

Datastore metadata

for kbmMW v. 2.00+



kbmMW v2 contains a couple of new components which provides a datastore independant API to managing metadata in the datastore.

Currently kbmMW supports obtaining information about and manipulating metadata for tables, indexes and sequences/generators.

kbmMW v2 bundles the following metadata component

TkbmMWNullMetaData	Gives no access to metadata as such, but can be used when metadata is of no interest for a specific connection pool.
TkbmMWGenericSQLMetaData	Supports obtaining metadata for generic SQL based datastores. Datastore specific metadata components will be able to utilize more of the specific datastore's special functionality. Further it contains several properties which are of importance for SQL based resolvers. This or one of its descendantta should be chosen when using resolvers that are SQL based.
TkbmMWOracleMetaData	Specific Oracle oriented SQL metadata component. Supports Oracle sequence metadata.
TkbmMWInterbaseMetaData	Specific Interbase oriented SQL metadata component. Supports Interbase generator metadata.
TkbmMWDBISAM3MetaData	Specific DBISAM v3 oriented SQL metadata component. Predefined SQL properties for resolving for DBISAM 3.
TkbmMWDBISAM4MetaData	Specific DBISAM v4 oriented SQL metadata component. Predefined SQL properties for resolving for DBISAM 4.
TkbmMWNexusDBMetaData	Specific NexusDB oriented SQL metadata component. Predefined SQL properties for resolving for NexusDB.

Metadata for SQL resolving

When a SQL based resolver is about to resolve changes to the database, it needs to know something about how the datastore wants the fieldnames, tablename and data formatted. In pre v2 releases, these informations were stored on the resolver itself. Starting in v2, they are instead part of the SQL metadata component or descendants thereof.

The following properties can be set on the generic SQL metadata component:

FieldNameQuote	Specify which character to use for quoting fieldnames if thats needed.
FieldNameCase	Set to one of kbmmwncUnaltered , kbmmwncUpper or kbmmwncLower which control the case of the fieldname.
TableNameQuote	Specify which character to use for quoting tablename if thats needed.
TableNameCase	Set to one of kbmmwncUnaltered , kbmmwncUpper or kbmmwncLower which control the case of the tablename.
QuoteAllFieldNames	If set to true, all field names are automatically quoted. Defaults is only to quote a tablename when it contains punctuation or space characters.
QuoteTableName	If set to true, all table names are automatically quoted. Defaults is only to quote a tablename when it contains punctuation or space characters.
StringQuote	Specify which character to use for quoting string datavalues if thats needed.
QuoteStringQuote	Specify which character to quote a quote with in a string, if the string actually contains the quote character as part of its data. Eg. if the string looks like this: don't do this and the StringQuote is set to ' and QuoteStringQuote is \ then the resulting quoted string will be: 'don't do this'
DateLayout	Specify the format to be used when converting a date from a TDateTime value to a string. The DateLayout follows the standard Delphi date-time format strings.
TimeLayout	Specify the format to be used when converting a time value from a TDateTime value to a string. The TimeLayout follows the standard Delphi date-time format strings.
DateTimeLayout	Specify the format to be used when converting a date and time value from a TDateTime value to a string. The DateTimeLayout follows the standard Delphi date-time format strings.
TrueValue	The string to use to specify a true value.
FalseValue	The string to use to specify a false value.
PrependTableName	Set to true if the tablename always should be prepended fieldnames.
SequenceTableName	Specify the name of the pivot table when operating sequence values without using the datastore's special sequence/generator capabilities.
UnicodeOptions	A combination of mwucAutoUTF8 , mwucStringsIsUnicode and mwucAutoParamUTF8 . If mwucAutoUTF8 is set kbmMW will automatically convert to and from UTF8 format when setting and getting string field values into a defined field on the query/stored proc component of type ftWideString. If mwucStringsIsUnicode is set, kbmMW will automatically see all datastore fields of type ftString and ftFixedChar as type ftWideString.

kbmMW

If **mwucAutoParamUTF8** is set, kbmMW will automatically convert to and from UTF8 format when setting and getting parameter values.

A typical setting for accessing Interbase with full Unicode support is to enable all 3 flags. Remember to set the character set to use for the Interbase API to UNICODE._ESS

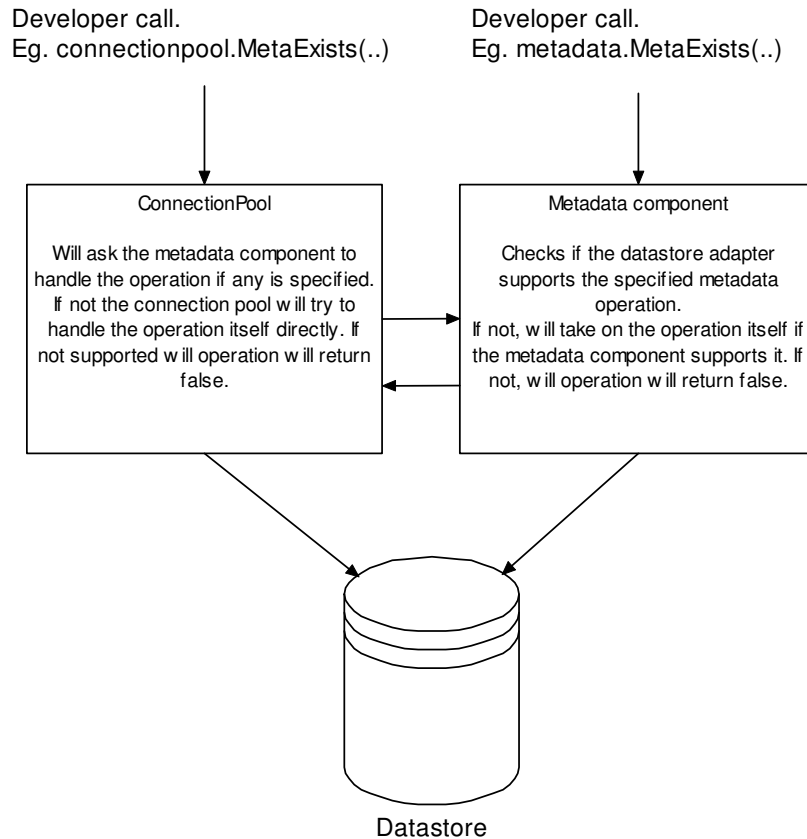
Manipulating datastore metadata

Metadata for a datastore currently include tables, indexes and sequences/generators/pivot tables. To manipulate those in a datastore independant way, the metadata component contain several functions and methods which can be called directly via the metadata component, or via the connection pool (which is the recommended way).

All functions accept a metadatatype value which identifies the type of metadata to operate:

- **mwmtdtTable**
- **mwmtdtSequence**
- **mwmtdtIndex**

If a function is not supported or fail using the current metadata component on the given datastore, the function returns false.





function MetaInitialize(const AMetaDataType:TkbmMWMetaDataType):boolean;

Should be called when initializing a particular type of metadata first time. Its forexample responsible for generating a pivot table for storing sequence numbers if the datastore doesnt have its own sequence generation mechanism, or the metadata component do not support that specific mechanism.

Example: `connectionpool.MetaInitialize (mwmdtSequence);`



**function MetaCreate(const AMetaDataType:TkbmMWMetaData; const AName:string;
const ADef:TkbmMWCustomDef):boolean;**

Function which can be called to create metadata of a specific type. AName must be a valid name for a metadata object, which means it usually should only contain alphanumeric characters (always starting with a letter). Punctuation should be avoided.

ADef is a collection of values for that particular type of metadata.

mwmtdTable : ADef should be of type TkbmMWFieldDefs

mwmtdIndex : ADef should be of type TkbmMWIndexDef

mwmtdSequence : ADef should be of type TkbmMWSequenceDef

Example: Creation of a table:

```
var
  def:TkbmMWFieldDefs;
begin
  def:=TkbmMWFieldDefs.Create;
  try
    // Add a field to the definition.
    with def.AddFieldDef do
      begin
        Name:='fld1';
        Unique:=false;    // Will for SQL type datastores
                          // automatically add an unique index.
        Primary:=false;  // Will for SQL type datastores
                          // define the field as a PRIMARY KEY.
        Required:=false; // Will for SQL type datastores
                          // define the field as NOT NULL.
        DataType:=ftInteger;
        // Size:=xx specify for forexample string fields.
        // Precision:=xx specify for forexample bcd fields.
      end;

      // Add more fields as needed.
      ...

      // Generate the table.
      if not connectionpool.MetaCreate(mwmtdTable,'TABLE1',def) then
        ShowMessage('Failed to create table TABLE1.');
    finally
      def.Free;
    end;
  end;
end;
```



Example: Creation of sequence:

```
var
  def:TkbnMWSequenceDef;
begin
  // Create a sequence definition with initial value of 10000.
  // Notice that some datastore sequencers/generators may not support
  // setting up an initial value in which case its ignored.
  def:=TkbnMWSequenceDef.Create(10000);
  try
    // Generate the sequence.
    if not connectionpool.MetaCreate(mwmdtSequence,'SEQ1',def) then
      ShowMessage('Failed to create sequence SEQ1.');
```



Example: Creation of index on existing table:

```
var
  def:TkbmMWIndexDef;
begin
  def:=TkbmMWIndexDef.Create;
  try
    // Setup general index definition.
    def.TableName:='TABLE1';
    def.Unique:=false;

    // Add a field to the definition.
    with def.AddIndexFieldDef do
    begin
      FileName:='fld1'; // Field to add index to.
      Descending:=false;
    end;

    // Add more fields as needed to the index.
    ...

    // Generate the index.
    if not connectionpool.MetaCreate(mwmdtIndex,'idxTABLE1',def) then
      ShowMessage('Failed to create table idxTABLE1.');
```

finally
 def.Free;
end;
end;



```
function MetaDelete(const AMetaData: TkbmMWMetaData;  
                    const AName: string): boolean;
```

Delete the metadata of the specified type.

Example: `connectionpool.MetaDelete(mwmdtTable, 'TABLE1');`



**function MetaReset(const AMetaDataType:TkbmMWMetaDataType; const AName:string;
const AValue:variant):boolean;**

Resets the specified metadata object. Tables will either be truncated, or all records deleted depending on datastore capabilities. Sequences will be reset to a starting value of AValue. Indexes currently doesnt support reset.

Example: `connectionpool.MetaReset (mwmdtSequence, 'SEQ1', 20000) ;`



**function MetaExists(const AMetaData:TkbmMWMetaDataType; const AName:string;
var AResult:boolean):boolean;**

Return true in AResult if the specified metadata object exists, otherwise returns false.

Example:

```
var  
    exists:boolean;  
begin  
    connectionpool.MetaExists(mwmdtTable,'TABLE1',exists);  
    if not exists then  
        ShowMessage('TABLE1 doesnt exist.');
```

end;



**function MetaList(const AMetaDataType:TkbmMWMetaDataType;
AResult:TStrings):boolean;**

List all the known names of the specified type of metadata.

Example:

```
var  
    sl:TStringList;  
begin  
    sl:=TStringList.Create;  
    try  
        connectionpool.MetaList(mwmdtTable,sl);  
        ShowMessage('The following tables are defined: '+sl.Text);  
    finally  
        sl.Free;  
    end;  
end;  
end;
```



```
function MetaValue(const AMetaData: TkbmMWMetaDataType; const AName: string;  
    var AResult: variant): boolean;
```

Obtain a value from the metadata object. Its currently primarily used for obtaining values from metadata objects of type `mwmtdtSequence`.

Example:

```
var  
    value: variant;  
begin  
    // Draw next value from the sequence named SEQ1.  
    connectionpool.MetaValue(mwmtdtSequence, 'SEQ1', value);  
    ShowMessage('Value drawn from sequence SEQ1 was '+value);  
end;
```



Creating new custom metadata components

Its easy to add new metadata components handling specific types of datastores.

Currently there are three choices to inherit from:

- TkbmMWCUSTOMMetaDATA
- TkbmMWCUSTOMSQLMetaDATA
- Any other existing meta data component

If your metadata component should support SQL datastores, you would generally select TkbmMWCUSTOMSQLMetaDATA as the ancestor of your new metadata component.

If you choose to base your new component on TkbmMWCUSTOMMetaDATA you would optionally need to override one or more of the following methods:

```
TYourMetaDATA=class (TkbmMWCUSTOMMetaDATA)
public
    function MetaInitialize(const AMetaDataType:TkbmMWMetaDataType):boolean; override;
    function MetaCreate(const AMetaDataType:TkbmMWMetaDataType; const AName:string;
        const ADef:TkbmMWCUSTOMDef):boolean; override;
    function MetaDelete(const AMetaDataType:TkbmMWMetaDataType;
        const AName:string):boolean; override;
    function MetaReset(const AMetaDataType:TkbmMWMetaDataType;
        const AName:string; const AValue:variant):boolean; override;
    function MetaExists(const AMetaDataType:TkbmMWMetaDataType;
        const AName:string; var AResult:boolean):boolean; override;
    function MetaList(const AMetaDataType:TkbmMWMetaDataType;
        AResult:TStrings):boolean; override;
    function MetaValue(const AMetaDataType:TkbmMWMetaDataType;
        const AName:string; var AResult:variant):boolean; override;
end;
```

Its important to return false if a method is not able to handle the request, and true if the method takes care of the specific request.

If you choose to base your new component on `TkbmMWCustomSQLMetaData`, you would optionally need to override one or more of the following methods:

```
TYourMetaData = class(TkbmMWCustomSQLMetaData)
protected
    function GenerateCreateTableSQL(const AName:string; const ADef:TkbmMWCustomDef;
        AResult:TStrings):boolean; override;
    function GenerateDeleteTableSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateResetTableSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateExistsTableSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateListTableSQL(AResult:TStrings):boolean; override;

    function GenerateCreateIndexSQL(const AName:string;
        const ADef:TkbmMWCustomDef; AResult:TStrings):boolean; override;
    function GenerateDeleteIndexSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateResetIndexSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateExistsIndexSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateListIndexSQL(AResult:TStrings):boolean; override;

    function GenerateInitializeSequenceSQL(AResult:TStrings):boolean; override;
    function GenerateCreateSequenceSQL(const AName:string;
        const ADef:TkbmMWCustomDef; AResult:TStrings):boolean; override;
    function GenerateDeleteSequenceSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateResetSequenceSQL(const AName:string;
        const AInitialValue:integer; AResult:TStrings):boolean; override;
    function GenerateExistsSequenceSQL(const AName:string;
        AResult:TStrings):boolean; override;
    function GenerateListSequenceSQL(AResult:TStrings):boolean; override;
    function GenerateValueSequenceSQL(const AName:string;
        AResult:TStrings):boolean; override;
end;
```

Again if any of the functions are not able to handle the request its important to return false, else true.

The `GeneratexxxSQL` methods job is to create the relevant SQL in the `AResult` argument which will match what the method is supposed to do.



For example:

```
function TYourMetaData.GenerateValueSequenceSQL(const AName:string;
        AResult:TStrings):boolean;
begin
    AResult.Add('!UPDATE '+FormatTableName(FSequenceTableName) +
        ' SET SEQVALUE=SEQVALUE+1 WHERE SEQNAME='+FormatString(AName));
    AResult.Add('=SELECT SEQVALUE FROM '+FormatTableName(FSequenceTableName) +
        ' WHERE SEQNAME='+FormatString(AName));
    Result:=true;
end;
```

Actually the specific sample shown here is already handled by the base `TkbmMWCustomSQLMetaData`.

Further you can setup the default property values for the resolving properties of the SQL meta data component for example in the constructor.

This concludes the whitepaper about the metadata components.

Kim Madsen
Components4Developers