Data Entry Input Validation With Delphi, Kylix, and C++Builder

Bob Swart

March 27, 2003

Introduction

In this article, I will demonstrate how to build data entry forms on top of IBM DB2® Universal Database as database tables and records, and how to perform data entry input validation using Borland Delphi Studio, Borland Kylix™, and Borland C++Builder® Studio.

Note that you'll use dbExpress™ again to connect to the DB2 Personal Edition v8.1 database, so make sure you've read the previous articles that describe how to connect from Delphi, Kylix or C++Builder.

Note: When I write about Delphi in this article, you can also substitute Kylix or C++Builder and perform the same steps. When showing the source code, I will provide both Delphi (Kylix-compatible) source code and C++(Builder) code snippets.

Build the connection

In previous articles, I've mainly used a TDBGrid component to show an overview of multiple records at the same time. This time, however, I'll focus on only one record at a time that the end user can edit (for the first time) or modify. To build the example application, start Delphi, and drop TSQLConnection, TSQLDataSet, TDataSetProvider, TClientDataSet, and TDataSource components.

Set the ConnectionName property of TSQLConnection to DB2Connection. Also specify the right values in the connection parameters, such as database name, user_name, and password. Now point TSQLDataSet's SQLConnection property to the TSQLConnection component, set the CommandType to ctTable, and select the EMPLOYEE table as the value for the CommentText property. Connect TDataSetProvider's DataSet property to the TSQLDataSet component, connect TClientDataSet's ProviderName property to the TDataSetProvider component, and finally connect TDataSource's DataSet property to the TClientDataSet component.
Design the data entry forms

You can double-click on the TClientDataSet component to start the **Fields Editor**. If you right-click in the **Fields Editor**, you can add persistent fields for the EMPLOYEE table. The table has 14 fields: EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, HIREDATE, JOB, EDLEVEL, SEX, BIRTHDATE, SALARY, BONUS, and COMM (see Figure 1). You need them all, so add all fields in the **Fields Editor**.

**Figure 1. Fields Editor**

As a quick-and-dirty way to create data entry controls on the form, you can now "drag" the fields from the **Fields Editor** and "drop" them onto the form. You can do this for one or more fields at the same time. If you select them all and drag them all at once, make sure to drop at the upper-left corner of the form, because each control will be created a little lower than the previous one, and if you start too low you'll probably need to reposition them again. This produces TLabel and TDBEdit components for each of the persistent fields. A TDBNavigator component, connected through its DataSource property to the TDataSource, finishes the rough data entry form for now (see Figure 2).
You need to write one line of code to make sure that any changes made to the data will be posted back to the original DB2 SAMPLE database. Select the TClientDataSet component, go to the **Object Inspector**, and write this code for the OnAfterPost event handler:

```delphi
ClientDataSet1.ApplyUpdates(0);
```

For C++Builder, use this syntax (an arrow instead of a dot):

```cpp
ClientDataSet1->ApplyUpdates(0);
```

You should also connect the OnAfterDelete event handler to the same event handler code using the **Object Inspector**.

Now you need to ensure that this call to ApplyUpdates (in OnAfterPost and OnAfterDelete) is executed only if the data that the end user submits is valid - that is, if the input contains no inconsistencies or incorrect values. To do this, you'll implement some input validation techniques at different levels.
Define the input validation

Some of the fields in Figure 2 have obvious restrictions in what the end user can enter. HIREDATE is a date field, which is always "fun" to allow an end user to enter without help. JOB is another string field, but from the looks of it, it should be taken from an existing list of string values ("PRES" being one of them). EDLEVEL is a small integer; SEX is a string field, probably with values "F" and "M" being the only acceptable ones; BIRTHDATE is another date field, which apart from a valid date should also be a sensible date in the past. Finally, SALARY, BONUS, and COMM are BCD fields, which probably shouldn't contain negative values. (Have you ever been asked to pay your boss to come to work?)

Mask the input fields

At the lowest level, you can define so-called input masks. For this, you must work with the individual fields that you added to the Fields Editor. String fields have an EditMask property that you can assign using the Input Mask Editor (see Figure 3), which already contains a number of predefined masks for telephone numbers and extensions, social security numbers, zip codes, dates, and times.

Figure 3. Delphi Input Mask Editor

The value for the EditMask property consists of three parts, separated by semi-colons. The first part is the actual mask, which uses 0 for a required digit, 9 for an optional digit, A for a required character, and so on (you can find details in the online help for the TEditMask topic). The second part of the mask specifies whether literals (such as brackets in phone numbers) should be included in the actual database field, and the third and final part of the mask is the dummy character used for blanks - an underscore, for example.

As an example, make sure that the end user can enter only digits for the PHONENO field: four digits, so the EditMask is set to 0000;1;_. For WORKDEPT, you can set the mask to A00;1;_ because it must start with a character followed by two digits.

You can also use an EditMask for the BIRTHDATE and HIREDATE fields, but this can lead to confusion if you plan to use the application in different countries - with different locales and hence different ways to enter a date. (yyyy/mm/dd is my personal preference, if only because it sorts easily, but I haven't yet found a country that supports this format.) I'll cover the date fields later with another solution that will support end users even more.
Sometimes the possible values are not easily placed in an EditMask. For example, the JOB field can be PRES, MANAGER, SALESREP, CLERK, ANALYST, DESIGNER, OPERATOR, or FIELDREP.

Putting these values in an EditMask will help only if the end user knows which values are possible, so a first letter can produce the rest of the value. However, I would feel better supporting the user by showing the possible values. Assuming no other JOB values exist, you can solve this problem using a TDBComboBox component.

Connect the TDBComboBox's DataSource property to the TDataSource and set the DataField to the JOB field. Enter the eight possible JOB values in the Items property (pad the values with spaces at the end until they are all eight characters long), and set the Style property to DropDownList - this will prevent the end user from selecting anything but the eight predefined values. If you want to allow entry of a new JOB, you can leave the Style value as DropDown, which means the user can either select one of the existing eight options or enter a new JOB title (obviously, it’s not easy to check the validity of that user-defined input).

The SEX field provides a perfect opportunity to use a TDBRadioGroup component: drop it on the form, point the DataSource property to the TDataSource, enter "Female" and "Male" in the Items property, and "F" and "M" in the Values property. This ensures that you see the long Female and Male options, but use only F and M in the SEX field itself.

Now take a look at the BIRTHDATE and HIREDATE fields. In my view, the best way to allow the end user to specify a date is to provide a calendar in which he or she can click on the specific day. Don’t confuse the end user with formatting details such as using / or - as separator character; the order of year, month, and day; and so on. Delphi offers a component called the TDateTimePicker that you can use for this purpose. It looks like a drop-down combobox showing the date value, but a month calendar will appear if the end user opens it. The user can click on the day required, as well as browse to the previous and next months, previous and next years, and so on. TDateTimePicker is quite intuitive to use.

The only disadvantage is that TDateTimePicker is not data-aware, so you have to "connect" it to the field of the ClientDataSet, and make sure that its DateTime property value is updated when the dataset changes: in an OnAfterScroll or OnAfterCancel (undo) event handler. Assuming you use two TDateTimePicker components - one for HIREDATE and one for BIRTHDATE - here’s the code to respond to the TClientDataSet’s OnAfterScroll and OnAfterCancel event handler:

```pascal
procedure TForm1.ClientDataSet1AfterScrollOrCancel(DataSet: TDataSet);
begin
  DateTimePickerHIREDATE.DateTime :=
  DataSet.FieldByName('HIREDATE').AsDateTime;
  DateTimePickerBIRTHDATE.DateTime :=
  DataSet.FieldByName('BIRTHDATE').AsDateTime
end;
```

For C++Builder, you can write the first statement like this:
DateTimePickerHIREDATE->DateTime =
DataSet->FieldByName('HIREDATE')->AsDateTime;

This will make sure the TDateTimePickers are updated when the dataset changes, but now you need to make sure that the field value is updated when the end user (manually) changes the value in the TDateTimePicker. As an additional step, check if the dataset is already in dsEdit or dsInsert mode; if not, call the Edit method (you can't just modify the values of the DATE fields). Here's the Delphi and Kylix code to do this:

```delphi
procedure TForm1.DateTimePickerHIREDATECloseUp(Sender: TObject);
begin
if not (ClientDataSet1.State in [dsEdit,dsInsert]) then ClientDataSet1.Edit;
ClientDataSet1.FieldByName('HIREDATE').AsDateTime :=
DateTimePickerHIREDATE.Date Time
end;

procedure TForm1.DateTimePickerBIRTHDATECloseUp(Sender: TObject);
begin
if not (ClientDataSet1.State in [dsEdit,dsInsert]) then ClientDataSet1.Edit;
ClientDataSet1.FieldByName('BIRTHDATE').AsDateTime :=
DateTimePickerBIRTHDATE.Date Time
end;
```

For C++Builder, you can write the code for HIREDATE like this:

```c++
if ((ClientDataSet1->State != dsEdit) && (ClientDataSet1->State != dsInsert))
    ClientDataSet1->Edit();
ClientDataSet1->FieldByName('HIREDATE')->AsDateTime =
    DateTimePickerHIREDATE->DateTime;
```

Check out the results of using the TDateTimePicker: you can see all component-level validations from this section, including an opened TDateTimePicker component for the BIRTHDATE field (see *Figure 4*).
Validate the field values

Even with all the special components to support the end user with data entry, you should still be prepared to perform some validation in your source code, such as confirming that a birthdate does not occur in the future or that someone wasn't hired before he or she was born. Each field has a special OnValidate event handler that you can use for this purpose, and I've implemented four of them (the last one is actually used by three fields: SALARY, BONUS, and COMM).

The format is always the same: the code performs a check and raises an exception with a helpful message if the check fails. For example, if the birthdate is less than 16 years in the past, an exception is thrown to inform the end user that this person is too young to be hired:
procedure TForm1.ClientDataSet1BIRTHDATEValidate(Sender: TField);
begin
if (Now - Sender.AsDateTime) < 4 * (365 + 365 + 365 + 366)
then
raise Exception.Create('You are too young to be hired by us!')
end;

procedure TForm1.ClientDataSet1HIREDATEValidate(Sender: TField);
begin
if Now < Sender.AsDateTime then
raise Exception.Create('We cannot hire someone from the future, yet!')
end;

procedure TForm1.ClientDataSet1EDLEVELValidate(Sender: TField);
begin
if (Sender.AsInteger < 12) then
raise Exception.Create('You need a higher education level!')
end;

procedure TForm1.ClientDataSet1SALARYValidate(Sender: TField);
begin
if Sender.AsFloat < 0 then
raise Exception.Create('People are not expecting to have to pay ' +
  'in order to come to work!')
end;

For C++Builder, you can write the last check (for SALARY, BONUS, and COMM) like this:

```cpp
if (Sender.AsFloat < 0)
throw Exception("People are not expecting to have to pay in order to come to work!");
```

If a negative amount has been entered for the SALARY, BONUS, or COMM fields, the user will get a message box describing the error (see Figure 5).

**Figure 5. Field validation error message**

![Error Message](image)

After the end user clicks on OK to get rid of this message, he or she can modify the specific field, or press escape to revert back to the original value of the record and undo the entire edit operation.

**Validate the record values**

Sometimes, two or more fields are related to each other, and you can perform a real validation check only when combining the field values. If you try to perform this check when not all field values have been entered, you might confront the end user with a premature error message or warning.

For that reason, you should wait until the end user is ready to post the new or modified record to the database. Right before the post happens, you can respond in the TClientDataSet's
OnBeforePost event handler and perform some multifield related tests, such as whether the HIREDATE comes after the BIRTHDATE:

```pascal
procedure TForm1.ClientDataSet1BeforePost(DataSet: TDataSet);
begin
  if DataSet.FieldByName('HIREDATE').AsDateTime <
      DataSet.FieldByName('BIRTHDATE').AsDateTime then
    raise Exception.Create('You cannot have been hired before you were born!');
end;
```

For C++Builder, use this syntax to write the check:

```cpp
if (DataSet->FieldByName('HIREDATE').AsDateTime <
    DataSet->FieldByName('BIRTHDATE').AsDateTime)
  throw Exception("You cannot have been hired before you were born!");
```

Obviously, this check is just an example, but if the end user tries to enter a record saying a person was born after being hired, an error message will appear (see Figure 6).

**Figure 6. Record validation error message**

![Record validation error message](image)

And the record will not be posted to the database (the exception that was raised in the OnBeforePost has cancelled the Post itself). The end user can now make modifications to resolve the inconsistent state and repost the changes.

**Respond to reconcile errors**

Now, assume that all validation checks have been passed, and the end user can post the record to the dataset. That's the TClientDataSet in this example, and in the OnAfterPost event handler you made sure to call the ApplyUpdates method to apply the updates all the way back to the DB2 SAMPLE database table itself.

However, someone else could have changed the same fields in the same record during the time the end user was working on it. This is a situation called a reconcile error, and you need to respond to it in the TClientDataSet's OnReconcileError event handler. This code shows the error message:

```pascal
procedure TForm1.ClientDataSet1ReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  ShowMessage('Reconcile Error: ' + E.Message);
end;
```
But you can find a special **Reconcile** dialog in the **Object Repository** (of Delphi Studio, Kylix, and C++Builder Studio) that might help you provide your end users with a dialog to resolve the reconcile errors, which is a topic for another day (see the online help for details).

### Connect to the table at runtime

I'd like to add a final section of code that connects to the table at runtime (as opposed to design time) and asks the end user whether to save the last changes to the database.

The advantage of connecting only at runtime is that you can load the example project in your copy of Delphi, without needing a DB2 SAMPLE database running on your machine:

```pascal
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientDataSet1.Active := True
end;

procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
  Answer: Word;
begin
  if ClientDataSet1.State in [dsEdit, dsInsert] then
  begin
    Answer := MessageDlg('Save changes?', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    if Answer = mrYes then ClientDataSet1.Post
    else
      if Answer = mrNo then
        ClientDataSet1.Cancel;
      CanClose := Answer <> mrCancel
  end
end;
```

By using the OnCloseQuery event handler, you can ensure that if the end user makes a change and closes the application without explicitly posting the change to the dataset, he or she will get a confirmation dialog (see **Figure 7**).

**Figure 7. Form-level update check**

Clicking on **Yes** posts the changes to the table, but invalid values can raise an exception that prevents the user from closing the application. After the end user has seen the exception message and clicked on **OK**, the application will remain open, and the end user can then make changes to resolve the problem or exit without saving.

As final option, I would like to mention input validation using check contraints in the database as opposed to doing input validation in the application. The constraints are included with the database...
table definition in DB2 and will be checked after a record has been posted to the table (so this is after the EditMark, OnValidate, and OnBeforePost event handlers have been used).

**Conclusion**

In this article, I've demonstrated several ways to perform data entry input validation: at the field level using edit masks, which prevents the user from entering incorrect values; at the OnValidate level, which checks after you're ready with a field; at the OnBeforePost level, which checks the relationship between the field values for an entire record; and finally at the OnReconcileError level, which prevents multiple users from modifying the same fields and/or the same record at the same time.

**A reminder:** One of the best techniques to support your end users when they're doing data entry is to ensure that the controls offer only valid choices, such as radiobuttons for the SEX field, a drop-down combobox for the JOB field, or a date picker for the BIRTHDATE and HIREDATE fields.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>source.zip</td>
<td>7 KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2003  
**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)