Using kbmMW as a query server

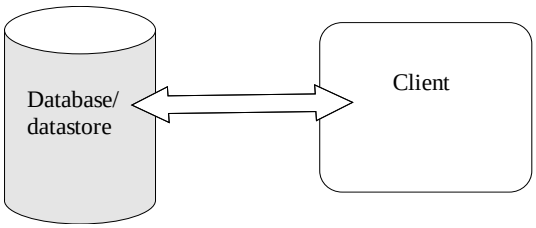
for kbmMW v. 1.00+

A little about n-tier

kbmMW contains quite advanced features for accepting queries from clients, sending resultsets back to the client and allow changes to the data in the resultset to be automatically updated back to a physical database or datastore.

As with everything else in kbmMW, this consists of two ends, the part on the server (the service) and some components for the client.

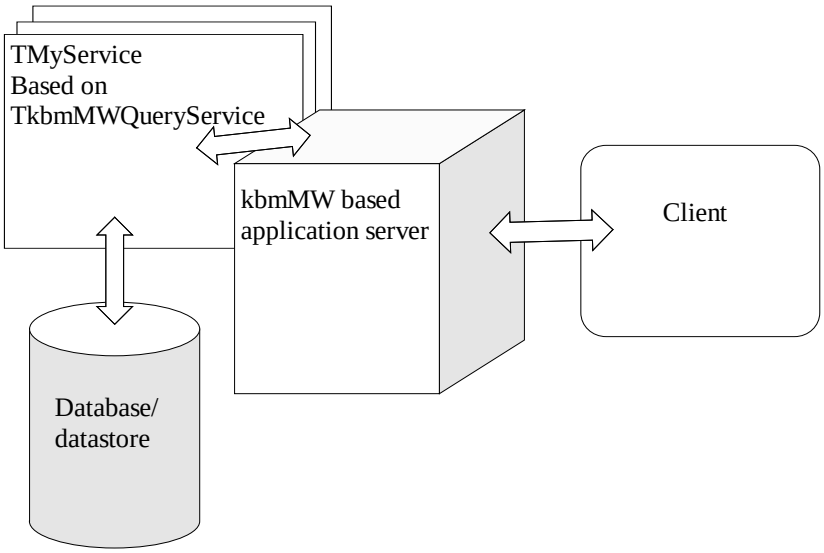
A client/server application usually is build like this:



The client access the database/datastore directly. Thus all calculations and rules are placed in the client or in some cases partly in the database via stored procedures if the database supports that. The communication between the client and the database is very different from database to database. Many different and not compatible methods are existing: ODBC, SQLLinks, JDBC, OLEDB, ADO, Native API's and more.

1. This scheme has several problems:
2. The client needs to be build to match the database it acccess.
3. The communication between the client and server is very different from method to method, and often involves serious reconfiguration of a firewall if any is placed between the client and server.
4. First.. and last defence against unauthorised access to the database is only configurable on the db itself.
5. If any other types of clients are needed (f.ex a web or WAP interface), all the business and validation rules that are existing in one client needs to be dupplicated to all other clients with all the troubles that give (chance of bugs, maintainance etc).
6. If the connection breaks between the client and the server, the programmer needs to handle that in the client. There are no automatic failover.
7. If the database server gets too busy, there is not much else to do than to try to replace it with some bigger (and usually more expensive) hardware. There are no scalability.

That is what a n-tier solution is trying to remedy.
A kbmMW 3-tier solution would look similar to this:



Now the client access an application usually running on another server (although its not required. It can run on the same physical machine as the client application) which in turn keeps contact with the database/datastore server via some services defined within the application server.

The communication between the client and the application server is no longer determined by the backend database. Instead the communication used can be selected to match whatever needs there are. F.ex. would it solve the firewall reconfiguration problem by choosing a communication method (in kbmMW called a transport) which emulates HTTP (the protocol used by web browsers and servers). Then the communication will pass right through the firewall without problems.
Further the client do not see the database/datastore directly. Its 'virtualized' for the client.
This means that the client dont have to know what database is actually behind the application server. The database can be exchanged for another one, or several different databases can be used at the same time. The client will think of the combined application server and backend databases as being one big database.

The application server

The kbmMW application server's job is to accept requests from the outer world, find and allocate a service which can handle the request, and send the result back to the originator.

A service is essentially a special TDatamodule (TkbmMWCustomService) which have a well defined interface to the application server. To create a special service, f.ex. adding two numbers and returning the result, one would create a new service (f.ex. TMyAddService) based on the TkbmMWCustomService and put some code in it that does the addition of the numbers given and returns the result to the application server. For more about creating specialized services see the document 'Creation of customized services'.

In addition to the basic specialized service, kbmMW also includes a high level query service which contains all the logic for accessing backend databases upon client requests.

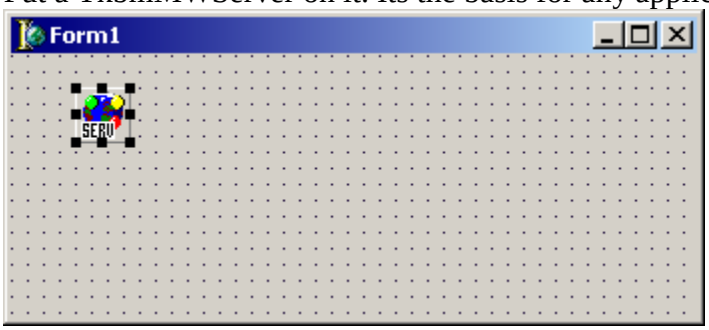
Using this high level query service (TkbmMWQueryService) one can easily publish data from a database/datastore which the client can request. Further the query service also support features for automatically updating client changes to the dataset back to the backend database.

Creating an application server

The first step is to create an application server. A standalone application server is created as a normal application. Thus choose File->New->Application.

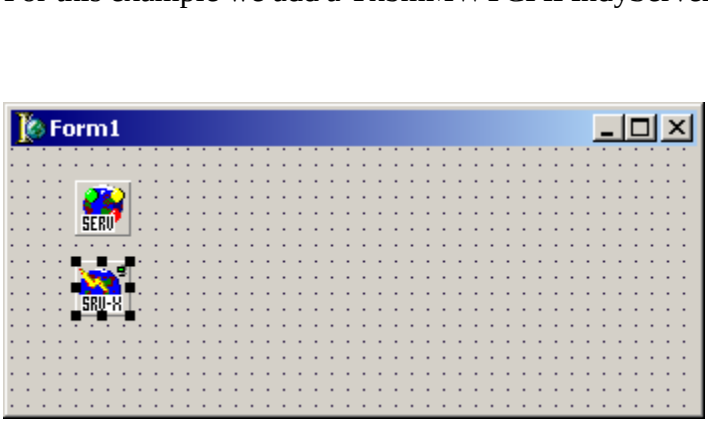
This gives you a new empty TForm.

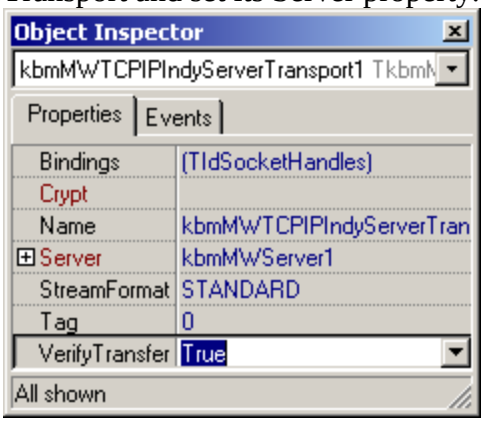
Put a TkbmMWServer on it. Its the basis for any application server.



The application server needs to know which means of communication it should handle. For this we use server transports. The client will use a matching client transport.

For this example we add a TkbmMWTCPIPIndyServerTransport and set its Server property:

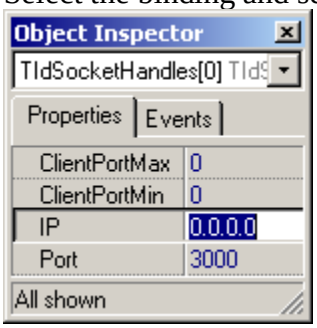




The Indy transport allow you to control which client IP addresses are allowed to access the server and on which ports. In this case we allow everybody on port 3000. The client needs to connect to the server on port 3000 too.

We select the Bindings property (click the ... there) and click its add icon to add a binding.

Select the binding and setup its properties:



0.0.0.0 means everybody is allowed to contact the server on port 3000.

Several other properties are available on the transport:

Crypt Can be used to add encryption to the transmissions.

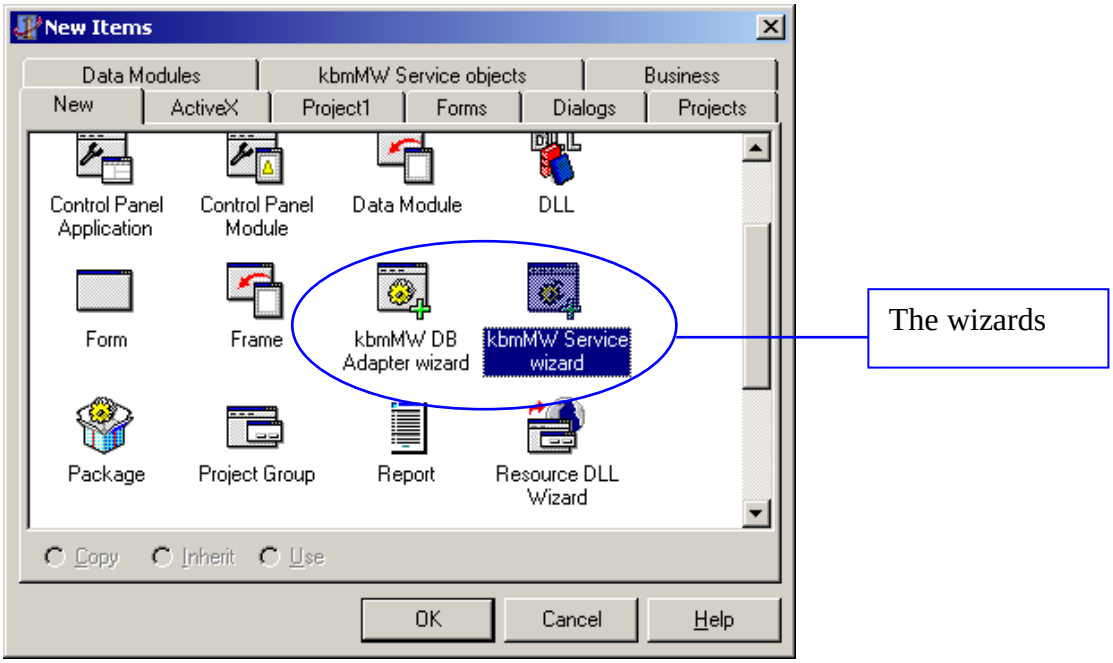
StreamFormat Select how the data is presented during the transmission.
STANDARD is a propritary binary protocol which is fast and effecient.
HTTP is STANDARD with HTTP compliant headers wrapped around it.
ZIPPED is STANDARD compressed with the ZLib compression library.
ZIPPED_HTTP is ZIPPED with HTTP compliant headers wrapped around it.
Using HTTP streamformat allows for the data to go through firewalls on ports usually used by web servers (port 80 etc.) The client and the server must use the same StreamFormat settings.

VerifyTransfer This is a setting which must be set the same on client and server. The purpose of it is to control if the server and client should make early verification of the data send. This is recommended to set to true to make the server immune to clients sending trash to it.

Creating a query service

The easiest way to create a new query service is to use the kbmMW Service Wizard.

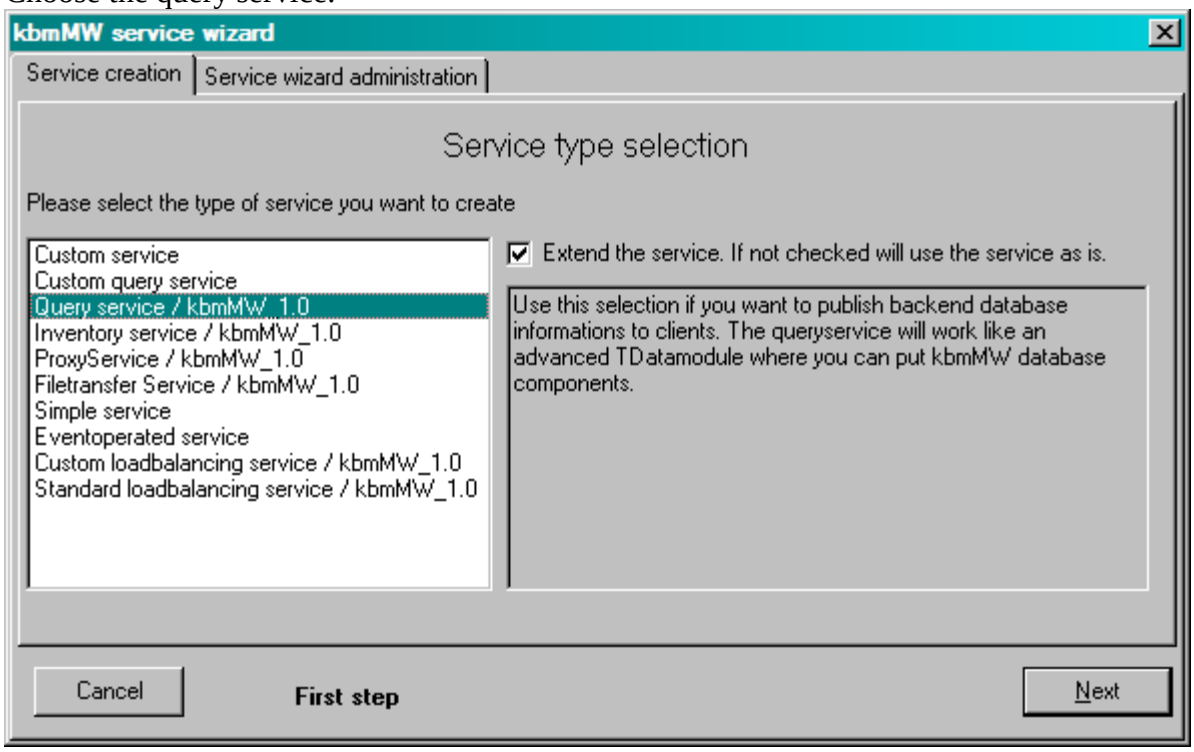
Find it in File->New (->Others for Delphi 6 or newer):



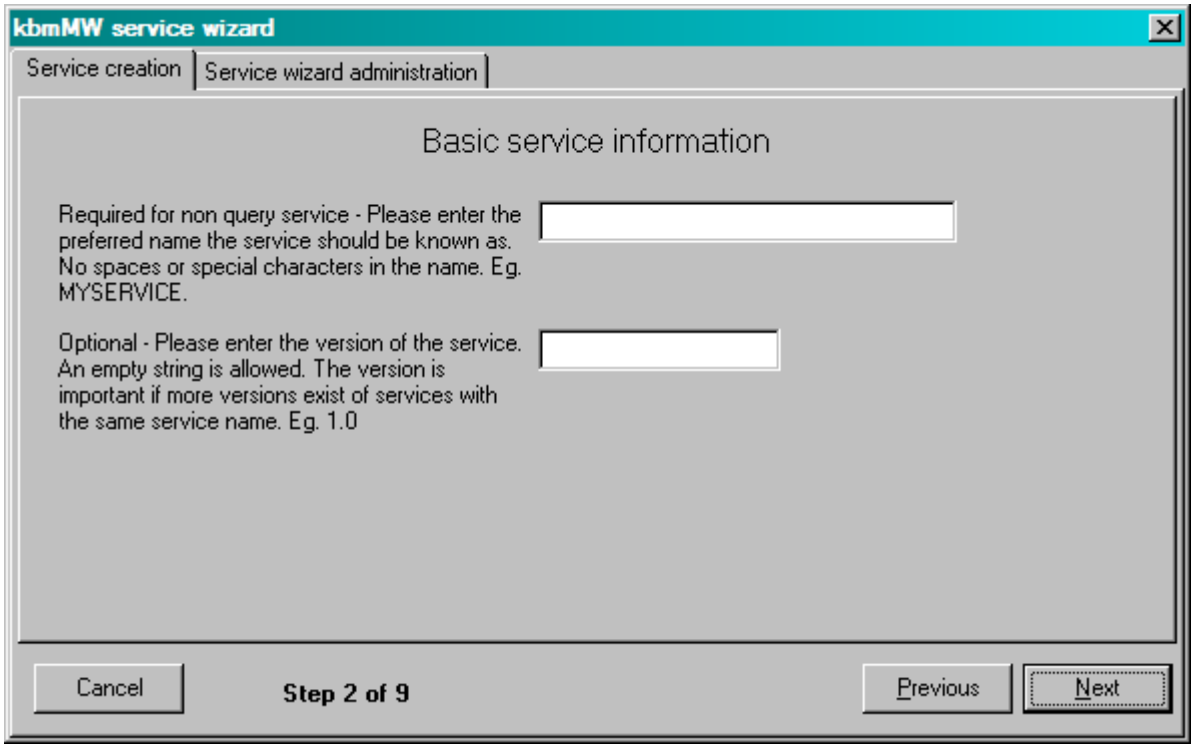
Select the kbmMW Service wizard and click OK.

This will start the wizard with this initial dialog where you are asked to choose the type of service you want to create. You have two choises... a query service (used for updateable remote datasets) or a custom service.

Choose the query service.

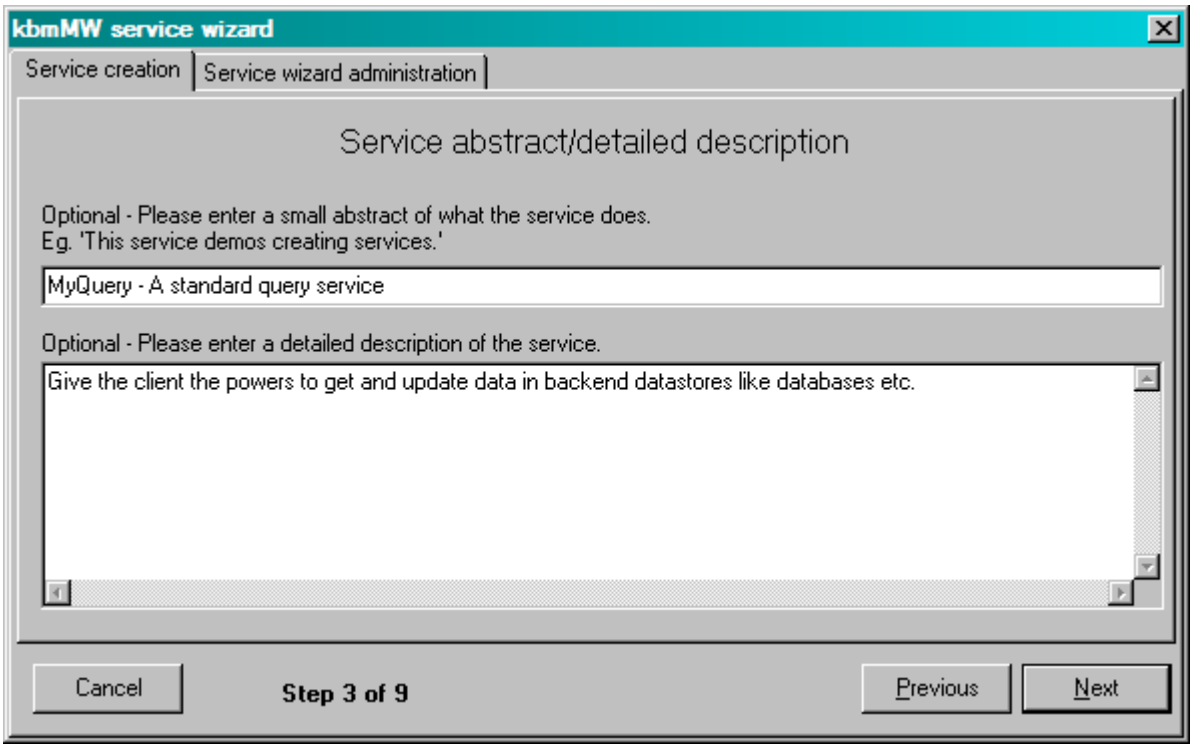


Click Next.

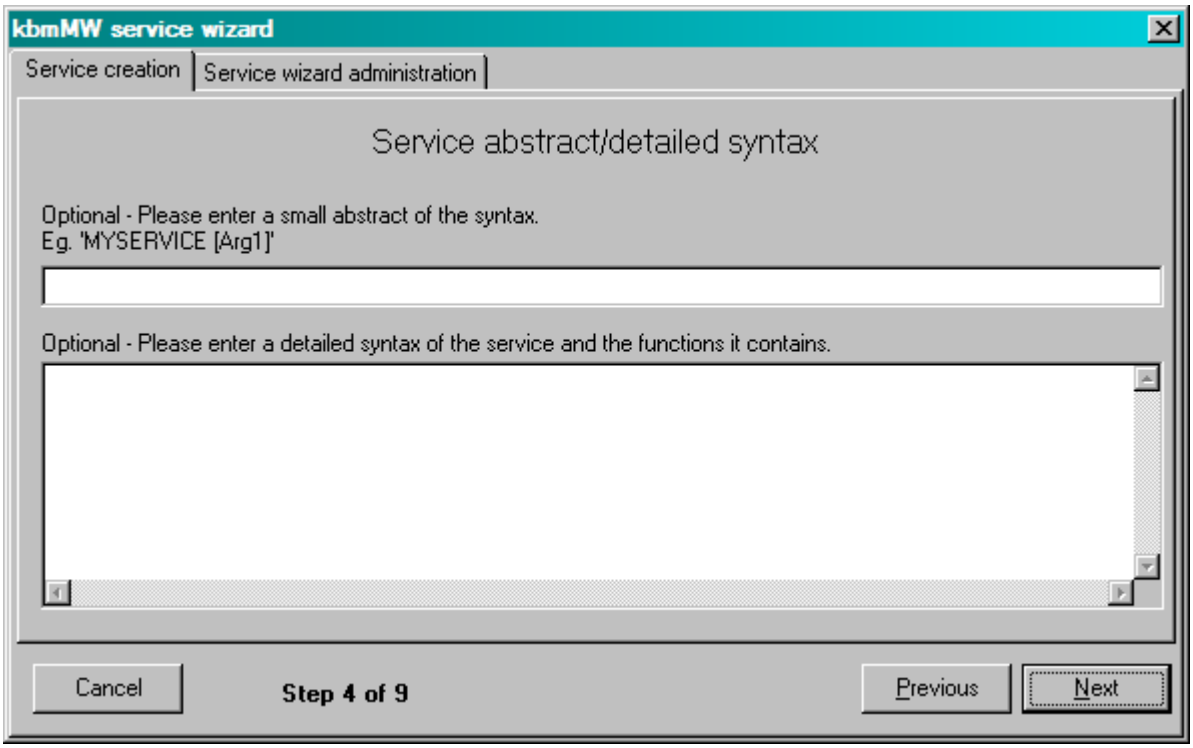


As with customized services the service can get a specific name. If this is the only service in your application server, you dont have to give it a preferred name. In that case it will automatically be named KBMMW_QUERY.

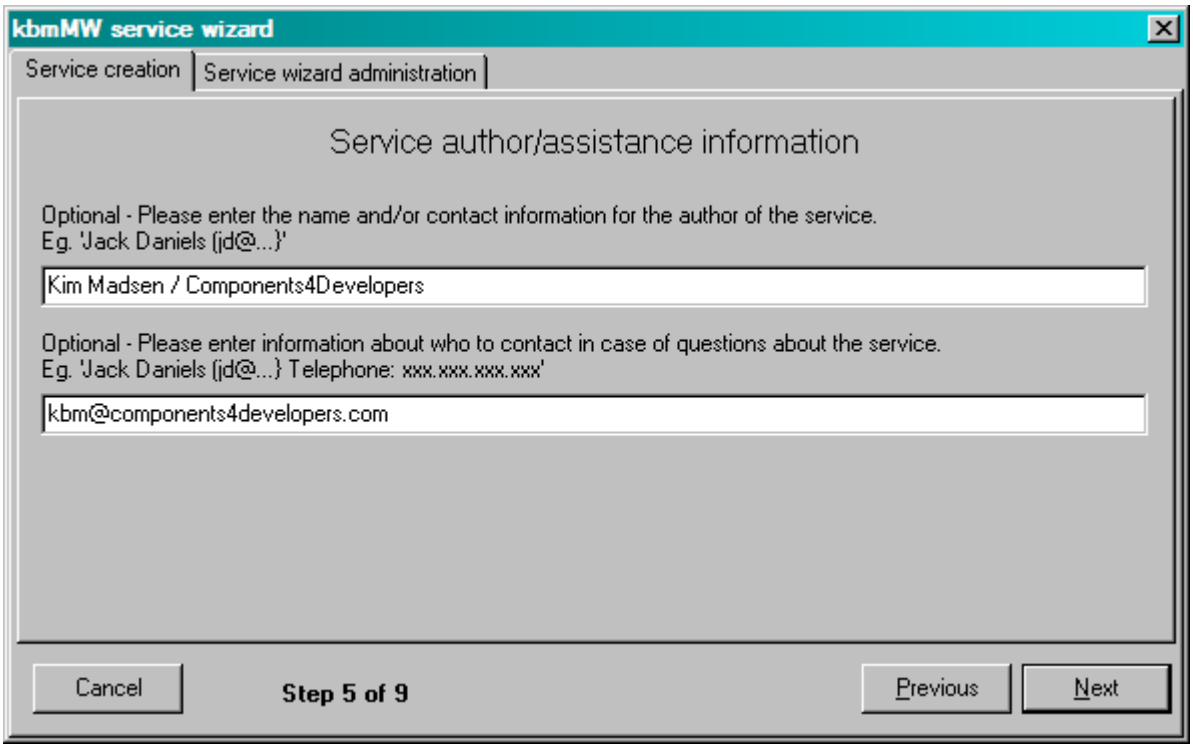
Click Next.



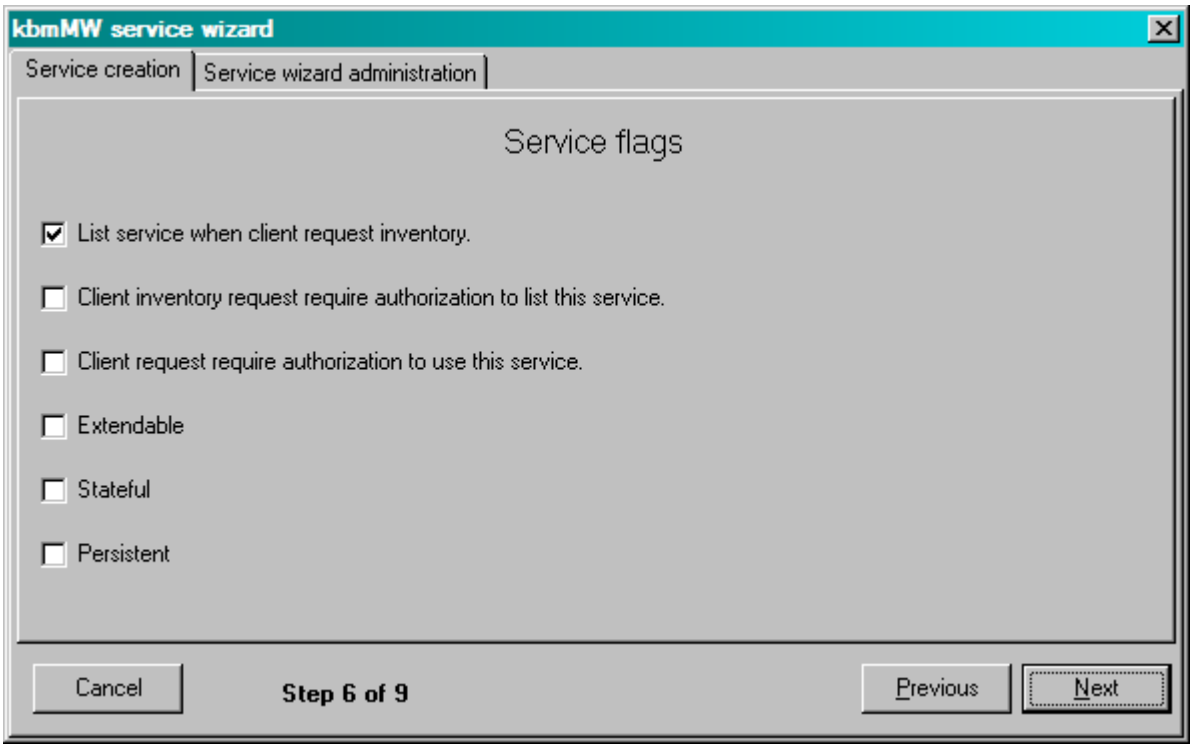
As with customized services, you can add some inventory information to the service.
Click Next.



Click Next.



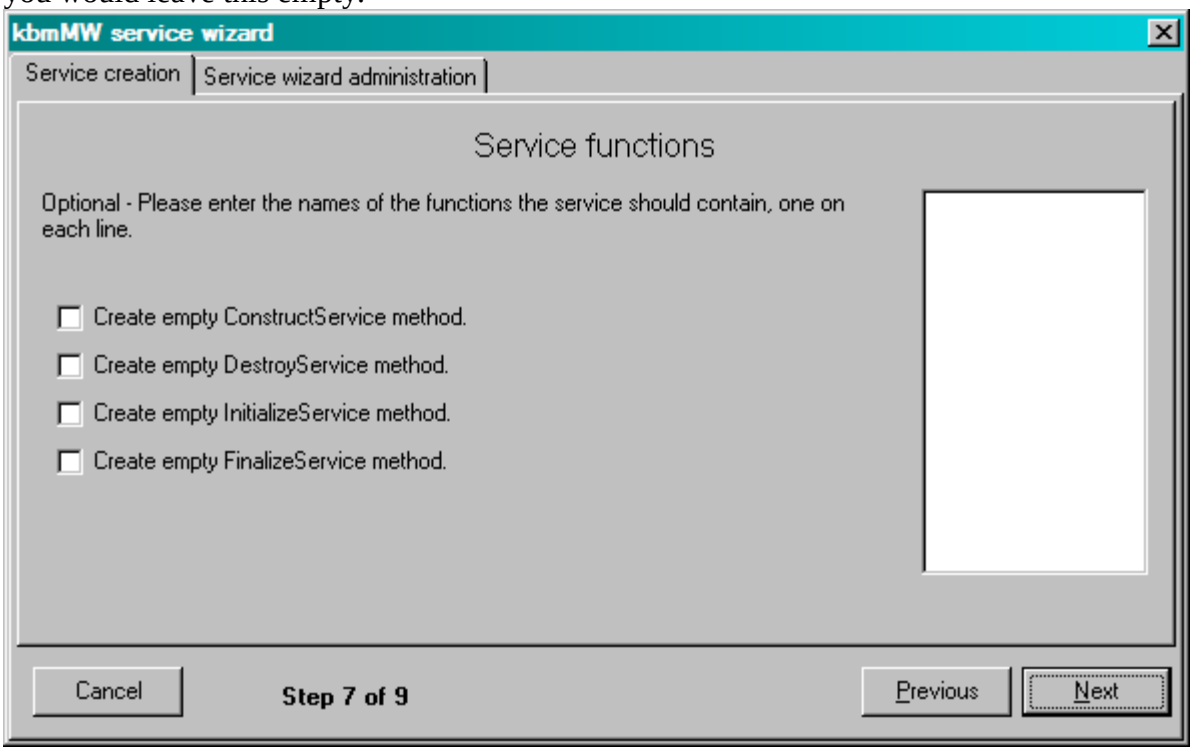
More inventory information can be added.
Click Next.



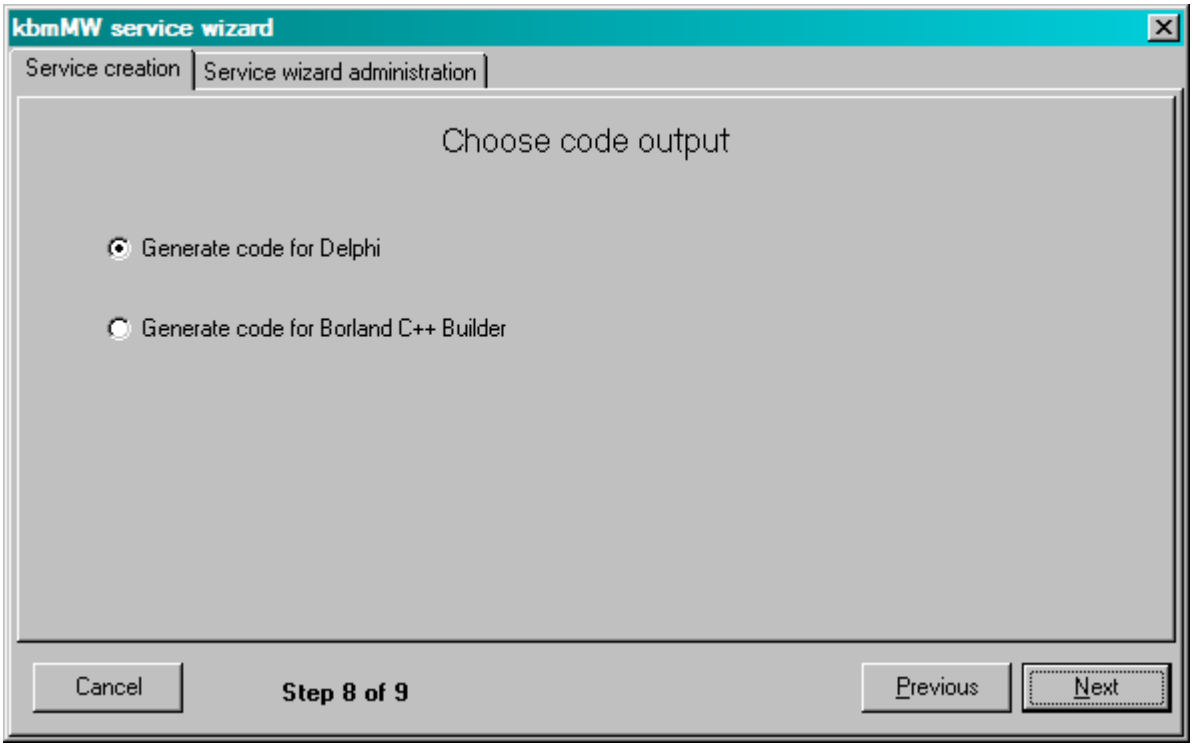
Some service flags can be set. For more information on these inventory things, please consult the 'Creating customized services' document.

Click Next.

Like with custom services you also get the posibility to add additional functionality to it. Generally you would leave this empty.

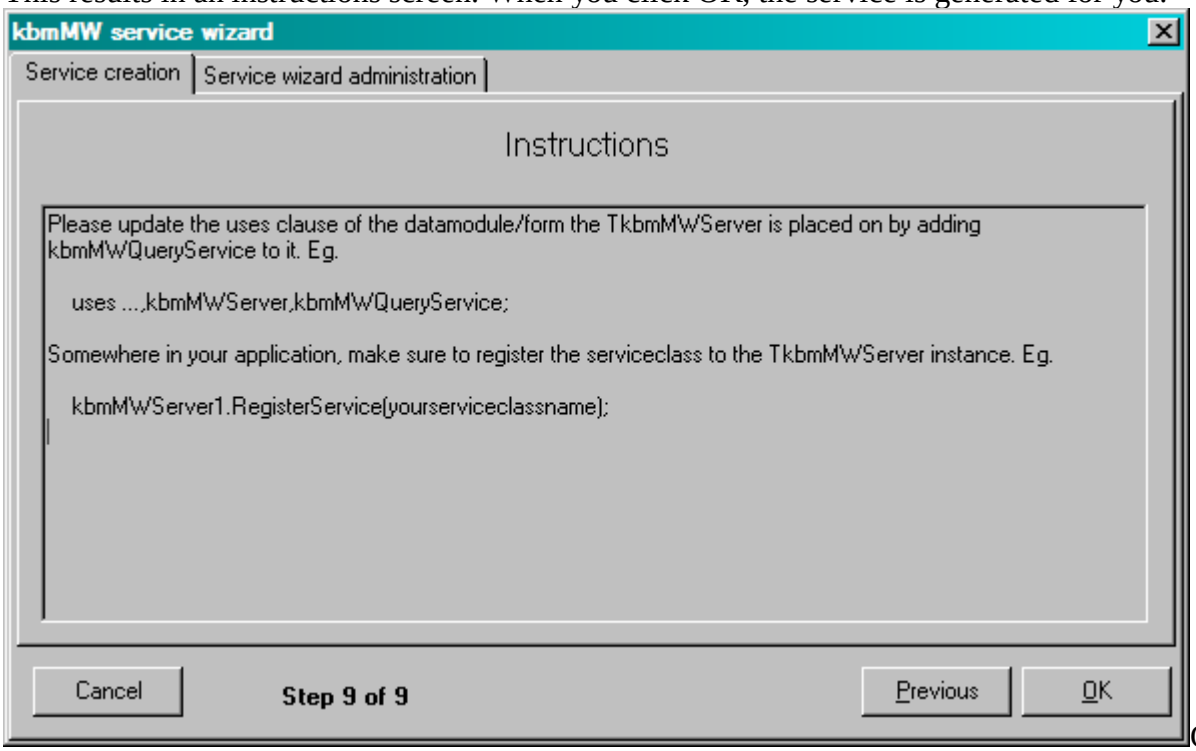


Click Next.



Choose the code generation. For more information on the inventory setups and code generation, please consult the 'Creating customized services' document.

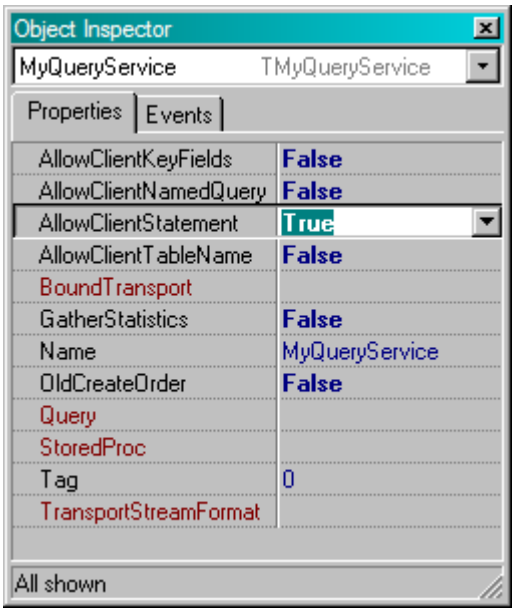
This results in an instructions screen. When you click OK, the service is generated for you.



Click

Click OK. This results in an empty query service datamodule.

Lets rename it to MyQueryService.



The basis for this service is essentially a TDataModule. Therefore all components which can be placed on a TDatamodule can also be placed on your new service object but if you want to benefit from kbmMW's connection pooling and data caching, you would use the kbmMW database adapter components.

The query module have some special properties which can be set:

AllowClient.... These controls what the client is allowed to send to the server as being part of a request. If f.ex. AllowClientStatement is false, the client is not allowed to specify the query statement (f.ex. SQL) that should be used to find the data requested.

GatherStatistics This exists for all services. It controls if access statistics should be gathered.

Query This should be set to the datamodule's default query. Thats the one that will be used if the client do not specify anything in its Query property.

StoredProc This should be set to the datamodule's default stored procedure. Thats the one that will be used if the client do not specify any name in the TkbmMWClientStoredProc.StoredProcName property.

TransportStreamFormat This is the format that should be used when transporting resultsets to and from the client.

Since this document will support a client which will define its own SQL statements (on the client), eg. a client side query, we need to allow client side statements by setting the AllowClientStatement to true.

For many reasons, a TQuery, TTable or TStoredProc is not the way to go. They simply do not contain all the nice things that make them universaly useable.

Instead kbmMW contains some special database components: TkbmMWxxQuery, TkbmMWxxStoredProc, TkbmMWxxConnectionPool and TkbmMWxxResolver.

The xx depends on the backend database type.

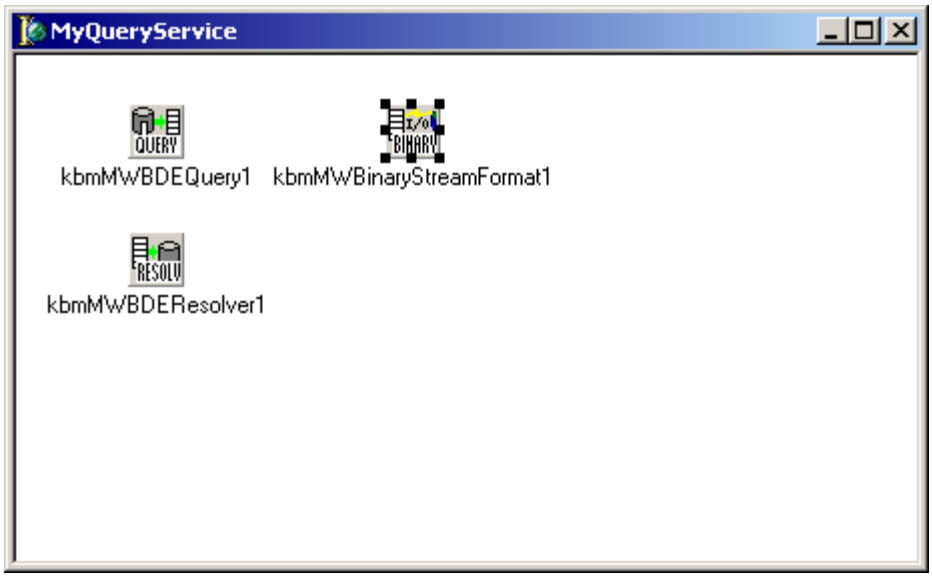
xx=ADOX	Borland ADO Express support.
xx=DBISAM3	Elevatesoft's DBISAM v. 3 support.
xx=FF2	TurboPower Flash Filer 2 support.
xx=IBX5	Borland Interbase Express 5 support.
xx=IBO	IBObjects Interbase support.
xx=ADS6	ExtendedSys's Advantage Database Server 6 support.
xx=BDE	Borland Database Engine support
xx=DADO	Deer-Soft ADO support
xx=MYDAC	Micro-OLAP MySQL support
xx=ZDB2	Zeos DB2 support
xx=ZIB	Zeos Interbase support
xx=ZMS	Zeos MSSQL support
xx=ZMY	Zeos MySQL support
xx=ZORA	Zeos Oracle support
xx=ZPG	Zeos Postgres support
xx=ZSY	Zeos Sybase support
xx=MT	Components4Developer's kbmMemTable support
xx=DBX	Borland dbExpress support
xx=DOA	Direct Oracle Access
xx=DAO	Diamond Access

These 4 components along with a class TkbmMWxxConnection is combined called a database adapter. A database adapter contains as minimum a descendent of TkbmMWCustomConnection, TkbmMWCustomConnectionPool and some data component like a descendent of TkbmMWCustomQuery.

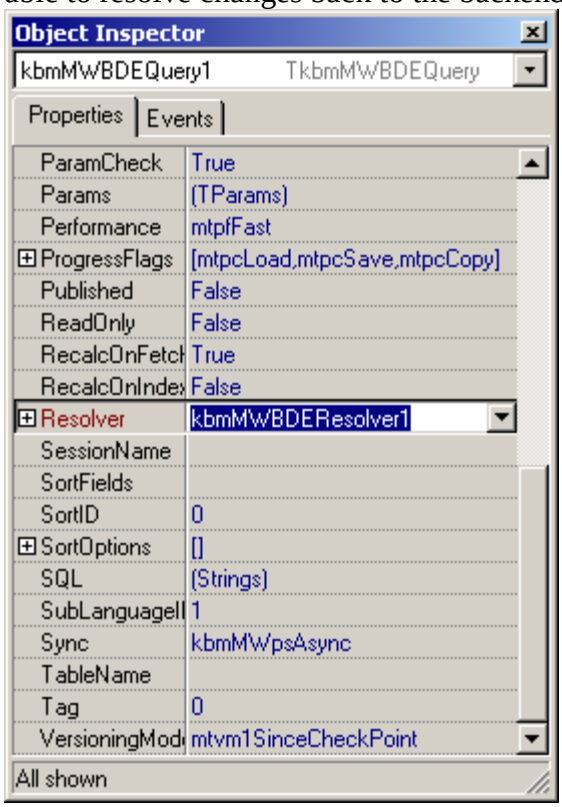
By creating a database adapter its possible to make kbmMW work with other special databases/file systems or other types of datastores in the same way regardless of choice.

This way the client will never need to know what database is behind the application server.

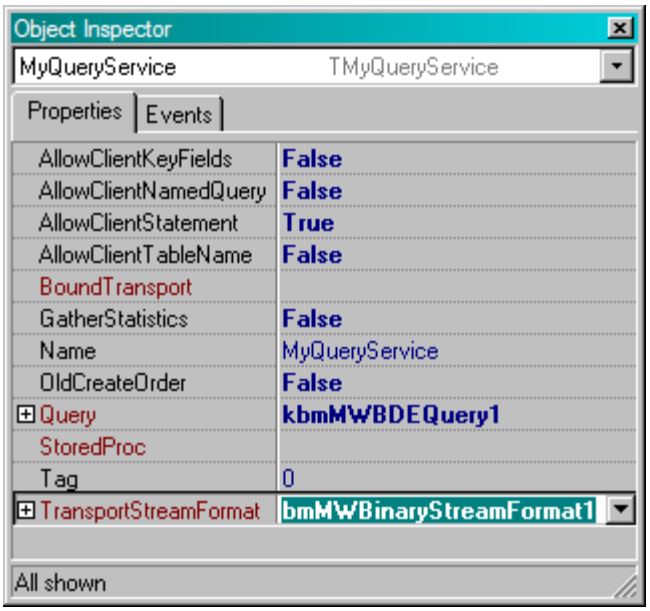
Lets say you want to access a table in a Paradox database using the BDE. Then you would add a TkbmMWBDEQuery, a TkbmMWBDEResolver and a TkbmMWBinaryStreamFormat on the MyQueryService module.



Make sure to tell the BDEQuery to use the BDEResolver (if you do not, your clients will not be able to resolve changes back to the backend database):



Also setup the query datamodule's properties TransportStreamFormat and Query:



Further you will need to add a TkbmMWBDEConnectionPool. This can be added two different places, either directly in the same query service module, or on a common place for all services, like the main TForm (for standalone application servers) or main TDatamodule (for non form based application servers) which is autocreated (recommended).

The job of a TkbmMWxxConnectionPool is to create and maintain connections to the backend databases and destroy them when they are not needed. A connection pool contains a pool of connections. Thus if a client makes a request, a connection from the pool will be selected. If all currently allocated connections are in use, it will automatically try to allocate a new one if there is room for it. The number of connections available in the pool is defined using the property editor on the connection pool component.

Further a connectionpool also contains a caching mechanism. This makes it possible to avoid calling the backend database on each client request, if f.ex. another client have just made the same request minutes or seconds before. This together with the connection pooling allow your database to be able to serve many more clients than it would otherwise be able to.

The properties are:
CachePerformance Can be mtpfFast, mtpfBalanced or mtpfSmall. This determines how recordsets are packed when they are stored in the cache.

ConnectionInactivityTimeout Specifies the number of seconds an actual database connection is allowed to be idle before its automatically closed. If set to 0, no connections are timed out.

Database Is dependent on the backend database. Its usually set to a TDatabase like component on the shared form. The database component chosen is used as a template for creating new connections.

EnableCache Controls if any recordsets should be cached or not.

CachePerformance Determines how much compression is made on records before they are stored in the cache. mtpfFast = no compression (each record takes up _max_ recordsize), mtpfBalanced = variable length fields (strings, widestrings, varchars) only takes up their actual length, not max.

GarbageCollection Controls if control of idle connections and stale cache entries should happen. If false, not connections that are timed out will be disconnected and cache entries that are out of date will not be removed.

GarbageInterval The number of seconds between each garbagecollection sweep.

MaxCacheAge This determines how old a cache entry is allowed to be (in secs) before it will be automatically removed. The idea is that some tables changes contents from time to time and to have that change propogate itself to the clients, the cached entries should be removed for a fresh selection to be fetched from the database on next client request.

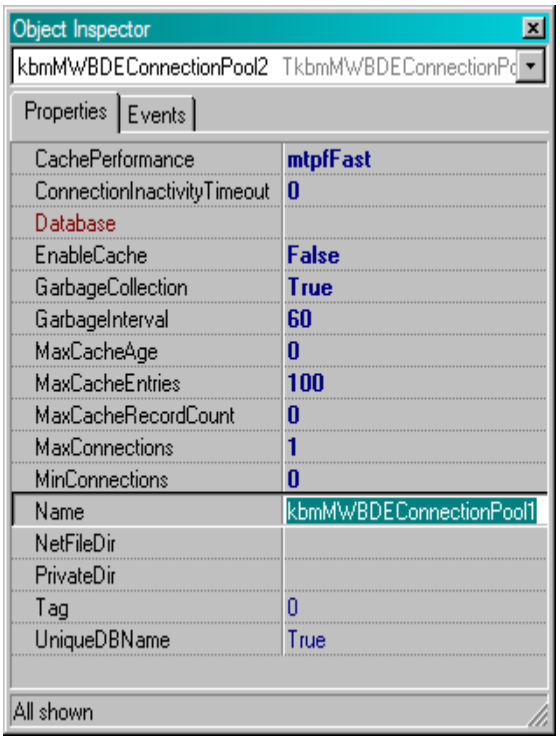
MaxCacheEntries This determines how many recordsets are allowed in the cache. Its always the x most used cache entries that are allowed to stay in the cache.

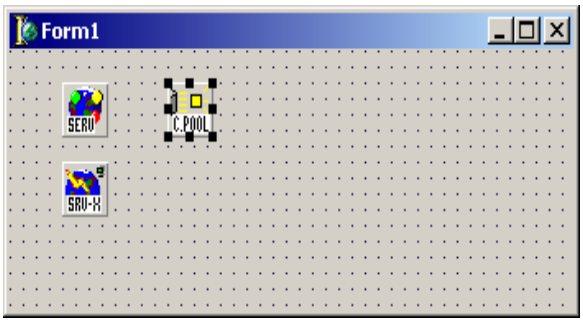
MaxCacheRecordCount This determines how big a resultset is allowed to be before it will not be cached.

MaxConnections Determines how many concurrent database connections are allowed in the pool.

UniqueDBName Determines if the internal database name should be made unique for each and every connection to the backend database.

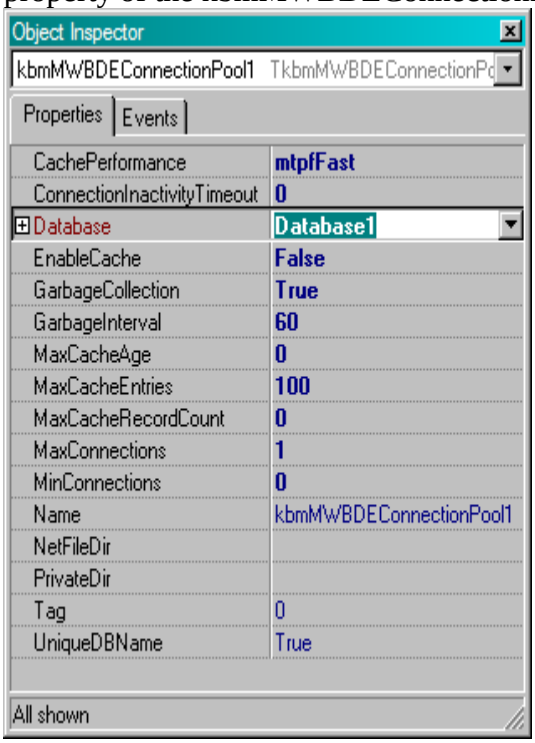
If the connectionpool is shared between many services, the number of connections toward the database can be limited to the max of that connectionpool. If a connectionpool is placed on each service, the number of connections towards the database is now only limited by the concurrent number of clients requesting something from the query service. Usually it is best to share one connectionpool between all the query services.

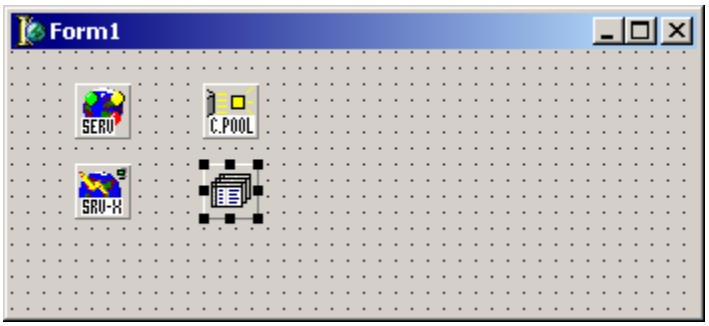




The connection pool needs to know which database to work against.

This is set by adding a TDatabase (for the BDE sample only... other adapters have other requirements) and set its properties so its able to connect to the database. Then set the Database property of the kbmMWBDEConnectionPool to the TDatabase.





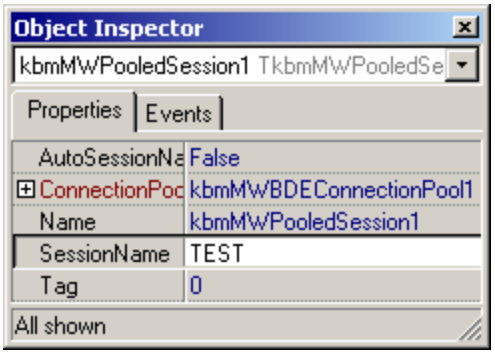
The added TDatabase component is actually not used directly by kbmMW, but rather its serving as a template for creating connections to the database.

To glue the service's query component to the connection pool, we now add a TkbmMWPooledSession.

The purpose of a TkbmMWPooledSession is to synchronize otherwise async requests in a threaded application along with being the linking component (similar to a TDatasource). For this document, we do not use its synchronization techniques.

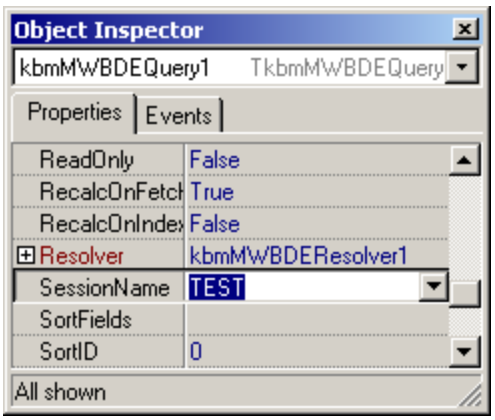
There can be many pooled session components, and they can be placed centrally or on each service module. For this setup, we put it to the main form.

We need to link its ConnectionPool property to the BDE connection pool and give it a unique SessionName:



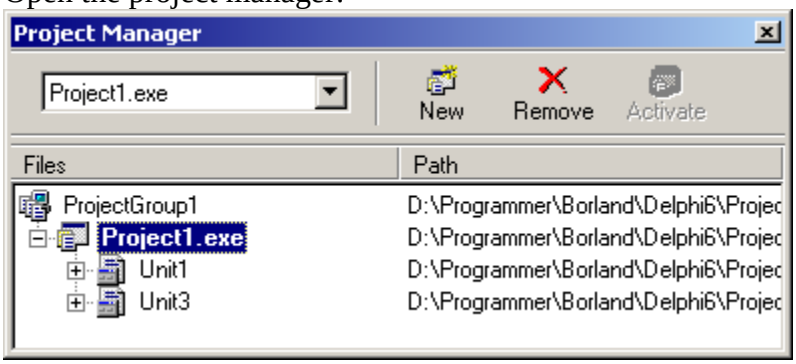
The SessionName is what links query components with the pooled session and thus with the connection pool.

Now lets set the SessionName on the query component on the service module to TEST:

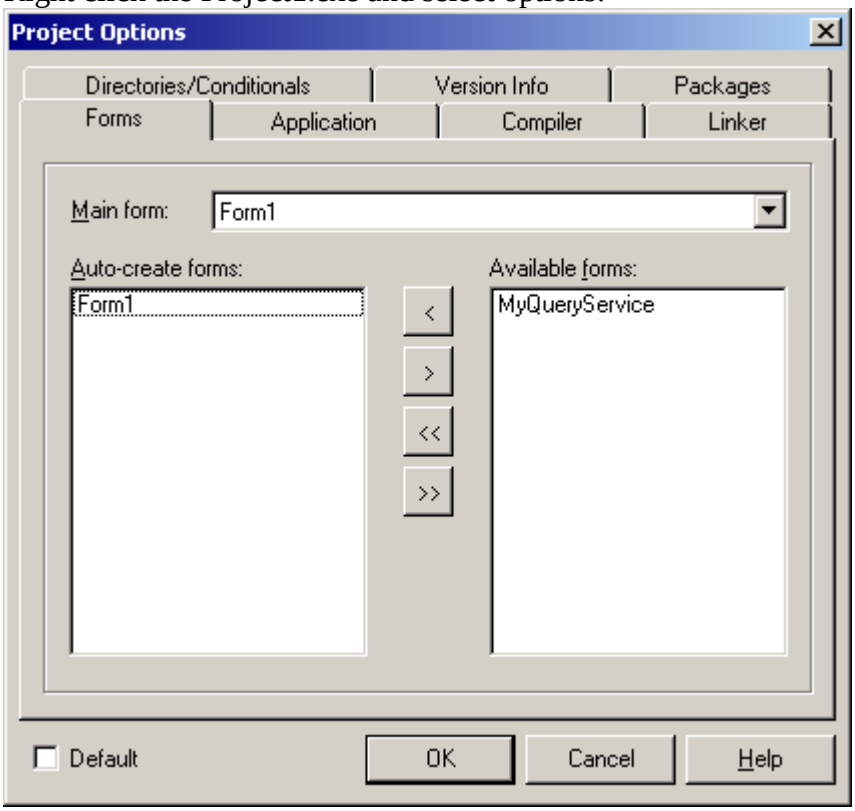


Now the database part of the application server is ready to use.
We just need to finish the application server itself.

Open the project manager:

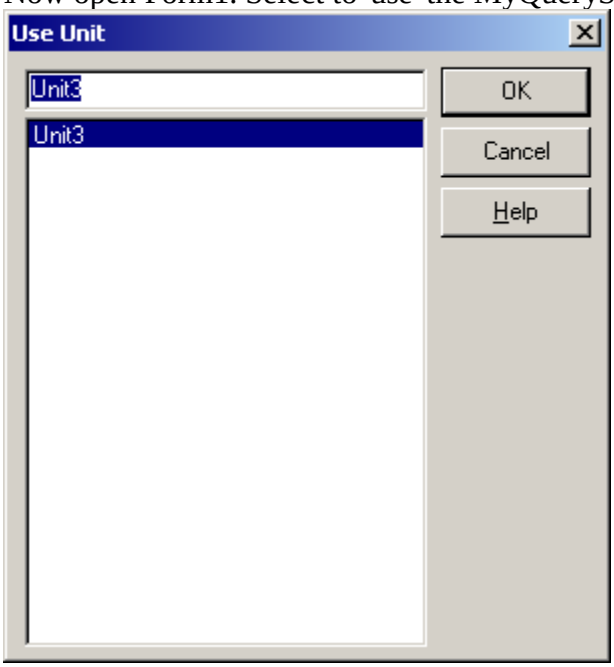


Right click the Project1.exe and select options:



Make sure that only Form1 is autocreated.

Now open Form1. Select to 'use' the MyQueryService datamodule in File->Use.



Then register MyQueryService (in the unit3.pas - name may vary) to the TkbmMWServer in the Form1.OnCreate event:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
     kbmMWServer1.RegisterService(TMyQueryService,false);
     // You might want to set up some of the many service definition options here.
     // Please check the whitepaper about creating customized services for more info.

     // Activate listening. If its not listening, no clients can connect.
     kbmMWServer1.Active:=true;
end;
```

Now there is a link between the new service and the application server and the server is listening for clients.

Save the project and compile.

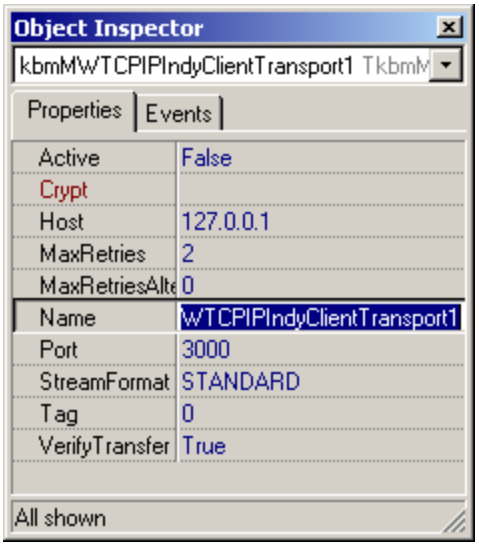
Creating a query client

Client applications follow the same scheme as the database adapter components on the application server. For the client we have: TkbmMWClientConnectionPool, TkbmMWClientQuery, TkbmMWClientStoredProc.

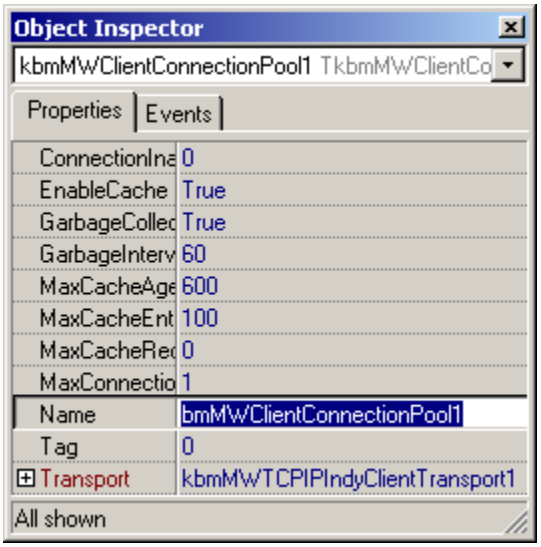
Create a new application: File->New->Application.

Add a TkbmMWTCPIPIndyClientTransport, and set its Host to the IP address or name of the maching the application server runs on (127.0.0.1 if its the same as the client application). Set the Port to 3000.

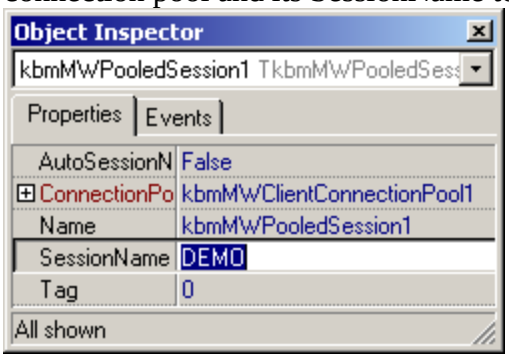
Make sure to set the StreamFormat and VerifyTransfer to the same as the server settings.



Add a TkbmMWClientConnectionPool and set its Transport property to point on the client transport component:



Add a TkbmMWPooledSession component, and set its ConnectionPool property to the client connection pool and its SessionName to something unique for the client eg. DEMO.



Add a TkbmMWBinaryStreamFormat which is used for streaming/unstreaming the dataset data.

Now the basis is ready.

Then add a TkbmMWClientQuery which is the client component that is going to communicate with MyQueryService on the application server.

Set its SessionName to 'DEMO' (the same as the pooled session component).
Set its TransportStreamFormat to point to the TkbmMWBinaryStreamFormat.

Now set its Query property to contain some statement (SQL or whatever the backend database supports). For example: select * from biolife

Add a TDatasource and point its Dataset property to the TkbmMWClientQuery.
Add a TDBGrid and point its DataSource property to the TDatasource.

Activate the TkbmMWClientQuery by setting Active to true.

This will result in the client trying to contact the server (which must be running and listening) and execute the query statement on the server after which the server returns the data to the client and you will see the contents in the DBGrid.

This is called a client side query because the SQL or whatever query language is used is specified on the client.

Its also possible to specify the statement on the server components instead. Then the client need not specify any statement at all, or only specify a named query statement.

A named query statement is a special syntax which is used to specify which query component on the service, that the client would like to use.

To specify a specific query component on the server set the Query property to:

@SOMENAME

where SOMENAME is the name of a query component on the service.
If the query on the service contains parameters, those definitions are send to the client which can choose to change them before activating the query.

The server can contain multiple query services, in which case they each _must_ be given a unique preferred name/version.

On the client set the QueryService property to the name of the query service in question. Default the QueryService property is set to KBMMW_QUERY which is the default name for a query service.

This concludes the query server session.

Kim Madsen
Components4Developers