COMPONENTS 4 DEVELOPERS

Example on datawarehousing using kbmMW

Prologue

What is a datawarehouse? It's a large collection of data from various sources, organized in a way that makes it possible to effeciently search to extact required data.

In other words, it require data to be organized in a way that makes it easy to access and search the data. Typically true datawarehouse centers, put out requirements for data to be entered into their systems and databases. The data must follow certain rules to be easily searchable, and that means in some situations that data must be reorganized to live up to that requirement, at the time the data is inserted into the datawarehouse.

However situations exist where data, of various reasons, cant be reorganized and must be provided into a datawarehouse as is. It can for example be in situations where the data is used as evidence in crime cases, or where the amount of data in each package simply makes it impractical to reorganize it everytime a new package arrives.

This article describes, how such a scenario has been handled using kbmMW, allowing users to search many terabytes of data (that number is constantly increasing) stored in thousands of separate databases (also constantly increasing).

The situation is that a company receives large (gigabyte sized) databases in SQLite format every day from various places. The data, amongst other things, contains hundred of thousands of records of records with detailed information about how a technical instrument is operating, and in addition contains a lot of video (frame by frame) video footage too.

Each technical device may (typically will) produce many databases over time, each containing the results of a single production run.

Each database files overall structure cant be modified, per requirement of the company. And they are not interested in duplicating information into a different database system of various reasons, one of them being security.

Earlier on, they were used to picking a database file at a time, and remotely over the net, analyze it using for example Matlab or other tools. However the performance of such operation was terrible, due to the enormous amount of data passing over the net. That lead to people trying to duplicate data into other local databases that were easier to handle, but that wasn't practical from several perspectives:

- Security… confidential raw data was then not as contained as they would like it to be.
- Security… few parts of the database was allowed to be updated by experts. However if duplicate versions of that data was floating around, then there were no guarantee that the updates actually entered into the original database. Manual processes were untolerable for this.

- Performance… Manually (or semi autimatically) copying parts of data from many database files to a local storage, was time consuming, space consuming and error prone.

Thus a better solution was sought. And that's where kbmMW comes into play.

As its probably well known, kbmMW is a middleware, that provides the glue between a server application (or in kbmMW terms.. application server) and one or more clients, that allow the clients to request various operations on the application server, and expect responses for those requests.

One of the many features provided by kbmMW is the ability to act as a middleman between a database and a client. By putting kbmMW inbetween the client and database, one gains several advantages including:

- Significantly reducing the network traffic required to access the database.
- Significantly increasing the security level around the database.
- Significantly increases the response time on the client.

Another feature of kbmMW Enterprise Edition is its ability to communicate completely in async mode via its transport framework called WIB (Wide Information Bus). When operating with the WIB all clients and servers are just nodes. The servers are nodes that publish certain info and subscribes for requests, while the clients typically are nodes publishing requests and subscribing for responses.

kbmMW v. 3.50.01 Enterprise Edition which is currently running in beta 2 and is about to be released any day now, contains features that allow a server provide incremental responses to a clients request for data from a database. Its not a simple fetch on demand scenario we are talking about where the client asks for more and more info. Its instead a server side push of data records (virtual or real) that are relevant for the ongoing search, whenever data is available.

Futher we also need some sort of connection management for accessing the thousands of databases. Preferably one where we can cache the connections to each database and reuse it later on.

Hence we would like to end up with a client that simply makes a request for data, for example asking for temperature data for all productions from a specific technical instrument. That data can then be analyzed in various ways afterwards which is out of the scope of this article. (In the specific company case, a scripted, vector oriented integrated development environment was developed for them, making it easy to analyze massive amounts of data using similar to Matlab methods, and under full control play synchronized video from the frames in the databases).

How building such an application server and client can be done using kbmMW, is what I will show in the remaining of the article.

The application server

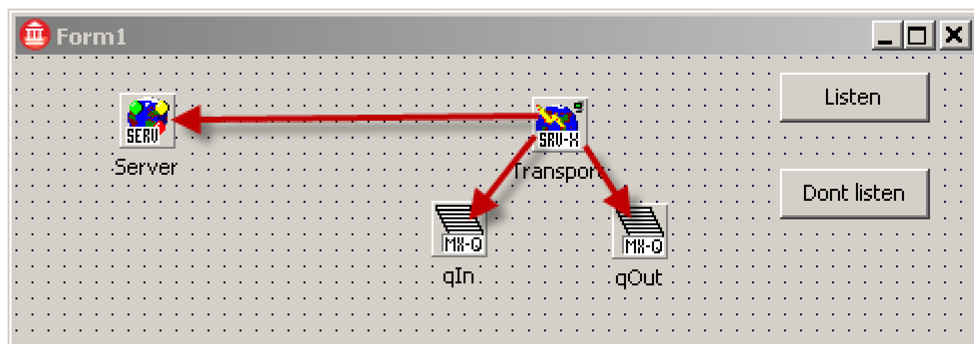
We will first build a simple standard kbmMW application server with a build in query service for accessing databases. Afterwards we will modify it bit by bit to give us the functionality we need as described in the prologue.

We start out with a new empty VCL forms appilcation. It means the application server will run as a regular application. In real life scenarios, one should create an application server as a service, which is illustrated in one of the many samples available on the kbmMW/Downloads/Samples section of Components4Developers home page. As producing Windows services is not the scope of this article, I just take the shortcut and create a simple windows application that operates as a server.

We add the following:

- TkbmMWServer component. Name it Server.
- TkbmMWTCPIPIndyMessagingServerTransport component and name it Transport. Set its property Server to point on Server, and its ClusterID to 'Demo'.
- Two message queue components for the in and out bound messages for the transport. Name one of them qIn and the other qOut. Set the properties Transport.InboundMessageQueue to point at qIn, and Transport.OutboundMessageQueue to point at qOut.
- We also add two buttons Listen and Don't listen to activate and deactivate the server.

The form should look something like this:



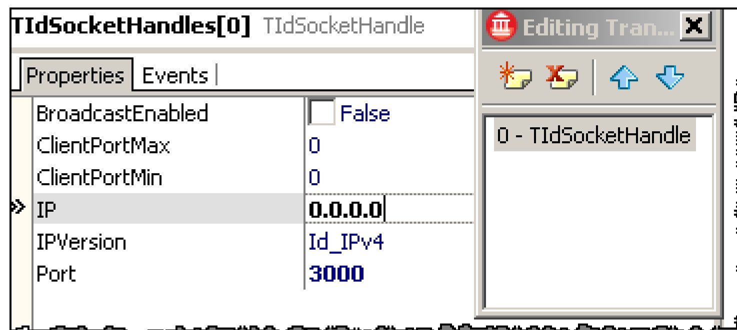
Lets add some code to the Listen and Dont Listen buttons:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     Transport.Subscriptions.Clear;

     // Subscribe for requests to this server.
     Transport.Subscriptions.Subscribe(
         kbmMWGenerateRequestSubscriptionSubject(Transport.ClusterID)
     );

     // Subscribe for subscriptions/unsubscriptions to this server.
     Transport.Subscriptions.Subscribe('SUB.'+Transport.ClusterID+'.>');
     Transport.Subscriptions.Subscribe('USB.'+Transport.ClusterID+'.>');

     // Activate server and transport.
     Server.Active:=true;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
     Server.Active:=false;
end;
```

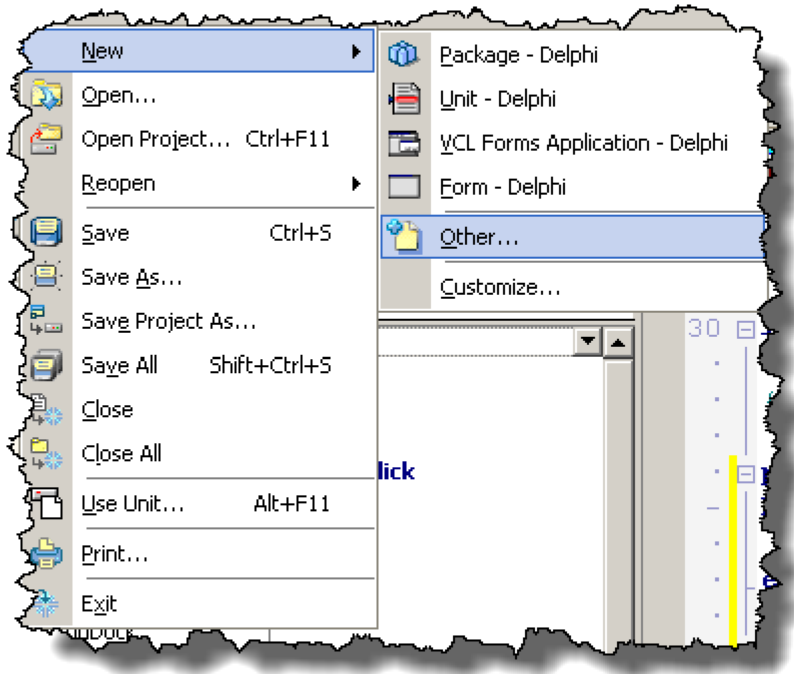
With messaging transports, correct subscriptions are crucial, otherwise the node (in this case the application server) will not receive the correct messages. As we will have a messaging based client later on, we need to tell the server that it should accept 3 type messages: Requests (REQ), Subscriptions (SUB) and Unsubscriptions (USB), the later not really required in this demo, but its good practice to pair up SUB and USB subscriptions. In all cases the application server subscribes only for requests etc. for the cluster called Demo (set in the transports ClusterID property).

Then lets tell the Transport that it should allow connections from anywhere. Open its Bindings property and at designtime add a new binding (Ip4).

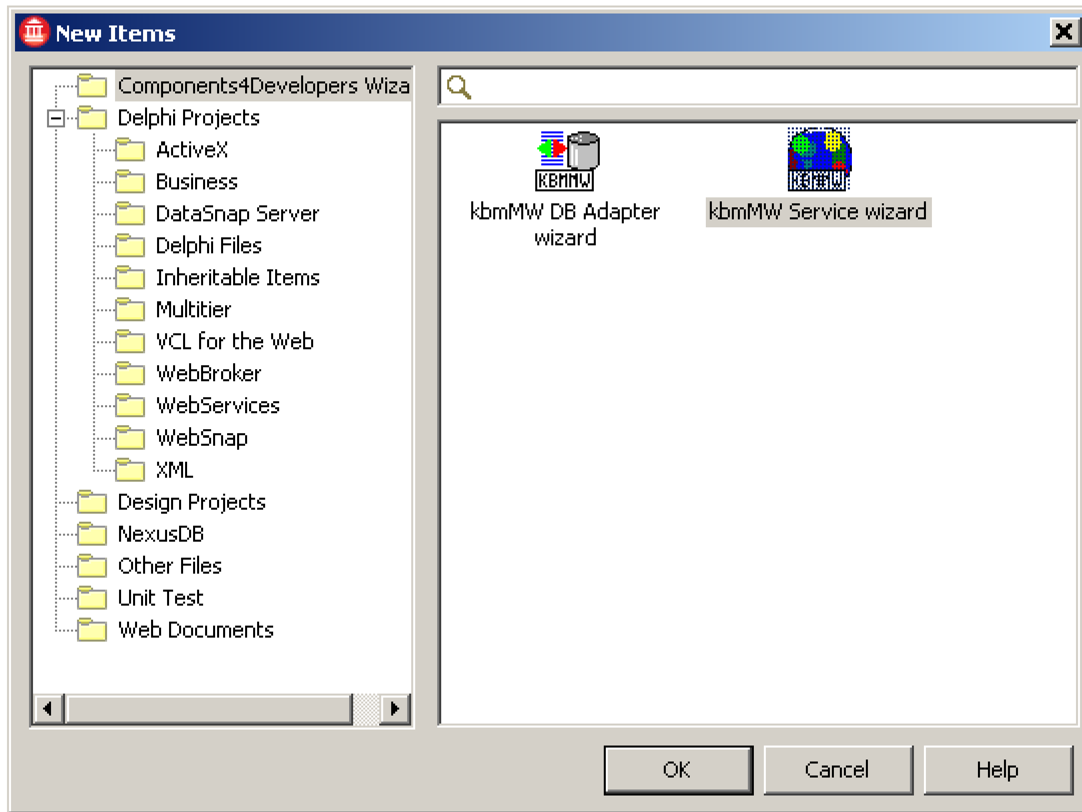
Set the port number to 3000 and the mask to 0.0.0.0.



Then we add the query service via the service wizard:



Locate the Components4Developers Wizards and clicj the kbmMW Service Wizard:



Now first page of the wizard is shown. Select the Query service/kbmMW_1.0 service type and click Next:

COMPONENTS 4 DEVELOPERS

kbmMW service wizard

Service admin

New custom service

Service type selection

Please select th

Custom servi
Simple servic
Eventoperated service
Inventory service / kbmMW_1.0
Custom query service
Query service / kbmMW_1.0
Custom loadbalancing service / kbmMW_1.0
Standard loadbalancing service / kbmMW_1.0
Filetransfer Service / kbmMW_1.0
ProxyService / kbmMW_1.0
Web server service / kbmMW_1.0
Event operated Web server service / kbmMW_1.0
Java based service / kbmMW_1.0
ActionScript based service / kbmMW_1.0

xtend the service. If not checked will use the service as is.

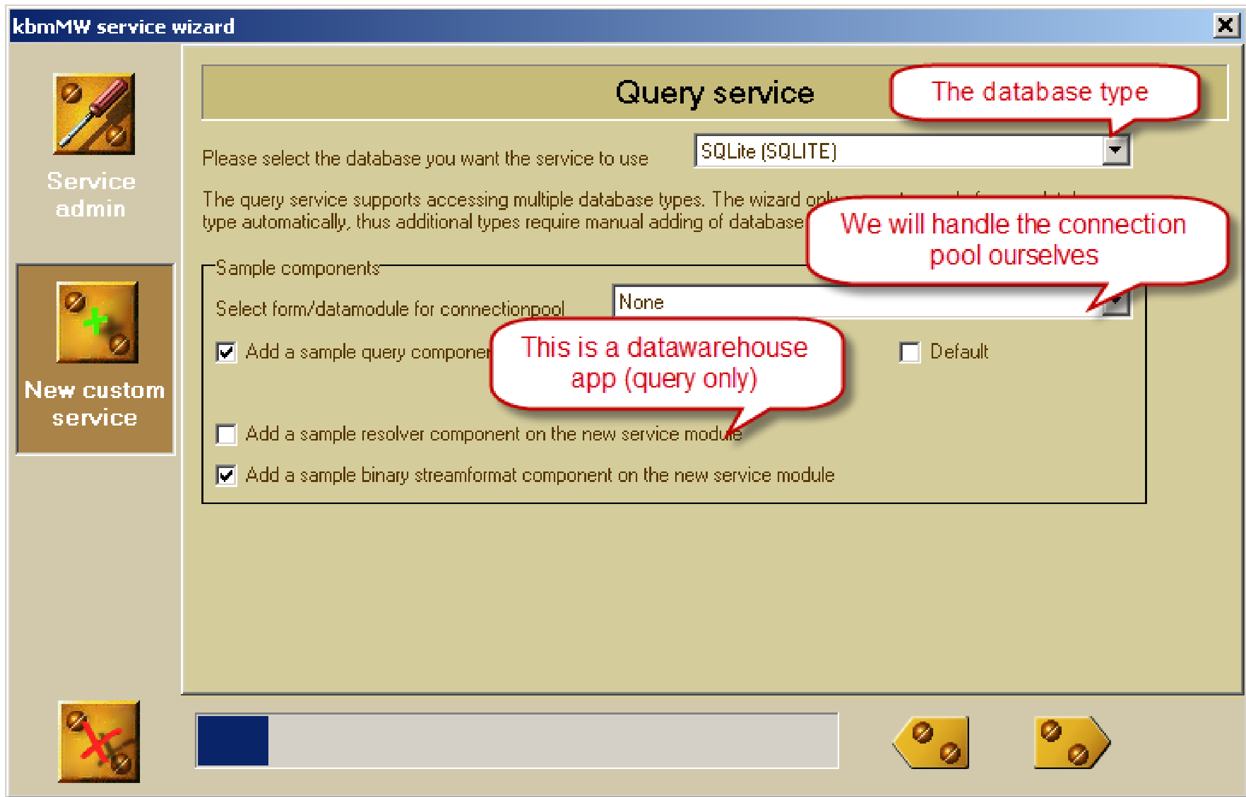
Define descriptive information

Use this selection if you want to publish backend database informations to clients. The queryservice will work like an advanced TDatamodule where you can put kbmMW database components.

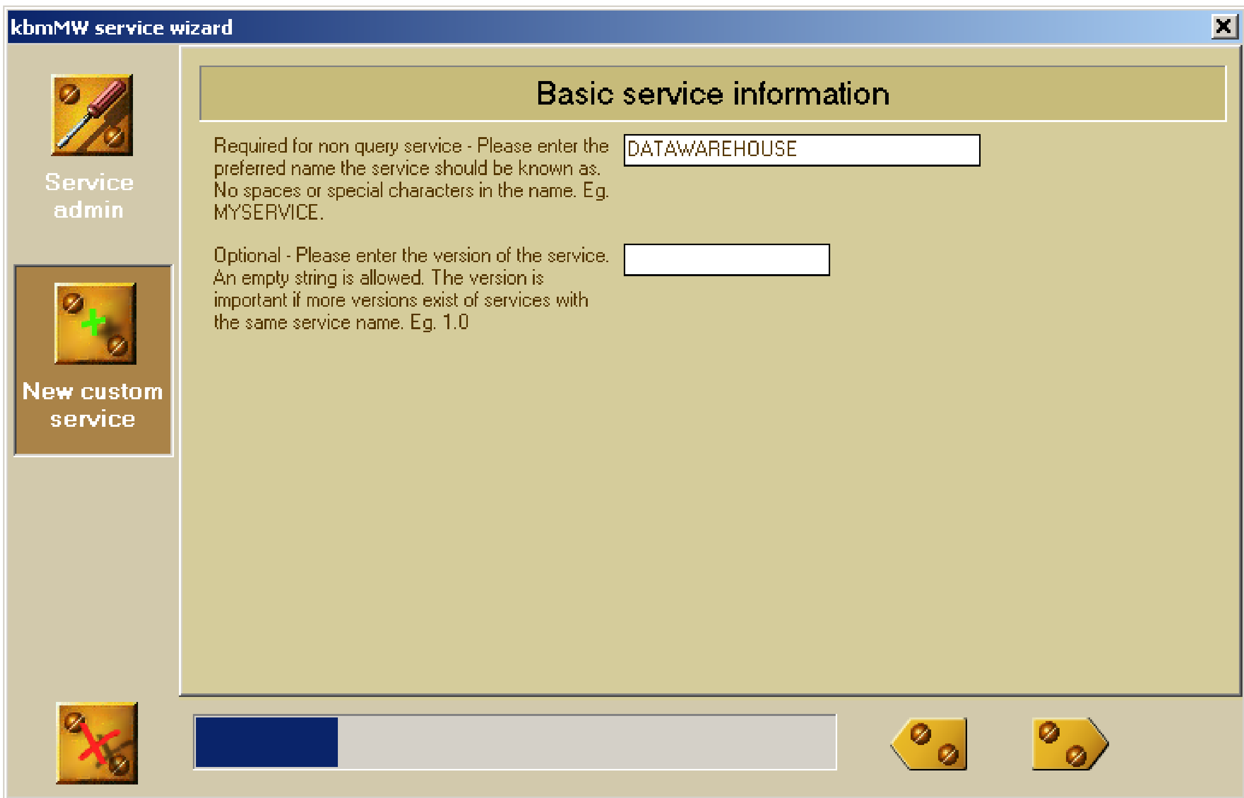
Service type selection

Next button

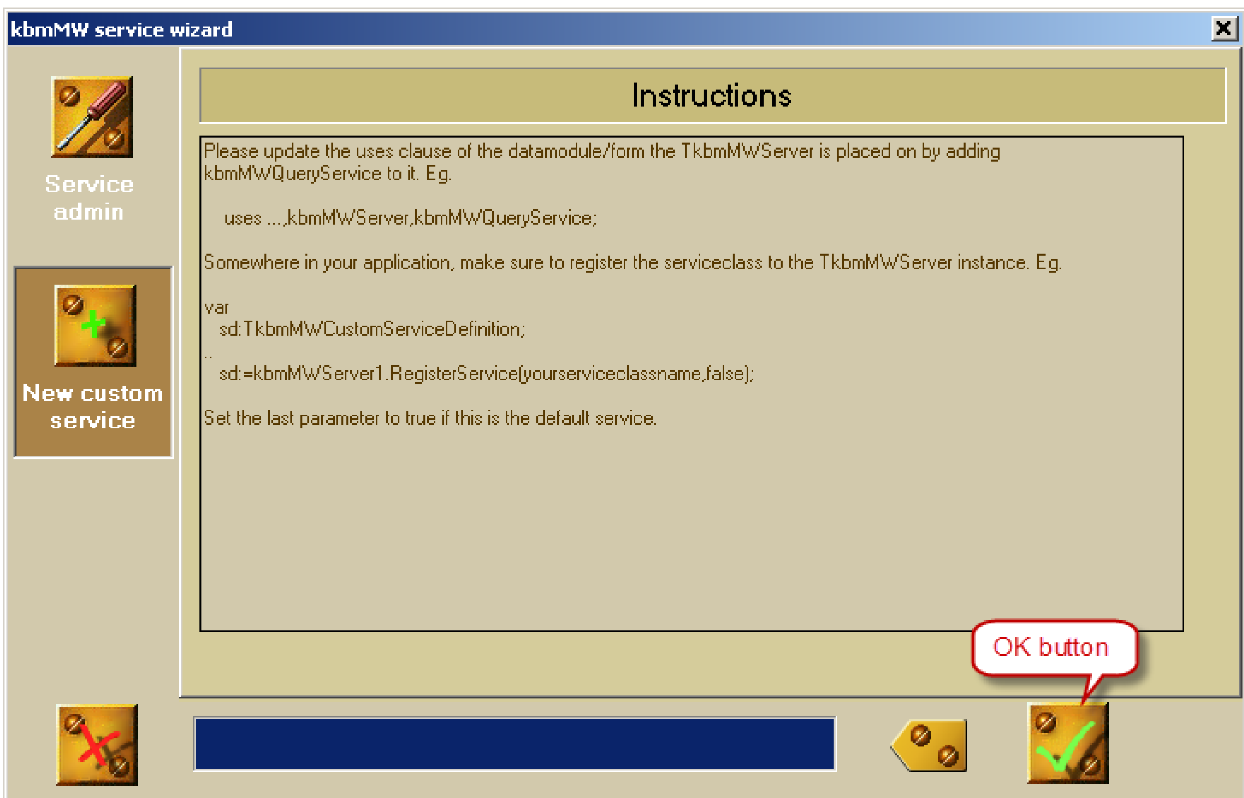
Now we will be given the option to choose what type database we would like to access. In this sample we choose to use SQLite that is supported by all kbmMW Editions. Then click Next.



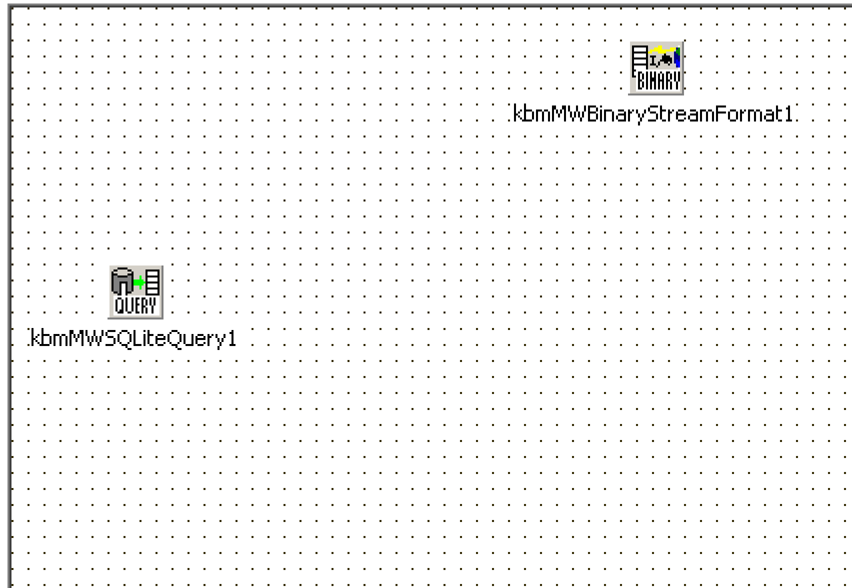
We will then be given the opportunity to name the service, and optionally give it a version. Versioning a service can be smart if, at some point, the service interface changes, and we want to support both older and newer clients. Lets name the service DATAWAREHOUSE and keep the version empty. Then click Next.



Now click next thru all following pages in the wizard. And then click the Ok button on last page.



This generates a new datamodule for us, with a couple of components on it:



The data module will be used by clients making database requests to the application server. Because we want to access a large number of SQLite databases, we have not defined a SQLite connection pool (as would usually have been normal to do) on the application servers main form (we deselected that option in the wizard). Instead we will make our own list of known SQLite databases, from which we dynamically can pick one or more when needed.

COMPONENTS DEVELOPERS 4

Next step is to register this service to the application server (Server component) on the main form. That only needs to be done once, thus the main form OnCreate event will be a fine place for doing that.

```
implementation

uses Unit2;

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
     Server.Active:=true;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
     Server.Active:=false;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
     Server.RegisterService(TkbmMWQueryService2,false);
end;

end.
```

For simplicity sake for this sample application , we design the application server to know about databases with filenames DB1.db, DB2.db, DB3.db.. DBn.db. The client will tell us what database number(s) we are to search in, at the appropriate time. This sample assumes that there is a table in each database called 'DATA', and that table has same structure in all databases.

Each database file needs its own TkbmMWSQLiteConnectionPool, that takes care of connections to the database, caching of resultsets and metadata and more. Thus when the client specify to access a specific database, we can choose to create a TkbmMWSQLiteConnectionPool on the fly, connect it up towards the relevant database and execute the client query request, or we can choose a better performing method, where we keep a list of all previously accessed databases, and thus keep the databases open and ready for use. A pool of connection pools. We'll show a simplistic way to do that now.

We define a thread safe hash table that will contain all the previously accessed database's connection pools. The unit kbmMWGlobal contains lots of nice containers and other goodies, so we add it to the uses clause.

```
uses
  Windows, Messages, SysUti
  Dialogs, kbmMWCustomMessa
  kbmMWCustomServerMessagin
  StdCtrls, kbmMWGlobal;
```

Then we define a field in the Tform1 class, which is to hold the connection pools for the databases we have accessed and thus to allow us to reuse the connection pools later on, without having to reopen the database files:

```
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    DBs:TkbmMWThreadHashStringList;
  end;
```

And add the relevant code to construct the instance and destruct it upon form creation and destruction time. We specify that objects that are are 'managed' by the hashlist are also deleted automatically by the hashlist when entries are delete from the list or the list itself is freed.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
     DBs:=TkbmMWThreadHashStringList.Create(100);
     DBs.FreeObjectsOnDestroy:=true;

     Server.RegisterService(TkbmMWQueryService2,false);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
     FreeAndNil(DBs);
end;
```

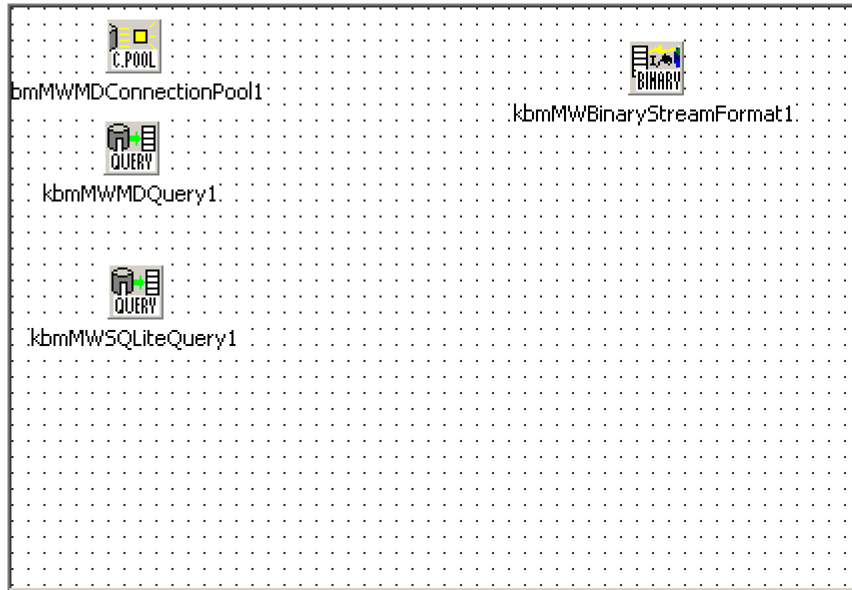
Then we make a couple of methods to fetch database connection pools from our hash list (our pool of connection pools), and automatically create new connection pools when new databases are accessed. We first add the unit kbmMWSQLite to the uses clause of the unit Unit1.pas. Then we add this method:

```
function TForm1.GetConnectionPool(ADatabaseName:string):TkbmMWSQLiteConnectionPool;
begin
     DBs.BeginWrite;
     try
        Result:=TkbmMWSQLiteConnectionPool(DBs.GetObject(ADatabaseName));
        if Result=nil then
        begin
             Result:=TkbmMWSQLiteConnectionPool.Create(nil);
             try
                Result.Database:='yourdbdirectory\'+ADatabaseName+'.db';
                Result.Active:=true;
                DBs.AddManagedObject(ADatabaseName,Result);
             except
                FreeAndNil(Result);
                raise;
             end;
        end;
     finally
        DBs.EndWrite;
     end;
end;
```

The method first ensures that all access to the contents of DBs is protected so only one thread at a time can access it. Then we look up a connection pool based on the database name. If none has been found, we create a new one, and add it 'managed' to the DBs storage. If the database the client is requesting, actually doesn't exist at all, this method will throw an exception. Other ways to indicate the issue to the client could also be coded.

Next step is to define a so called virtual dataset on the query service. Right now we have a TkbmMWSQLiteQuery component there, and that will be used for the query against a specific database. However we would like to interact with the client in such a way, that we don't just send the complete result (containing the combined matching records for all client specified databases) in one go to the client. Instead wewould like to send incremental resultsets. In this sample we choose to send all matching records from one database to the client as one incremental resultset.

On the queryservice (unit2.pas) we put a TkbmMWMDQuery and a TkbmMWMDConnectionPool.

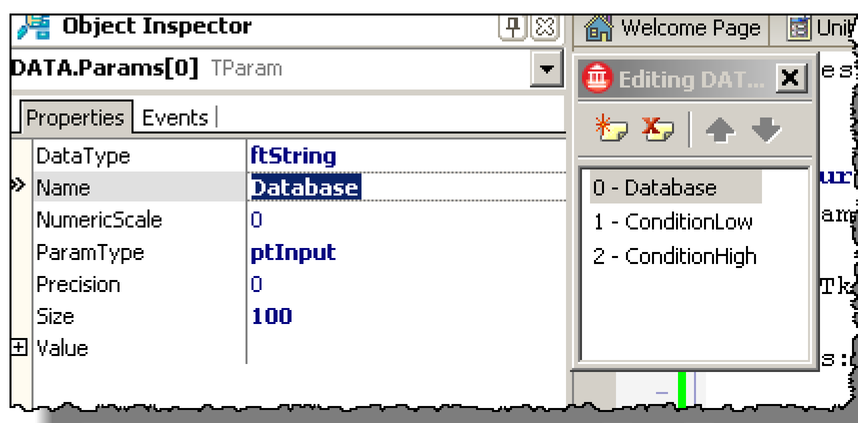


We rename the kbmMWMDQuery1 to ‘DATA’, and set its published property to true and its connection pool to point on the kbmMWMDConnectionPool1 component. That way clients can request data from this particular component. The virtual memory dataset (thus the MD acronym) provides some interesting events for us, namely the OnPerformFieldDefs and OnPerformQuery events.

The OnPerformFieldDefs allow us to easily define field definitions at runtime on the fly. We could also define them at designtime for this demo because we know the structure of the SQLite table DATA doesn’t change for different databases served by this application server. However we’ll define the definitions in code. Its also possible to define parameters in the same method, but since we need a way the client query can provide information about database name and some search criteria, we will define parameters for that at designtime.

```
procedure TkbmMWQueryService2.DATAPerformFieldDefs(Sender: TObject;
  AParamsOnly: Boolean);
var
   ds:TkbmMWMDQuery;
begin
     ds:=TkbmMWMDQuery(Sender);

     if not AParamsOnly then
     begin
          // Define the fields that should be returned to the client.
          ds.FieldDefs.Clear;
          ds.FieldDefs.Add('ID',ftString,20,false);
          ds.FieldDefs.Add('X',ftFloat,0,false);
          ds.FieldDefs.Add('Y',ftFloat,0,false);
     end;
end;
```

Then we add the parameters at designtime via the property Params of the DATA (TkbmMWMDQuery) component.

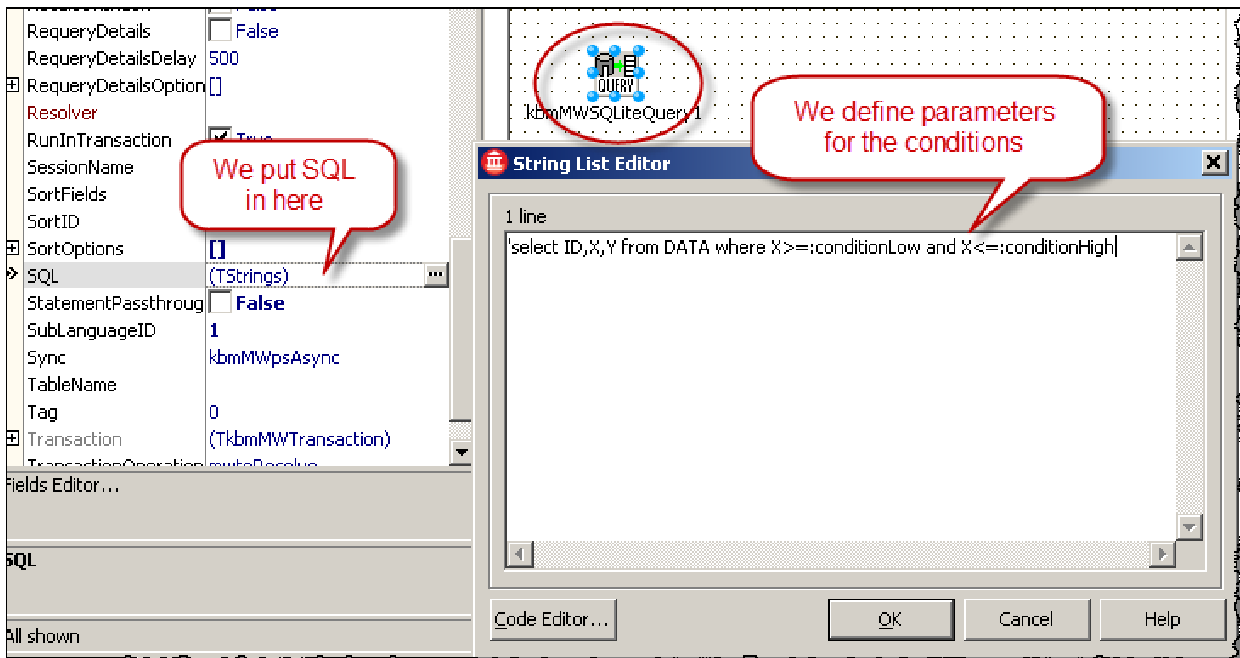


3 parameters has been created:

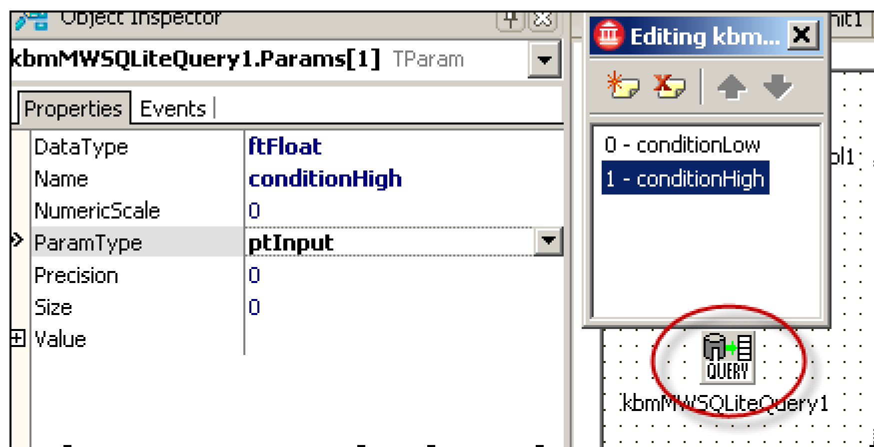
- Database, ftString, ptInput, Size 100
- ConditionLow, ftFloat, ptInput
- ConditionHigh, ftFloat, ptInput

The purpose of these parameters is to allow the client to indicate database number, and some conditions we may choose to use for the selection from a database. By convention we define that Database may contain one or more commaseparated numbers indicating database number for which a result is requested.

The we define the SQL statement to query a single database with. That can be done at runtime or designtime. Since the SQL is the same regardless which database is queried, we define it at designtime.



The parameters must then be configured in the Params property. We have 2 parameters to define: conditionLow and conditionHigh. They need to be of DataType ftFloat and ParamType ptInput.



Finally we need to add some code to do the actual search and return the incremental data. This code should be put in the OnPerformQuery event .

First part of the performQuery event extracts parameter values provided from the client.

```
procedure TkbmMWQueryService2.DATAPerformQuery(Sender: TObject; var ACanCache,
  ACallerMustFree: Boolean; var ADataset: TDataSet);
var
   i:integer;
   sDBID:string;
   ds:TkbmMWMDQuery;
   sDatabase:string;
   slDatabase:TStringList;
   conditionLow,conditionHigh:double;
   cp:TkbmMWSQLiteConnectionPool;
   bFirst:boolean;
   mt:TkbmMemTable;
begin
     ds:=TkbmMWMDQuery(Sender);

     // Get parametervalues.
     sDatabase:=ds.ParamByName['Database'].AsString;
     conditionLow:=ds.ParamByName['conditionLow'].AsFloat;
     conditionHigh:=ds.ParamByName['conditionHigh'].AsFloat;
```

Next part contains a loop that runs for each database that the client have asked the application server to query. For each database, a connection pool is requested, and a native TkbmMWSQLite query is executed. The resulting dataset (if any) is then sent asynchronously to the client via the SendPartialResultDataset method. The method need to know if it's the first partial resultset, or an intermediate one.

The last partial resultset MUST be sent by returning from the eventhandler with a dataset that it can send to the client. That final dataset will be marked as either the complete dataset (if SendPartialResultDataset was never called) or the final partial dataset (if SendPartialResultDataset has been called at least once). kbmMW keeps track of that internally.

The purpose of that, is 2 ways:

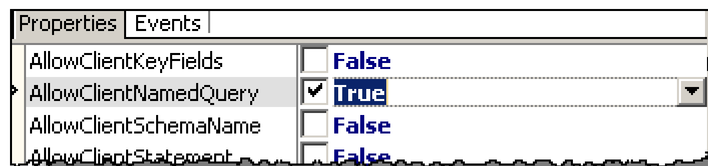
1) The server don't need to send complete field definitions for intermediate or final partial datasets.
2) The client know if there are about to come more partial datasets or if all has been received.

```
    conditionHigh:=ds.ParamByName('conditionHigh').AsFloat;

    // Extract list of database id's to search.
    slDatabase:=TStringList.Create;
    try
       slDatabase.CommaText:=sDatabase;

       // Loop for all database ids.
       bFirst:=true;
       for i:=0 to slDatabase.Count-1 do
       begin
            sDBID:=slDatabase.Strings[i];

            // Lets ignore any exceptions related to accessing a single database.
            try
               // Find or allocate a connection pool access to the database.
               cp:=Form1.GetConnectionPool(sDBID);

               // Setup the SQLite query component.
               kbmMWSQLiteQuery1.ConnectionPool:=cp;
               try
                  kbmMWSQLiteQuery1.ParamByName('conditionLow').AsFloat:=conditionLow;
                  kbmMWSQLiteQuery1.ParamByName('conditionHigh').AsFloat:=conditionHigh;
                  kbmMWSQLiteQuery1.Open;

                  // Now we should have data that we can return to the client.
                  // Its important that the last partial result is sent by
                  // kbmMWs internals, and thus not by us using SendPartialResultDataset.
                  // Since we actually dont know if there is data in the last database
                  // requested, we will let the final package be empty.
                  if kbmMWSQLiteQuery1.RecordCount>0 then
                  begin
                       SendPartialResultDataset(Form1.Transport,
                                                kbmMWSQLiteQuery1,
                                                KBMMW_MESSAGEPRIORITY_NORMAL,
                                                bFirst);
                       bFirst:=false;
                  end;
```

The final part of the event closes the native query, detaches the connection pool from it, and generates a final (empty) dataset to return. Because we don't know if we infact are processing the last dataset (database) at any point in the loop, due to condition may result in 0 records, or database may not exist or other reasons, we send this empty, but confirming dataset as the last one. We also tell the system that it has the responsibility to get rid of our temporary TkbmMemTable, when its done with it.

```
                        bFirst:=false;
                   end;
                finally
                   kbmMWSQLiteQuery1.Close;
                   kbmMWSQLiteQuery1.ConnectionPool:=nil;
                end;
             except
                // Do nothing.
             end;
        end;
     finally
        slDatabase.Free;
     end;

      // If no data collected, complain.
      if bFirst then
         raise Exception.Create('No slides was found to be matching.');

     // Now prepare a final (empty) dataset package.
     mt:=TkbmMemTable.Create(nil);
     mt.CreateTableAs(ds,[mtcpoStructure]);
     mt.Open;
     ADataset:=mt;
     ACallerMustFree:=true;
end;
```

Finally we need to tell the query service that clients are allowed to access its published query components on it. That is done via the property AllowClientNamedStatement on the query service data module.



COMPONENTS DEVELOPERS 4

The client

Now all that's left is to build a client that can talk with the application server.

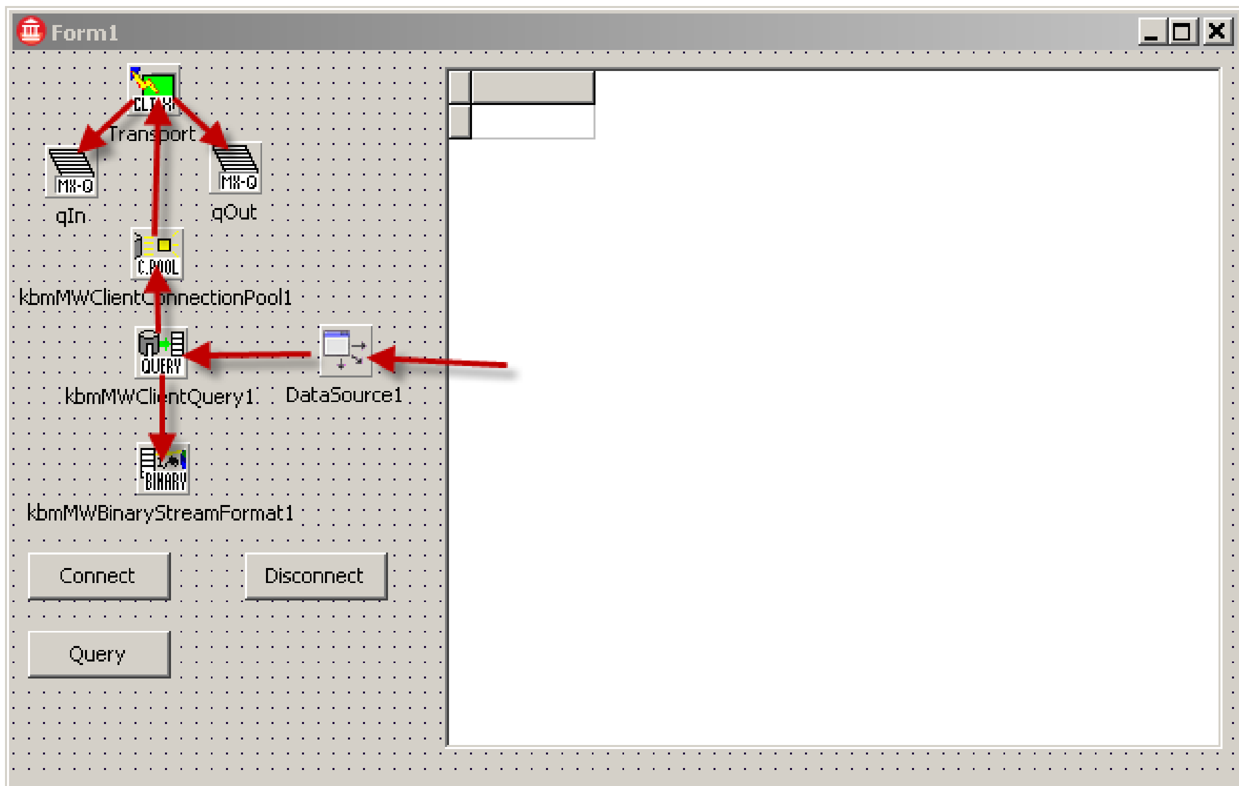
We start out with creating a new VCL Forms application for the client. We add several components:

- TkbmMWTCPIPIndyClientMessagingTransport
- 2 x TkbmMWMemoryMessageQueue
- TkbmMWClientConnectionPool
- TkbmMWClientQuery
- TkbmMWBinaryStreamFormat
- And a datasource, dataaware grid and a couple of buttons.

Transport.InboundMessageQueue is set to point at qIn and Transport.OutboundMessageQueue is set to pont at qOut. The TkbmMWClientConnectionPool.Transport property is set to point at Transport, and kbmMWClientQuery1.ConnectionPool is set to point at the kbmMWClientConnectionPool component.

Further kbmMWClientQuery1.TransportStreamFormat is set to point at kbmMWBinaryStreamFormat1 (matching a similar setting on the query service in the application server).The DB grid is hooked up to the datasource that is hooked up to kbmMWClientQuery1, and Transport.ClusterID property is set to 'Demo'.

The QueryService property of the kbmMWClientQuery1 component is set to the name of the service on the server that we would like to provide data to our client query. That is 'DATAWAREHOUSE'. Hence set the property to that string. As we didn't define a service version on the DATAWAREHOUSE service we should clear out the property QueryServiceVersion.



Lets put some code in the connect/disconnect buttons.

```
procedure TForm1.btnConnectClick(Sender: TObject);
begin
     kbmMWClientConnectionPool1.Active:=true;
end;

procedure TForm1.btnDisconnectClick(Sender: TObject);
begin
     kbmMWClientConnectionPool1.Active:=false;
     kbmMWClientConnectionPool1.KillConnections;
end;
```

We don't in fact immediately connection, but instead tell the connection pool, that maintains connections from the client to the application server, that it can connect when it needs to similarly we tell the connection pool that it should kill its pool of connections when the user click the Disconnect button.

COMPONENTS 4 DEVELOPERS

Then we write some code for the button that makes the request to the server:

```
procedure TForm1.QueryClick(Sender: TObject);
begin
     // We want to access the published query named DATA on the server.
     kbmMWClientQuery1.Query.Text:='@DATA';

     // Lets get the parameters from it, so we can fill them out with
     // relevant query information.
     kbmMWClientQuery1.FetchDefinitions;

     // Setup the parameter values for the query.
     // Search all databases identified with the values 3 and 4 and 5.
     // Search for values in the range 10-20.
     kbmMWClientQuery1.ParamByName['Database'].AsString:='3,4,5';
     kbmMWClientQuery1.ParamByName['ConditionLow'].AsInteger:=10;
     kbmMWClientQuery1.ParamByName['ConditionHigh'].AsInteger:=20;
     kbmMWClientQuery1.AsyncOpen;

     // Now the server will go thru all the specified databases,
     // and search for the relevant criterias, and give us a message
     // for each non empty resultset, via the Transports OnAsyncResponse
     // event.
end;
```

And finally we need to write some code to handle the asynchronous responses, of which there will come minimum 1 and potentially many of for each time one click the Query button.

```
procedure TForm1.TransportAsyncResponse(Sender: TObject;
  const TransportStream: IkbmMWCustomResponseTransportStream;
  const RequestID: Integer; const Result: Variant;
  UserStream: TkbmMWMemoryStream);
var
   ps:TkbmMWDatasetPartialState;
begin
     // Check if response to request made via query component?
     // Compare LastRequestID from the query component with the
     // RequestID for the incoming response message.
     if kbmMWClientQuery1.ActiveClient.LastRequestID=RequestID then
     begin
          ps:=kbmMWClientQuery1.SetQueryResult(Result,UserStream);
          case ps of
            mwdpsAll: ; // We got all in one response message.
                        // No more messages are coming for this request.

            mwdpsInitial: ; // We got initial response message, and thus
                            // the dataset has now been defined with
                            // structure (fields) and some data.

            mwdpsData: ; // We got an intermediate response message.
                         // It was appended to the already existing data.

            mwdpsFinal: ; // We got the final response message for this
                          // request. Now the dataset is complete.
          end;
     end;
end;
```

Provided one have the correct SQLite databases available and configured the kbmMWSQLiteConnectionPool correctly on the application server, its now possible to make asynchronous queries with developer controlled incremental on the fly transfer of partial resultsets to the client.

To sum up what we have shown in this:

- How to create a basic server and client using kbmMW Enterprise Edition
- How to work with virtual memory data sets
- How to work with SQLite databases
- How to work with the Wide Information Bus (WIB) messaging framework
- How to work with developer controlled incremental resultsets