Resolving
for kbmMW v. 2.50+
Pro, ProPlus and Enterprise Editions

Introduction ... 2
One tier resolving ... 2
  The application ... 3
  Error handling ... 8
  Raising errors manually ... 10
  Modifying values during resolve ... 11
  Doing additional database operations while resolving ... 14
  Resolving autoinc fields ... 15
  Field selective resolves ... 16
  Resolving joins ... 17
  Resolving multiple server-side datasets in one transaction ... 20
N-tier client side resolving ... 22
  The application server ... 22
  The client application ... 23
  Resolving multiple client-side datasets in one transaction ... 25
Creating a custom resolver ... 26

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

COMPONENTS 4 DEVELOPERS

Introduction

Through the query service, one can fetch resultsets based on queries, but its also important to be able to make sure that changes in those resultsets are reflected back to the datastore/databases, usually, but not always, to the same tables from where the data originated.

This concept is called 'resolving' in kbmMW.

Although resolving sounds pretty easy, its rather complex and lots of parameters can play in. kbmMW try to hide the complex parts and allow the developer to resolve data in even very complex situations easily.

This whitepaper describes different resolving situations in order of complexity. To get the best understanding, please read the document in its entirety.

One tier resolving

To start simple, we build a standard TForm based application only using the database adapter components.

The purpose of this application is to select all records from the BDE DBDEMOS's table 'customer' and allow the user to modify its contents via the kbmMW BDE adapter components.

The exercise may seem trivial, but it does show lots of the basic aspects of resolving in kbmMW. Later in this documents we will use the same knowledge to create n-tier resolving.

As minimum a **TkbmMWBDEConnectionPool**, a **TkbmMWPooledSession**, a **TkbmMWBDEQuery** and a **TkbmMWBDEResolver** is needed. Remember to check the document *'Using kbmMW as a query server'* for more information about the use of these.

In addition the resolver needs information about the type of database that are being resolved against. For that purpose we need a metadata component. In our case we choose **TkbmMWGenericSQLMetaData**. It's specially important to choose a more specific/correct one, if autoinc fields are being resolver and you want to get the correct value of those back to the user.

The resolver components responsibility is to handle how to reflect dataset changes back on the datastore/database. Thus it has lots of properties depending on the resolver type.

kbmMW supports both SQL and non SQL datastores/databases and thus also different types of resolvers. The SQL orientated resolvers have SQL oriented properties which can be used to fine tune the resolving process.

Other resolver types may have other properties controlling their behaviour. We will primarely look at SQL based resolving as that is what people mostly use.

The resolver works in close corporation with the query component. The query component is responsible for keeping track of all deletes, updates and inserts made to the resultset.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

It does that by employing something called versioning of records. That means that if you delete a record, the original record is actually retained, but a marked as deleted. If you modify a record, the original record is still kept, and a new version with the edited data added etc.

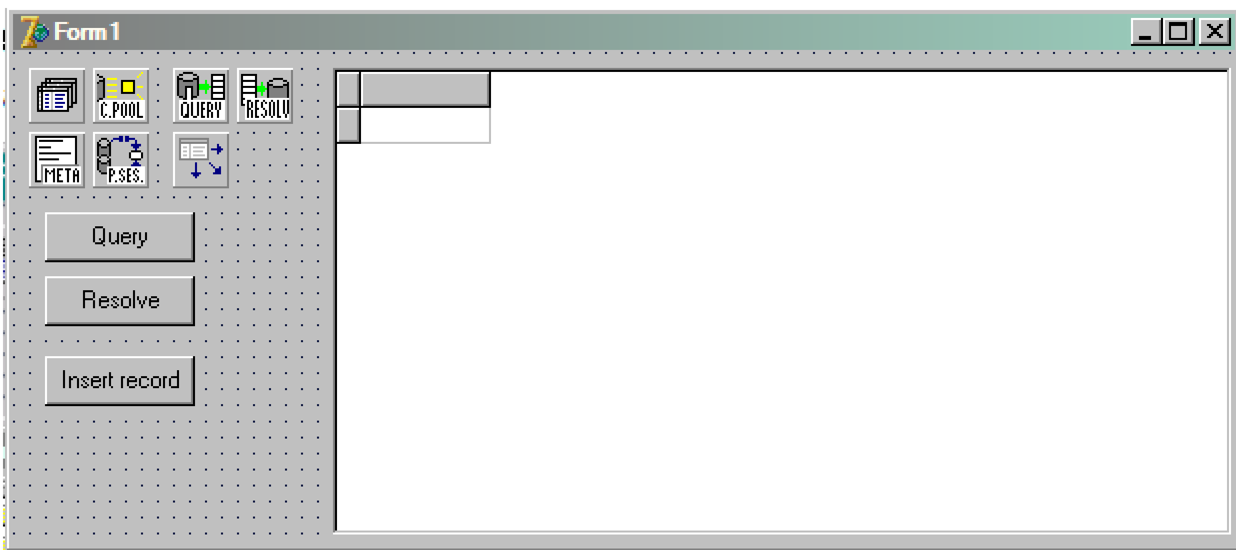
Thus its important that the query component has **EnableVersioning**:=true to be able to resolve changes from it (its default true).

Its **VersioningMode** can be set to either of the values, but the 1sinceCheckpoint is the one taking up less memory. The other mode is useful if you want to build in multiple steps of undo in your application. (for example using StartTransaction, Commit and Rollback, or using the Undo method).

To configure a query for resolving, the **Resolver** property must be set to point on a resolver of some type.

The application

So we build a simple sample application looking like this:



The following properties are set:

```
kbmMWConnectionPool1.Database:=Database1
kbmMWConnectionPool1.MetaData:=kbmMWGenericSQLMetaData1
Database1.AliasNames:='DBDEMOS'
kbmMWPooledSession1.ConnectionPool:=kbmMWConnectionPool1
kbmMWPooledSession1.SessionName:='DEMO'
kbmMWBDEQuery1.SessionName:='DEMO'
kbmMWBDEQuery1.SQL.Text:='select * from customer'
kbmMWBDEQuery1.Resolver:=kbmMWBDEResolver1
DataSource1.DataSet:=kbmMWBDEQuery1
DBGrid1.DataSource:=DataSource1
```

The event handler of the Query button contains:

```
kbmMWBDEQuery1.Open;
```

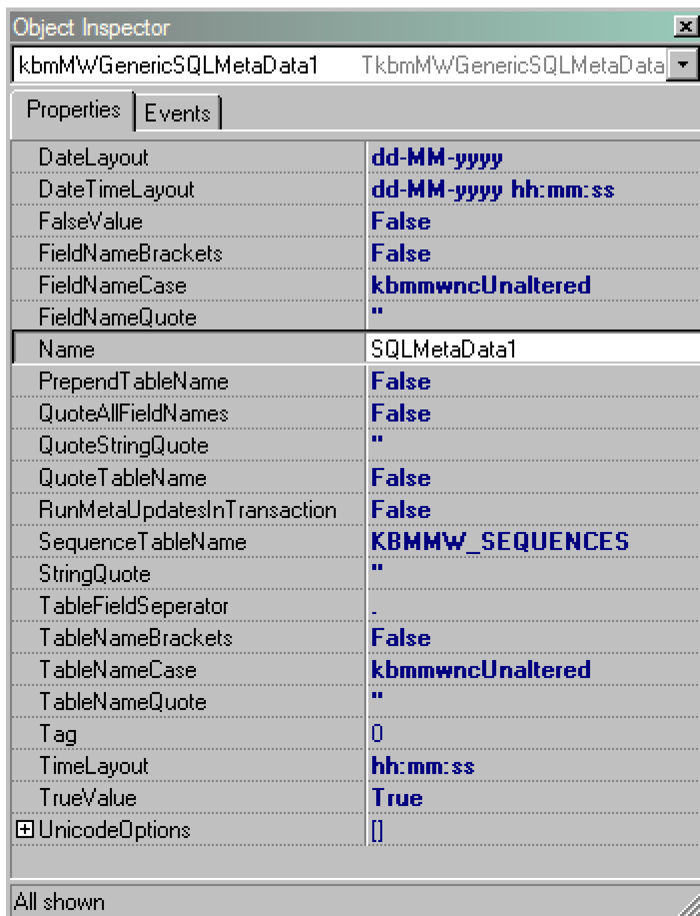
The event handler of the Resolve button contains:

```
kbmMWBDEQuery1.Resolve;
```


Copyright 2001-2005 Components4Developers
http://www.components4developers.com

In this sample application our query is a BDE based query, using the TkbmMWBDEQuery component and therefore we have chosen to use a TkbmMWBDEResolver to be able to resolve back to the database type queried from.

For the resolver to know how to generate the correct SQL syntax for the backend database, a correct metadata component has to be chosen and configured. Several metadata components exists, some specially designed for a specific database, and others, like the one we use now TkbmMWGenericSQLMetaData, more general, where you can modify most of its settings to match the appropriate database.



DateLayout – Specifies which layout the database prefers to receive dates in. It's normally not used during parameterized resolves (which most resolvers are).

DateTimeLayout – Specifies which layout the database prefers to receive date/time values in. It's normally not used during parameterized resolves (which most resolvers are).

TimeLayout - Specifies which layout the database prefers to receive time info in. It's normally not used during parameterized resolves (which most resolvers are).

FalseValue – Specifies the value that the database use to indicate a boolean false. It's normally not used during parameterized resolves (which most resolvers are).

TrueValue – Specifies the value that the database use to indicate a boolean true. It's normally not used during parameterized resolves (which most resolvers are).

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

FieldNameBrackets – Specifies if fieldnames should be bracketed. If set to true, **FieldNameQuote** determines what the starting bracket should be (one of '{ [< ('). kbmMW automatically choose the correct ending bracket based on those settings.

FieldNameCase - Unaltered means use fieldnames as is. Lower means convert fieldnames to lowercase. Upper means convert fieldnames to uppercase. Directly affect the SQL generation.

FieldNameQuote - The character to put around a fieldname. If **FieldNameBrackets** is set to true, FieldNameQuote should be one of **(**, **{**, **[** or **<** indicating the starting bracket. kbmMW will automatically select the correct matching ending bracket.

PrependTableName - If the name of the table should be put in front of the fieldname.

QuoteAllFieldNames - If true, will format all fieldnames according to **FieldNameQuote** and **FieldNameBrackets** settings (If false, only fieldnames which contains invalid characters will be quoted.

QuoteTableName - If true, will put **TableNameQuote** around the table name.

QuoteStringQuote – Is the character to use to quote (prepend) the **StringQuote** character when its part of the actual string being quoted. Eg. *QuoteStringQuote* = \, and *StringQuote*=' means that a text like: The lady's purse
will be quoted like this: 'The lady\'s purse'

TableFieldSeperator – Specifies the character to use when using compound tablename/fieldname pairs. Typically a dot (.) is used.

TableNameBrackets – Same as **FieldNameBrackets**, except for table names.

TableNameCase - Like with **FieldNameCase**.

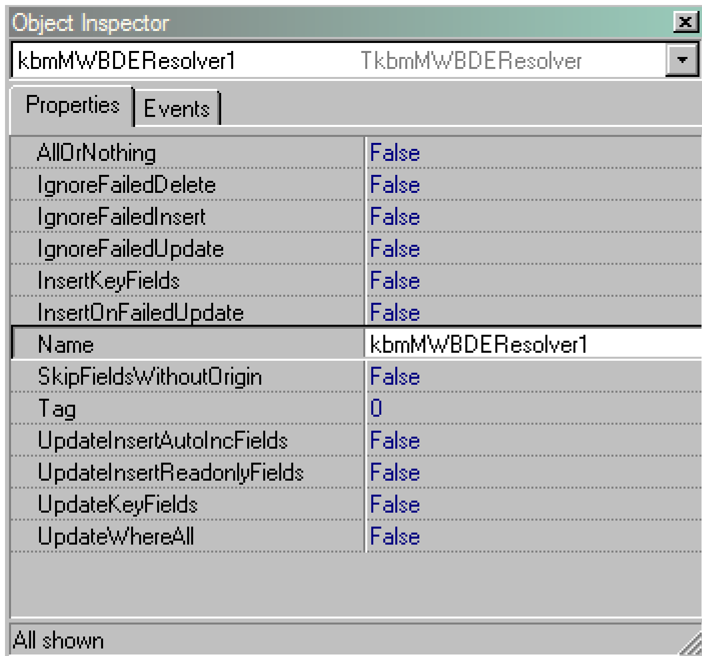
TableNameQuote - The quote to use, when quoting table names. If **TableNameBrackets** is set to true, TableNameQuote should be one of **(**, **{**, **[** or **<** indicating the starting bracket. kbmMW will automatically select the correct matching ending bracket. If TableNameQuote is set to #0 (chr(0)) there wont be any quoting of the table name.

StringQuote – Is the character to use when quoting strings. Check QuoteStringQuote for how to quote quotes within the string.

For more information about the meta data component, please check the whitepaper '**kbmMW and generalized metadata management**'

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

The resolver has several settings that control the resolving process:



AllOrNothing - True if all records must be resolved as an atomic operation. Thus if one of the record operations of some reason cannot be resolved to the backend database/datastore, all changes will be rolled back. False means that it will resolve what it can regardless of errors. Often this property is set to true by the developer.

IgnoreFailedDelete - If true, will not report an error when a record cannot be deleted.

IgnoreFailedInsert - If true, will not report an error when a record cannot be inserted.

IgnoreFailedUpdate - If true, will not report an error when a record cannot be modified.

InsertKeyFields - If true will also insert values for key fields.

InsertOnFailedUpdate - If true, will try to insert a new record if the record could not be updated.

SkipFieldsWithoutOrigin - If true, fields which have an empty Origin value will not be included in the resolve operation.

UpdateInsertAutoIncFields - If true, fields which are of ftAutoInc type will be included in update and insert statements.

UpdateInsertReadonlyFields - If true, fields which are ReadOnly will anyway be included in update and insert statements.

UpdateKeyFields - If true, fields which are part of the unique key of the record will also be included in update statements.

UpdateWhereAll - If true, includes all fields in the 'where' clause of the SQL. If false, only include defined key fields.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

COMPONENTS 4 DEVELOPERS

If we run this application and press the Query button we will see lots of records in the DBGrid. What happens is that a cursor/connection to the DBDEMOS database is opened by the connection pool, the connection reserved by the query component, the query performed, all records fetched into the kbmMWBDEQuery component. The connection is then released back to the connection pool for later reuse, instead of being closed.

Modifying a record in the DBGrid does not modify the data directly in the database, since the query component is no longer connected to it. That's called disconnected datasets.

What happens is that we can modify all records we want, delete, insert and do whatever we like, but none of those changes will ever be seen in the DBDEMOS database…. unless one decides to resolve the changes.

Resolving changes means that the resolver checks the recordstorage in the query component for changed, inserted and deleted records. Each of those changes are then attempted to be reflected back on the database by building parameterized SQL statements like 'INSERT …VALUES….', 'DELETE…. WHERE…', and 'UPDATE…. SET....' and then execute the appropriate statement with the appropriate values against the DBDEMOS database.

To be able to generate the correct SQL statements, the resolver needs to know the name of the table to receive the data. We know that it's the 'customer' table. The resolver obtains this information from the query component's TableName property. Thus set
`kbmMWBDEQuery.TableName:='customer'`

Another thing the resolver needs to know is the name of the field(s) that constitutes a unique key. We know that it's the 'custno' field that represents the unique key. Thus set
`kbmMWBDEQuery.KeyFieldNames:='custno'`

If the unique key consists of multiple fields, separate the fieldnames with semicolon. E.g.
`'field1;field2'`.

Now everything is ready to be able to resolve changes back to the datastore/database.

All what's needed is to call the **Resolve** method of the query component at an appropriate time or in the case of this application click the Resolve button.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Error handling

Then what happens if the resolve of some reason cannot be fulfilled either partially or completely?

During the resolve process, the resolver builds an error table (accessible via the kbmMWBDEQuery1.ErrorTable property) which, amongst other things, contains references to all the records that cannot be resolved and where no resolver setting prevents the registration of the error (the Ignorexxxx properties).

The error table contains 4 standard fields which names are given via the following constants:

```
KBMMW_ERROR_RESOLVER_RECORDID_FIELDNAME = 'KBMMW_RECORDID';
KBMMW_ERROR_RESOLVER_TYPE_FIELDNAME = 'KBMMW_ERRORTYPE';
KBMMW_ERROR_RESOLVER_MESSAGE_FIELDNAME = 'KBMMW_ERRORMESSAGE';
KBMMW_ERROR_RESOLVER_DATA_FIELDNAME = 'KBMMW_DATA';
```

You can then access the **ErrorTable** directly after a resolve, and check the records there to determine if any errors has occurred during resolving. You will find a message describing the reason in the field 'KBMMW_ERRORMESSAGE'.
And if the contents of the field 'KBMMW_ERRORTYPE' has the value 0 (kbmMWErrorResolverDB) then the resolve didn't succeed for the specified record. Please notice that other values, for example: 1 (kbmMWErrorResolverModified) also exists. That error type doesn't really indicate an error, but rather that a field value has been changed while resolving, and the user probably should be informed about it. It doesn't indicate a failed resolve.

To locate the actual record in the query that didn't resolve, use search for the RecordID using the **SearchRecordID** method which returns a recordnumber which is the record asked for. Eg.

```
var
  i:integer;
begin
  q.CurIndex.SearchRecordID(
             q.ErrorTable.FieldByName('KBMMW_RECORDID').AsInteger,
             i);
  if (i<0) then Raise Exception.Create('Record not found');
  q.RecNo:=i+1;

  // Now the failing record has been found.
  ShowMessage('Record with value '+q.Fields[0].AsString+' didn''t resolve.');
end;
```

The 'KBMMW_RECORDID' field identifies the record. If the resolve originated from a client query, the record id is the record id of the client record. If not, the record id is the id of the server side query records.

It's possible that error records are registered without any record ID. That means that the error isn't specific to any record, and thus it can't be looked up.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

A less manual and much easier way is to write some code in the **OnResolveError** event of the query or stored procedure component.

The event will be called for each record which didn't resolve with the type (always `kbmMWErrorResolverDB`) and information about the error message.

If possible, it automatically makes sure that the current record in the query/storedprocedure dataset, is the failing record and thus information about the record can be obtained through normal field operations. Check the **Current** argument to see if a current record could be determined.

Finally the **OnResolveError** event have a **Retry** argument which is default true. It can be set to false if this particular record should not be re-resolved on next resolve operation. All failing records are normally automatically marked to be resolved again on next opportunity.

Original field values for the current record can be obtained like this:

```
var
   oldvalue:variant;
   n:integer;
   us:TUpdateStatus;
begin
   // Get the original value for a field as a variant.
   oldvalue:=q.GetVersionFieldData(q.FieldByName('somefieldname'),9999);

   // oldvalue contain the value of the field 9999 versions ago…
   // which in reality means the original value.
   // The number of versions of the record can be obtained by
   n:=q.GetVersionCount;

   // And the updatestatus of a particular version can be obtained by:
   us:=q.GetVersionStatus(x);

   // where x is the version number. 0=current version, 1=older version,
   // 2=even older version etc.
end;
```


Copyright 2001-2005 Components4Developers
http://www.components4developers.com

COMPONENTS 4 DEVELOPERS

Raising errors manually

On occasions, it's practical to validate the resolve operation while it's taking place, and potentially stop it or signal a problem back to the user.
Let's say you want to check the contents of a new record before its being inserted into the database.

Write some code for the OnInsert event handler of the resolver component. E.g

```
procedure TForm1.kbmMWBDEResolver1Insert(Sender: TkbmMWCustomResolver;
  var Skip, Abort: Boolean);
begin
     if Sender.ValueByFieldName['afieldname']=22 then
        Skip:=true;
end;
```

What this does is that it checks the value of the field named 'afieldname' for its value. If it's 22, we ask the resolver to skip the insert silently.

We could also have forced an abort of the resolving operation by setting Abort:=true.
The client would immediately get to know that the resolve operation was aborted. However the user wouldn't get any information that may have been provided in the errortable by previous records being resolved in the same operation.

If we would like to signal a problem with a specific record in a way that the user would be able to receive via the error table, and without aborting the resolve operation as such, we can use the LogError and LogRecordError methods. E.g.

```
procedure TForm1.kbmMWBDEResolver1Insert(Sender: TkbmMWCustomResolver;
  var Skip, Abort: Boolean);
begin
     if Sender.ValueByFieldName['afieldname']=22 then
     begin
          Sender.LogRecordError('We don'''t like the value 22 in afieldname');
          Skip:=true;
     End;
end;
```

LogRecordError logs an error in the errortable along with a record reference, in such way that the record in question can be selected automatically for the user to scrutinize.

If there are errors that can't really be related to a specific record, instead use LogError.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Modifying values during resolve

Its possible by the developer to change the values that the resolver is operating on either right before the resolving process, or as part of the actual resolving.

The resolver has an event named **OnResolveInitialization**. Its triggered whenever a resolve operation is about to start up i.e. when the dataset.Resolve method is called.

In it its valid to modify the values of the complete dataset that are to be resolved. You can access the dataset via the resolver's **Dataset** property.
Remember that whatever changes you make at this point will be reflected directly to the resolving operation. I.e. if you delete a record from the dataset, the resolver will try its best to delete it from the backend database.

Hence the **OnResolveInitialization** event allows you to operate the complete dataset. However don't try to add or delete fields, as the structure of the dataset must not be changed.

Another way to change the values that the resolver obtains from the dataset while resolving, is via the OnGetValue event of the resolver.

It allows you to provide a new value for a specific field for the resolve operation of the current record. E.g.

```
procedure TForm1.kbmMWBDEResolver1GetValue(
  ADeltaHandler: TkbmCustomDeltaHandler; AField: TField;
  var AValue: Variant);
begin
     if AField.FieldName='field1' then
        AValue:=22;
end;
```

In this case, the resolver will always get the value 22 for the field named field1. That may be of relatively little use unless you know what the value is supposed to be used for. Is it an insert operation, you may want to impose a specific value, but if it's an update, you may want to use the value coming from the dataset itself. E.g.

```
procedure TForm1.kbmMWBDEResolver1GetValue(
  ADeltaHandler: TkbmCustomDeltaHandler; AField: TField;
  var AValue: Variant);
begin
     if (TkbmMWCustomResolver(ADeltaHandler).State=mwrsInsert) and
        (AField.FieldName='field1') then
        AValue:=22;
end;
```

Now we check for if the operation is for a record that is being inserted. In that case we force the value 22 to the field named field1 in the resolve operation.

You can at anytime check for the resolvers state which can be one of: **mwrsInsert**, **mwrsDelete**, **mwrsModify** and **mwrsIdle** (when the resolver isn't resolving).

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Fine… we can now impose new values while resolving… but isn't this going to confuse our user? The user sees one value in his disconnected dataset, and the database contains another.

We need to let the user know about a value change has been made. This is easily done by using the resolvers **LogFieldValueChange** method. E.g.

```
procedure TForm1.kbmMWBDEResolver1GetValue(
  ADeltaHandler: TkbmCustomDeltaHandler; AField: TField;
  var AValue: Variant);
begin
     with TkbmMWCustomResolver(ADeltaHandler) do
     begin
          if (State=mwrsInsert) and (AField.FieldName='field1') then
          begin
               AValue:=22;
               LogFieldValueChange([AField],[AValue],
                      'We chose 22 instead for field1');
          end;
     end;
end;
```

The **LogFieldValueChange** method takes 3 arguments, an array of fields within the dataset being resolved that are being changed, a matching array of variant values indicating the new values, and finally a message of your choice.

Default the changes logged via this method will automatically be updated in the originating dataset, and hence the user will see the new values immediately after the resolve.

If you want to notify the user about the changes, you can do that via the dataset's **OnResolveFieldValueChange**. E.g.

```
procedure TForm1.kbmMWBDEQuery1ResolveFieldValueChange(Sender: TObject;
  Fields: array of TField; Message: String; RecordID: Integer;
  Current: Boolean; var Skip: Boolean);
var
   s,a:string;
   i:integer;
begin
     s:='';
     a:='';
     for i:=low(Fields) to high(Fields) do
     begin
          s:=s+a+Fields[i].FieldName+' to '+Fields[i].AsString;
          a:=',';
     end;
     ShowMessage('The following fields have changed their value:'+s);
end;
```


Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Notice that the Fields array actually references the fields in the errortable, and not the fields in the dataset. That way you can still obtain the original field value like this:

```
Originalvalue:=yourquery.FieldByName(Fields[i].FieldName).AsString;
```

The M**essage** argument contains the message that was provided when the field(s) changed values during resolve. The S**kip** argument can be set to true (default false), to ignore all the changes reported for the current record. That means that those values aren't updated in the users disconnected dataset.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

COMPONENTS DEVELOPERS 4

Doing additional database operations while resolving

There may be requirements to update other tables out of the resolving process, perhaps based on special conditions.

To do this you can access the Connection property of the resolver which contains the connection currently used for the database operations. By using this connection, all the updates you may be adding to your database will also be part of the complete transaction. Eg.

```
procedure TForm1.kbmMWBDEResolver1Delete(Sender: TkbmMWCustomResolver;
  var Skip, Abort: Boolean);
var
   c:TkbmMWBDEConnection;
   q:TQuery;
begin
     c:=TkbmMWBDEConnection(Sender.Connection);
     q:=TQuery.Create;
     try
        q.Database:=c.Database;
        q.SQL:='INSERT INTO TRACE (MESSAGE) VALUES (''DELETING '+ValueByName['somefield']+''')';
        q.ExecSQL;
     finally
        q.Free;
     end;
end;
```


Copyright 2001-2005 Components4Developers
http://www.components4developers.com

COMPONENTS 4 DEVELOPERS

Resolving autoinc fields

Autoincrement fields are usually not resolved per the default settings of the resolver property **UpdateInsertAutoIncFields**.

However while the record is inserted, the autoincrement field will automatically be assigned a new unique value by the database. kbmMW supports getting that value for a good part of the supported databases (although not guaranteed for all).

When a record has been inserted, you can use the following resolver properties to obtain the recorded new autoinc value for that record (only one autoinc field supported per table).

```
property LastAutoIncValueByIndex[AIndex:integer]:integer
property LastAutoIncValueByTableName[ATableName:string]:integer
```

LastAutoIncValueByIndex require you to provide an index from 0 to the number of tables being resolved less one (see resolving joins).
LastAutoIncValueByTableName allow you to provide the table name for which you want to get the latest autoinc value. The table obviously must be part of the tables being resolved.

The resolver automatically logs field changes when an autoinc field change value during resolve.

When we later talk about resolving joins, we will see how to use this to effectively resolve master/detail relations which involves autoinc fields.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Field selective resolves

Then what if you have a statement in which you have some field which should not be resolved at all? E.g.

```
SELECT * from customer order by custno
```

And you don't want to resolve the field *Contact*, or perhaps a derived field which contains the content of adding two other fields together.

There are several ways to avoid resolving a specific field:

1) Setup **Origin** on all fields to be `customer.` except for the field you don't want to resolve which you leave empty. Set **SkipFieldsWithoutOrigin** to **true** on the resolver.
2) Set the **ProviderFlags** of a field not to include `pfInUpdate`. This way you can also control if a field should be considered part of the unique key or not during resolve by setting `pfInKey` and `pfInWhere`.
3) Use the database adapter's resolver's event **ExcludeFromUpdateInsert** and **ExcludeFromWhere**. Simply return a set of matching field provider flags in the Flags argument of the event excluding the operation you don't want the field to participate in. The returned set can contain `mwpfInUpdate`, `mwpfInInsert` and `mwpfInKey`.

Further the resolve have several properties which can be set to include or exclude resolving fields marked as readonly in the dataset, key fields and autonumeric fields namely **UpdateInsertReadonlyFields**, **UpdateInsertAutoIncFields**, **InsertKeyFields**, **UpdateKeyFields**.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Resolving joins

Let’s extend the project to use a more advanced select statement.

```
SELECT * from customer,orders where orders.custno=customer.custno order by
customer.custno
```

This is a join of two tables which makes the resolving process a little bit more complex.
When the statement is executed, all fields from customer and orders are returned in one single record. Some of the field names collide in the two tables, eg. custno and taxrate which both exists in the customer and the orders tables. Since no two fields can have the same name in a result set, the two nonunique field names are automatically given a unique name. Thus we will have customer.custno, customer.taxrate, orders.custno_1 and orders.taxrate_1 as resulting fields.

Trying to resolve this directly clearly wont work since custno_1 or taxrate_1 do not exist in the base tables. Thus we have to specify field origins, i.e. which table and fieldname a field originates from. Some database adapters are able to automatically give this information, like the ADOX adapter, but some are not – like the BDE in join situations, simply because the database API does not support reporting that type of information.

This means that the developer will have to specify it.
Before resolving one should thus make sure that all fields which need to be resolved back have their origin values set. This is usually not needed if only one table is involved, as the returned field names match the actual field names in the database table.

In our sample, we enter the SQL statement into the SQL property of the **TkbmMWBDEQuery** component.

Then double click the **TkbmMWBDEQuery** component to bring up the field editor.
Right click and select *‘Add all fields’*.
This persistently defines all fields in the query component. If you are using the same query component for many different types of queries, you need to setup the field origins at runtime in code.

If you have persistently defined fields, you can setup the field origins at designtime. In the field designer multiple fields can be selected at the same time using the Shift or Ctrl buttons while clicking on a field.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Select all fields which originate from the Customer table. Bring up the object inspector (press F11). In the **Origin** property type: `customer.`

Notice the trailing dot!

This means that these fields originate from the table named ‘customer’. If the trailing dot was missing, kbmMW would think the fields originated from a database field named ‘customer’.

If the Origin ends with a dot, it means that the field name in the database table is the same as the TField’s fieldname.

Then select all fields which originate from the Orders table and set their Origin to: `orders.`
Again notice and remember entering the trailing dot!

Then we have to make sure that the fields that were assigned a unique name during the query get special treatment.

Select the field in the field list named CustNo_1. Set its Origin property to: `orders.custno`
Select the field in the field list named TaxRate_1. Set its Origin property to `orders.taxrate`

Now we have specified which table and which database field a field originates from.
Next step is to enter some information for **TableName** and **KeyFieldNames** as we did in the *‘Simple resolving’* chapter.

In this case we want changes to both customer fields and orders fields to be resolved.
Thus we set

```
TableName:='customer;orders'
```

This tells the resolver that two tables needs to be resolved. Only fields belonging to tables listed in the TableName property are resolved. Thus it’s perfectly legal to have a join, but only resolve changes for one or some of the tables in the join.

Finally we set

```
KeyFieldNames:='customer.custno;orders.orderno'
```

This specifies the fields on the tables which are to be known as unique key fields.
If a unique key is defined by two fields in one table, simply specify both fieldnames fully qualified with table name.

The property **SkipFieldsWithoutOrigin** on the resolver would normally be set to true to avoid confusion about where a field - which have no origin - should go. If all fields have an origin value set, SkipFieldsWithoutOrigin have not effect.

Resolving joins is performed under full transaction control (provided the backend databases support it) which means that if an error occurs during the resolve on one of the tables, all updates for all tables are rolled back.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

That's all what's needed to resolve this type join.

The keys for the master/detail relation are manually determined by the user or application when adding new records.

However what if the keyfields were of type autoinc? If that would be the case, the values provided from the disconnected user dataset for the customer table would not be the values that would be stored in the database, simply because the database automatically assigns a new unique number to the autoinc (key) field on insertion of a new customer record.
In that case the field value in the orders table (detail table) that reference the relevant record in the customer table (master table for which the orders belong) will not match any longer.

There is a solution to that too.

Make sure that the master table is listed first in the TableName property, followed by the detail table. E.g. `q.TableName:='customer;orders'.`
Doing that ensures that fields belonging to the customer (master table) is resolved first, followed by fields from the orders (detail) table.

Because of that we can use the LastAutoIncValue…methods to obtain the correct newly created autoinc value for the customer table and use that value as the reference value in the orders table. E.g.

```
procedure TForm1.kbmMWBDEResolver1GetValue(
  ADeltaHandler: TkbmCustomDeltaHandler; AField: TField;
  var AValue: Variant);
begin
     with TkbmMWCustomResolver(ADeltaHandler) do
     begin
          if (State=mwrsInsert) and (AField.FieldName='CustNo') then
          begin
               AValue:=LastAutoIncValueByTableName['customer'];
               LogFieldValueChange([AField],[AValue],
                       'Referenced customer record was changed to '+AValue);
          end;
     end;
end;
```

At the same time we remember to log the change so the disconnected dataset can be automatically updated with the new valid values without having to reopen the query.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Resolving multiple server-side datasets in one transaction

If you have multiple datasets, possibly originating from different database servers of even different types, and you resolve the changes in each dataset, one by one, you can have the problem that one dataset is resolved ok, and another is not which may leave you with a set of partially updated databases.
To remedy that, kbmMW employs a server side (db adapter side) component called **TkbmMWTransactionResolver**.

This component handle distributed transaction control. It means that it automatically will start, commit or rollback transactions on multiple databases at the same time, virtually operating like the updates on all the databases are all running in one big transaction.

It's very easy to use. Add an instance of **TkbmMWTransactionResolver** to your application at design time, or create one at runtime. Eg.

```
var
   t:TkbmMWTransactionResolver;
begin
   t:=TkbmMWTransactionResolver.Create(nil);
   try
     t.Resolve([dataset1, dataset2, …, datasetn]);
   finally
     t.Free;
   end;
end;
```

Each dataset will automatically be resolved according to the normal resolving settings used when resolving the dataset standalone. The difference with the transaction resolver is that if one of the datasets does not resolve properly, all the resolves for all datasets will be rolled back. It however requires that the backend database fully supports transactions.

What if you want not only to resolve changes in datasets, but also do other custom SQL operations within the same transaction?

That's simple.

We add a query component containing the SQL you want to introduce to your database while resolving. E.g.

```
qSpecialStuff.SQL.Text:='INSERT INTO sometable (X,Y,Z) VALUES (10,22,33)';
```

Now, remember to set its **TransactionOperation** property to **mwtoExecute** (it's default **mwtoResolve**).

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Then put qSpecialStuff into the list of datasets to resolve in the transaction resolver. E.g.

```
t.Resolve([dataset1, dataset2, qSpecialStuff, …, datasetn]);
```

Now the transaction resolver will resolve dataset1 and dataset2 in that order, then execute the SQL you have in qSpecialStuff and continue with resolving the remaining datasets.

You can introduce any number of SQL statements in different places in the transactional resolving operation this way.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

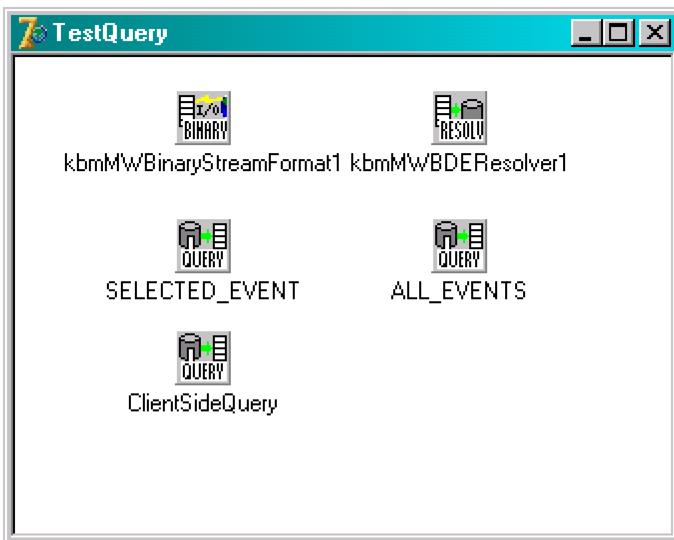
N-tier client side resolving

Keeping the knowledge from the previous chapters present, we will now resolve data in an n-tier setup from the client. The interesting part is that now we have an application server in-between through which the client communicates.

The application server

First create an application server containing a query service following the guidelines from '*Using kbmMW as a query server*' whitepaper, or use one of the predefined ones like the BDE application server.

The query service could look like this (as copied from the BDE server):



The query components all have the **Resolver** property hooked up to the kbmMWBDEResolver1 component.

This query service contains 3 query components where two of them contain a predefined SQL statement while one (ClientSideQuery) require the client to provide the SQL statements.

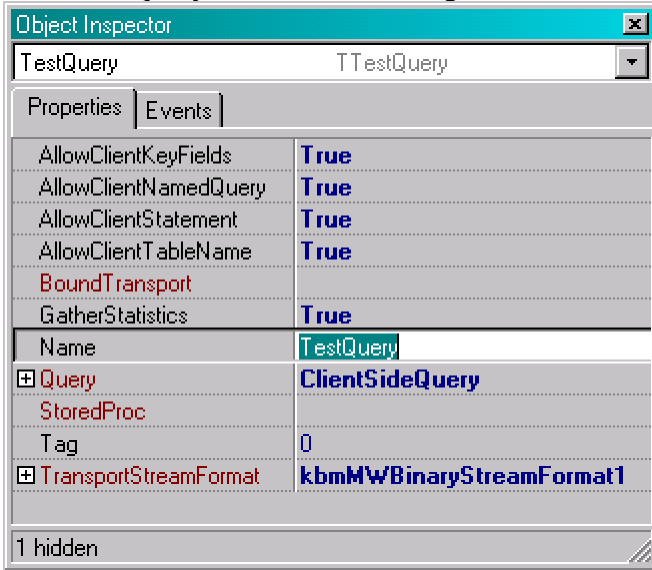
In case of SELECTED_EVENT and ALL_EVENTS, the **TableName** and **KeyFieldNames** properties should be given at server side. This makes sure that the client do not have to worry about that when the client want to resolve.

Also field **Origins** and/or **Providerflags** should be specified on the server side for those two queries if they are needed at all (eg. in join situations etc.). The client will automatically get to know the values defined on the server.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

The ClientSideQuery is another story. Since the client can give any SQL statement it cares for, predefining **TableName**, **KeyFieldNames**, **Origins** and **ProviderFlags** wont make sense on the server. Its the responsibility of the client to setup that.

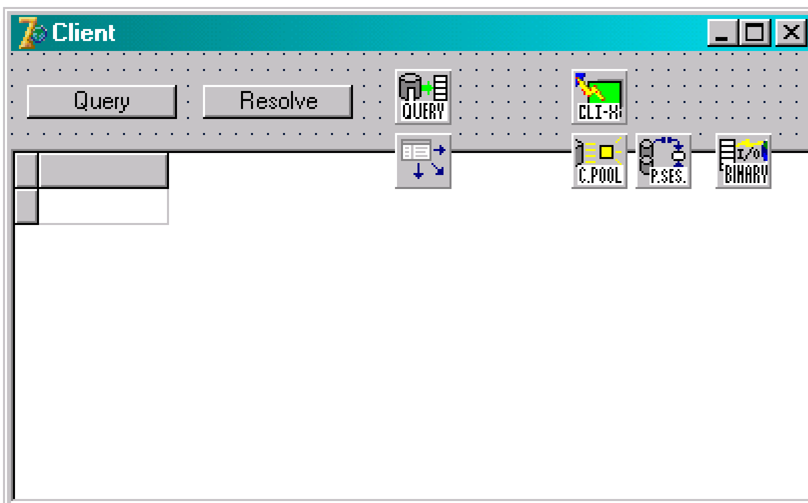
To let the client provide TableName and KeyFieldNames settings, a couple of properties must be set on the query service authorizing the client to do so.



AllowClientKeyFields and **AllowClientTableName** must be true. In addition, **AllowClientStatement** should be true to let the client specify a query statement (SQL) of its own.

AllOrNothing on the resolver component should usually also be set to true to disallow partial successful resolves.

The client application



Copyright 2001-2005 Components4Developers
http://www.components4developers.com

The client contains a **TkbmMWClientTCPIPIndyTransport**, a **TkbmMWClientConnectionPool**, a **TkbmMWPooledSession**, a **TkbmMWClientQuery** and a dataset streamformatter in addition to the dataaware controls.
All components are hooked together according to the 'Using kbmMW as a query server' whitepaper.

If the client should access the predefineded server query ALL_EVENTS, the query statement in the client should be '@ALL_EVENTS' utilizing a named query statement.

When opening the query, all informations needed to resolve are automatically transferred from the server to the client so that the client can advice the server about them later on when/if the client choose to resolve changes in the resultset.

In other words, all what the developer needs to do is to open the query, let the user alter data in the grid, and press a key which calls the **Resolve** method of the client query component.

If the client will itself provide query/SQL statements, its now the responsibility of the client to setup **TableName**, **KeyFieldNames**, optionally field **Origins** and **ProviderFlags**.
In some cases, the server is able to deliver correct Origin values, but this is very much dependent on the database and its API as mentioned earlier on.

Error handling after a resolve event on the client is exactly the same as on the server, described in a previous chapter.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Resolving multiple client-side datasets in one transaction

We have already looked at the TkbmMWTransactionResolver which, on the server side, you be used to resolve multiple datasets under full transactional control as one atomic operation.

The same functionality exists on the client side via **TkbmMWClientTransactionResolver** .

To use it the following properties must be set:

ConnectionPool	Should point on the **TkbmMWClientConnectionPool**
QueryService	Should contain the name of the query service on the app server that is responsible for the resolving.
QueryServiceVersion	Should contain the version string matching the query service.
TransportStreamFormat	Should point on a dataset transport format component matching the one on the server side (eg. **TkbmMWBinaryStreamFormat**).

All that is needed now is to call the Resolve method to resolve the client datasets. Eg:

```
Myclienttransactionresolver.Resolve([clientdataset1,…, clientdatasetN]);
```

If the resolve call returns false, the resolve process didn't succeed, and ErrorTable's should be checked to see reason (or the **OnResolveError** event used).

As with the server side transactional resolving component, you can introduce special SQL statements by including a dataset containing the special SQL in the list of datasets to resolve, and set its **TransactionOperation** to **mwtoExecute** instead of **mwtoResolve**.

Copyright 2001-2005 Components4Developers
http://www.components4developers.com

Creating a custom resolver

In some very special situations you might want to be in full control of what the resolver is doing. It could for example be that the resolving process should not operate against the database but instead update a file system or something else.

In these cases, its often easier to create a custom resolver. This is an example of a simple resolver.

```
TYourResolver = class(TkbmMWCustomResolver)
protected
  function DoInsertRecord(AObject:TkbmMWCustomResolverObject):boolean; override;
  function DoModifyRecord(AObject:TkbmMWCustomResolverObject):boolean; override;
  function DoDeleteRecord(AObject:TkbmMWCustomResolverObject):boolean; override;
end;

function TYourResolver.DoInsertRecord(AObject:TkbmMWCustomResolverObject):boolean;
var
   i:integer;
   v:variant;
   s:string;
begin
     // Getting new values for the record to be inserted.
     for i:=0 to FieldCount-1 do
     begin
        v:=Values[i];     // The value for a field.
        s:=FieldNames[i]; // s contain the name for the field.

        // Do whatever you need here.
        …

        // If you raise an exception it will automatically add the exception message to
        // the ErrorTable and continue with next record, unless AllOrNothing is true.
     end;

     // If you don't return true, an error record will be added to the ErrorTable.
     Result:=true;
end;

function TYourResolver.DoModifyRecord(AObject:TkbmMWCustomResolverObject):boolean;
begin
     Result:=true;
end;

function TYourResolver.DoDeleteRecord(AObject:TkbmMWCustomResolverObject):boolean;
begin
     Result:=true;
end;
```

Then simply assign an instance of the resolver to the **Resolver** property of the server query/stored procedure component. You can register the resolver as a component to be able to use it at designtime just like any other custom component.

This completes the whitepaper about resolving.

Kim Madsen
Components4Developers

Copyright 2001-2005 Components4Developers
http://www.components4developers.com