Overview

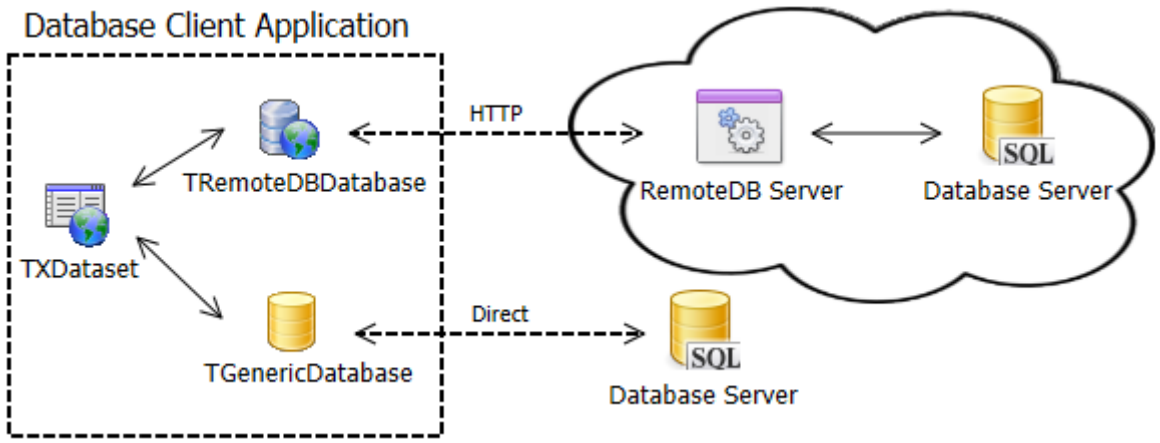
TMS RemoteDB is a set of Delphi components that allows you to create 3-tier database applications. It allows your client db application to execute SQL statements through an Http server, instead of directly accessing the database server. It is targeted for performance, stability, and as an easy path to convert client-server applications into 3-tier applications. At the server-side you can choose a wide range of database-access components to connect to database server, including FireDac, dbExpress, dbGO (ADO) and many others. TMS RemoteDB uses TMS Sparkle as the core communication library.

TMS RemoteDB product page: https://www.tmssoftware.com/site/remotedb.asp

TMS Software site: http://www.tmssoftware.com

TMS RemoteDB allows you to create database applications that perform SQL operations on a remote HTTP server, instead of a database server, using a TDataset descendant named TXDataset. This makes it easy to convert existing Delphi client-server applications into 3-tier applications with minimal changes in source code.

The following picture provides an overview of RemoteDB architecture.



The RemoteDB Server is a Delphi application/service that listen to HTTP requests. When clients perform requests, the RemoteDB Server will forward the requests to the actual database server being used, using the specified database-access component. The SQL is performed and the results are returned back to the client. You can choose the components that will perform the SQL statements from a wide-range of options such as FireDac, dbExpress, dbGo (ADO), among others, using component adapters. The SQL database can be any database server supported by the components used, and must be thread-safe (most are).

From the Delphi client application, you can use as many TXDataset components as you need to perform the regular SQL operations. All TXDataset components are linked to a TRemoteDBDatabase component via a Database property. The RemoteDBDatabase component is the one in charge of forwarding the SQL requests to the RemoteDB Server and retrieving back the values. All operations on the client are transparent and just like any TDataset usage: you specify the SQL statement, Open the dataset (or ExecSQL), define the list TField components, Params, connect TDatasource components to it, etc..

TMS RemoteDB also allows you to use the TXDataset to connect directly to the database server in a traditional client-server approach. For that, you could just use TGenericDatabase instead of TRemoteDBDatabase. TXDataset component can be connected to any of the two. This allows you to have a single application, a single dataset component, but switch the database access between a remote, cloud HTTP access to the database, or a regular, local network direct access to the database.

In this section:

Creating RemoteDB Server

How to create a new TMS RemoteDB server and how to configure it.

RemoteDB Client Applications

How to create Delphi client applications that connect to a TMS RemoteDB server.

Creating RemoteDB Server

The following topics explain how to create a new TMS RemoteDB server and how to configure it.

Ways to Create the RemoteDB Server

You have four different ways to create a RemoteDB Server app, as follows.

RemoteDB Server Wizard

The easiest and more straightforward way to get started with RemoteDB is using the wizard.

1. From Delphi IDE, choose **File > New > Other** and then look for the "**TMS RemoteDB**" category under "Delphi Projects".
2. There you find the following wizard to create a new XData Server Application:

   TMS RemoteDB VCL Server: Creates a VCL application that runs a RemoteDB server using http.sys
3. Choose the wizard you want, double-click and the application will be created.

The wizard will create the design-time components for you. You still need to drop the database-access component to be used to connect to the database - e.g., TFDConnection (FireDac), TUniConnection (UniDac), TSQLConnection (dbExpress), etc. - and then associated it to the TAureliusConnection component.

You can also create the server manually, using design-time components or from non-visual code.

Using Design-Time Components

Another way to create a TRemoteDBServer is by using the design-time components. If you want the RAD, component dropping approach, this is the way to go.

1. **Drop a dispatcher component on the form** (for example, *TSparkeHttpSysDispatcher*);
2. **Drop a TRemoteDBServer component on the form**;
3. **Associate the TRemoteDBServer component with the dispatcher through the Dispatcher property**;
4. **Specify the BaseUrl property of the server** (for example, *http://+:2001/tms/remotedb*);
5. **Set the Active property of the dispatcher component to true**;
6. **Drop a TAureliusConnection on the form and configure it so that it connects to your database** (you will need to drop additional database-access components, e.g. TFDConnection if you want to use FireDac, and then associate it to the TAureliusConnection.AdaptedConnection).
7. **Associate the TRemoteDBServer component to the Aurelius connection through the Connection property**.

That is enough to have your RemoteDB server up and running!

Legacy Wizard for RemoteDB Server

There is a legacy wizard which don't use design-time components but you can still use.

To create a new RemoteDB Server using the legacy wizard:

1. Choose **File > New > Other** and then look for "**TMS Business**" category under "Delphi Projects". Then double click "**TMS RemoteDB Server**".
2. Chose the kind of applications you want to server to run on, then click *Next*. Available options are **VCL**, **FMX**, **Service** and **Console**. You can choose multiple ones (for example, you can have a VCL project for quick test the server and a Service one to later install it in production, both sharing common code.
3. Chose the **Host Name**, **Port** and **Path** for the server, then click *Next*. Usually you don't need to change the host name, for more info check URL namespace and reservation. Port and Path form the rest of base URL where your server will be reached.
4. Select the **Driver** (component to access database) and the **SQL Dialect** (type of database server), then click *Create*.

The new server will be created and ready to run.

Creating the Server Manually

If you don't want to use the RemoteDB Server wizard and do not want to use design-time components, you can create a server manually, from code. This topic describes how to do it, and it's also a reference for you to understand the code used "behind-the-scenes" by the design-time components.

TMS RemoteDB is based on TMS Sparkle framework. The actual RemoteDB Server is a Sparkle server module that you add to the Sparkle Http Server.

Please refer to the following topics to learn more about TMS Sparkle servers:

- Overview of TMS Sparkle Http Server
- Creating an Http Server to listen for requests
- TMS Sparkle Server Modules

To create the RemoteDB Server, just create and add a RemoteDB Server Module (TRemoteDBModule object, declared in unit RemoteDB.Server.Module) to the Sparkle Http Server. To create the RemoteDB Module, you just need to pass the base URL address of the server, and an IDBConnectionFactory interface so that the server can create connections to the actual SQL database server. Here is an example (`try..finally` blocks removed to improve readability):

```
uses
  {...},

  Sparkle.HttpSys.Server, RemoteDB.Drivers.Base,
  RemoteDB.Drivers.Interfaces, RemoteDB.Server.Module;

function CreateNewIDBConnection: IDBConnection;
var
  SQLConn: TSQLConnection;
begin
  // Create the IDBConnection interface here
  // Be sure to also create a new instance of the database-access component here
  // Two different IDBConnection interfaces should not share the same database-
access component

  // Example using dbExpress
  SQLConn := TSQLConnection.Create(nil);


  { Define SQLConn connection settings here, the server
     to be connected, user name, password, database, etc. }
  Result := TDBExpressConnectionAdapter.Create(SQLConn, true);
end;

var
  Module: TRemoteDBModule;
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;
  Module := TRemoteDBModule.Create('http://localhost:2001/tms/business/remotedb',
    TDBConnectionFactory.Create(
      function: IDBConnection
      begin
        Result := CreateNewIDBConnection;
      end
    ));
  Server.AddModule(Module);
  Server.Start;
  ReadLn;
  Server.Stop;
  Server.Free;
end;
```

The code above will create a new RemoteDB server which base address is *http://localhost:2001/tms/business/remotedb*. That's the address clients should use to connect to the server. The server will use dbExpress to connect to the database, and the TSQLConnection component must be properly configured in the CreateNewIDBConnection function.

There are many other ways to create the IDBConnection interface, including using existing TDataModule. You can refer to the following topics for more info.

IDBConnectionFactory Interface

The IDBConnectionFactory interface is the main interface needed by the RemoteDB server to work properly. As client requests arrive, RemoteDB Server might need to create a new instance of a database-access component in order to connect to the database. It does that by calling IDBConnectionFactory.CreateConnection method to retrieve a newly created IDBConnection interface, which it will actually use to connect to database.

To create the factory interface, you just need to pass an anonymous method that creates and returns a new IDBConnection interface each time it's called.

```
uses
  {...}, RemoteDB.Drivers.Base;

var
  ConnectionFactory: IDBConnectionFactory;
begin
  ConnectionFactory := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        SQLConn: TSQLConnection;
      begin
        // Create the IDBConnection interface here
        // Be sure to also create a new instance of the database-access component
here
        // Two different IDBConnection interfaces should not share the same
database-access component

        // Example using dbExpress
        SQLConn := TSQLConnection.Create(nil);
        { Define SQLConn connection settings here, the server
           to be connected, user name, password, database, etc. }
        Result := TDBExpressConnectionAdapter.Create(SQLConn, true);
      end
    ));

  // Use the ConnectionFactory interface to create a RemoteDB Server
end;
```

It's possible that you already have your database-access component configured in a TDataModule and you don't want to create it from code. In this case, you can just create a new instance of the data module and return the IDBConnection associated to the component. But you must be sure to destroy the data module (not only the database-access component) to avoid memory leaks:

```
var
  ConnectionFactory: IDBConnectionFactory;
begin
  ConnectionFactory := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        MyDataModule: TMyDataModule;
      begin
        MyDataModule := TMyDataModule.Create(nil);
        // The second parameter makes sure the data module will be destroyed
        // when IDBConnection interface is released
        Result := TDBExpressConnectionAdapter.Create(MyDataModule.SQLConnection1,
MyDataModule);
      end
    ));

  // Use the ConnectionFactory interface to create a RemoteDB Server
end;
```

IDBConnection Interface

The IDBConnection interface represents a connection to a database in RemoteDB. Every connection to a database in the server is represented uses this interface to send and receive data from/to the database.

IDBConnection wraps the data access component you are using, making it transparent for the framework. Thus, regardless if you connect to the database using dbExpress, ADO, IBX, etc., you just need an IDBConnection interface.

To obtain an IDBConnection interface you use existing adapters (drivers) in RemoteDB. The adapters just take an existing data access component (TSQLConnection, TADOConnection, etc.) and give you back the IDBConnection interface you need to use. To create database connections it's important to know the available:

- Component Adapters
- SQL Dialects

In summary, **to obtain an IDBConnection interface**:

1. **Create and configure (or even use an existing one) component that makes a connection to your database**.

If you use dbExpress, for example, you need to create a TSQLConnection component, and create the adapter that wraps it:

```
function CreateDBExpressConnection: TSQLConnection;
begin
  Result := TSQLConnection.Create(nil);
  // Configure Result with proper connection settings
  // Don't forget setting LoginPrompt to false
end;
```

2. **Instantiate an adapter passing the connection component**.

```
function CreateIDBConnection: IDBConnection;
var

begin
  MyConnection := TDBExpressConnectionAdapter.Create(CreateDBExpressConnection, T
rue);
  // return the newly created IDBConnection to the caller
  Result := MyConnection;
end;
```

Note the second parameter when calling Create constructor. It indicates that when IDBConnection interface is destroyed, the wrapped TSQLConnection component is also destroyed.

For more information about how to create adapters, see Component Adapters.

Component Adapters

There is an adapter for each data-access component. For dbExpress, for example, you have TDBExpressConnectionAdapter, which is declared in unit RemoteDB.Drivers.dbExpress. All adapters are declared in unit RemoteDB.Drivers.*XXX* where *XXX* is the name of data-access technology you're using. You can create your own adapter by implementing IDBConnection interfaces, but RemoteDB already has the following adapters available:

Technology	Adapter class	Declared in unit	Adapted Component	Vendor Site
Advantage	TAdvantageConnectionAdapter	RemoteDB.Drivers.Advantage	TAdsConnection	http://www.sy
dbExpress	TDBExpressConnectionAdapter	RemoteDB.Drivers.dbExpress	TSQLConnection	Delphi Native
dbGo (ADO)	TDbGoConnectionAdapter	RemoteDB.Drivers.dbGo	TADOConnection	Delphi Native
ElevateDB	TElevateDBConnectionAdapter	RemoteDB.Drivers.ElevateDB	TEDBDatabase	http://elevate
FireDac	TFireDacConnectionAdapter	RemoteDB.Drivers.FireDac	TFDConnection	Delphi native
NexusDB	TNexusDBConnectionAdapter	RemoteDB.Drivers.NexusDB	TnxDatabase	http://www.n
SQL-Direct	TSQLDirectConnectionAdapter	RemoteDB.Drivers.SqlDirect	TSDDatabase	http://www.sc
UniDac	TUniDacConnectionAdapter	RemoteDB.Drivers.UniDac	TUniConnection	http://www.d

You can also use native database drivers:

Database	Driver Name	Connection class	Declared in unit
Microsoft SQL Server	MSSQL	TMSSQLConnection	Aurelius.Drivers.MSSQL

For more information on using native drivers, please refer to TMS Aurelius documentation.

Creating the adapter

To create the adapter, you just need to instantiate it, passing an instance of the component to be adapted. In the example below, a dbExpress adapter constructor receives a TSQLConnection component.

```
MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1, True);
```

The adapter usually detects the SQL Dialect automatically, but you can force the adapter to use a specific dialect, using one of the following overloaded constructors.

Note that for RemoteDB, the SQLConnection1 cannot be shared between different IDBConnection interfaces. Thus, you must create one database-access component for each IDBConnection interface you create.

Overloaded constructors

There are some overloaded versions of the constructor for all adapters:

```
constructor Create(AConnection: T; AOwnsConnection: boolean); overload; virtual;

constructor Create(AConnection: T; ASQLDialect: string; AOwnsConnection:
boolean); overload; virtual;

constructor Create(AConnection: T; OwnedComponent: TComponent); overload;
virtual;

constructor Create(AConnection: T; ASQLDialect: string; OwnedComponent: TComponen
t); overload; virtual;
```

- *AConnection*: specify the database-access component to be adapted.
- *AOwnsConnection*: if true, the component specified in AConnection parameter will be destroyed when the IDBConnection interface is released. If false, the component will stay in memory.
- *ASQLDialect*: defines the SQL dialect to use when using this connection. If not specified, Aurelius will try to discover the SQL Dialect based on the settings in the component being adapted.
- *OwnedComponent*: specifies the component to be destroyed when the IDBConnection interface is released. This is useful when using data modules (see below).

Memory Management

Note the second boolean parameter in the Create constructor of the adapter. It indicates if the underlying connection component will be destroyed when the IDBConnection interface is destroyed. This approach is borrowed from TMS Aurelius, but for RemoteDB, you should not keep the component alive after the IDBConnection interface is released. Always destroy the component with the interface (parameter must be true).

In the example above ("Creating the adapter"), the SQLConnection1 component will be destroyed after MyConnection interface is out of scope and released. Quick examples below:

```
var
  MyConnection: IDBConnection;
begin
  MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1, True);
  // ...
  MyConnection := nil;
  { MyConection is nil, the TDBExpressConnectionAdapter component is destroyed,
    and SQLconnection1 is also destroyed}
end;
```

Alternatively, you can inform a different component to be destroyed when the interface is released. This is useful when you want to create an instance of a TDataModule (or TForm) and use an adapted component that is owned by it. For example:

```
MyDataModule := TConnectionDataModule.Create(nil);
MyConnection := TDBExpressConnectionAdapter.Create(MyDataModule.SQLConnection1, M
yDataModule);
```

The previous code will create a new instance of data module TConnectionDataModule, then create a IDBConnection by adapting the SQLConnection1 component that is in the data module. When MyConnection is released, the data module (MyDataModule) will be destroyed (and in turn, the SQLConnection1 component will be destroyed as well). This is useful if you want to setup the connection settings at design-time, or have an existing TDataModule with the database-access component already properly configured. Then you just use the code above in RemoteDB server to create one IDBConnection for each instance of the data module.

dbGo (ADO) Adapter

Currently dbGo (ADO) is only officially supported when connecting to Microsoft SQL Server databases. Drivers for other databases might work but were not tested.

SQL Dialects

When creating an IDBConnection interface using an adapter, you can specify the SQL dialect of the database server that RemoteDB server connects to.

Currently this is only used by TMS Aurelius and **does not affect** RemoteDB behavior if you are just using TXDataset and not using any Aurelius clases. But RemoteDB might need it in future for some operations, so we suggest you create the server passing the correct SQL Dialect.

When you create an IDBConnection interface using a component adapter, usually the adapter will automatically retrieve the correct SQL dialect to use. For example, if you are using dbExpress components, the adapter will look to the DriverName property and tell which db server you are using, and then define the correct SQL dialect name that should be used.

Note that in some situations, the adapter is not able to identify the correct dialect. It can happen, for example, when you are using ODBC or just another data access component in which is not possible to tell which database server the component is trying to access. In this case, when creating the adapter, you can use an overloaded constructor that allows you to specify the SQL dialect to use:

```
MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1, 'MSSQL',
False);
```

The following table lists the valid SQL Dialect strings you can use in this case.

SQL dialect	String identifier	Database Web Site
Advantage	Advantage	http://www.sybase.com
DB2	DB2	http://www.ibm.com
ElevateDB	ElevateDB	http://www.elevatesoftware.com
Firebird	Firebird	http://www.firebirdsql.org
Interbase	Interbase	http://www.embarcadero.com
Microsoft SQL Server	MSSQL	http://www.microsoft.com/sqlserver
MySQL	MySQL	http://www.mysql.com
NexusDB	NexusDB	http://www.nexusdb.com
Oracle	Oracle	http://www.oracle.com
PostgreSQL	PostgreSQL	http://www.postgresql.org
SQLite	SQLite	http://www.sqlite.org

TRemoteDBModule settings

Before creating the RemoteDB Server by adding the TRemoteDBModule object to the Http Server, you can use some of its properties to configure the server module.

Basic authentication properties

```
property UserName: string;
property Password: string;
```

Use these properties to specify UserName and Password required by the server, using Basic authentication. By default, the values are: UserName = *remotedb*, Password = *business*. Since basic authentication is used, be sure to use Http secure (Https) if you don't want your user name/password to be retrieved by middle-man attack. If you don't use Http secure, user name and password are transmitted in plain text in http requests.

Instance Timeout

```
property InstanceTimeout: integer;
```

TMS RemoteDB Server keeps database-access components in memory while clients are connected. It will destroy everything when client is normally closed. However, there might be situations where the client crashes and never notifies the server to destroy the database-access component. In this case, the server will eventually destroy the component after the instance timeout is reached, i.e., the time which the component didn't receive any request from the client.

This value must be specified in milliseconds, and default value is 60000 (1 minute).

TRemoteDBServer Component

TRemoteDBServer component wraps the TRemoteDBModule module to be used at design-time.

Properties

Name	Description
Connection: TAureliusConnection	Contains a reference to a TAureliusConnection component. This will be used as the connection factory for the TRemoteDB server-side database instances.
DefaultInstanceTimeout: Integer	Defines for how long a connection component should stay alive in the server without any contact from the client. After such time, the instance will be destroyed and any further request from the client will recreate a new component.
UserName: string; Password: string;	Defines user name and password to be used for Basic authentication.

Events

```
OnModuleCreate: TRemoteDBModuleEvent
```

Fired when the TRemoteDBModule instance is created.

```
TRemoteDBModuleEvent = procedure(Sender: TObject; Module: TRemoteDBModule) of obj
ect;
```

Module parameter is the newly created TRemoteDBModule instance.

Server-Side Events

TRemoteDBServer (and TRemoteDBModule) publish several events that you can use to monitor what's going on server-side. The only difference between the two is that TRemoteDBServer includes a "Sender" parameter of type TObject, which is the standard for design-time events. All the other parameters are the same for TRemoteDBServer and TRemoteDBModule. Of course, you can define event-handlers for TRemoteDB server from the IDE (since it's a design-time component), and for TRemoteDBModule you need to set it from code.

The events are mostly used for logging purposes.

RemoteDB events

OnDatabaseCreate and *OnDatabaseDestroy*

OnDatabaseCreate event is fired right after a database-access component is created in the server (from a call to IDBConnectionFactory interface). On the other hand, *OnDatabaseDestroy* is called right before the component is about to be destroyed. Event signature is the following:

```
procedure(Sender: TObject; Info: IDatabaseInfo)
```

Look below to see what's available in IDatabaseInfo interface.

BeforeStatement and *AfterStatement*

BeforeStatement and *AfterStatement* events are fired right before (or after) an SQL statement is executed server-side. Event signature is the following:

```
procedure(Sender: TObject; Info: IStatementInfo)
```

Look below to see what's available in IStatementInfo interface. It's worth noting that when the SQL statement execution raises an exception, *BeforeStatement* event is fired, but *AfterStatement* is not.

IDatabaseInfo interface

Represents a database connection in the server. The following properties are available.

Name	Description
Id: string	The internal Id for the database connection.
LastAccessed: TDateTime	The last time (in server local time zone) the connection was requested (used) by the client.
ClientID: string	The ID of the client which created the connection.
ClientIP: string	The IP address of the client which created the connection.

Name	Description
Connection: IDBConnection	The underlying IDBConnection interface used to connect to the database.

IStatementInfo interface

Represents the SQL statement being executed. The following properties are available.

Name	Description
Database: IDatabaseInfo	The IDatabaseInfo interface (database connection) associated with the statement being executed.
Sql: string	The SQL statement to be executed.
Params: TEnumerable<TDBParam>	The parameters to be bound to SQL statement. TDBParam is a TMS Aurelius object which contain properties ParamName, ParamType and ParamValue.
Operation: TStatementOperation	The type of operation being performed with the statement. It can be one of the these values. Note that depending on client behavior, statement-related events can be fired more than once: one with *FieldDefs* operation (to retrieve SQL fields) and then a second one with *Open* operation, to return actual data. Sometimes, a single operation that does both will be executed (*FieldDefsOpen*).
Dataset: TDataset	The underlying TDataset component used to retrieve data. Note Dataset can be nil (in the case of *Execute* operation, for example).

TStatementOperation

- *Open*: Execution of a statement that returns data (SELECT).
- *Execute*: Execution of a statement that does not return data (INSERT, UPDATE, DELETE).
- *FieldDefs*: Retrieval of field definitions of a statement that returns data. The SQL will not be actually executed.
- *FieldDefsOpen*: Retrieval of field definitions and data return.

Administration API

RemoteDB provides an administration API that helps you to know status of existing database connections in the server and drop existing connections, if needed. The API is disabled by default. To enable, you have to set EnableAPI property to true (in either TRemoteDBServer or TRemoteDBModule):

```
Module.EnableApi := True;
```

The API provides the following endpoints (relative to server base URL):

Retrieve database connections

```
GET api/databases
```

Returns a JSON array with information about the existing database connections. For example:

```
[
    {
        "Id": "8FFDF133-286E-4C04-94D0-4479342FE389",
        "LastAccess": "2019-07-04T18:50:41.068Z",
        "ClientId": "Client A",
        "ClientIP": "::1",
        "Connected": true,
        "InTransaction": false
    }
]
```

Drop existing connection

```
DELETE api/databases/{id}
```

Drops an existing database connection, identified by is Id.

RemoteDB Client Applications

The following topics provide detailed info about how to create Delphi client applications that connect to a TMS RemoteDB Server.

TRemoteDBDatabase Component

TRemoteDBDatabase component is the one you should use to configure connection settings to a RemoteDB Server. Once you have created and configured the component, you can link any TXDataset to it to execute SQL statements in the RemoteDB server. It's a regular Delphi component so you can drop it in a form or data module to use it.

Usage example:

```
uses
  {...}, RemoteDB.Client.Database;

function CreateRemoteDBDatabase: TRemoteDBDatabase;
var
begin
  Result := TRemoteDBDatabase.Create(nil);
  Result.ServerUri := 'http://localhost:2001/tms/business/remotedb/';
  Result.UserName := 'remotedb';
  Result.Password := 'business';
  Result.Connected := true;
end;
```

Key Properties

```
property ServerUri: string;
```

Specifies the Url of the RemoteDB server.

```
property Connected: boolean;
```

Set this property to true to establish a connection to the RemoteDB server. As most of Delphi database components, TRemoteDBDatabase component will try to automatically connect to the server when a TXDataset tries to execute an SQL statement.

```
property UserName: string;
property Password: string;
```

Defines the UserName and Password to be used to connect to the server. These properties are initially set with the default values (*remotedb:business*). In production environment, build a server with different values, set these properties accordingly to make a connection, and use Https to ensure user name and password are encrypted in client/server communications.

```
property Timeout: integer;
```

Defines the lifetime of inactive **server-side** database objects. The server-side database connection will be kept alive while the client keeps sending requests to it. If the client suddenly interrupts the requests without explicitly destroying the database, the object will remain in the server for the period specified by the server by default. You can use Timeout property to define such time at client-side in a per-database manner. Value must be in milliseconds.

```
property ClientID: string;
```

Defines a value to identity the current client. This is useful to identify the client from the server. Such information is avaliable, for example, in server-side events, or administration API, to identify the client ID associated with an existing database connection.

Key Methods

```
procedure BeginTransaction;
procedure Commit;
procedure Rollback;
function InTransaction: boolean;
```

Use the above methods to start, commit and rollback transactions, respectively. The InTransaction property allows you to check if a transaction is already active. Only a single transaction can exist per database component.

Key Events

```
TRemoteDBHttpClientEvent = procedure(Sender: TObject; Client: THttpClient) of obj
ect;
property OnHttpClientCreate: TRemoteDBHttpClientEvent
```

OnHttpClientCreate event is fired when a new Sparkle THttpClient object is created by the RemoteDB database. THttpClient is used for the low-level HTTP communication with RemoteDB server and this event is an opportunity to do any custom configuration you want in that object.

TGenericDatabase Component

TGenericDatabase component is used if you want to connect to your database server directly, in a traditional client-server architecture, using your preferred database-access component through component adapters.

By switching the Database property of a TXDataset between TGenericDatabase and TRemoteDBDatabase you can easily build one single client application that can communicate directly with the database server in traditional client-server approach (TGenericDatabase) and communicate with a RemoteDB server through http (TRemoteDBDatabase). And you can do that without needing to change the dataset component you use to access the local or remote database: just use the same TXDataset component.

TGenericDatabase doesn't actually contains the code to connect to the database. It's just a wrapper for the component adapter. To use it, you need to set its Connection property to point to an IDBConnection interface.

Usage example using FireDac to connect to the database:

```
uses
  {...}, RemoteDB.Client.GenericDatabase, RemoteDB.Drivers.FireDac;

function CreateGenericDBDatabase: TGenericDatabase;
var
begin
  Result := TGenericDatabase.Create(nil);
  Result.Connection := TFireDacConnectionAdapter.Create(FDConection1, False);
  Result.Connected := true;
end;
```

Key Properties

```
property Connection: IDBConnection;
```

Specifies the IDBConnection interface used to connect to the database.

```
property Connected: boolean;
```

Set this property to true to establish a connection to the database server. Available at runtime.

Key Methods

```
procedure BeginTransaction;
procedure Commit;
procedure Rollback;
function InTransaction: boolean;
```

Use the above methods to start, commit and rollback transactions, respectively. The InTransaction property allows you to check if a transaction is already active. Only a single transaction can exist per database component.

TXDataset Component

TXDataset is the main component you will use to perform SQL statements on the RemoteDB Server. It's a TDataset descendant so you can use it anywhere in your Delphi application that supports TDataset components.

Its usage is very similar to any TDataset component:

1. Set the Database property to the property TRemoteDBDatabase component to be used.
2. Use the SQL property to define the SQL statement.
3. Use Params property to define SQL param values, if any.
4. Create persistent TField components, if needed.
5. Open the dataset to retrieve results (for SELECT statements) or call Execute method to execute the SQL statement

Simple example:

```
uses
  {...}, RemoteDB.Client.Dataset;

var
  Dataset := TXDataset.Create(Self);
begin
  Dataset.Database := RemoteDBDatabase1;
  Dataset.SQL.Text := 'Select * from Customers';
  Dataset.Open;
  while not Dataset.EOF do
  begin
    {process}
    Dataset.Next;
  end;
  Dataset.Close;
  Dataset.Free;
end;
```

See additional topics about TXDataset below.

Updating Records

When you call Post or Delete methods in TXDataset, data is modified in the internal database buffer, but no modification is automatically done directly in database. The only way to modify data is by executing INSERT, UPDATE or DELETE SQL statements. You have two options here: let RemoteDB do it automatically for you, using AutoApply property, or do it manually using events.

Automatic update

You can set TXDataset.AutoApply property to true (default is false) to let RemoteDB automatically perform SQL statements to modify data when data is posted or deleted in the dataset.

There is only one thing you need to define manually: tell RemoteDB what are the key (primary key) fields of the table. This way it can build the proper WHERE clause when executing the SQL statements.

You have two ways for doing that:

- Use TXDataset.KeyFields property, providing the key field names in a semicolon-separated list.

or

- Set pfInKey flag in ProviderFlags property of dataset persistent fields.

In summary, use the following properties (you can also set at desing-time):

```
XDataset1.AutoApply := True;
XDataset1.KeyFields := 'Id';
```

RemoteDB will automatically try to retrieve the name of the table to be updated, from the SQL statement. It might not be able to do it in some more complex SQL statements. In this case you can provide the name of the table to be udpated using UpdateTableName property:

```
XDataset1.UpdateTableName := 'Customers';
```

Updating manually using events

When AutoApply is false (default), calls to Post and Delete update the in-memory cache, but do not perform any SQL update/insert/delete operation on the RemoteDB Server (and, in turn, in the SQL database server).

In this mode you have more flexibility to perform updates, but then you must manually provide the code to perform such operations, using TXData events:

- OnRecordInsert
- OnRecordUpdate
- OnRecordDelete

Those events are called in the proper time you need to execute the SQL statement to respectively INSERT, UPDATE and DELETE a record in the database server.

```
property ModifiedFields: TList<TField>;
```

You can then use ModifiedFields property to verify which fields were modified by the dataset. It can be useful in case you want to perform UPDATE or INSERT SQL statements and only update/insert the fields modified by the user.

You also need to execute the SQL statements yourself, either using a different TXDataset component for that, or calling TRemoteDBDatabase.ExecSQL method directly.

Master-Detail Setup

You can setup a master-detail relationship between two TXDataset components using the Datasource property. That property behaves as specified in Delphi documentation. According to this source:

Setting DataSource property will automatically fill parameters in a query with field values from another dataset. Parameters that have the same name as the fields in the other dataset are filled with the field values. Parameters with names that are not the same as the fields in the other dataset do not automatically get values, and must be programmatically set. For example, if the SQL property of the TXDataset contains the SQL statement below and the dataset referenced through DataSource has a CustNo field, the value from the current record in that other dataset is used in the CustNo parameter:

```
SELECT *
FROM Orders O
WHERE (O.CustNo = :CustNo)
```

Other Methods and Properties

This topic lists some key methods and properties of TXDataset component, in addition to those inherited from TDataset component.

```
procedure Execute;
```

Executes an SQL for data modification (Insert, Delete, Update statements).

```
procedure FetchAllRecords;
```

Retrieves all remaining records for the dataset and closes the internal data provider.

```
property ModifiedFields: TList<TField>;
```

Provides a list of fields which values were modified. Useful in record update operations.

```
property SQL: TStrings;
```

Contains the SQL statement to be executed in the server.

```
property Params: TParams;
```

Use to to define the values for the params in SQL statement. Params should be declared in SQL using commands (:paramname). The list of params is updated automatically when you change the SQL property.

```
property DataSource: TDataSource;
```

Defines the datasource where param values will be automatically retrieved. Used to setup master-detail datasets.

```
property Database: TXDatabase;
```

Points to the database (usually TRemoteDBDatabase) where the SQL statements will be executed to.

```
property OnRecordInsert: TNotifyEvent;
property OnRecordUpdate: TNotifyEvent;
property OnRecordDelete: TNotifyEvent;
```

Events for updating records.

```
property Unidirectional: boolean;
```

Set Unidirectional property to true to save memory and increase performance if you are going to retrieve records from the dataset in a unidirectional way. If Unidirectional is set to true and you try to return to a previous record (using First or Prior methods for example), an error will be raised.

Batch Updates

If you want to insert, update or delete several records at the same time, using the same SQL statement, you can use the batch update feature - also known as Array DML.

In this mode, a single SQL statement is sent to the server, and multiple values are passed for each parameter. For example, consider the following SQL statement:

```
XDataset1.SQL.Text := 'INSERT INTO Cities (Id, Name) VALUES (:Id, :Name)';
```

If you want to insert three records using the same statement, this is how you should do it:

```
XDataset1.ParamByName('Id').DataType := ftInteger;
XDataset1.ParamByName('Name').DataType := ftString;
XDataset1.Params.ArraySize := 3;
XDataset1.ParamByName('Id').Values[0] := 1;
XDataset1.ParamByName('Name').Values[0] := 'London';
XDataset1.ParamByName('Id').Values[1] := 2;
XDataset1.ParamByName('Name').Values[1] := 'New York';
XDataset1.ParamByName('Id').Values[2] := 3;
XDataset1.ParamByName('Name').Values[2] := 'Rio de Janeiro';
XDataset1.Execute;
```

The advantage of this approach is that a single HTTP request containing the SQL statement and all parameters will be send to the server. This increases performance, especially on environments with high latency.

In addition to that, if the database-access component you are using server-side supports Array DML (like FireDAC or UniDAC), then it will also increase performance server-side significantly, by also using Array DML to actually save data in the database. Otherwise, a batch update will be simulated, by preparing the SQL statement and executing it for each row of parameters.

Connecting TMS Aurelius to RemoteDB Server

From the client application, you can use TMS Aurelius to retrieve objects from a RemoteDB server, instead of a regular database server. All you need is to use the proper Aurelius IDBConnection interface. Just like TMS Aurelius can connect to SQL databases using FireDac, dbExpress, etc., it also provides a RemoteDB driver adapter which you can use to perform the database connection. Once you do that, Aurelius usage is exactly the same as with any IDBConnection and you can create a TObjectManager object and use Find, Update, Delete methods, perform queries, and other operations.

Aurelius provides an adapter for a TRemoteDBDatabase component, which in turn is used to connect to a RemoteDB server. The following code illustrates how to use it, for more information please refer to the Component Adapters topic in TMS Aurelius documentation.

```
uses
  RemoteDB.Client.Database, Aurelius.Drivers.RemoteDB;

function CreateClientConnection: IDBConnection;
var
  XDB: TRemoteDBDatabase;
begin
  XDB := TRemoteDBDatabase.Create(nil);
  Result := TRemoteDBConnectionAdapter.Create(XDB, true);
  XDB.ServerUri := 'http://localhost:2001/tms/business/remotedb/';
end;

{ Aurelius usage is exactly the same }
Connection := CreateClientConnection;
Manager := TObjectManager.Create(Connection);
Customers := Manager.Find<TCustomer>.List;
```

Note that in client you don't need (and shouldn't) create both TRemoteDBDatabase and IDBConnection for each manager you use. The code above is just an explanation about how to create those classes. In a real client application, you would create the IDBConnection interface and share it among different TObjectManager instances.

About

This documentation is for TMS RemoteDB.

In this section:

What's New

Copyright Notice

Getting Support

Breaking Changes

What's New

Version 2.17 (Nov-2023)

- **New: Delphi 12 Support.**

Version 2.16 (Jan-2023)

- **Fixed:** Sporadic Access Violation when destroying TRemoteDBDatabase component. Ticket #19922.

Version 2.15 (Dec-2022)

- **New: TXDataset.AutoFillDetailFields property**.
- **Fixed:** Detail dataset not being updated after master dataset was closed and reopened (regression).

Version 2.14 (Sep-2022)

- **Improved**: Design-time components were greyed out in component palette if current platform was different than Win32.
- **Fixed:** Field Not Found error with master detail link when changing SQL. (ref: https://support.tmssoftware.com/t/field-not-found-error-with-master-detail-link-when-changing-sql/19183)
- **Fixed:** Access Violation when closing form with TXDataset in dsInsert (ref: https://support.tmssoftware.com/t/access-violation-when-closing-form-with-txdataset-in-dsinsert/19234)
- **Fixed:** Access Violation when using Master-Detail field alias with double quotes as parameter (ref: https://support.tmssoftware.com/t/access-violation-when-using-master-detail-field-alias-with-double-quotes-as-parameter/19236)

Version 2.13 (Sep-2021)

- **New:** Delphi 11 support.

Version 2.12 (Mar-2021)

- **New: Linux support (Indy-based server only)**. You can now use RemoteDB server on Linux, using Indy-based servers.

Version 2.11 (Sep-2020)

- **Improved:** Performance increase when executing SQL statements with a big number of parameteres.

Version 2.10 (Aug-2020)

- **Fixed:** Events BeforeStatement and AfterStatement were not correctly passing the value of SQL parameters used in the statement.

Version 2.9 (Jun-2020)

- **Fixed:** InvalidFieldSize exception in float fields when using RemoteDB with native SQL Server driver (MSSQL).
- **Fixed:** Duplicated field name when using native SQL Server driver (MSSQL) and the SQL statement returned ambiguous (same name) fields.

Version 2.8 (Jun-2020)

- **New: Support for native database-access drivers.** You can now use TRemoteDB using the database native drivers available in TMS Aurelius. Up until this version, RemoteDB servers and TGenericDatabase could only be used with dataset-based components like FireDAC, UniDAC, dbExpress, etc.. Now you can use the native Aurelius database drivers available (except for SQLite), which don't require any component at all, and are even faster!
- **Improved:** Support for batch updates when using TGenericDatabase. Up to this version, only TRemoteDBDatabase was supporting batch updates.
- **Improved:** Significant performance improvement when defining dataset pararameters (ArraySize property) when using batch updates.
- **Fixed:** Settings null parameter values when using batch updates was not working.

Version 2.7 (May-2020)

- **New: Support for Delphi 10.4 Sydney.**
- **Fixed:** TXDataset.AutoApply property not working when the key field itself was modified.

Version 2.6 (Apr-2020)

- **Fixed:** "Field not found" when using master-detail mechanism (regression from last version).
- **Fixed:** Using two parameters with the same name, differentiated just by a number (param1, param2), the same param value were being used for both params.

Version 2.5 (Apr-2020)

- **New: Batch updates (Array DML) mechanism for improved performance when inserting, updating or deleting several records at the same time.** It is used both client-side (to use a single request to send SQL statement and all parameters at once) and server-side (to execute a single SQL statement in the database passing all parameters for all rows.

Version 2.4 (Mar-2020)

- **New: Integration plugin for Report Builder from Digital Metaphors. More info in this Youtube video.**

Version 2.3 (Nov-2019)

- **New: Support for Android 64-bit platform (Delphi 10.3.3 Rio).**

Version 2.2 (Oct-2019)

- **Fixed:** Error "Field <fieldname> is of unknown type" in blob fields when updating records using AutoApply.
- **Fixed:** AutoApply was not working when field names had spaces. If you were editing a record and modified a field which name had a space (like "Contact Name") and tried to Post the record using AutoApply, an error would be raised.
- **Fixed:** Correct handle of fields of type ftOraClob. Now they have the same behavior as ftMemo.

Version 2.1 (Sep-2019)

- **Improved:** Optimizations have been implemented, performance has been improved, especially when retrieving big amount of data and/or executing many statements that use many query parameters each.
- **Fixed:** A SQL syntax error were being raised when using AutoApply and more than one field were modified by the dataset.
- **Fixed:** Previous version had broken compatibility with NexusDB integration, it's now fixed.

Version 2.0 (Jul-2019)

- **New: Administration API for retrieving server status (database connections including last accessed time, client id, etc.) and dropping database connections.**
- **New: Server-side events OnDatabaseCreate, OnDatabaseDestroy, BeforeStatement and AfterStatement.** Data is provided in IDatabaseInfo and IStatementInfo with full information about the database connections and statements being executed.

- **New: Automatic data modification using TXDataset properties: AutoApply and KeyFields.** Allow for automatic data modification by executing UPDATE, INSERT and DELETE SQL statements.
- **New: macOS 64 support in Delphi Rio 10.3.2.**
- **New: TRemoteDBDatabase.ClientId property allows identifying clients in server.**
- **New: TMS RemoteDB VCL Server Wizard** makes it easy to create a new RemoteDB server application using design-time components.

Version 1.15 (Jun-2019)

- **Fixed:** Basic authentication was being enforced even when UserName and Password properties were empty strings. (1.15.1 fixed a regression on this).

Version 1.14 (May-2019)

- **New: TRemoteDBServer component provides design-time support to create server-side RemoteDB server.**
- **Improved:** RemoteDB simple demo updated to use the new TRemoteDBServer component.

Version 1.13 (Jan-2019)

- **Improved:** Increased performance when using TGenericDatabase and executing SQL statements (non-SELECT).
- **Fixed:** Error when using SQL syntax with double colon, e.g. PostgreSQL type cast - ::varchar(20).

Version 1.12 (Dec-2018)

- **Fixed:** TXDataset was creating invalid parameters in some situations, for example in stored procedures with code like A := B.

Version 1.11 (May-2018)

- **Improved:** Exception raised by the client is now ERemoteDBRequestException and includes the status code of the HTTP response.

Version 1.10 (Feb-2018)

- **New: TRemoteDBDatabase.OnHttpClientCreate event.** This allows more customization of the underlying THttpClient object used to perform HTTP requests to RemoteDB server.
- **New: TRemoteDBDatabase.BeforeConnect and AfterConnect events.**

- **Improved:** TXDataset.Execute function now returns the number of rows affected by the operation.

Version 1.9.1 (Oct-2017)

- **New: TXDataset.ParamByName method.** Just an alias for TXDataset.Params.ParamByName.

Version 1.9 (Jul-2017)

- **New: TRemoteDBDatabase.OnRequestSending property.** You can now use this event (equivalent to Sparkle's OnSendingRequest) to customize the HTTP request sent to the RemoteDB server (to send custom HTTP headers, for example).

Previous Versions

Version 1.8 (May-2017)

- Improved: RemoteDB IDBStatement now implements IDBDatasetStatement
- Fixed: AV when setting TXDataset.Database property to a TGenericDatabase without having is Connection property set.

Version 1.7 (Mar-2017)

- New: Delphi 10.2 Tokyo Support
- Fixed: Detail dataset in master-detail set was being closed/open upon editing master record.

Version 1.6 (Sep-2016)

- Fixed: Memo parameter values not being sent correctly when using TDatasetProvider.
- Fixed: Index out of bounds when using TXDataset as a dataset provider and dbGo (ADO) at RemoteDB server side.

Version 1.5 (Aug-2016)

- New: Support for IProviderSupport allows using RemoteDB dataset (TXDataset) as a dataset provider - for example, in a setup using TClientDataset + TDatasetProvider + TXDataset.

Version 1.4 (May-2016)

- New: Delphi 10.1 Berlin support.

Version 1.3 (Feb-2016)

- New: Design-time wizard to create new RemoteDB Server with a few clicks.
- Fixed: Error when executing SQL statements with string literals containing quotes or double quotes.

Version 1.2.4 (Sep-2015)

- New: Delphi 10 Seattle support.

Version 1.2.3 (Aug-2015)

- Fixed: Setting TRemoteDBDatabase.Connected to true at design-time was causing error at runtime.

Version 1.2.2 (Apr-2015)

- New: Delphi XE8 support.

Version 1.2.1 (Mar-2015)

- Fixed: Client sending wrong data in memo fields with empty strings.

Version 1.2 (Oct-2014)

- New: TGenericDatabase component allows using TXDataset to connect to database servers directly in old client-server style.
- New: Support for Advantage Database Server.
- Improved: Server database objects lifetime: now objects expire right after timeout period has passed (in previous versions it took longer).
- Improved: Client transactions now can get longer than timeout of server db objects, RemoteDB keeps server-side objects alive automatically.
- Fixed: Error with long-running queries that take longer than timeout of server db objects. Now RemoteDB ensures queries will execute not matter how long they take.
- Fixed: Server issues when queries caused infinite deadlocks.
- Fixed: Sporadic "CoInitialize not called" error in RemoteDB demo.
- Fixed: Wrong initial values when inserting a record using TXDataset in Delphi XE4 and up.

Version 1.1.1 (Sep-2014)

- New: Delphi XE7 support.

Version 1.1 (Aug-2014)

- New: TRemoteDBDatabase.Timeout property allows specifying lifetime of inactive database connections on server side.
- New: TXDataset.Unidirectional property improves memory usage and performance when using dataset in forward-only (unidirectional) mode.
- Improved: Server now returns a better error message if a transaction commit/rollback request fails.
- Fixed: Error when executing data modification SQL statements (Insert, Delete, Update) using Open method instead of Execute, when using SQL-Direct and some other specific components.
- Fixed: Server instability in rare situations when connection/disconnection to database at server side raised errors.

Version 1.0 (Apr-2014)

- First public release.

Copyright Notice

The trial version of this product is intended for testing and evaluation purposes only. The trial version shall not be used in any software that is not run for testing or evaluation, such as software running in production environments or commercial software.

For use in commercial applications or applications in production environment, you must purchase a single license, a small team license or a site license. A site license allows an unlimited number of developers within the company holding the license to use the components for commercial application development and to obtain free updates and priority email support for the support period (usually 2 years from the license purchase). A single developer license allows ONE named developer within a company to use the components for commercial application development, to obtain free updates and priority email support. A small team license allows TWO developers within a company to use the components for commercial application development, to obtain free updates and priority email support. Single developer and small team licenses are NOT transferable to another developer within the company or to a developer from another company. All licenses allow royalty free use of the components when used in binary compiled applications.

The component cannot be distributed in any other way except through TMS Software web site. Any other way of distribution must have written authorization of the author.

Online registration/purchase for this product is available at http://www.tmssoftware.com. Source code & license is sent immediately upon receipt of payment notification, by email.

Getting Support

General notes

Before contacting support:

- Make sure to read this whole manual and any readme.txt or install.txt files in component distributions, if available.
- Search TMS support forum and TMS newsgroups to see if you question hasn't been already answer.
- Make sure you have the latest version of the component(s).

When contacting support:

- Specify with which component is causing the problem.
- Specify which Delphi or C++Builder version you're using and preferably also on which OS.
- For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.

Send email from an email account that

1. allows to receive replies sent from our server
2. allows to receive ZIP file attachments
3. has a properly specified & working reply address

Getting support

For general information: info@tmssoftware.com

Fax: +32-56-359696

For all questions, comments, problems and feature request for our products: help@tmssoftware.com

---

IMPORTANT
All topics covered by this manual are officially supported and it's unlikely that future versions will break backward compatibility. If this ever happens, all breaking changes will be covered in this manual and guidelines to update to a new version will be described. However, it's important to note that parts of the source code of this product that are undocumented are not officially supported and are **subject to change**, which includes breaking backward compatibility. In case you are using an unsupported/undocumented feature we will not provide support for upgrading and will not officially support it.

Breaking Changes

List of changes in each version that breaks backward compatibility from a previous version.

No breaking changes so far.