This tutorial is part of a set. Find out more about data access with ASP.NET in the Working with Data in ASP.NET 2.0 section of the ASP.NET site at http://www.asp.net/learn/dataaccess/default.aspx.

Working with Data in ASP.NET 2.0 :: Handling BLL- and DAL-Level Exceptions

Introduction

In the Overview of Editing and Deleting Data in the DataList tutorial, we created a DataList that offered simple editing and deleting capabilities. While fully functional, it was hardly user-friendly, as any error that occurred during the editing or deleting process resulted in an unhandled exception. For example, omitting the product's name or, when editing a product, entering a price value of "Very affordable!", throws an exception. Since this exception is not caught in code, it bubbles up to the ASP.NET runtime, which then displays the exception's details in the web page.

As we saw in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial, if an exception is raised from the depths of the Business Logic or Data Access Layers, the exception details are returned to the ObjectDataSource and then to the GridView. We saw how to gracefully handle these exceptions by creating `Updated` or `RowUpdated` event handlers for the ObjectDataSource or GridView, checking for an exception, and then indicating that the exception was handled.

Our DataList tutorials, however, aren't using the ObjectDataSource for updating and deleting data. Instead, we are working directly against the BLL. In order to detect exceptions originating from the BLL or DAL, we need to implement exception handling code within the code-behind of our ASP.NET page. In this tutorial, we'll see how to more tactfully handle exceptions raised during an editable DataList's updating workflow.

Note: In the *An Overview of Editing and Deleting Data in the DataList* tutorial we discussed different techniques for editing and deleting data from the DataList, Some techniques involved using an ObjectDataSource for updating and deleting. If you employ these techniques, you can handle exceptions from the BLL or DAL through the ObjectDataSource's `Updated` or `Deleted` event handlers.

Step 1: Creating an Editable DataList

Before we worry about handling exceptions that occur during the updating workflow, let's first create an editable DataList. Open the `ErrorHandling.aspx` page in the `EditDeleteDataList` folder, add a DataList to the Designer, set its `ID` property to `Products`, and add a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class's `GetProducts()` method for selecting records; set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

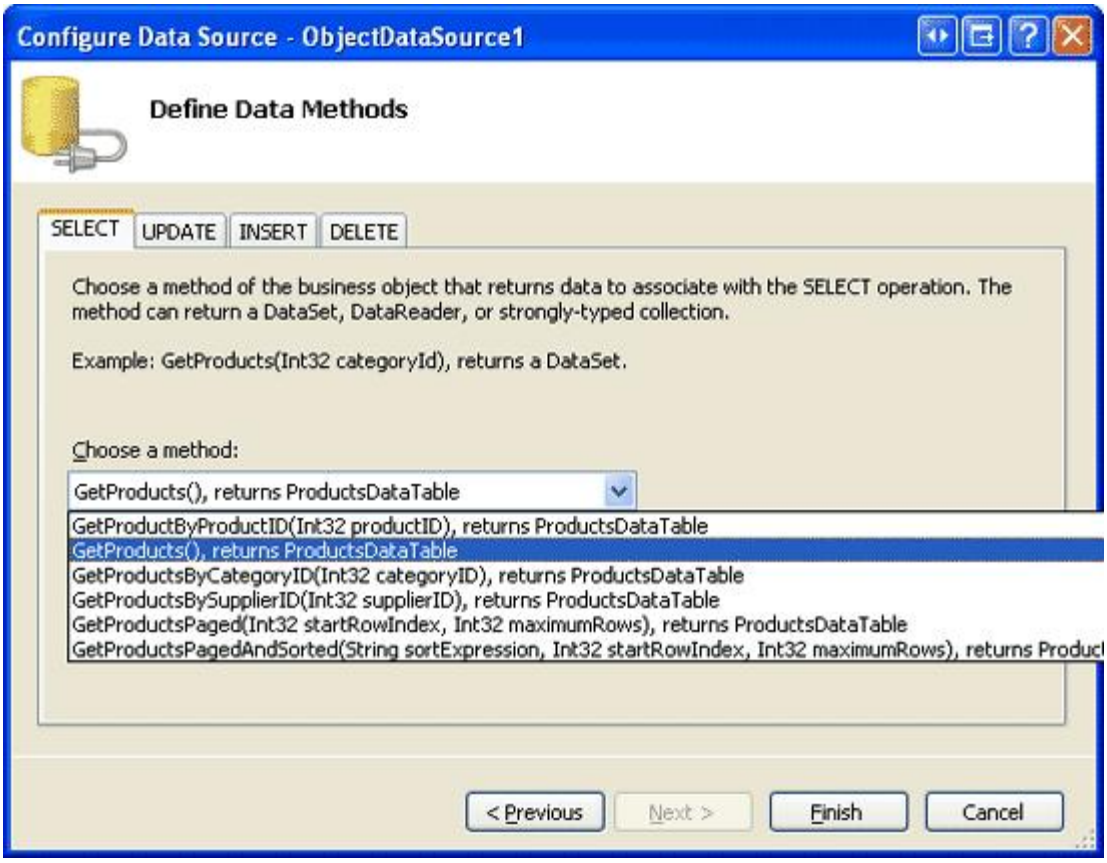

Figure 1: Return the Product Information Using the `GetProducts()` Method

After completing the ObjectDataSource wizard, Visual Studio will automatically create an `ItemTemplate` for the DataList. Replace this with an `ItemTemplate` that displays each product's name and price and includes an Edit button. Next, create an `EditItemTemplate` with a TextBox Web control for name and price and Update and Cancel buttons. Finally, set the DataList's `RepeatColumns` property to 2.

After these changes, your page's declarative markup should look similar to the following. Double-check to make certain that the Edit, Cancel, and Update buttons have their `CommandName` properties set to "Edit", "Cancel", and "Update", respectively.

```
<asp:DataList ID="Products" runat="server" DataKeyField="ProductID"
    DataSourceID="ProductsDataSource" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>' />
        </h5>
        Price:
            <asp:Label runat="server" ID="Label1"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
            <asp:Button runat="server" id="EditProduct" CommandName="Edit"
                Text="Edit" />
        <br />
        <br />
    </ItemTemplate>
    <EditItemTemplate>
        Product name:
            <asp:TextBox ID="ProductName" runat="server"
                Text='<%# Eval("ProductName") %>' />
```

```
        <br />
        Price:
            <asp:TextBox ID="UnitPrice" runat="server"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
            <asp:Button ID="UpdateProduct" runat="server" CommandName="Update"
                Text="Update" />
            <asp:Button ID="CancelUpdate" runat="server" CommandName="Cancel"
                Text="Cancel" />
    </EditItemTemplate>
</asp:DataList>

<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    OldValuesParameterFormatString="original_{0}">
</asp:ObjectDataSource>
```

Note: For this tutorial the DataList's view state must be enabled.

Take a moment to view our progress through a browser (see Figure 2).

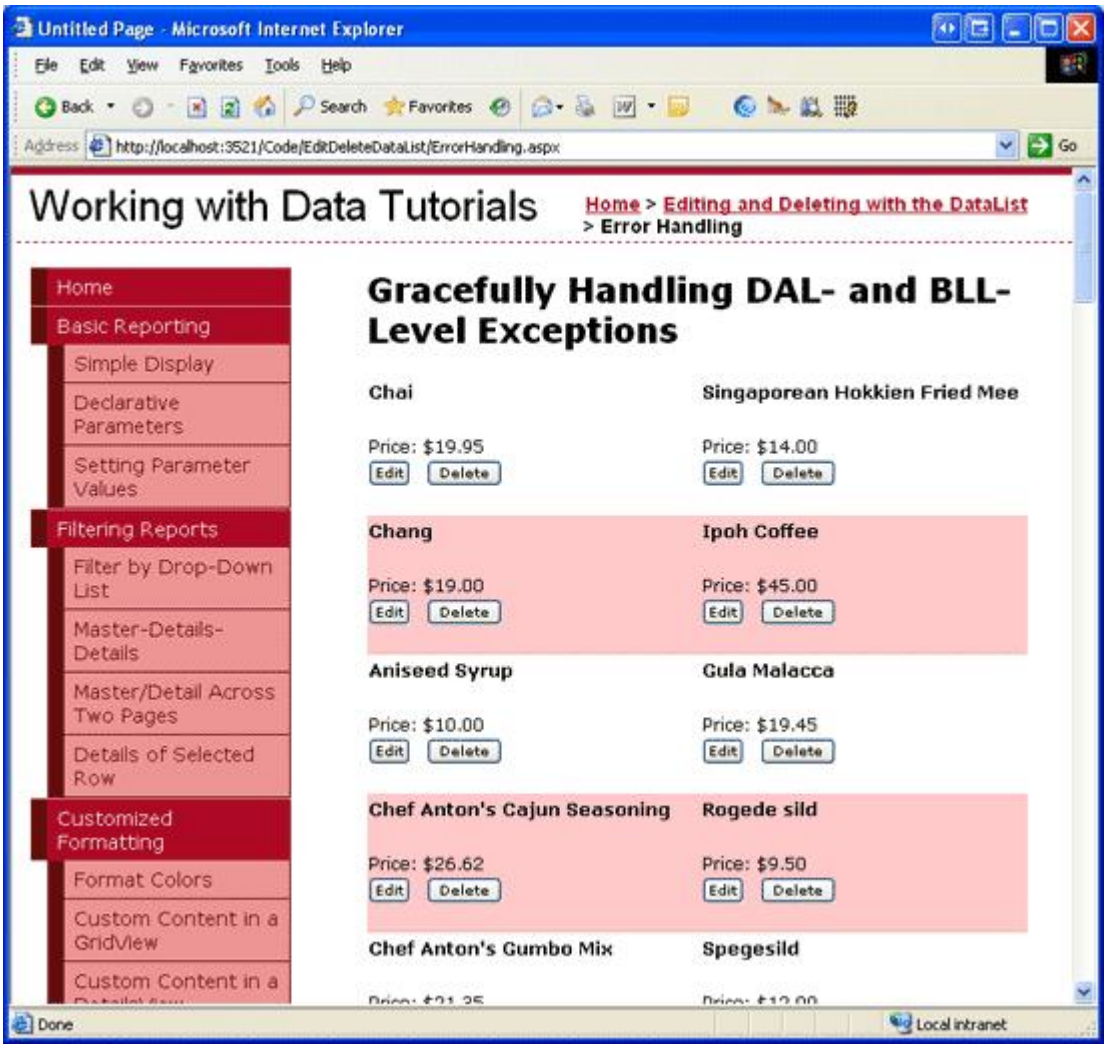

Figure 2: Each Product Includes an Edit Button

Currently, the Edit button only causes a postback — it doesn't yet make the product editable. To enable editing, we need to create event handlers for the DataList's `EditCommand`, `CancelCommand`, and `UpdateCommand` events.

The `EditCommand` and `CancelCommand` events simply update the DataList's `EditItemIndex` property and rebind the data to the DataList:

```
protected void Products_EditCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property to the
    // index of the DataListItem that was clicked
    Products.EditItemIndex = e.Item.ItemIndex;

    // Rebind the data to the DataList
    Products.DataBind();
}

protected void Products_CancelCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property to -1
    Products.EditItemIndex = -1;

    // Rebind the data to the DataList
    Products.DataBind();
}
```

The `UpdateCommand` event handler is a bit more involved. It needs to read in the edited product's `ProductID` from the `DataKeys` collection along with the product's name and price from the TextBoxes in the `EditItemTemplate`, and then call the `ProductsBLL` class's `UpdateProduct` method before returning the DataList to its pre-editing state.

For now, let's just use the exact same code from the `UpdateCommand` event handler in the *Overview of Editing and Deleting Data in the DataList* tutorial. We'll add the code to gracefully handle exceptions in step 2.

```
protected void Products_UpdateCommand(object source, DataListCommandEventArgs e)
{
    // Read in the ProductID from the DataKeys collection
    int productID = Convert.ToInt32(Products.DataKeys[e.Item.ItemIndex]);

    // Read in the product name and price values
    TextBox productName = (TextBox)e.Item.FindControl("ProductName");
    TextBox unitPrice = (TextBox)e.Item.FindControl("UnitPrice");

    string productNameValue = null;
    if (productName.Text.Trim().Length > 0)
        productNameValue = productName.Text.Trim();

    decimal? unitPriceValue = null;
    if (unitPrice.Text.Trim().Length > 0)
        unitPriceValue = Decimal.Parse(unitPrice.Text.Trim(),
            System.Globalization.NumberStyles.Currency);

    // Call the ProductsBLL's UpdateProduct method...
    ProductsBLL productsAPI = new ProductsBLL();
    productsAPI.UpdateProduct(productNameValue, unitPriceValue, productID);


    // Revert the DataList back to its pre-editing state
    Products.EditItemIndex = -1;
    Products.DataBind();
}
```

In the face of invalid input — which can be in the form of an improperly formatted unit price, an illegal unit price value like "-$5.00", or the omission of the product's name — an exception will be raised. Since the `UpdateCommand` event handler does not include any exception handling code at this point, the exception will

bubble up to the ASP.NET runtime, where it will be displayed to the end user (see Figure 3).

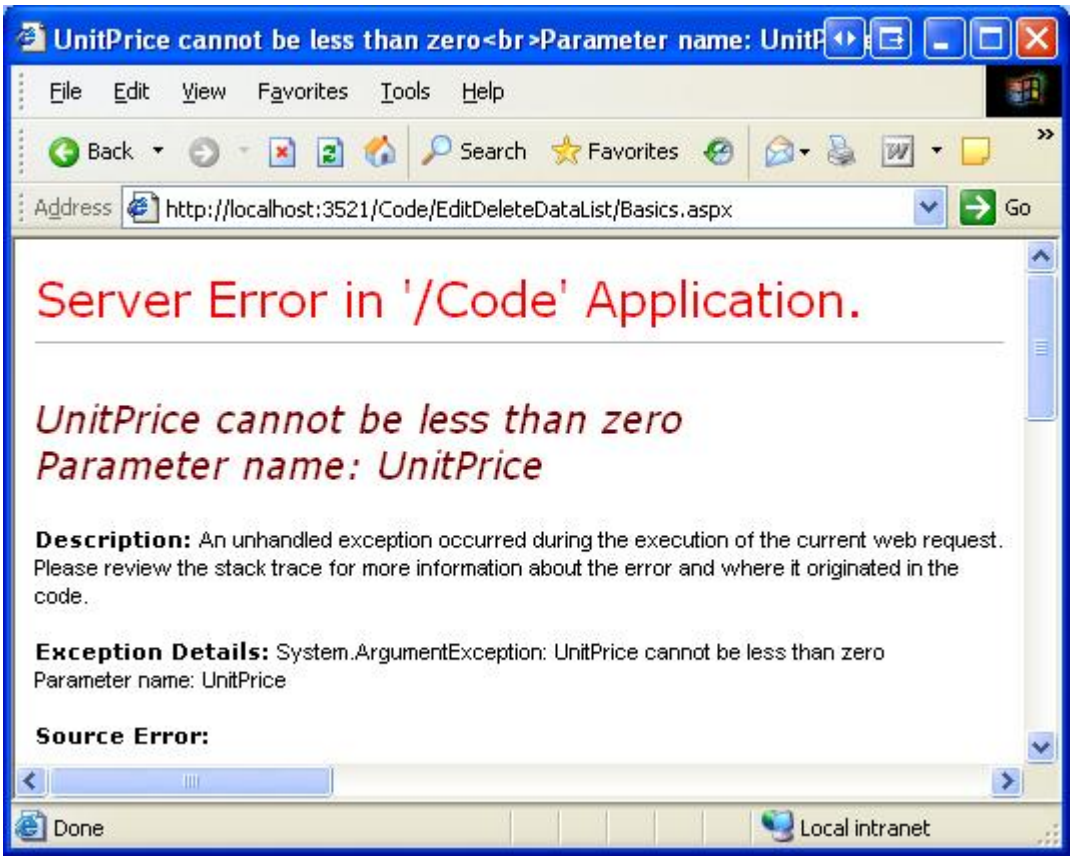

Figure 3: When an Unhandled Exception Occurs, the End User Sees an Error Page

Step 2: Gracefully Handling Exceptions in the UpdateCommand Event Handler

During the updating workflow, exceptions can occur in the `UpdateCommand` event handler, the BLL, or the DAL. For example, if a user enters a price of "Too expensive", the `Decimal.Parse` statement in the `UpdateCommand` event handler will throw a `FormatException` exception. If the user omits the product's name or if the price has a negative value, the DAL will raise an exception.

When an exception occurs, we want to display an informative message within the page itself. Add a Label Web control to the page whose `ID` is set to `ExceptionDetails`. Configure the Label's text to display in a red, extra-large, bold and italic font by assigning its `CssClass` property to the `Warning` CSS class, which is defined in the `Styles.css` file.

When an error occurs, we only want the Label to be displayed once. That is, on subsequent postbacks, the Label's warning message should disappear. This can be accomplished by either clearing out the Label's `Text` property or settings its `Visible` property to `False` in the `Page_Load` event handler (as we did back in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial) or by disabling the Label's view state support. Let's use the latter option.

```
<asp:Label ID="ExceptionDetails" EnableViewState="False" CssClass="Warning"
runat="server" />
```

When an exception is raised, we'll assign the details of the exception to the `ExceptionDetails` Label control's `Text` property. Since its view state is disabled, on subsequent postbacks the `Text` property's programmatic changes will be lost, reverting back to the default text (an empty string), thereby hiding the warning message.

To determine when an error has been raised in order to display a helpful message on the page, we need to add a `Try ... Catch` block to the `UpdateCommand` event handler. The `Try` portion contains code that may lead to an exception, while the `Catch` block contains code that is executed in the face of an exception. Check out the Exception Handling Fundamentals section in the .NET Framework documentation for more information on the `Try ... Catch` block.

```
protected void Products_UpdateCommand(object source, DataListCommandEventArgs e)
{
    // Handle any exceptions raised during the editing process
    try
    {
        // Read in the ProductID from the DataKeys collection
        int productID = Convert.ToInt32(Products.DataKeys[e.Item.ItemIndex]);

        ... Some code omitted for brevity ...
    }
    catch (Exception ex)
    {
        // TODO: Display information about the exception in ExceptionDetails
    }
}
```

When an exception of any type is thrown by code within the `Try` block, the `Catch` block's code will begin executing. The type of exception that is thrown — `DbException`, `NoNullAllowedException`, `ArgumentException`, and so on — depends on what, exactly, precipitated the error in the first place. If there's a problem at the database level, a `DbException` will be thrown. If an illegal value is entered for the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, or `ReorderLevel` fields, an `ArgumentException` will be thrown, as we added code to validate these field values in the `ProductsDataTable` class (see the Creating a Business Logic Layer tutorial).

We can provide a more helpful explanation to the end user by basing the message text on the type of exception caught. The following code — which was used in a nearly identical form back in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial — provides this level of detail:

```
private void DisplayExceptionDetails(Exception ex)
{
    // Display a user-friendly message
    ExceptionDetails.Text = "There was a problem updating the product. ";

    if (ex is System.Data.Common.DbException)
        ExceptionDetails.Text += "Our database is currently experiencing problems.
            Please try again later.";
    else if (ex is NoNullAllowedException)
        ExceptionDetails.Text += "There are one or more required fields that are
            missing.";
    else if (ex is ArgumentException)
    {
        string paramName = ((ArgumentException)ex).ParamName;
        ExceptionDetails.Text +=
            string.Concat("The ", paramName, " value is illegal.");
    }
    else if (ex is ApplicationException)
        ExceptionDetails.Text += ex.Message;
}
```

To complete this tutorial, simply call the `DisplayExceptionDetails` method from the `Catch` block passing in the caught `Exception` instance (`ex`).

With the `Try ... Catch` block in place, users are presented with a more informative error message, as Figures 4 and 5 show. Note that in the face of an exception the DataList remains in edit mode. This is because once the exception occurs, the control flow is immediately redirected to the `Catch` block, bypassing the code that returns the DataList to its pre-editing state.

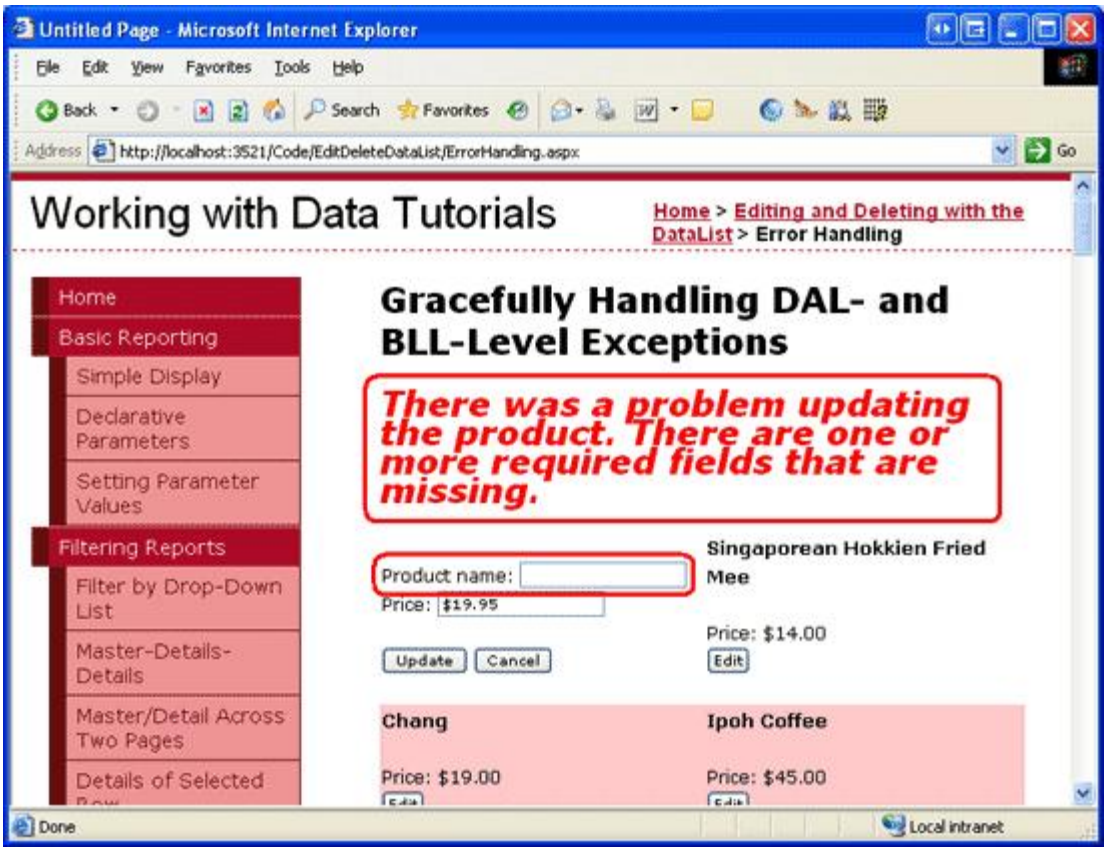

Figure 4: An Error Message is Displayed if a User Omits a Required Field

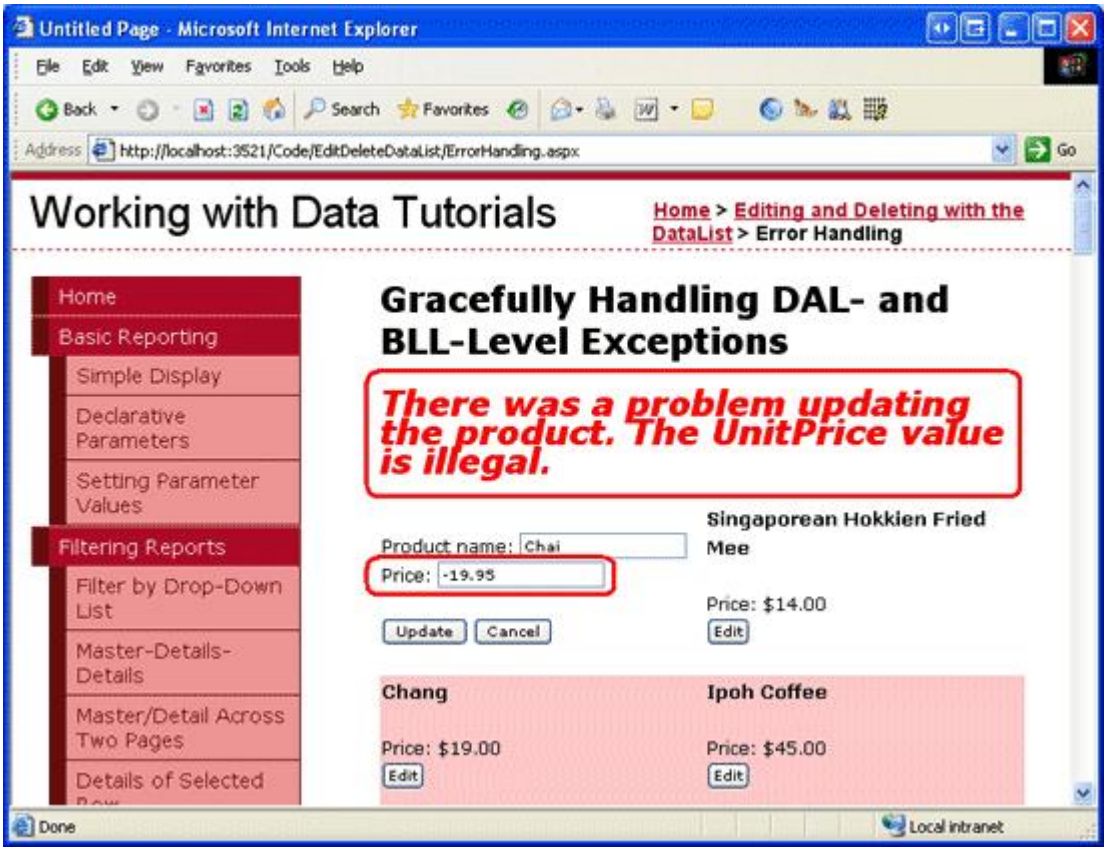

Figure 5: An Error Message is Displayed When Entering a Negative Price

Summary

The GridView and ObjectDataSource provide post-level event handlers that include information about any exceptions that were raised during the updating and deleting workflow, as well as properties that can be set to indicate whether or not the exception has been handled. These features, however, are unavailable when working with the DataList and using the BLL directly. Instead, we are responsible for implementing exception handling.

In this tutorial we saw how to add exception handling to an editable DataList's updating workflow by adding a `Try ... Catch` block to the `UpdateCommand` event handler. If an exception is raised during the updating workflow, the `Catch` block's code executes, displaying helpful information in the `ExceptionDetails` Label.

At this point, the DataList makes no effort to prevent exceptions from happening in the first place. Even though we know that a negative price will result in an exception, we haven't yet added any functionality to proactively prevent a user from entering such invalid input. In our next tutorial we'll see how to help reduce the exceptions caused by invalid user input by adding validation controls in the `EditItemTemplate`.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Design Guidelines for Exceptions
- Error Logging Modules and Handlers (ELMAH) (an open-source library for logging errors)

- Enterprise Library for .NET Framework 2.0 (includes the Exception Management Application Block)

About the Author

Scott Mitchell, author of six ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer, recently completing his latest book, Sams Teach Yourself ASP.NET 2.0 in 24 Hours. He can be reached at mitchell@4guysfromrolla.com or via his blog, which can be found at ScottOnWriting.NET.

Special Thanks To…

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Ken Pespisa. Interested in reviewing my upcoming articles? If so, drop me a line at mitchell@4guysfromrolla.com.