Overview

TMS Echo is a Delphi framework for data replication. It's part of TMS Business product line, and it's relies on TMS Aurelius to function.

What does it do?

It allows you to have two or more databases and synchronize data between them. Changes you make to one client database (inserts, updates, deletes) can be sent to another database. It's bidirectional, so you can send and receive changes between databases.

How does it work?

It creates a log of all the changes made to a database (using TMS Aurelius), and when requested, it sends those changes to the other nodes (databases) registered to receive changes from that one database. Thanks to relying on TMS Aurelius, it works very smoothly and almost automatically, since it benefits from existing information and mechanisms like logging, mapping, model information, etc.

How can I benefit from it, in what types of applications can I use it?

TMS Echo is great if you want to distribute your databases so you don't need to have a direct connection to a central database for your application to work. It will perform data on a local database and synchronize data to central database when needed. Some examples of scenarios:

- Client applications that need to work offline
    - Applications on mobile devices with slow connection (3G, bad Wifi)
    - Users that go out of office with notebooks/mobile devices to places where there are no internet connections
    - Any scenario where connection to server is not 100% reliable and you need to be sure your application will be available to work all the time (save data)
- Better performance by using local databases
    - Improve the application overall performance by saving data in a local database (or a database in a local server network)
    - Send data to a remote server in a background/service process, while keeping the user experience smooth by saving data locally
- Client data is only a subset of central, server data (multi-tenancy)
    - You might have a distributed system where you need a central database server with all data for static purposes, but each client will only have access to a subset of that data (multi-tenancy architecture)

In this section:

Initial Setup

Steps to set up databases before implementing the replication process in your application.

Replication Process

Details about each step of the replication process performed by TMS Echo.

Reference

Key classes and interfaces used in TMS Echo.

TEcho Class

IEchoRemoteNode Interface

IEchoNodeManager Interface

Initial Setup

TMS Echo replicates data between two or more databases. For TMS Echo to work, **every database** that will be part of this replication system needs to go through an initial setup process.

1. Update the database schema

TMS Echo saves lots of data in internal tables in the database. Those tables need to be created before you can use TMS Echo in that database.

2. Register Self Node

Each database in the replication system is called a node. A node is actually a concept and the database "becomes" a node when it's registered as a node (when some identity data is saved in some TMS Echo internal tables). This is needed to each database in the replication system has an "Id", so TMS Echo knows how to identify each different database.

3. Create the Echo Server for node communication (optional)

Nodes need to communicate with each other. This communication between the local node and a Remote Node might be a direct database connection or through an Echo server. If your case is the latter, then you need to start up the server.

4. Register Remote Node

Once you have turned all databases into nodes (i.e., registering the self node in each database, giving each of them an Id), you need to make each node know each other. This is done by registering the remote node. In TMS Echo there is no concept as "client/server", just nodes. So when you register a remote node, it just means makes node "A" knows that node "B" exists and vice-versa (a reference to node A is saved in the database B and vice-versa). But usually, you will have a typical client/server setup, where the "client" application/database is the replica of a central server database. So usually, it's the client application that will register itself as a node in the server.

Once you have done the steps above, the ecosystem is set up, and you can then implement the replication process in your application.

Database Schema

For TMS Echo to work it needs some tables to be present in the database schema. It uses those tables to save the nodes, the log of changes, batches, among other things.

It's very easy to create those tables using the TDatabaseManager object provided by TMS Aurelius. You just need to pass the correct TMappingExplorer object to the TDatabaseManager which holds all the mapped classes needed by Echo. Such mapping explorer is available in the TEcho.Explorer class property.

```
uses {...}, Echo.Main, Aurelius.Engine.DatabaseManager;

var
  DB: TDatabaseManager;
begin
  // Create needed tables and fields for TMS Echo usage
  DB := TDatabaseManager.Create(Echo.Pool.GetConnection, TEcho.Explorer);
  try
    DB.UpdateDatabase;
  finally
    DB.Free;
  end;
```

In the example above, the IDBConnection needed by the TDatabaseManager is provided from the connection pool of the TEcho object. This is just for convenience, you can use any IDBConnection just as you use in your TMS Aurelius application.

The above code is enough to create the needed tables in an existing database. If you want to validate the database update, you might get some warnings though, because the updater will indicate that there are already existing tables in the database (the tables used by your application).

If you want to avoid such warnings (which don't affect the update itself, it's just informative), you can mix classes from both mappings: the ones needed by your application, and the ones needed by TMS Echo. To do that, all you have to do is pass an TArray<TMappingExplorer> as the second parameter of the Create constructor, with all explorers you need. For example:

```
  // Create needed tables and fields for TMS Echo usage
  DB := TDatabaseManager.Create(Echo.Pool.GetConnection,
    TArray<TMappingExplorer>.Create(
      TMappingExplorer.Default, // include tables needed by application
      TEcho.Explorer // include tables for TMS Echo
  );
{...}
```

Nodes

A node represents a copy (replica) of the database. In other words, in a replication system where you have multiple databases, each database is named a node. Each database has a list of nodes that it knows of including the self node. The self node represents the local database itself, and the other nodes listed in the local database represent the other databases that will replicate with the local one.

Self Node

To define that a database is a node in the replication system, you need to create a node and then define it as the self node. The Echo variable in the example is an instance of a TEcho class.

```
var
  NodeManager: IEchoNodeManager;
begin
  { First thing TMS Echo requires is to define the SelfNode. Each database
    in the TMS Echo replication system is known as a "node" and must be
    identified. So if the SelfNode of this database is not defined yet, we must
do it }

  // Get the NodeManager for the local database
  NodeManager := Echo.GetNodeManager;

  // If SelfNode is already defined, no need to define it again
  if NodeManager.SelfNode <> nil then Exit;

  // SelfNode is not defined, so let's create a Node instance with the id
  // specified by the NodeId variable. It can be any string value, it just needs
  // to be unique among all other databases being replicated.
  NodeManager.CreateNode('client1');

  // Now define this newly created node as the SelfNode
  NodeManager.DefineSelfNode(DBName);
```

Remote Node

Strictly speaking, there is no such "local node" and "remote node". All databases in the replication system are just nodes. But we often refer to a node either as "local" or "remote", and this has a meaning from the programming point of view.

A local node is the node (database) your application is working with, and the database you always have direct access to (a "local" database).

A remote node is a node that you push data to or pull data from, and is represented in TMS Echo by the IEchoRemoteNode interface. It's just another database, but the difference is that the database at remote node can be accessed in two ways:

1. Using a direct database connection

In this case, the application will access the remote database (node) directly, in a client/server manner. To push data to or pull data from the remote node, the application will in essence perform SQL operations directly in the remote database. This is how to retrieve the IEchoRemoteNode interface in this scenario:

```
var
  ServerDatabasePool: IDBConnectionPool;
{...}
  RemoteNode := Echo.GetRemoteNode(ServerDatabasePool);
```

2. Using a Echo server (based on XData)

In this case, the application will communicate with the remote node using an Echo server. To push data to or pull data from the remote node, the application will perform Http requests to the server, which in turn will save/load data in the remote database. This is useful when the remote database is not directly accessible. For this scenario to work, you have to be sure that an Echo server is running at the remote node. This is how to retrieve the IEchoRemoteNode interface in this scenario:

```
var
  RemoteNode: IEchoRemoteNode;
{...}
  RemoteNode := Echo.GetRemoteNode('http://serverhost/tms/echo/servernode');
```

Registering Remote Nodes

Once you have defined the self node for the database, you can register additional remote nodes that represent the database which the local node will exchange data with.

This is needed because when Echo is routing the data, it needs to know to which nodes the data will be routed to. The registration needs to be made just once and it represents a bidirectional communication.

To register the local node in a remote node, you just need to retrieve the IEchoRemoteNode interface and call RegisterNode method:

```
var
  RemoteNode: IEchoRemoteNode;
...
  RemoteNode.RegisterNode;
```

When you do this TMS Echo will:

1. Create a node entry in the remote database representing the local node. So when routing happens at the remote database server, changes will be routed to local node.
2. Create a note entry in the local database representing the remote node. So when routing happens at the local database, changes will be routed to the remote node.

Creating the Echo Server

A local node communicates with a remote node via direct database connection or through an Echo server. If you want to have the nodes communicate via the server, you need to be sure the server is running.

The Echo server is just a TMS Sparkle module using the TMS XData technology. You can refer to documentation of both products, mainly TMS Sparkle, to learn more about how to create the server. This is basically what you need to setup (assuming the ConnectionPool variable has a valid reference to an IDBConnectionPool interface):

```
uses Sparkle.HttpSys.Server, Echo.Server, {...};

{...}

  FSparkleServer: THttpSysServer;
  FEchoModule: TEchoServerModule;

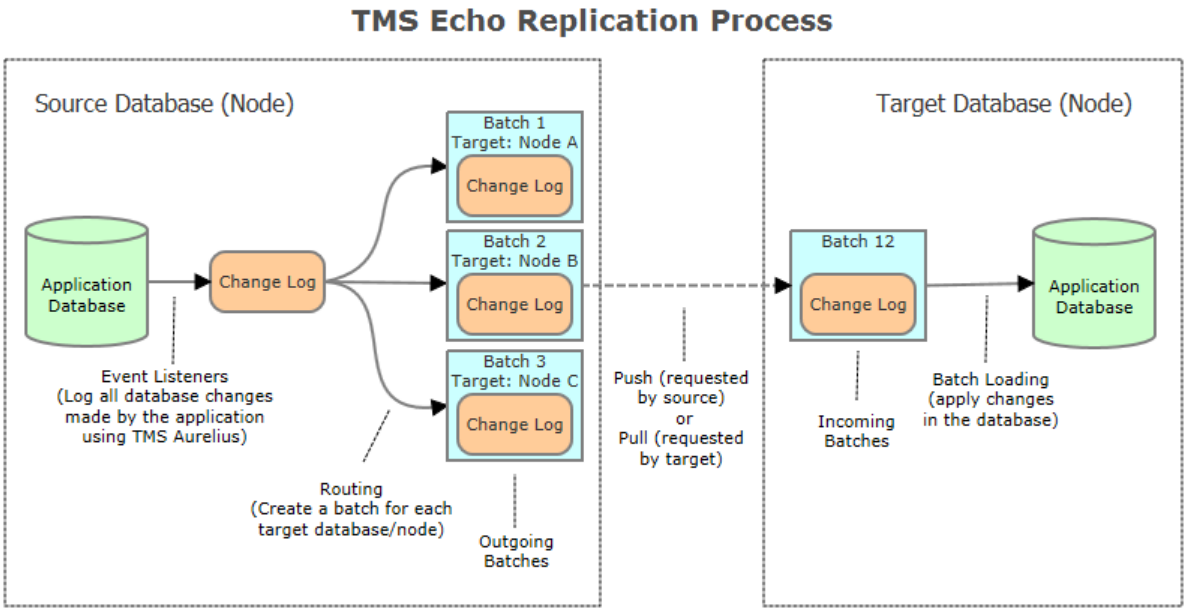
{..}

  FSparkleServer := THttpSysServer.Create;
  FEchoModule := TEchoServerModule('http://+:80/myechoserver', ConnectionPool);
  FSparkleServer.AddModule(FEchoModule);
  FSparkleServer.Start;
```

The URL for the server in the example is just an example, you should use the one that fits your needs.

Replication Process

The following picture illustrates how TMS Echo performs data replication.



The whole process is detailed here:

1. Logging Changes

Every database operation the application performs through TMS Aurelius is captured and logged, creating a "Change Log" in the database.

2. Routing

For each node (database) registered as a remote node (i.e., that should receive data from the source code), the router creates an outgoing batch with the log of changes. From the original change log, each outgoing batch can will receive its own change log, which can be the same or a subset of the original one, so each target node will have its own set of data to receive.

3. Transfering Data (Push/Pull)

This step is the communication with the remote node. Or, in other words, is how the batch is transferred from one database (node) to another. The transfer can be requested by the application being executed in the source code (which will be a "Push" to target node) or can be requested by the application being executed in the target node (which will be a "Pull" from the source node). The way data is transferred depends on how the IEchoRemoteNode interface is created. It can be through a TMS XData Rest/Json server, or can be direct database connection (i.e., the application in the source node access the database in the target node directly).

The data transferred will be come an incoming batch at the target database (node).

4. Batch Loading

This is the step where all the incoming batches are read, the change log in the batch is processed and for each item in the log the respective operation is performed in the target database. In other words, that's when the data is effectively updated in the remote database.

Logging Changes

For TMS Echo to replicate data correctly, it needs to log all changes made to the database. Strictly speaking, TMS Echo replicates changes, not data. So logging the changes is a crucial thing for it to work. Also, Echo only logs data changed through Aurelius, so **all your database manipulation must be done using TMS Aurelius**. If you perform any operation directly in database with using Aurelius, or if you do not subscribe the events to log changes made by Aurelius, such data will **not be replicated**.

To log changes for replication, TMS Echo relies on TMS Aurelius event system. You need to create the Echo listener and subscribe it:

```
uses {...}, Echo.Listeners;

  // Subscribe listeners at the begining of your application,
  // before any database operation is performed
  // The subscription is made by TMappingExplorer. In this case we're
  // subscribing to the default mapping explorer (calling SubscribeListeners
without params)
  FEchoSubscriber := TEchoEventSubscriber.Create;
  FEchoSubscriber.SubscribeListeners;

  // At the end of application, unsuscribe the listeners,
  // this will stop logging
  FEchoSubscriber.UnsubscribeListeners;
  FEchoSubscriber.Free;
```

Keep FEchoSubscriber instance reference alive for the whole time of the application (or at least while logging is important), and that is enough. Remember that Echo will save the log to internal tables it uses, so you must be sure the database schema is updated and has those tables created.

As described in TMS Aurelius documentation about the event system, listeners are registered per TMappingExplorer object. The SubscribeListeners method, when used without parameters, will subscribe to the default explorer (TMappingExplorer.Default). If you have multiple explorers in your application (for example because you use a multi-model design or any other reason), then you have to be sure to subscribe the listeners to all the TMappingExplorer objects used by your application which maps to data that you want to be replicated:

```
FEchoSubscriber.SubscribeListeners(TMappingExplorer.Get('Finance'));
FEchoSubscriber.SubscribeListeners(TMappingExplorer.Get('Marketing'));
FEchoSubscriber.SubscribeListeners(AnotherMappingExplorerInstance);
```

and also unsubscribe from all of them:

```
FEchoSubscriber.UnsubscribeListeners(TMappingExplorer.Get('Finance'));
FEchoSubscriber.UnsubscribeListeners(TMappingExplorer.Get('Marketing'));
FEchoSubscriber.UnsubscribeListeners(AnotherMappingExplorerInstance);
```

Routing

Routing is the process of processing logged changes and create batches targeting each remote node registered to receive data. This is a more detailed description of the routing mechanism:

1. Get a list of the logged changes that has not been routed yet (status is "unrouted");
2. Create a batch for each registered remote node;
3. Include the list of changes in the batch, so those changes are scheduled to be sent to that specific node;
4. Flag the list of logged changes as routed (status becomes "routed") so they are not sent again.

Once batches are created, they can be transferred to the remote nodes.

To execute the routing process, you just need to call TEcho.Route method:

```
Echo.Route;
```

And that is enough. It's important that you call Route from time to time in your application to create the batches. If you don't do so, batches are never created, and data is never transferred. It's up to you to decide the best way (how and when) you should execute the routing process.

It's also while routing that you can filter which data is transferred to each node. For example, supposed you are routing data at the server database to two different nodes, "Client1" and "Client2", and you don't want to send invoices to Client1. You can implement a custom logic this way:

```
uses {...}, Echo.Main, Echo.Entities;

Echo.Route(
    procedure(Log: TEchoLog; Node: TEchoNode; var Route: boolean)
    begin
      if SameText(Log.EntityClass, 'AppEntities.TEchoInvoice')
        and (Node.Id = 'Client1') then
        Route := false;
    end;
);
```

The above code means: if you are routing to the node with id "Client1", and you are trying to route a change in an object of class "AppEntities.TEchoInvoice", then don't route it.

So any change in TEchoInvoice object will only be send to "Client2", not "Client1".

Be aware that routing happens on both client and server side (or in better words, in all nodes) so you need to know exactly the source code (local database) and the target node (Node parameter) to implement your logic.

Transfering Data (Push/Pull)

Once your application is periodically logging changes and routing data, you will eventually have several outgoing batches targeting several remote nodes, waiting to be transferred to them. So you can transfer data to and from remote nodes, using the Push/Pull mechanism.

Here is how **Push** (to a specific remote node) works:

1. Get all outgoing batches targeting the remote node that has not been sent yet (status "Pending");
2. Send each of those batches to the remote node;
3. An incoming batch with same data will be locally saved in the database represented by the remote node;
4. The outgoing batch in the source node will be flagged as "Sent", to avoid being resent;
5. The incoming batch in the target node will be flagged as "Imported", meaning it's ready to be loaded.

Here is how **Pull** (from a specific remote node) works:

1. Asks for all outgoing batches in the remote node targeting the local node (status "Pending");
2. Receive each of those batches from the remote node;
3. An incoming batch with same data will be locally saved in the database represented by the local node;
4. The incoming batch in the local node will be flagged as "Imported", meaning it's ready to be loaded.
5. The remote node will be asked to flag the outgoing batch as "Sent", to avoid being resent;

So in very short words, "Push" sends local changes to remote node, and "Pull" receives remote changes to local node.

To Push or Pull, just call the respective method using the IEchoRemoteNode interface:

```
var
  RemoteNode: IEchoRemoteNode;
{...}
  RemoteNode.Push; // push changes to server
{...}
  RemoteNode.Pull; // pull changes from server
```

It's up to your application to periodically call Push and Pull to transfer pending batches from one node to another. If you don't do that, batches will be indefinitely pending to be sent and nodes won't receive data from each other.

Push/pull just transfer batches from one node to the other. It doesn't mean data will be updated in the database. You need to perform batch loading to finish the replication process and effectively have your data present in the database.

Batch Loading

Batch loading is the last stage of the replication process. After an outgoing batch is transferred from from source node to target node, an incoming batch is created in the target node. The incoming batch then needs to be loaded into the database, meaning that the changes in the batch are effectively applied (inserts, deletes, updates are performed).

Batch loading process described:

1. Retrieve all incoming batches pending to be loaded (status is "imported");
2. For each pending batch, get the list of changes in the batch;
3. Apply each change (insert, update, etc.) in the database;
4. Flag the batch so it won't be loaded again (status becomes "loaded").

To perform the batch load, just call TEcho.BatchLoad method:

```
Echo.BatchLoad;
```

It's up to your application to periodically call BatchLoad to apply received changes to the database.

Batch loading is the most critical process because it's where the replication effectively happens, i.e., where data from source node is effectively applied in the target node. This means that this is where errors and conflicts can arise. TMS Echo provides several mechanisms to minimize and solve those problems:

Fallback

When loading a batch, errors might arise because the data being loaded might be incompatible with existing data in the database. To minimize such problems, TMS Echo provides a fallback mechanism that automatically tries to solve some problems.

On Insert

If an Insert operation fails, TMS Echo checks if the object being inserted already exists in the database. If it does, it automatically tries an Update on that object with the same data and continue normally.

Fallback fails if the object does not exist (which means the Insert operation failed due to other reason than a key conflict with existing object) or if the fallback Update operation also fails.

On Update

If an Update operation fails, TMS Echo checks if the object being updated exists in the database. If it does not, it automatically tries to Insert such object again and continue normally.

Fallback fails if the object does exist in the database (which means the Update operation failed due to other reason than the object not existing in the database) or if the fallback Insert operation also fails.

On Delete

If a Delete operation fails, TMS Echo checks if the object being deleted exists in the database. If it does not, it just ignores the Delete operation and continue normally, since object has already been deleted.

Fallback fails if the object does exist in the database, which means the Delete operation failed due to other reason than the object not existing in the database.

If a fallback operation is performed and fails, an EFallbackException exception is raised, and it includes information about both exceptions: the original one which caused the fallback to be tried, and the one that was raised when fallback operation was performed.

Predictive Fallback

The automatic fallback can be predictive or not. By default, it's not predictive, which means Echo will try to execute the load operation normally, and if an exception is raised, then the fallback operation will be tried, if applicable.

You an alternatively set the fallback to predictive. This means for Echo will first check (execute an SQL) to see if the load operation will fail. Then it will already execute the fallback operation, if applicable. Predictive fallback is a **must if you are using PostgreSQL**, since it doesn't allow a transaction to continue if an exception is raised. To set the predictive fallback, just set the PredictiveFallback to true in the TEcho object:

```
Echo.PredictiveFallback := True;
```

Even though predictive fallback avoids raising exceptions in advance and allows fallback operations to be used in PostgreSQL, it makes the batch loading operation slower because it will execute an SQL SELECT statement before each load operation where fallback applies.

Disabling fallback

If for some reason you don't want Echo to perform the automatic fallback, you can disable it by setting the `FallbackEnabled` property to false:

```
Echo.FallbackEnabled := False;
```

TEcho Class

The TEcho class groups most of the operations you can perform in a single, helper class. Using this class you will be able to create and register nodes, perform routing, batch loading, push/pull data, etc..

TEcho class is declared in the Echo.Main unit, and since most of its operations are performed at the local database level, to create a TEcho instance you need to pass an IDBConnectionPool interface that provides connections to the local database. Please refer to the Database Connections topic for more information about database pools.

```
uses Echo.Main,
  Aurelius.Drivers.Base, Aurelius.Drivers.FireDac;

var
  Echo: TEcho;
{...}
  Echo := TEcho.Create(TDBConnectionFactory.Create(
    function: IDBConnection
    var
      MyDataModule: TMyDataModule;
    begin
      // This example creates a pool using a FireDac connection placed in the
      // TMyDataModule data module.
      // Please refer to proper topic to learn more about creating connection
      // pools and factories
      MyDataModule := TMyDataModule.Create(nil);
      Result := TFireDacConnectionAdapter.Create(MyDataModule.FDConnection1, MyDa
taModule);
    end
  );
```

Once you create a TEcho instance for a local database, you can keep it alive for the whole lifetime of your application, and use its methods and properties to perform the needed operations.

Methods

Name	Description
procedure BatchLoad;	Loads all the pending incoming batches into the database, using the default TMappingExplorer.
procedure BatchLoad(Explorer: TMappingExplorer)	Loads all the pending incoming batches into the database, using the TMappingExplorer object specified by the Explorer parameter.

Name	Description
procedure Route(AOnRouteToNode: TEchoRouteToNodeProc = nil)	Routes pending changes to all registered nodes.
function GetNodeManager: IEchoNodeManager	Retrieves the IEchoNodeManager interface for managing nodes in local database
function GetRemoteNode(const AServerUri: string): IEchoRemoteNode	Retrieves an IEchoRemoteNode interface representing the remote node accessible through the server specified by the ServerUri parameter.
function GetRemoteNode(TargetPool: IDBConnectionPool): IEchoRemoteNode	Retrieves an IEchoRemoteNode interface representing the remote node accessible through the database connection specified by the TargetPool parameter.

Properties

Name	Description
Pool: IDBConnectionPool	Contains a reference to the connection pool passed in the constructor method.
class property Explorer: TMappingExplorer	The TMappingExplorer object containing all classes and mappings used by TMS Echo
PredictiveFallback: Boolean	If True, Echo will use predictive fallback to anticipate possible errors in the batch loading and perform the fallback operation before the error happens.

Events

Name	Description
OnClientCreate: TProc<TXDataClient>	This event is fired whenever a TXDataClient object is created by TMS Echo to perform requests to a XData Server. You can use this event to configure the client (and the Sparkle HTTP Client using TXDataClient.HttpClient property), for example to add authentication, as in the following code.

```
Echo1.OnClientCreate :=
  procedure(XClient: TXDataClient)
  begin
    XClient.HttpClient.OnSendingRequest :=
      procedure(Req: THttpRequest)
      begin
        Req.Headers.SetValue('Authorization', 'Bearer ' + AuthToken);
      end;
  end;
```

Database Connections (Pools)

TMS Echo normalizes any database connection through the IDBConnectionPool interface. Whenever there is a need to connect to a database, it will get a connection from the pool. Most classes in TMS Echo that need a database connection will ask you to provide a connection pool.

The process of creating a connection pool is more detailed explained in the topic IDBConnection Pool interface in the TMS XData manual.

For TMS Echo, it's recommended that you create a pool using the TDBConnectionFactory class, since there is no strong need for pooling mechanism, just creating a connection each time is needed is enough.

So, for a quick reference, here is an example of how to create the connection (pool) from an existing TFDConnection (FireDac) placed in the data module TMyDataModule:

```
uses {...}, Aurelius.Drivers.Base, Aurelius.Drivers.Interfaces,
  Aurelius.Drivers.FireDac;

var
  ConnectionPool: IDBConnectionPool;
begin
  ConnectionPool := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        MyDataModule: TMyDataModule;
      begin
        // Creates a datamodule which contains a
        // TFDConnection component that is already configured
        MyDataModule := TMyDataModule.Create(nil);

        // The second parameter makes sure the data module will be destroyed
        // when IDBConnection interface is released
        Result := TFireDacConnectionAdapter.Create(MyDataModule.FDConnection1, My
DataModule);
      end
    ));
end;
```

IEchoRemoteNode Interface

The IEchoRemoteNode interface represents a remote node. More specifically, it actually represents the link between the local node and a remote node, since the IEchoRemoteNode interface is always retrieved from an existing local node context.

From the main TEcho class, you can retrieved the IEchoRemoteNode interface using the method GetRemoteNode. This method has two overloaded signatures, each one representing one of the two different ways of communication between nodes.

1. Rest/Json Server

In this case communication with the remote node will happen through a Rest/Json Echo server. To get the interface, pass the URL of the Echo server:

```
var
  RemoteNode: IEchoRemoteNode;
{...}
  RemoteNode := Echo.GetRemoteNode('http://serverhost/tms/echo/servernode');
```

2. Direct database connection

In this case the server database will be accessed directly. To get the interface, pass an IDBConnectionPool interface that connects with the remote database:

```
var
  ServerDatabasePool: IDBConnectionPool;
{...}
  RemoteNode := Echo.GetRemoteNode(ServerDatabasePool);
```

Methods

Name	Description
procedure Push;	Pushes all pending batches from the local node to the remote node.
procedure Pull;	Pulls all pending batches from remote node to local node.
procedure RegisterNode;	Registers the bidirectional replication link between local node and remote node

IEchoNodeManager Interface

The IEchoNodeManager interface is a helper interface to manage the nodes registered in the local database. It's also used to register the self node.

You can get the IEchoNodeManager interface form the TEcho object using the TEcho.GetNodeManager method:

```
NodeManager := Echo.GetNodeManager;
```

This interface has methods that receive and return object instances (like TEchoNode objects). The interface **manages** such instances, meaning that you don't need to destroy them. But note that once the interface reference count reaches zero and then destroyed, all the objects it manages are also destroyed. Pay attention to that when reusing objects returned by the interface.

Methods

Name	Description
function SelfNode: TEchoNode;	Retrieves the TEchoNode object representing the self node. If no self node is registered, nil is returned. The TEchoNode object is destroyed when the node manager is destroyed.
procedure DefineSelfNode(const NodeId: string);	Defines an existing node entry as the self node for the database. The node must be already created and the NodeId parameter must contain the id of the node.
function CreateNode(const Id: string): TEchoNode;	Creates a new node entry in the local database, with the specified Id. The newly created TEchoNode object with that id, representing the node entry, is returned. The node entry is just created in the local database and doesn't imply a communication with a remote node. You will rarely use this method besides for the purpose of creating the self node. For other node entries, you should register remote nodes using the IEchoRemoteNode interface. If a node with that Id already exists, an exception will be raised. The TEchoNode object is destroyed when the node manager is destroyed.

About

This documentation is for Echo.

In this section:

What's New

Copyright Notice

Getting Support

Breaking Changes

Online Resources

What's New

Version 1.10 (Oct-2021)

- **New:** Linux platform support.
- **New:** TEcho.FallbackEnabled property allows you to disable the default automatic fallback mechanism.

Version 1.9 (Sep-2021)

- **New:** Delphi 11 support.

Version 1.8 (Mar-2021)

- **Fixed:** Fallback insert operations not working when entities being loaded were belonging to a mapping explorer other than the default.

Version 1.7 (May-2020)

- **New: Support for Delphi 10.4 Sydney.**

Version 1.6 (Nov-2019)

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

Version 1.5 (Nov-2018)

- **New: Support for Delphi 10.3 Rio.**
- **Fixed:** Echo table structure was not capable of holding Unicode values. Echo Aurelius entities were modified to include a [DBTypeMemo] attribute in those fields.

Version 1.4 (Jul-2018)

- **New:** Demo application showing how to replicate data between two nodes using XData server as the channel to Push/Pull data.
- **Fixed:** Replicating data with version-control mapping ([Version] attribute) was not working properly, raising unneeded EVersionedConcurrencyControl exceptions.

Previous Versions

Version 1.3.2 (Oct-2017)

- Improved: Compatibility with latest TMS BCL library.

Version 1.3.1 (Jul-2017)

- Improved: Compatibility with latest TMS BCL library.

Version 1.3 (Mar-2017)

- New: Delphi 10.2 Tokyo Support.

Version 1.2 (Jan-2017)

- New: TEcho.OnClientCreate event for custom configuration of the HTTP client object used to connect to Echo server.
- Fixed: When OnRouteToNode event fires, sometimes empty batches (with no log) were being created to target nodes. Now only batches with log items are created.

Version 1.1 (May-2016)

- New: Delphi 10.1 Berlin support.

Version 1.0 (Feb-2016)

- 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 https://www.tmssoftware.com. Source code & license is sent immediately upon receipt of payment notification, by email.

Copyright © TMS Software. ALL RIGHTS RESERVED.

No part of this help may be reproduced, stored in any retrieval system, copied or modified, transmitted in any form or by any means electronic or mechanical, including photocopying and recording for purposes others than the purchaser's personal use.

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.

Online Resources

This topic lists some links to internet resources - videos, articles, blog posts - about TMS Echo.

Official Online Documentation: https://download.tmssoftware.com/business/echo/doc/web