Resolving

for kbmMW v. 1.05+

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 completely.

Simple 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.

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 coorporation with the query component. The query component is responsible for keeping track of all deletes, updates and inserts made to the resultset.

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).

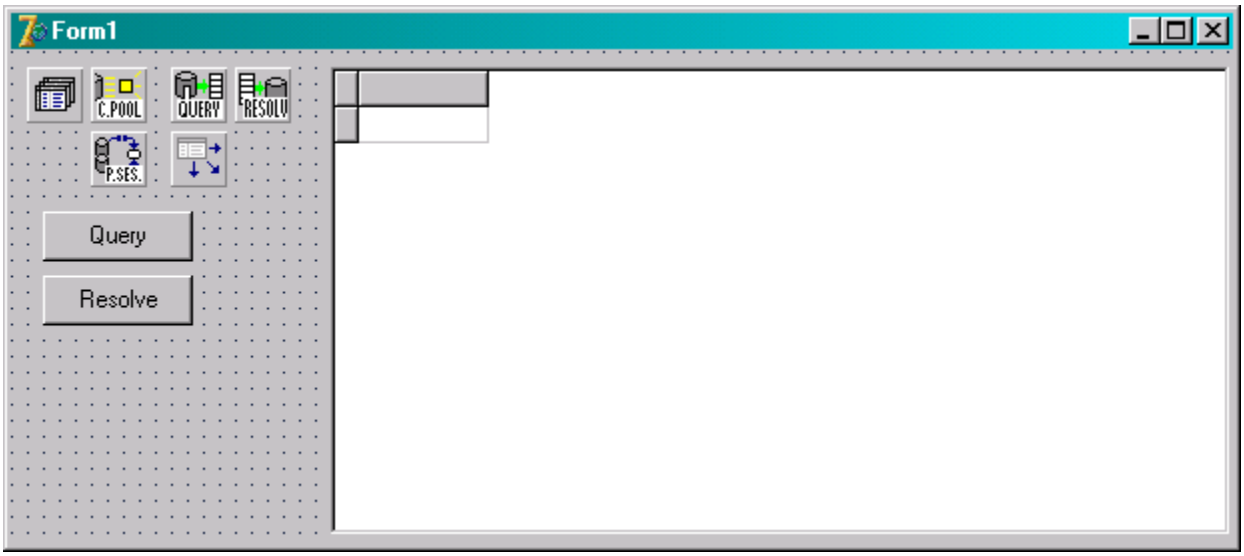
Copyright 2001-2003 Components4Developers
http://www.components4developers.com

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. (using StartTransaction, Commit and Rollback).

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
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;
```

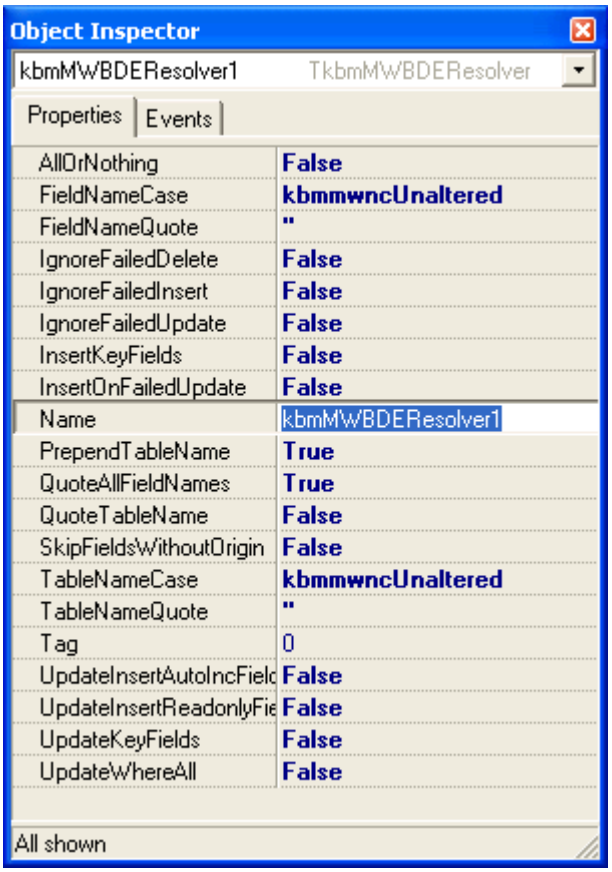
The event handler of the Resolve button contains:

```
kbmMWBDEQuery1.Resolve;
```

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.

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

The BDE Resolver is like many of the bundled resolvers SQL compatible. Thus a set of SQL related properties are available and can be modified if needed (usually its not required).



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.

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.

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.

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

QuoteAllFieldNames - If true, will put FieldNameQuote around all fieldnames. If false, only fieldnames which contains invalid characters will be quoted.

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

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

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

TableNameCase - Like with FieldNameCase.

TableNameQuote - The quote to put around table names.

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.

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 and the connection released back to the connection pool.

Thus modifying a record in the DBGrid do 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 need to know is the name of the field(s) that constitute 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 consist of multiple fields, separate the fieldnames with semicolon. Eg.

```
'field1;field2'.
```

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

All whats 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 2001-2003 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 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 3 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';
```

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.

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.

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 and message information given. It also automatically makes sure that the current record is the failing record and thus information about the record can be obtained through normal field operations.

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.

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

Original field values for the record in question 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 2001-2003 Components4Developers
http://www.components4developers.com

Resolving joins

Let's extend the project a bit to contain a bit 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, ie 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 do not support reporting that type of information.

This means that the developer will have to specify it.
Thus one should before resolving make sure that all fields which needs to be resolved back must have an origin value 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.

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

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

Then we have to make sure that the fields that was 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 its 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.

This is all whats needed to do to resolve the data changes in joined situations.

Copyright 2001-2003 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? Eg.

```
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 2001-2003 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.

Its 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;
```

IMPORTANT: If some of the datasets originate from the same connection pool, that connection pool must have **MaxConnetions** set to either -1 (unlimited) or a value higher than the number of datasets from that connection pool resolved within the same transaction. If not, a deadlock situation will occur.

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

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

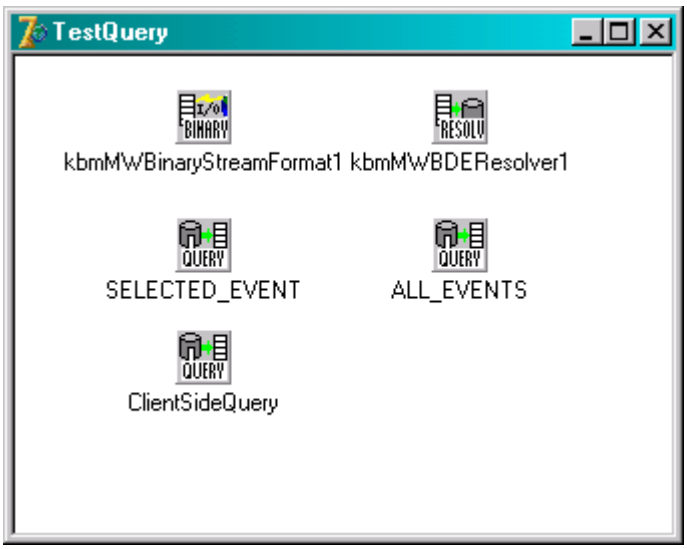
Simple client side resolving

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

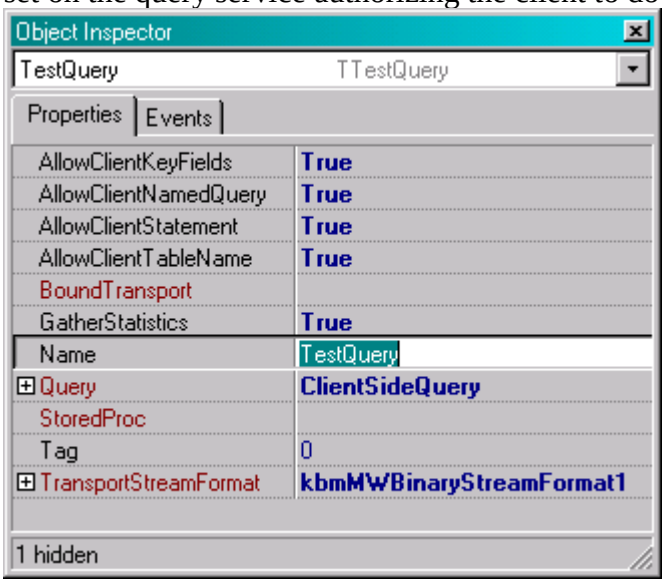
In case of SELECTED_EVENT and ALL_EVENTS, the **TableName** and **KeyFieldNames** properties should be given at server side. This makes sure that the client do not have to worry about that when the client want to resolve.

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 2001-2003 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.

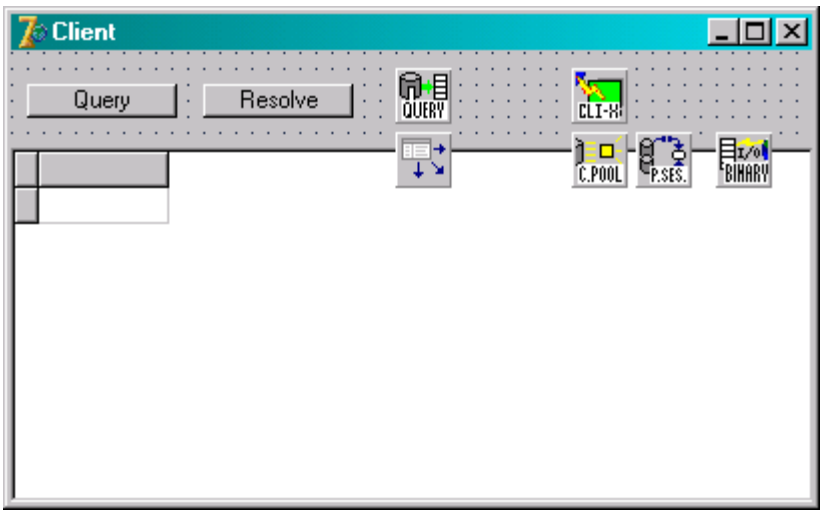
To let the client provide TableName and KeyFieldNames settings, a couple of properties must be set on the query service authorizing the client to do so.



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 2001-2003 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 2001-2003 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 multiple dataset resolving, the server side connection pools must be configured to allow for the number of open connections needed by the resolving process.

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

Advanced topics

Server side constraint checking

Its easy to add complex constraints to what is allowed to resolve and what is not.
The resolver's `OnDelete`, `OnModify` and `OnInsert` events are called before the actual operation against the database is happening.

In the events code can be added to check for special cases which should not be allowed to resolve. Eg.

```
procedure TForm1.kbmMWBDEResolver1Delete(Sender: TkbmMWCustomResolver;
  var Skip, Abort: Boolean);
begin
     if Sender.ValuesByName['fieldname']=10 then
        Skip:=true;
end;
```

This example simply skips resolving the current record if the value of the field 'fieldname' is 10.
The resolving is not interrupted for other records.
The client will see the resolve operation as successful.

```
procedure TForm1.kbmMWBDEResolver1Delete(Sender: TkbmMWCustomResolver;
  var Skip, Abort: Boolean);
begin
     if Sender.ValuesByName['fieldname']=10 then
        Abort:=true;
end;
```

This example raise an **EkbmMWAbortException** immediately on the client, and operation will not continue. No errortable will be sent to client. Usually make sure to set the resolver to **AllOrNothing** to ensure an automatic rollback of the resolves.

```
procedure TForm1.kbmMWBDEResolver1Delete(Sender: TkbmMWCustomResolver;
  var Skip, Abort: Boolean);
begin
     if Sender.ValuesByName['fieldname']=10 then
        raise Exception.
           Create('Fieldname must never be 10 of some obscure reason!');
end;
```

This example raise a user defined exception which is automatically converted to an entry in the **ErrorTable**, after which the resolving operation may continue if **AllOrNothing** is false.

The same way, one can do more things at the same time while a record is being inserted, updated or deleted. You might have a trace table on the same database which you want to update at the same time to reflect the changes made by the resolver. Or you want to enforce referential integrity when records are deleted etc.

Copyright 2001-2003 Components4Developers
http://www.components4developers.com

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 2001-2003 Components4Developers
http://www.components4developers.com

Altering values during resolve programmatically

In a few cases you might want to modify the values given by the dataset to something else.
A thought example could be that you allow the user to enter string values with case, but when the data should be stored in the database, you want it converted to uppercase, or you want to store a completely different value all together.

In this case one can use the `OnGetValue` event on the resolver. Eg.

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


Copyright 2001-2003 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 2001-2003 Components4Developers
http://www.components4developers.com