile:List id="lstOrderLines" styleReference="styListAndLink"
               runat="server" /><br />
  <mobile:Label id="lblTotal" Font-Bold="true" runat="server" /><br />
  <mobile:Link NavigateUrl="#frmOrderSelect" SoftKeyLabel="Order"
               Text="Select Order"
               styleReference="styListAndLink" runat="server" />
  <mobile:Link NavigateUrl="#frmCustType" SoftKeyLabel="Customer"
               Text="Select Customer"
               styleReference="styListAndLink" runat="server" />
</mobile:Form>
```

The next screenshot shows what this screen looks like when order number 10692 is selected:



The Code in the ShowOrderDetail Event Handler

The `ShowOrderDetail` function in our page runs when the user selects an order from the list in the previous screen. It is somewhat more complex than the `ShowOrderLines` event handler we saw in the HTML version of our application, because it has to display values from both of the tables in our `DataSet`. We first collect the order ID from the arguments passed to this event handler (the value selected in the list), and the customer ID from the "status" label in the previous screen. Then we can collect the `DataSet` from our `GetOrderDataSetFromSessionOrServer` function:

```
Sub ShowOrderDetail(objSender As Object, objArgs As ListCommandEventArgs)
'create order line details to display on page 5

  'get OrderID from selection made in List control on page 4
  Dim strOrderID As String = objArgs.ListItem.Value

  'get CustomerID by parsing out of Label control on page 4
  Dim strCustID As String = Mid(lblStatus2.Text, _
                                InStr(lblStatus2.Text, ":") + 1)

  'get DataSet using function elsewhere in this page
  Dim objDataSet As DataSet = GetOrderDataSetFromSessionOrServer(strCustID)
  ...
```

The `DataSet` will contain details of all orders for this customer, not just the order selected in the previous screen. This might seem an inefficient approach, but in most cases we will collect this `DataSet` from the user's session rather than hitting the database again. We used the same `DataSet` in the previous screen, and it will also be reused if the user chooses a different order to view afterwards.

We need to apply a filter to both of the tables in the `DataSet` so that only the details of the selected order are displayed. We filter both tables on the current order ID:

```
...
'create filtered DataView from Orders table in DataSet
Dim objOrderView As DataView = objDataSet.Tables("Orders").DefaultView
objOrderView.RowFilter = "OrderID = " & strOrderID

'create filtered DataView from OrderLines table in DataSet
Dim objLinesView As DataView = objDataSet.Tables("OrderLines").DefaultView
objLinesView.RowFilter = "OrderID = " & strOrderID
...
```

Now we can calculate the order total by summing the values in the `LineTotal` column that we added to the `OrderLines` table in our `GetOrderDataSetFromSessionOrServer` function. Then we extract the shipping details from the single row in the filtered `DataView` of the `Orders` table and display these in the `Label` controls:

```
...
'calculate total value of order
Dim dblTotal As Double = 0
Dim objDataRowView As DataRowView
For Each objDataRowView In objLinesView
  dblTotal += objDataRowView("LineTotal")
Next

'check that there are some matching order lines
If objLinesView.Count > 0 Then

  'display the shipping details in Labels
  lblOrderNo.Text = "Order No: " & strOrderID
  lblCustName.Text = "Customer: " & objOrderView.Item(0)("ShipName")
  Dim datThisDate as DateTime = objOrderView.Item(0)("OrderDate")
  lblOrdered.Text = "Ordered: " & datThisDate.ToString("d")
  If IsDbNull(objOrderView.Item(0)("ShippedDate")) Then
    lblShipped.Text = "Awaiting shipping"
  Else
    datThisDate = objOrderView.Item(0)("ShippedDate")
    lblShipped.Text = "Shipped: " & datThisDate.ToString("d")
  End If
  lblVia.Text = "via " & objOrderView.Item(0)("CompanyName")

  'display the total value of the order
  lblTotal.Text = "Total value: " & dblTotal.ToString("$#,##0.00")
  ...
```

Finally, we bind the filtered `DataView` of the `OrderLines` table to the `List` control to display the order line details. Again, we bind to the new column named `DisplayCol` that we added to the `OrderLines` table in our `GetOrderDataSetFromSessionOrServer` function. Then we can activate this screen so that the order details are displayed to the user:

```
      ...
      'set DataSource and bind the List control
      lstOrderLines.DataSource = objLinesView
      lstOrderLines.DataTextField = "DisplayCol"
      lstOrderLines.DataBind()

    Else

      lblOrderNo.Text = "No order lines found for this order..."

    End If

    'display page 4 containing the details
    ActiveForm = frmOrderDetail

  End Sub
```

Summary

In this chapter we've looked at two versions of our example application that lists orders for customers of a fictional food distribution corporation. We saw an overview of its capabilities, and discussed some of the design decisions and how we detect the client type when the application first starts.

We also looked at how we provide the data to drive the application, using components that we developed in Chapters 2 and 3, to create a separate **data tier**. From there we moved on to examine in more detail the version of the application designed for down-level HTML clients, and the version for small screen and mobile devices such as PDAs and cellular phones.

The topics we covered were:

- What the application looks like
- Some of the design considerations involved
- Specific client detection techniques for the example application
- The version of the application aimed at HTML 3.2 clients
- The version aimed at small screen and mobile devices

In the next chapter we'll continue to look at this application by examining some of the other versions that we've provided as examples. In particular, we'll see how we can work with "rich" clients, using XML and delimited text as the data transfer and storage medium.