Overview

TMS XData is a Delphi framework that allows you to create HTTP/HTTPS servers that expose data through REST/JSON. It is highly integrated into TMS Aurelius in a way that creating XData services based on applications with existing Aurelius mappings are just a matter of a few lines of code. XData defines URL conventions for adressing resources, and it specifies the JSON format of message payloads. It is heavily based on the OData standard. Such conventions, with the benefit of existing Aurelius mapping, allow building a full REST/JSON server with minimum writing of code. TMS XData uses TMS Sparkle as its core communication library.

TMS XData product page: https://www.tmssoftware.com/site/xdata.asp

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

TMS XData is a full-featured Delphi framework that allows you to create REST/JSON servers, using server-side actions named service operations, and optionally exposing TMS Aurelius entities through REST endpoints. Consider that you have an Aurelius class mapped as follows:

```
[Entity, Automapping]
TCustomer = class
strict private
  FId: integer;
  FName: string;
  FTitle: string;
  FBirthday: TDateTime;
  FCountry: TCountry;
public
  property Id: Integer read FId write FId;
  property Name: string read FName write FName;
  property Title: string read FTitle write FTitle;
  property Birthday: TDateTime read FDateTime write FDateTime;
  property Country: TCountry read FCountry write FCountry;
end;
```

With a few lines of code you can create an XData server to expose these objects. You can retrieve an existing TCustomer with an id equal to 3 using the following HTTP request, for example:

```
GET /tms/xdata/Customer(3) HTTP/1.1
Host: server:2001
```

And the JSON representation of the customer will be returned in the body of the HTTP response:

```
{
  "$id": 1,
  "@xdata.type": "XData.Default.Customer",
  "Id": 3,
  "Name": "Maria Anders",
  "Title": "Sales Representative",
  "Birthday": "1980-05-20",
  "Country": null
}
```

You can perform changes to objects through the REST interface, using a POST method to create new objects, DELETE to remove objects, and PUT or PATCH to update the objects. The following example will change the value of the *FTitle* property of the customer resource specified in the previous example:

```
PATCH /tms/xdata/Customer(1) HTTP/1.1
Host: server:2001

{
  "Title": "Marketing Manager"
}
```

You can also perform queries on existing objects. The following example will retrieve all customers with a country name equal to "USA", ordered by the customer's name.

```
GET /tms/xdata/Customer?$filter=Country/Name eq 'USA'&$orderby=Name&$top=10 HTTP/
1.1
Host: server:2001
```

The server will return a JSON array of objects containing all the filtered objects. You can use query paging to restrict the number of objects returned in each request.

Also, you can use service operations to implement custom business logic that uses Aurelius objects. By defining an interface and an implementation...

```
type
  [ServiceContract]
  IMyService = interface(IInvokable)
  ['{F0BADD7E-D4AE-4521-8869-8E1860B0A4A0}']
    function GetTopCustomersByState(const State: string): TList<TCustomer>;
  end;

{...}
function TMyService.GetTopCustomersByState(const State: string): TList<TCustome>;
begin
  Result := TXDataOperationContext.Current.GetManager.Find<TTCustomer>
    .Where(TLinq.Eq('State', State) and TLinq.Eq('Category', TStatus.VIP))
    .List;
end;
```

...you can easily invoke them from a Delphi client...

```
Client := TXDataClient.Create;
Client.Uri := 'http://server:2001/tms/xdata';
MyService := Client.Service<IMyService>;
TopNYCustomers := MyService.GetTopCustomersByState('NY');
// process customers
```

...or from an HTTP client:

```
POST /tms/xdata/MyService/GetTopCustomersByState HTTP/1.1
Host: server:2001

{
  "State": "NY"
}
```

Please refer to the Introduction topic of this manual which lists all major topics to learn more about XData and its features.

Features

Here is a list of main features of the TMS XData framework:

- Server based on the REST/JSON architecture.
- Easily accessible from different client platforms. For example: .NET, Java, JavaScript (since it is based on REST/JSON).
- Uses standard POST, GET, PUT and DELETE HTTP methods for data request and data modification operations.
- Service Operations for custom server-side business logic.
- Partial update of objects (PATCH).
- Full-featured query mechanism.
- Well-defined JSON representation of resources including entities, associations, streams and proxies.
- Support for streams (blobs).
- Several databases supported at the back end: SQL Server, MySQL, PostgreSQL, Oracle, Firebird, etc. (using TMS Aurelius).
- HTTP/HTTPS server architecture based on TMS Sparkle which provides:
    - HTTP server based on the Windows http.sys stack;
    - Built-in authentication mechanism with JWT (JSON Web Token) or Basic methods;
    - Support for HTTP Secure (HTTPS);
    - Kernel-mode caching and kernel-mode request queuing (less overhead in context switching);

- Multiple applications/processes can share (respond to) the same port (at different addresses/endpoints);
- Secure Sockets Layer (SSL) support in kernel-mode.

In this section:

Getting Started

Getting your first XData server and client applications running.

Service Operations

How to implement and use service operations to add business logic to your server and invoke it from clients.

TMS Aurelius CRUD Endpoints

CRUD endpoints defined by XData for applications using TMS Aurelius.

TXDataClient

Using TXDataClient object to send and receive objects to/from a XData server in a straightforward way.

JSON Format

XData representation of different structures in JSON format.

Design-Time Components

Overview about XData components for design-time usage.

XData Model

TXDataAureliusModel: description of the available service operations and entities published from the CRUD endpoints.

Server-Side Events

Events that can be used to implement additional server-side logic, customize XData behavior, among other tasks.

Authentication and Authorization

How to implement authentication and authorization to protect your API from unauthorized access.

Other Tasks

How-tos and examples about basic tasks you can do with XData in code.

Web Applications with TMS Web Core

The TMS XData Web-Client Framework: using XData servers from TMS Web Core applications.

Getting Started

It's very easy to get your first XData server and client applications running:

1. Create and run an "empty" server
2. Add your server-side logic using service operations
3. Send requests to the server from clients
4. (Optional) Automatically publish your existing Aurelius entities

1. Create and run an "empty" server

a. From Delphi IDE, choose *File* > *New* > *Other*;

b. From the dialog that appears, navigate to *Delphi Projects* > *TMS XData*;

c. Double click "*TMS XData VCL Server*" to create the server.

Done: a new project will be created, run it, and your server will be running at the address "*http://localhost:2001/tms*".

You have several different options for this step:

- Creating the Server Using the XData Server Wizards
- Creating the Server Using Design-Time Components
- Creating the Server Manually

Using the "XData Server Wizards" is just the more straightforward way to create a new server.

If you don't like wizards, you can simply create a new blank application and drop a couple of design-time components to create your server.

If you don't like design-time components and you want to do it 100% from code, just create the server manually.

2. Add your server-side logic using service operations

a. From Delphi IDE, choose *File* > *New* > *Other*;

b. From the dialog that appears, navigate to *Delphi Projects* > *TMS XData*;

c. Double click "*TMS XData Service*" to create a new service. Use the default settings for now.

Done: Two new units will be created, including two server-side sample methods: *Sum* and *EchoString*. Your server is doing something!

Using "XData Service Wizard" is just the more straightforward way to add server-side logic.

If you don't like wizards, you can simply create your service operations manually from code, creating a ServiceContract interface and ServiceImplementation class.

Your server is ready! Let's connect to it now!

3. Send requests to the server from clients

Connecting from Delphi client applications

If you are connecting to server from a **Delphi** application, accessing your server could not be easier. All you need is to use the TXDataClient object:

```
uses {...},  MyService, XData.Client;

var
  Client: TXDataClient;
  MyService: IMyService;
  SumResult: Double;
begin
  Client := TXDataClient.Create;
  Client.Uri := 'http://localhost:2001/tms/xdata';
  SumResult := Client.Service<IMyService>.Sum(10, 5);
  Client.Free;
end;
```

And that's it! You invoke XData methods as if they were regular procedures in your client application. The *MyInterface* unit and *IMyService* interface were created in the previous step above. You just *reuse* the same code so that in client you don't have to do anything else but call the interface method.

Of course, there are much more advanced features you can use, so you can learn more about TXDataClient.

Connecting from non-Delphi client applications

To invoke the same *Sum* operation above without TXDataClient, just perform an HTTP request:

```
GET /tms/xdata/myservice/sum?a=10&b=5 HTTP/1.1
Host: localhost:2001
```

And you will get your response in JSON format:

```
{
  "value": 15
}
```

Of course you can simply go to your web browser and navigate to address *"http://localhost:2001/tms/xdata/myservice/sum?a=10&b=5"* to test it.

XData server is a standard REST/JSON server. Meaning you can access it from any client application that can simply handle JSON and perform HTTP applications. That means virtually all kinds of applications: desktop, web, mobile, IoT, etc., regardless if those applications were built in C# .NET, Java, PHP, JavaScript, TypeScript, C/C++, Swift, etc.

4. (Optional) Automatically publish your existing Aurelius entities

If you use TMS Aurelius, then TMS XData can automatically create CRUD endpoints for some or all of your Aurelius entities. It's a nice feature that saves you time. Note that this is optional, TMS XData doesn't require you to use TMS Aurelius at all.

How to continue from this

Now you can implement your real server. Of course XData allows you to create complex service operations, not just simple ones like the Sum above.

You can receive and return parameters of many different types: primitive types like integers, doubles, strings, guids, dates; complex types like object and arrays; and several specific supported types like strings, streams, etc..

When using non-Delphi clients, it's interesting to learn how routing and parameter binding works, so you know exactly how to invoke a service operation from a non-Delphi application. It's also important to understand how JSON serialization works (how each Delphi type is represented as JSON) to know how to build the JSON to be sent, and how to interpret the received JSON.

NOTE
Any XData HTTP server is based on the TMS Sparkle framework. According to the Sparkle documentation, to use the Windows (http.sys) based server, you must first reserve the URL your service will process its requests from. If you have installed TMS Sparkle on your computer, the URL "*http://+:2001/tms*" is already reserved, so you can create your XData server under that address (all examples in this documentation use the base address "*http://server:2001/tms/xdata*". If you change your base URL to a different port, or to a URL that doesn't start with "tms", you must reserve that URL otherwise your server might fail to start.

Creating the Server Using the XData Server Wizards

The easiest and more straightforward way to get started with XData is using the wizards.

1. Choose *File* > *New* > *Other* and then look for the *TMS XData* category under "Delphi Projects".
2. There you find several wizards to create a new XData Server Application:
    - *TMS XData VCL Server*: Creates a VCL application that runs an XData server using http.sys
3. Choose the wizard you want, double-click and the application will be created.

As soon as you execute your application, the server is run. The application generated uses the design-time components. Simply learn about the components to configure them. For example, you can drop a *TFDConnection* component and set it to the *TAureliusConnection.AdaptedConnection* property to associate a database connection with the XData server.

You can also create the server manually. The wizard is not mandatory and it is one way to get started quickly.

Creating the Server Using Design-Time Components

If you don't want to use the XData Server Wizard, another way to create a XData Server is by manually dropping 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/data module** (for example, *TSparkeHttpSysDispatcher*).

2. **Drop a *TXDataDBServer* component on the form/data module**.

3. **Associate the TXDataDBServer component with the dispatcher through the *Dispatcher* property**.

4. **Specify the *BaseUrl* property of the server** (for example, *http://+:2001/tms/xdata*).

5. **Set the *Active* property of the dispatcher component to true.**

That's all you need. If you want to have a ready-to-use database connection pool, you can use the following extra steps:

6. **Drop a TAureliusConnection component on the form/data module 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. **Drop a TXDataConnectionPool component on the form/data module and associate it to the *TAureliusConnection* component through the *Connection* property.**

8. **Associate the *TXDataServer* component to the *TXDataConnectionPool* component through the *Pool* property.**

Creating the Server Manually

Here we describe the steps to create the XData server manually, from code, without using XData Server Wizard or the design-time components:

1. Create an IDBConnectionFactory interface
2. Create an IDBConnectionPool interface
3. Create a TXDataServerModule object
4. Create a THttpSysServer, add the module to it and start the server

1. Create an IDBConnectionFactory interface

The *IDBConnectionFactory* interface is used to create TMS Aurelius IDBConnection interfaces which will be used by the server to connect to the desired database (in that database Aurelius objects will be persisted). You can read more details on the specific topic here: IDBConnectionFactory interface.

Below we provide a code example that creates an IDBConnectionFactory interface which produces IDBConnection interfaces that connect to an MS SQL Server database, based on an existing TSQLConnection component named SQLConnection1 in a data module TMyConnectionDataModule.

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

var
  ConnectionFactory: IDBConnectionFactory;
begin
  ConnectionFactory := TDBConnectionFactory.Create(
    function: IDBConnection
    var
      MyDataModule: TMyConnectionDataModule;
    begin
      MyDataModule := TMyConnectionDataModule.Create(nil);
      Result := TDBExpressConnectionAdapter.Create(MyDataModule.SQLConnection1, M
yDataModule);
    end
  ));

  // Use the ConnectionFactory interface to create an IDBConnectionPool interface
end;
```

2. Create an IDBConnectionPool interface

The IDBConnectionPool interface is used by the server to retrieve an IDBConnection interface when it is needed. It holds a list of such interfaces, and if there is none available, it uses the IDBConnectionFactory interface to create a new one. Example:

```
uses
  {...}, Aurelius.Drivers.Interfaces, XData.Aurelius.ConnectionPool;

var
  ConnectionPool: IDBConnectionPool;
  ConnectionFactory: IDBConnectionFactory;
begin
  {...}
  ConnectionPool := TDBConnectionPool.Create(
    50, // maximum of 50 connections available in the pool
        // Define a number that best fits your needs
    ConnectionFactory
  );

  // Use the IDBConnectionPool interface to create the XData server module
end;
```

3. Create a TXDataServerModule object

The *TXDataServerModule* is the main class of the XData server. It is a Sparkle server module that is added to the Sparkle server for a specific address. Example:

```
uses
  {...}, XData.Server.Module;
var
  XDataModule: TXDataServerModule;
  ConnectionPool: IDBConnectionPool;
begin
  {...}
  XDataModule := TXDataServerModule.Create('http://+:2001/tms/xdata', ConnectionP
ool);
```

4. Create a THttpSysServer, add the module to it and start the server

Finally, create a TMS Sparkle THttpSysServer object, add the XData module to it, and start the server. Example:

```
uses
  {...}, Sparkle.HttpSys.Server, XData.Server.Module;
var
  XDataModule: TXDataServerModule;
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;
  Server.AddModule(XDataModule);
  Server.Start;
end;
```

Later on, do not forget to destroy the *Server* object instance when the application finishes.

Example 1: In-memory SQLite for testing/development

The following example is a minimum console application that illustrates how to start an XData server at address "http://localhost:2001/tms/music" using an in-memory SQLite database. It uses an unit named `AureliusEntities` which is not shown here for simplification. Such a unit should just contain the Aurelius mapped classes that will be exposed by the server.

An extra note about this example: Since it is an in-memory database, the database will be empty every time the server starts. Thus, before the server starts, the method "UpdateDatabase(Connection)", which is declared in unit `DatabaseUtils` (also not listed here), will be called to create the required tables and data. Such a thing is a regular TMS Aurelius procedure and is out of the scope of this example as well. The purpose of this example is to show how to start an XData server.

```
program SQLiteConsoleServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Aurelius.Drivers.Interfaces,
  Aurelius.Drivers.SQLite,
  Aurelius.Sql.SQLite,
  Aurelius.Schema.SQLite,
  Sparkle.HttpSys.Server,
  XData.Aurelius.ConnectionPool,
  XData.Server.Module,
  AureliusEntities,
  DatabaseUtils;

procedure StartServer;
var
  Server: THttpSysServer;
  Connection: IDBConnection;
begin
  Server := THttpSysServer.Create;
  try
    Connection := TSQLiteNativeConnectionAdapter.Create(':memory:');
    UpdateDatabase(Connection);
    Server.AddModule(TXDataServerModule.Create('http://localhost:2001/tms/music',
      TDBConnectionPool.Create(1,
        function: IDBConnection
        begin
          Result := Connection;
        end)));
    Server.Start;
```

```
    WriteLn('Server started. Press ENTER to stop.');
    ReadLn;
  finally
    Server.Free;
  end;
end;

begin
  StartServer;
end.
```

Example 2: MySQL Server with dbExpress (from Delphi code)

The following example is a minimum console application that illustrates how to start an XData server at address "http://localhost:2001/tms/music" using dbExpress to connect to a MySQL database. It is configured in code. It uses an unit named `AureliusEntities` which is not shown here for simplification. Such a unit should just contain the Aurelius mapped classes that will be exposed by the server.

The connection pool is configured to hold up to 25 connections to the database.

```
program DBExpressConsoleServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils, SqlExpr, DBXMySQL,
  Aurelius.Drivers.Base,
  Aurelius.Drivers.Interfaces,
  Aurelius.Drivers.dbExpress,
  Aurelius.Sql.MySQL,
  Aurelius.Schema.MySQL,
  Sparkle.HttpSys.Server,
  XData.Aurelius.ConnectionPool,
  XData.Server.Module,
  AureliusEntities in '..\common\AureliusEntities.pas';

procedure StartServer;
var
  Server: THttpSysServer;
  ConnFactory: IDBConnectionFactory;
begin
  Server := THttpSysServer.Create;
  try
    // Example using dbExpress. Create a connection factory to use later
    ConnFactory := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        SqlConn: TSQLConnection;
```

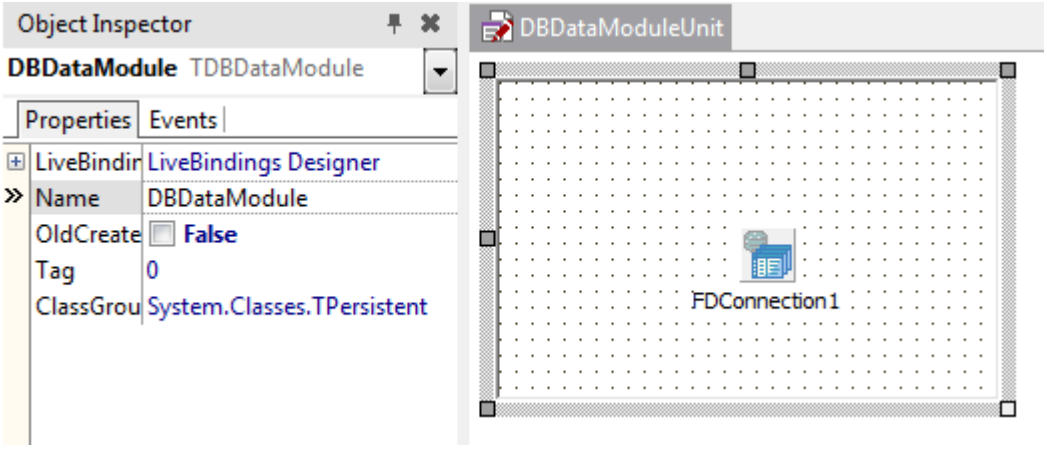
```
      begin
        SqlConn := TSQLConnection.Create(nil);
        SqlConn.DriverName := 'MySQL';
        SqlConn.GetDriverFunc := 'getSQLDriverMySQL';
        SqlConn.VendorLib := 'libmysql.dll';
        SqlConn.LibraryName := 'dbxmys.dll';
        SqlConn.Params.Values['HostName'] := 'dbserver';
        SqlConn.Params.Values['Database'] := 'xdata';
        SqlConn.Params.Values['User_Name'] := 'user';
        SqlConn.Params.Values['Password'] := 'mysql';
        SqlConn.LoginPrompt := false;
        Result := TDBExpressConnectionAdapter.Create(SqlConn, true);
      end
    );
    Server.AddModule(TXDataServerModule.Create(
      'http://localhost:2001/tms/music',
      TDBConnectionPool.Create(25, ConnFactory)
    ));
    Server.Start;
    WriteLn('Server started. Press ENTER to stop.');
    ReadLn;
  finally
    Server.Free;
  end;
end;

begin
  StartServer;
end.
```

Example 3: MS SQL Server with FireDAC (using TDataModule)

The following example illustrates how to start an XData server at address "http://localhost:2001/tms/music" using FireDac to connect to any database (in the example, a MS SQL Server database). Compared to Example 2, this example uses a *TFDConnection* dropped onto a TDataModule to configure the database connection at design-time. It uses an unit named `AureliusEntities` which is not shown here for simplification. Such a unit should just contain the Aurelius mapped classes that will be exposed by the server. The connection pool is configured to hold up to 15 connections to the database.

Thus, consider you have a data module with an existing configured database connection using FireDac:



You can use such a configuration and create one instance of that data module when required by the server. You can use the TFDConnection component from it.

```
program FireDacConsoleServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Aurelius.Drivers.Base,
  Aurelius.Drivers.Interfaces,
  Aurelius.Drivers.FireDac,
  Aurelius.Sql.MSSQL,
  Aurelius.Schema.MSSQL,
  Sparkle.HttpSys.Server,
  XData.Aurelius.ConnectionPool,
  XData.Server.Module,
  AureliusEntities in '..\common\AureliusEntities.pas',
  DBDataModuleUnit in 'DBDataModuleUnit.pas' {DBDataModule: TDataModule};

procedure StartServer;
var
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;
  try
    Server.AddModule(TXDataServerModule.Create('http://localhost:2001/tms/music',
      TDBConnectionPool.Create(15, TDBConnectionFactory.Create(
        function: IDBConnection
        var
          DBDataModule: TDBDataModule;
        begin
          // Example using FireDac and Data Module
          DBDataModule := TDBDataModule.Create(nil);
          Result := TFireDacConnectionAdapter.Create(DBDataModule.FDConnection1,
DBDataModule);
        end
    ))));
    Server.Start;
```

```
    WriteLn('Server started. Press ENTER to stop.');
    ReadLn;
  finally
    Server.Free;
  end;
end;

begin
  StartServer;
end.
```

Service Operations

Service operations are the core mechanism you use in XData to add business logic to your server and later invoke it from clients.

Service operations are invoked via HTTP requests, providing the correct URL and sending/receiving data via JSON payloads. This chapter explain hows to implement and use service operations.

XData Service Wizard

Easiest way to create service operations is by using the XData Service wizard.

From the Delphi IDE, choose *File* > *New* > *Other...*

From the dialog that appears, navigate to *Delphi Projects* > *TMS Business* and choose the wizard *TMS XData Service*.

That will launch the "New XData Service" dialog, which will provide you with the following options:

- *Service Name*: Specifies the base name for the service contract interface and service implementation class, as well the unit names to be generated.
- *Generate interface and implementation in separated units*: If checked, service contract interface and service implementation classes will be created in two different units. This makes it easy to reuse the interface for use in client applications using TXDataClient. This is true by default. If you are not going to use interfaces at client-side or if you simply prefer to put all code in a single unit, uncheck this option.
- *Add sample methods*: If checked, the wizard will add some sample methods (service operations) as an example.
- *Use a specific model*: If checked, adds a Model attribute to the service contract/implementation with the name specified in the edit box.

After options are set, click *Finish* and the full source code with the service contract and service implementation will be generated for you.

If you prefer to create the source code manually, or simply want to learn more about the generated source code, refer to the following topics:

- Service Operations Tutorial
- Creating Service Contract
- Service Implementation

Service Operations Overview

This chapter describes basic steps you need to follow to implement service operations at server side and invoke them from Delphi side. For detailed information about the steps, please refer to Service Operations main topic.

The presented source code samples are not complete units and only display the relevant piece of code for understanding service operations usage. Some obvious code and keywords might have been removed to improve readability. Here we will implement two server operations, one using scalar types (*Sum*), and another using entities (*FindOverduePayments*).

This tutorial also assumes that you have already created your XData Server application.

A service operation is built by creating two elements:

a. A Service Contract interface, which describes the methods and how they will be accessible from clients. This interface can be shared between server and client applications;

b. A Service Implementation class, which effectively implement the service contract. This is used only by the server app.

The following steps explains how to use service operations, including the creation of mentioned types. The XData Service Wizard automates steps 1 and 2 for you.

1. Define a service contract interface and declare the operations (methods) in it.

```
unit MyServiceInterface;

uses
  {...}, XData.Service.Common;

type
  [ServiceContract]
  IMyService = interface(IInvokable)
  ['{F0BADD7E-D4AE-4521-8869-8E1860B0A4A0}']
    function Sum(A, B: double): double;
    function FindOverduePayments(CustomerId: integer): TList<TPayment>;
  end;

initialization
  RegisterServiceType(TypeInfo(IMyService));
end.
```

This will add the contract to the XData model. You can use the Model attribute to specify the model where the contract belongs to:

```
uses {...}, Aurelius.Mapping.Attributes;
{...}
[ServiceContract]
[Model('Sample')] // adds interface to "Sample" model
IMyService = interface(IInvokable)
{...}
```

2. Create a service implementation class that implements the interface.

```
uses
  {...}, MyServiceInterface,
  XData.Server.Module,
  XData.Service.Common;

type
  [ServiceImplementation]
  TMyService = class(TInterfacedObject, IMyService)
  private
    function Sum(A, B: double): double;
    function FindOverduePayments(CustomerId: integer): TList<TPayment>;
  end;

implementation

function TMyService.Sum(A, B: double): double;
begin
  Result := A + B;
end;

function TMyService.FindOverduePayments(CustomerId: integer): TList<TPayment>;
begin
  Result := TList<TPayment>.Create;
  TXDataOperationContext.Current.Handler.ManagedObjects.Add(Result);
  // go to the database, instantiate TPayment objects,
  // add to the list and fill the properties.
end;

initialization
  RegisterServiceType(TMyService);

end.
```

3. Run the XData server.

It will automatically detect the service interfaces and implementations declared in the application. Your operations can now be invoked from clients. When the operation is invoked, the server creates an instance of the service implementation class, and invokes the proper method passing the parameters sent by the client.

4. If using Delphi applications: invoke the operation using XData client.

This tutorial assumes you created the xdata server at base address "http://server:2001/tms/xdata".

```
uses
  {...}
  MyServiceInterface,
  XData.Client;

var
  Client: TXDataClient;
  MyService: IMyService;
  SumResult: double;
  Payments: TList<TPayment>;
begin
  Client := TXDataClient.Create;
  Client.Uri := 'http://server:2001/tms/xdata';
  MyService := Client.Service<IMyService>;
  SumResult := MyService.Sum(5, 10);
  try
    Payments := MyService.FindOverduePayments(5142);
  finally
    // process payments
    Payments.Free;
  end;
  Client.Free;
end;
```

5. If using non-Delphi applications: invoke the operation using HTTP, from any platform and/or development tool.

Perform an HTTP Request:

```
POST /tms/xdata/MyService/Sum HTTP/1.1
Host: localhost:2001

{
  "a": 5,
  "b": 8
}
```

And get the response:

```
HTTP/1.1 200 OK

{
  "value": 13
}
```

Creating Service Contract

Service operations are grouped in service contracts, or service interfaces. You can define multiple service interfaces in your XData server, and each interface can define multiple operations (methods). You create a service contract by defining the service interface (a regular interface type to your Delphi application). The following topics in this chapter describes the steps and options you have to create such contract.

Defining Service Interface

These are the basic steps you need to follow to create an initial service contract interface:

1. Declare an interface inheriting from *IInvokable*.
2. Create a GUID for the interface (Delphi IDE creates a GUID automatically for you using *Shift+Ctrl+G* shortcut key).
3. Add `XData.Service.Common` unit to your unit uses clause.
4. Add *[ServiceContract]* attribute to your interface.
5. Declare methods in your interface.
6. Call *RegisterServiceType* method passing the typeinfo of your interface (usually you can do that in initialization section of your unit).

When you add the unit where interface is declared to either your server or client application, XData will automatically detect it and add it to the XData model as a service contract, and define all interface methods as service operations. The client will be able to perform calls to the server, and the server just needs to implement the interface. The source code below illustrates how to implement the service interface.

```
unit MyServiceInterface;

interface

uses
  System.Classes, Generics.Collections,
  XData.Service.Common;

type
  [ServiceContract]
  IMyService = interface(IInvokable)
  ['{BAD477A2-86EC-45B9-A1B1-C896C58DD5E0}']
    function Sum(A, B: double): double;
    function HelloWorld: string;
  end;

implementation

initialization
  RegisterServiceType(TypeInfo(IMyService));
end.
```

You can use the Model attribute to specify the model where the contract belongs to. This way you can have a server with multiple server modules and models.

```
uses {...}, Aurelius.Mapping.Attributes;
{...}
[ServiceContract]
[Model('Sample')] // adds interface to "Sample" model
IMyService = interface(IInvokable)
{...}
```

Routing

Each service operation is associated with an endpoint URL and an HTTP method. In other words, from clients you invoke a service by performing an HTTP method in a specific URL. The server mechanism of receiving an HTTP request and deciding which service operation to invoke is called routing.

NOTE
In all the examples here, the server base URL is assumed to be *http://localhost:2001*. Of course, use your own base URL if you use a different one.

Default Routing

By default, to invoke a service operation you would perform a POST request to the URL <service>/<action>, where <service> is the interface name without the leading "I" and <action> is the method name. For example, with the following service interface declaration:

```
IMyService = interface(IInvokable)
  function Sum(A, B: double): double;
```

you use the following request to invoke the *Sum* method:

```
POST /MyService/Sum
```

Note that the parameters follow a specific rule, they might be part of the routing, or not, depending on routing settings and parameter binding settings.

Modifying the HTTP method

The method can respond to a different HTTP method than the default POST. You can change that by using adding an attribute to method specifying the HTTP method the operation should respond to. For example:

```
IMyService = interface(IInvokable)
  [HttpGet] function Sum(A, B: double): double;
```

The above declaration will define that *Sum* method should be invoked using a GET request instead of a POST request:

```
GET /MyService/Sum
```

You can use attributes for other HTTP methods as well. You can use attributes: *HttpGet*, *HttpPut*, *HttpDelete*, *HttpPatch* and *HttpPost* (although this is not needed since it's default method).

Using Route attribute to modify the URL Path

By using the *Route* attribute, you can modify the URL path used to invoke the service operation:

```
[Route('Math')]
IMyService = interface(IInvokable)
  [Route('Add')]
  function Sum(A, B: double): double;
```

which will change the way to invoke the method:

```
POST /Math/Add
```

You can use multiple segments in the route, in both interface and method. For example:

```
[Route('Math/Arithmetic')]
IMyService = interface(IInvokable)
  [Route('Operations/Add')]
  function Sum(A, B: double): double;
```

will change the way to invoke the method:

```
POST /Math/Arithmetic/Operations/Add
```

An empty string is also allowed in both interface and method Route attributes:

```
[Route('Math/Arithmetic')]
IMyService = interface(IInvokable)
  [Route('')]
  function Sum(A, B: double): double;
```

The above method will be invoked this way:

```
POST /Math/Arithmetic
```

And also, the way you bind parameters can affect the endpoint URL as well. For example, the Route attribute can include parameters placeholders:

```
[Route('Math')]
IMyService = interface(IInvokable)
  [Route('{A}/Plus/{B}')]
  function Sum(A, B: double): double;
```

In this case, the parameters will be part of the URL:

```
POST /Math/10/Plus/5
```

Replacing the root URL

By default XData returns a service document if a GET request is performed in the root URL. You can replace such behavior by simply using the *Route* attribute as usual, just passing empty strings to it:

```
[ServiceContract]
[Route('')]
IRootService = interface(IInvokable)
['{80A69E6E-CA89-41B5-A854-DFC412503FEA}']
  [HttpGet, Route('')]
  function Root: TArray<string>;
end;
```

A quest to the root URL will invoke the *IRootService.Root* method:

```
GET /
```

You can of course use other HTTP methods like PUT, POST, and the *TArray<string>* return is just an example, you can return any type supported by XData, just like with any other service operation.

Conflict with Automatic CRUD Endpoints

TMS XData also provides endpoints when you use Aurelius automatic CRUD endpoints. Depending on how you define your service operation routing, you might end up with service operations responding to same endpoint URL as automatic CRUD endpoints. This might happen inadvertently, or even on purpose.

When that happens XData will take into account the value specified in the RoutingPrecedence property to decide if the endpoint should invoke the service operation, or behave as the automatic CRUD endpoint.

By default, the automatic CRUD endpoints always have precedence, starting with the entity set name. This means that if there is an entity set at URL `Customers/`, any subpath under that path will be handled by the automatic CRUD endpoint processor, even if it doesn't exist. For example, `Customers/Dummy/` URL will return a 404 error.

Setting `RoutingPrecedence` to `TRoutingPrecedence.Service` will change this behavior and allow you to override such endpoints from service operations. Any endpoint routing URL you define in service operations will be invoked, even if it conflicts with automatic CRUD endpoints.

Parameter Binding

Service operation parameters can be received from the HTTP request in three different modes:

- From a property of a JSON object in request body (*FromBody*);
- From the query part of the request URL (*FromQuery*);
- From a path segment of the request URL (*FromPath*).

In the method declaration of the service contract interface, you can specify, for each parameter, how they should be received, using the proper attributes. If you don't, XData will use the default binding.

Default binding modes

This are the rules used by XData to determine the default binding for the method parameters that do not have an explicity binding mode:

1. If the parameter is explicitly used in a Route attribute, the default mode will be *FromPath*, regardless of HTTP method used:

```
[Route('orders')]
IOrdersService = interface(IInvokable)
  [Route('approved/{Year}/{Month}')]
  [HttpGet] function GetApprovedOrdersByMonth(Year, Month: Integer):
TList<TOrder>;
```

Year and *Month* parameters should be passed in the URL itself, in proper placeholders:

```
GET orders/approved/2020/6
```

2. Otherwise, if the HTTP method is GET, the default mode will be *FromQuery*:

```
[Route('orders')]
IOrdersService = interface(IInvokable)
  [Route('approved')]
  [HttpGet] function GetApprovedOrdersByMonth(Year, Month: Integer):
TList<TOrder>;
```

Meaning *Year* and *Month* parameters should be passed as URL query parameters:

```
GET orders/approved/?Year=2020&Month=6
```

3. Otherwise, for all other HTTP methods, the default mode will be *FromBody*:

```
[Route('orders')]
IOrdersService = interface(IInvokable)
  [Route('approved')]
  [HttpPost] function GetApprovedOrdersByMonth(Year, Month: Integer): TList<TOrde
r>;
```

In this case, *Year* and Month parameters should be passed as properties of a JSON object in request body:

```
POST orders/approved/

{
  "Year": 2020,
  "Month": 6
}
```

FromBody parameters

FromBody parameters are retrieved from a JSON object in the request body, where each name/value pair corresponds to one parameter. The pair name must contain the parameter name (the one declared in the method), and value contains the JSON representation of the parameter value, which can be either scalar property values, the representation of an entity, or the representation of a collection of entities.

As specified above, FromBody parameters are the default behavior if the HTTP method is not GET and no parameter is explicitly declared in the *Route* attribute.

Example:

```
[HttpPost] function Multiply([FromBody] A: double; [FromBody] B: double): double;
```

How to invoke:

```
POST /tms/xdata/MathService/Multiply HTTP/1.1
Host: server:2001

{
  "a": 5,
  "b": 8
}
```

FromQuery parameters

FromQuery parameters are retrieved from the query part of the request URL, using *name=value* pairs separated by "&" character, and the parameter value must be represented as URI literal.

NOTE
You can only use scalar property values, or objects that only have properties of scalar value types.

If the *Multiply* operation of previous example was modified to respond to a GET request, the binding mode would default to FromQuery, according to the following example.

You can add the [FromQuery] attribute to the parameter to override the default behavior.

Example:

```
[HttpPost] function Multiply([FromQuery] A: double; [FromQuery] B: double): doubl
e;
```

How to invoke:

```
POST /tms/xdata/MathService/Multiply?a=5&b=8 HTTP/1.1
```

FromQuery is the default parameter binding mode for GET requests.

You can also use DTOs as query parameters, as long the DTO only have properties of scalar types. In this case, each DTO property will be a separated query param. Suppose you have a class `TCustomerDTO` which has properties `Id` and `Name`, then you can declare the method like this:

```
    [HttpGet] function FindByIdOrName(Customer: TCustomerDTO): TList<TCustomer>;
```

You can invoke the method like this:

```
GET /tms/xdata/CustomerService/FindByIdOrName?Id=10&Name='Paul' HTTP/1.1
```

WARNING
Using objects as parameters in query strings might introduce a breaking change in TMS Web Core applications using your XData server. If your TMS Web Core application was built a version of TMS XData below 5.2, the connection to the XData server will fail.

If you have TMS Web Core client applications accessing your XData server, and you want to use DTOs in queries, recompile your web client applications using TMS XData 5.2 or later.

FromPath parameters

FromPath parameters are retrieved from path segments of the request URL. Each segment is a parameter value, retrieved in the same order where the FromPath parameters are declared in the method signature. As a consequence, it modifies the URL address used to invoke the operation. The following example modified the *Multiply* method parameters to be received from path.

Example:

```
[HttpGet] function Multiply([FromPath] A: double; [FromPath] B: double): double;
```

How to invoke:

```
GET /tms/xdata/MathService/Multiply/5/8 HTTP/1.1
```

Parameter "A" was the first declared, thus it will receive value 5. Parameter "B" value will be 8. If the parameters are explicitly declared in the Route attribute, FromPath mode is the default mode. All the remaining parameters flagged with FromPath that were not included in the Route attribute must be added as additional segments.

Example:

```
[Route('{A}/Multiply')]
[HttpGet] function Multiply(A: double; [FromPath] B: double): double;
```

How to invoke:

```
GET /tms/xdata/MathService/5/Multiply/8 HTTP/1.1
```

Mixing binding modes

You can mix binding modes in the same method operation, as illustrated in the following example.

Method declaration:

```
procedure Process(
  [FromPath] PathA: Integer;
  [FromQuery] QueryA: string;
  BodyA, BodyB: string;
  [FromQuery] QueryB: Boolean;
  [FromPath] PathB: string;
): double;
```

How to invoke:

```
POST /tms/xdata/MyService/Process/5/value?QueryA=queryvalue&QueryB=true HTTP/1.1

{
  "BodyA": "one",
  "BodyB": "two"
}
```

Here are the values of each parameter received:

- *PathA*: 5
- *QueryA*: queryvalue
- *BodyA*: one
- *BodyB*: two
- *QueryB*: true
- *PathB*: value

Methods with a single FromBody object parameter

If the method has a single *FromBody* parameter (parameters with other binding type can exist) and that parameter is an object, for example:

```
procedure UpdateCustomer(C: TCustomer);
```

then clients must send the JSON entity representation of the customer directly in request body, without the need to wrap it with the parameter name (*C*, for example).

Methods with a single FromBody scalar parameter

If the method receives a single scalar parameter, for example:

```
procedure ChangeVersion(const Version: string);
```

In this case, clients can send the value of *Version* parameter in a name/value pair named "value", in addition to the original "Version" name/value pair. Both can be used.

Supported Types

When defining service interfaces, there are several types you can use for both parameter and return types of the operations.

Scalar types

You can use the regular simple Delphi types, such as Integer, String, Double, Boolean, TDateTime and TGUID, and its variations (like Longint, Int64, TDate, etc.). The server will marshal such values using the regular JSON representation of simple types. Variant type is also supported as long the variant value is one of the supported simple types (String, Integer, etc.).

Examples:

```
function Sum(A, B: double): double;
function GetWorld: string;
```

Enumerated and Set Types

Enumerated types and set types are also supported. For example:

```
type
  TMyEnum = (First, Second, Third);
  TMyEnums = set of TMyEnum;

procedure ReceiveEnums(Value: TMyEnums);
```

Enumeration values are represented in JSON as strings containing the name of the value ("First", "Second", according to the example above). Sets are represented as a JSON array of enumeration values (strings).

Simple objects - PODO (Plain Old Delphi Objects)

Any Delphi object can be received and returned. The objects will be marshaled in the HTTP request using object representation in JSON format. One common case for simple objects is to use it as structure input/output parameters, or to use them as DTO classes.

```
function ReturnClientDTO(Id: Integer): TClientDTO;
function FunctionWithComplexParameters(Input: TMyInputParam): TMyOutputParam;
```

Aurelius Entities

If you use Aurelius, then you can also use Aurelius entities. Aurelius entities are a special case of simple objects. The mechanism is actually very similar to entity representation, with only minor differences (like representation of proxies and associated entities). The entities will be marshaled in the HTTP request using entity representation in JSON Format.

Examples:

```
function ProcessInvoiceAndReturnDueDate(Invoice: TInvoice): TDateTime;
function AnimalByName(const Name: string): TAnimal;
```

Generic Lists: TList<T>

You can use *TList<T>* types when declaring parameters and result types of operations. The generic type T can be of any supported type. If T in an Aurelius entity, the the list will be marshaled in the HTTP request using the JSON representation for a collection of entities. Otherwise, it will simply be a JSON array containing the JSON representation of each array item.

```
procedure HandleAnimals(Animals: TList<TAnimal>);
function GetActiveCustomers: TList<TCustomer>;
function ReturnItems: TList<string>;
```

Generics arrays: TArray<T>

Values of type *TArray<T>* are supported and will be serialized as a JSON array of values. The generic type T can be of any supported type.

```
procedure ProcessIds(Ids: TArray<Integer>);
```

TJSONAncestor (XE6 and up)

For a low-level transfer of JSON data between client and server, you can use any *TJSONAncestor* descendant: TJSONObject, TJSONArray, TJSONString, TJSONNumber, TJSONBool, TJSONTrue, TJSONFalse and TJSONNull. The content of the object will be serialized/deserialized direct in JSON format.

Example:

```
procedure GenericFunction(Input: TJSONObject): TJSONArray;
```

TCriteriaResult

If you use Aurelius, you can return Aurelius *TCriteriaResult* objects, or of course a *TList<TCriteriaResult>* as a result type of a service operation. This is very handy to implement service operations using Aurelius projections.

For example:

```
function TTestService.ProjectedCustomers(NameContains: string): TList<TCriteriaRe
sult>;
begin
  Result := TXDataOperationContext.Current.GetManager
    .Find<TCustomer>
    .CreateAlias('Country', 'c')
    .SetProjections(TProjections.ProjectionList
      .Add(TProjections.Prop('Id').As_('Id'))
      .Add(TProjections.Prop('Name').As_('Name'))
      .Add(TProjections.Prop('c.Name').As_('Country'))
    )
    .Where(TLinq.Contains('Name', NameContains))
    .OrderBy('Name')
    .ListValues;
end;
```

The JSON representation for each TCriteriaResult object is a JSON object which one property for each projected value. The following JSON is a possible representation of an item of the list returned by the method above:

```
{
  "Id": 4,
  "Name": "John",
  "Country": "United States"
}
```

TStrings

Values of type *TStrings* are supported and will be serialized as JSON array of strings. Since TStrings is an abstract type, you should only use it as property type in objects and you should make sure the instance is already created by the object. Or, you can use it as function result, and when implementing you should also create an instance and return it. You cannot receive TStrings in service operation parameters.

TStream

You can also use *TStream* as param/result types in service operations. When you do that, XData server won't perform any binding of the parameter or result value. Instead, the TStream object will contain the raw content of the request (or response) message body. Thus, if you declare your method parameter as TStream, you will receive the raw message body of the client request in the server. If you use TStream as the result type of your method, the message body of the HTTP response send back to the client will contain the exact value of the TStream. This gives you higher flexibility in case you want to receive/send custom or binary data from/to client, such as images, documents, or just a custom JSON format.

```
function BuildCustomDocument(CustomerId: integer): TStream;
procedure ReceiveDocument(Value: TStream);
```

For obvious reasons, when declaring a parameter as TStream type, it must be the only input parameter in method to be received in the body, otherwise XData won't be able to marshal the remaining parameters. For example, the following method declaration is invalid:

```
// INVALID declaration
[HttpPost] procedure ReceiveDocument(Value: TStream; NumPages: Integer);
```

Since the method is Post, then NumPages will by default be considered to be received in the request body (FromBody attribute), which is not allowed. You can, however, receive the NumPages in the query part of from url path:

```
// This is valid declaration
[HttpPost] procedure ReceiveDocument(Value: TStream; [FromQuery] NumPages: Intege
r);
```

Value will contain the raw request body, and *NumPages* will be received from the URL query string.

Return Values

When a service operation executes succesfully, the response is 200 OK for operations that return results or 204 No Content for operations without a return type. If the operation returns a value, it will also be in the same JSON format used for parameters. The value is sent by the server wrapped by a JSON object with a name/value pair named "value". The value of the "value" pair is the result of the operation. In the *Multiply* example, it would be like this:

```
HTTP/1.1 200 OK

{
    "value": 40
}
```

There are some exceptions for the general rule above:

Methods with parameters passed by reference

In this case, the result will be a JSON object where each property relates to a parameter passed by reference. If the method is also a function (returns a value), then an additional property with name "result" will be included. For example, suppose the following method:

```
function TMyService.DoSomething(const Input: string; var Param1, Param2:
Integer): Boolean;
```

This will be a possible HTTP response from a call to that method:

```
HTTP/1.1 200 OK

{
    "result": True,
    "Param1": 50,
    "Param2": 30
}
```

Method which returns a single object

If the method returns a single object parameter, for example:

```
function FindCustomer(const Name: string): TCustomer;
```

then the server will return the JSON entity representation of the customer in the response body, without wrapping it in the "value" name/value pair.

Default Parameters

You can declare service operations with default parameters. For example:

```
function Hello(const Name: string = 'World'): string;
```

This will work automatically if you are invoking this service operation from Delphi clients. However, it will not work if you invoke this service operation directly performing an HTTP request. In this case you would have to provide all the parameter values otherwise a server error will be raised informing that a parameter is missing.

This is because the default value declared in the function prototype above is not available at runtime, but at compile time. That's why it works when calling from Delphi clients: the compiler will provide the default value automatically in the client call - but that's a client feature: the HTTP request will always have all parameter values.

To make the above method also work from raw HTTP requests and do not complain about missing parameters, you will need to inform (again) the default value using the [XDefault] attribute. This way XData server will know what is the default value to be used in case the client did not provide it:

```
function Hello([XDefault('World')] const Name: string = 'World'): string;
```

For more than one parameter, just add one Default attribute for each default parameter:

```
procedure DoSomething(
    [XDefault('Default')] Name: string = 'Default';
    [XDefault(0)] Value: Integer = 0
  );
```

You only have to add attributes to the method declaration in the service contact interface. Attributes in methods in service implementation class will be ignored.

Service Implementation

Once you have defined the service interface, you need to write the server-side implementation of the service. To do this you must:

1. Create a class implementing the service interface.
2. Inherit the class from *TInterfacedObject*, or be sure the class implement automatic referencing counting for the interface.
3. Add `XData.Service.Common` unit to your unit uses clause.
4. Add *[ServiceImplementation]* attribute to your interface.
5. Implement in your class all methods declared in the interface.
6. Call *RegisterServiceType* method passing the implementation class (usually you can do that in initialization section of your unit).

When you create a TXDataServerModule to build your XData server, it will automatically find all classes that implement interfaces in the model and use them when operations are invoked by the client. When a client invokes the operation, the server will find the class and method implementing the requested service through routing, will create an instance of the class, bind input parameters, and invoke the method. If the method returns a value, the server will bind the return values and send it back to the client. The instance of the implementation class will then be destroyed.

The following source code illustrates how to implement the *MyService* interface defined in the topic "Defining Service Interface":

```
unit MyService;

interface

uses
  System.Classes, Generics.Collections,
  MyServiceInterface,
  XData.Service.Common;

type
  [ServiceImplementation]
  TMyService = class(TInterfacedObject, IMyService)
  private
    function Sum(A, B: double): double;
    function HelloWorld: string;
  end;

implementation

function TMyService.Sum(A, B: double): double;
begin
  Result := A + B;
end;

function TMyService.GetWorld: string;
begin
  Result := 'Hello, World';
end;

initialization
  RegisterServiceType(TMyService);

end.
```

Server Memory Management

When executing service operations, XData provides a nice automatic memory management mechanism that destroy all objects involved in the service operation execution. This makes it easier for you to implement your methods (since you mostly don't need to worry about object destruction) and avoid memory leaks, something that is critical at the server-side.

In general, all objects are automatically destroyed by XData at server-side. This means that when implementing the service, you don't need to worry to destroy any object. However, it's important to know how this happens, so you implement your code correctly, and also knowing that in

some specific situations of advanced operations, the previous statement might not be completely true. So strictly speaking, the objects automatically destroyed by XData are the following:

- Any object passed as parameter or returned in a function result (called param object or result object);
- Any object managed by the context TObjectManager;
- The context *TObjectManager* is also destroyed after the operation method returns;
- Any associated object that is also deserialized or serialized together with the param object or result object.

Example

Suppose the following meaningless service implementation:

```
function TMyService.DoSomething(Param: TMyParam): TMyResult;
var
  Entity: TMyEntity;
begin
  Entity := TMyEntity.Create;
  Entity.SomeProperty := Param.OtherProperty;
  TXDataOperationContext.Current.GetManager.Save(Entity);
  Result := TMyResult.Create('test');
end;
```

You have three objects involved here:

- *Param*, of type *TMyParam* and was created by XData;
- function *Result*, of type *TMyResult*, being created and returned in the last line of service implementation;
- *Entity*, of type *TMyEntity*, being created and then passed to context TObjectManager to be saved in the database.

You **don't need** to destroy any of those objects. Whatever objects are passed as param (*Param*) of function result are always destroyed by XData automatically. The context manager is automatically destroyed by XData, and since you have added an entity to it, it will be destroyed as well upon manager destruction (this is a TMS Aurelius feature, not XData).

Managed objects

To control what will be destroyed and avoid double destruction of same object instance (for example, if the same object is received as parameter and returned as function result), XData keeps a collection of object instances in a property named *ManagedObjects*, available in the *TXDataRequestHandler* object. You can access that property using the operation context:

```
TXDataOperationContext.Current.Handler.ManagedObjects
```

You can also add object to that collection in advance, to make sure the object you are creating is going to be destroyed eventually by XData. This can help you out in some cases. For example, instead of using a construction like this:

```
function TMyService.DoSomething: TMyResult;
begin
  Result := TMyResult.Create;
  try
    // do complex operation with Result object
  except
    Result.Free;
    raise;
  end;
end;
```

You could write it this way:

```
function TMyService.DoSomething: TMyResult;
begin
  Result := TMyResult.Create;
  // Make sure Result will be eventually destroyed no matter what
  TXDataOperationContext.Current.Handler.ManagedObjects.Add(Result);

  // now do complex operation with Result object
end;
```

Specific cases

As stated before, in general you just don't need to worry, since the above rules cover pretty much all of your service implementation, especially if you use the context TObjectManager to perform database operations instead of creating your own manager. The situations where something might go wrong are the following:

1. If you create your own *TObjectManager* and save/retrieve entities with it that were also passed as parameter or are being returned in function result.

This will cause an Access Violation because when you destroy your object manager, the entities managed by it will be destroyed. But then XData server will also try to destroy those objects because they were passed as parameter or are being returned. To avoid this, you must set your *TObjectManager.OwnsObjects* property to false, or use the context TObjectManager so XData knows that the objects are in the manager and don't need to be destroyed. The latter option is preferred because it's more straightforward and more integrated with XData behavior.

2. If you are creating associated objects that are not serialized/deserialized together with your object

For example, suppose your method returns an object *TCustomer*. This TCustomer class has a property *Foo: TFoo*, but such property is **not mapped** in Aurelius as an association (or not marked to be serialized if TCustomer is a PODO). You then create a TFoo object, assign it to Foo property and return:

```
Result := TCustomer.Create;
Result.Foo := TFoo.Create;
```

Since the Foo property will not be serialized by the server, XData will not know about such object, and it will not be destroyed. So you must be sure to destroy temporary/non-mapped/ non-serializable objects if you create them (or add them to *ManagedObjects* collection as explained above). This is a very rare situation but it's important that developers be aware of it.

There is a specific topic for client-side memory management using TXDataClient.

TXDataOperationContext

To help you implement your service operations, XData provides you some information under the context of the operation being executed. Such information is provided in a *TXDataOperationContext* object (declared in unit `XData.Server.Module`). The simplified declaration of such object is as followed.

```
TXDataOperationContext = class
public
  class function Current: TXDataOperationContext;
  function GetConnectionPool: IDBConnectionPool;
  function Connection: IDBConnection;
  function GetManager: TObjectManager;
  function CreateManager: TObjectManager; overload;
  function CreateManager(Connection: IDBConnection): TObjectManager; overload;
  function CreateManager(Connection: IDBConnection; Explorer: TMappingExplorer):
TObjectManager; overload;
  function CreateManager(Explorer: TMappingExplorer): TObjectManager; overload;
  procedure AddManager(AManager: TObjectManager);
  function Request: THttpServerRequest;
  function Response: THttpServerResponse;
end;
```

To retrieve the current instance of the context, use the class function *Current*.

```
Context := TXDataOperationContext.Current.GetManager;
```

Then you can use the context to retrieve useful info, like a ready-to-use, in-context *TObjectManager* to manipulate Aurelius entities (using the *GetManager* function) or the connection pool in case you want to acquire a database connection and use it directly (using *GetConnectionPool* method).

This way your service implementation doesn't need to worry about how to connect to the database or even to create a TObjectManager (if you use Aurelius) with proper configuration and destroy it at the end of process. All is being handled by XData and the context object. Another interesting thing is that when you use the TObjectManager provided by the context object, it makes it easier to manage memory (destruction) of entity objects manipulated by the server, since it can tell what objects will be destroyed automatically by the manager and what objects need to be destroyed manually. See Memory Management topic for more information.

The example is an example of an operation implementation that uses context manager to retrieve payments from the database.

```
uses
  {...}
  XData.Server.Module;

function TCustomerService.FindOverduePayments(CustomerId: integer): TList<TPaymen
t>;
begin
  Result := TXDataOperationContext.Current.GetManager.Find<TPayment>
    .CreateAlias('Customer', 'c')
    .Where(TLinq.Eq('c.Id', CustomerId) and TLinq.LowerThan('DueDate', Now))
    .List;
end;
```

Note that the database connection to be used will be the one provided when you created the TXDataServerModule. This allows you keep your business logic separated from database connection. You can even have several modules in your server pointing to different databases, using the same service implementations.

Inspecting request and customizing response

The context also provides you with the request and response objects provided by the TMS Sparkle framework.

You can, for example, set the content-type of a stream binary response:

```
function TMyService.GetPdfReport: TStream;
begin
  TXDataOperationContext.Current.Response.Headers.SetValue('content-type', 'appli
cation/pdf');
  Result := InternalGetMyPdfReport;
end;
```

Or you can check for a specific header in the request to build some custom authentication system:

```
function TMyService.GetAppointment(const Id: integer): TVetAppointment;
var
  AuthHeaderValue: string;
begin
  AuthHeaderValue := TXDataOperationContext.Current.Request.Headers.Get('custom-
auth');
  if not CheckAuthorized(AuthHeaderValue) then
    raise EXDataHttpException.Create(401, 'Unauthorized'); // unauthorized

  // Proceed to normal request processing.
  Result := TXDataOperationContext.Current.GetManager.Find<TVetAppointment>(Id);
end;
```

The default connection

The default database connection interface (IDBConnection) is provided in the property *Connection*:

```
DefConnection := TXDataOperationContext.Current.Connection;
```

It's the same connection used by the default manager, and it's retrieved from the pool when needed.

Additional managers

You might need to create more Aurelius managers in a service operation. You can of course do it manually, but if by any chance you want to return one entity inside one of those managers, you can't destroy the manager you've created because otherwise the object you want to return will also be destroyed.

In these scenario, you can use the context methods *CreateManager* or *AddManager*. The former will create a new instance of a TObjectManager for you, and you can use it just like the default manager: find objects, save, flush, and don't worry about releasing it.

```
function TSomeService.AnimalByName(const Name: string): TAnimal;
var
  Manager: TObjectManager;
begin
  Manager := TXDataOperationContext.Current.CreateManager(TMappingExplorer.Get('O
therModel'));
  Result := Manager.Find<TAnimal>
    .Where(TLinq.Eq('Name', Name)).UniqueResult;
end;
```

To create the manager you can optionally pass an IDBConnection (for the database connection), or the model (TMappingExplorer), or both, or none. It will use the default Connection if not specified, and it will use the model specified in the XData server (module) if model is not specified.

If you want to create the manager yourself, you can still tell XData to destroy it when the request processing is finished, by using AddManager.

```
function TSomeService.AnimalByName(const Name: string): TAnimal;
var
  Manager: TObjectManager;
begin
  Manager := TObjectManager.Create(SomeConnection, TMappingExplorer.Get('OtherMod
el'));

  TXDataOperationContext.Current.AddManager(Manager);
  Result := Manager.Find<TAnimal>
    .Where(TLinq.Eq('Name', Name)).UniqueResult;
end;
```

XData Query

XData offers full query syntax to query entities from automatic CRUD endpoints. But you can also benefit from XData query mechanism in service operations. You can receive query information sent from the client and create an Aurelius criteria from it to retrieve data from database.

Receiving query data

To allow a service operation to receive query data like the automatic CRUD endpoint, declare a parameter of type `TXDataQuery` (declared in unit `XData.Query`):

```
IMyService = interface(IInvokable)
  [HttpGet] function List(Query: TXDataQuery): TList<TCustomer>;
```

The class `TXDataQuery` is declared like this (partial):

```
TXDataQuery = class
strict private
  [JsonProperty('$filter')]
  FFilter: string;
  [JsonProperty('$orderby')]
  FOrderBy: string;
  [JsonProperty('$top')]
  FTop: Integer;
  [JsonProperty('$skip')]
  FSkip: Integer;
```

Which means clients can invoke the endpoint passing the same `$filter`, `$orderby`, `$top` and `$skip` parameters accepted also by the automatic CRUD endpoint, like this:

```
/MyService/List/?$filter=Name eq 'Foo'&$orderby=Name&$top=10&$skip=30
```

Or, of course, using the `TXDataClient` passing the raw string:

```
Customer := Client.Service<IMyService>
  .List('$filter=Name eq 'Foo'&$orderby=Name&$top=10&$skip=30');
```

Or using the XData query builder:

```
Customer := Client.Service<IMyService>.List(
  CreateQuery.From(TCustomer)
    .Filter(Linq['Name'] eq 'Foo')
    .OrderBy('Name')
    .Top(10).Skip(30)
    .QueryString
);
```

Creating a criteria

From your service operation, you can then use the received `TXDataQuery` object to create an Aurelius criteria, using the `CreateCriteria` method of the current `TXDataOperationContext`:

```
function TMyService.List(Query: TXDataQuery): TList<TCustomer>;
begin
  Result := TXDataOperationContext.Current
    .CreateCriteria<TCustomer>(Query).List;
end;
```

You can of course create the criteria and then modify it as you wish. Since you are implementing a service operation, instead of using an automatic CRUD endpoint, you have full control of the business logic you want to implement. You could even, for example, create a criteria based on a query on a DTO object:

```
function TMyService.List(Query: TXDataQuery): TList<TCustomer>;
begin
  Result := TXDataOperationContext.Current
    .CreateCriteria<TCustomer>(Query, TCustomerDTO).List;
end;
```

In the example above, even though you are creating a criteria for the `TCustomer` class, the query will be validated from the `TCustomerDTO` class. This means XData will only accept queries that use properties from `TCustomerDTO`. For example, if the filter string is `$filter=Status eq 2`, even if the `Status` property exists in `TCustomer`, if it does not exist in class `TCustomerDTO` the query will not be accepted and an error "property Status does not exist" will return to the client.

TMS Aurelius CRUD Endpoints

XData doesn't require TMS Aurelius to work. You declare your service operations and implement any custom logic, you want, including, of course, database operations. But if you use TMS Aurelius, XData offers a nice integration feature named *TMS Aurelius CRUD endpoints*.

This chapter assumes you have existing, basic knowledge about TMS Aurelius and your application already has some objects mapped to the database. Please refer to the TMS Aurelius documentation, "Quick Start" section for a brief introduction.

The following are the topics related to automatic TMS Aurelius CRUD Endpoints.

Overview of Aurelius CRUD Endpoints

If you have TMS Aurelius entities declared in your application, once your server is running, Aurelius entities are accessible through a REST/JSON architecture. The XData server defines URL conventions for the CRUD enpoints so you know how to build the URL to access the resources you need. The message payload in HTTP requests and responses must be in JSON format that also follows an XData specification. Then, XData clients can retrieve Aurelius entities by requesting data from the server or request data modification on Aurelius entities.

To illustrate this, suppose you have an Aurelius entity *TCustomer* associated with a *TCountry* entity, both declared in your application, you will have the following URL addresses published by TMS XData:

- *http://server:2001/tms/xdata/Customer* (lists all TCustomer entities)
- *http://server:2001/tms/xdata/Country* (lists all TCountry entities)
- *http://server:2001/tms/xdata/Customer(1)* (retrieves TCustomer entity with id = 1)
- *http://server:2001/tms/xdata/Customer(2)/Country* (retrieves the TCountry object associated to the customer with id = 2 via Country property)
- *http://server:2001/tms/xdata/Customer?$filter=Country/Name eq 'USA'&$orderby=Name&$top=10* (retrieves top 10 customers in the USA, ordered by name)

Furthermore, you can perform GET requests to retrieve entities, POST to create new entities, PUT/PATCH to update existing entities, and DELETE to remove entities.

You can also easily execute service operations from Delphi code through the service interfaces using a TXDataClient object instance:

```
uses {...},  MyServiceInterface, XData.Client;

var
  Client: TXDataClient;
  MyService: IMyService;
  Customer: TCustomer;

begin
  Client := TXDataClient.Create;
  Client.Uri := 'http://server:2001/tms/xdata';
  MyService := Client.Service<IMyService>;
  Customer := Client.CustomerByName('Joe');
  {...}
end;
```

Do not forget that you have to use those Aurelius entities anywhere in your server application otherwise they will be removed by the linker, at the very least, you should call *RegisterEntity(TCustomer)*, for example, to make sure TCustomer will not be removed.

Aurelius Equivalence to XData Model

When you create the XData server module, the CRUD endpoints are automatically created from the current Aurelius mapping, using a default entity model builder, and you don't need to define it yourself. This topic describes the rules used by the model builder to create the model based on Aurelius mapping.

Note that the XData server module creates the EM based on a *TMappingExplorer* object, which contains all the Aurelius mapping information.

Classes become entity types
Every mapped class in Aurelius becomes an entity type. By default, the name of entity type is the class name. So, if for example your mapping has classes TCustomer, TOrder and TCountry, each of those classes become an entity type in the EM.

Entity sets
XData creates an entity set for each entitytype, so that all entities can be easily accessible from the server. The entity set associated with the class contains all entities (objects) of that class. By default the name of the entity set is the name of the entity type associated to it. Thus, if you have a mapped class named "TCustomer", XData will define an entity type "Customer" in the EM, and will also define an entity set "Customer" that represents all entities of that type. You can prevent the "T" from being removed from class name by using the entity model builder.

Properties
Every mapped property or field in an Aurelius class becomes either a property or navigation property in the entity type. If the class property/field is an association, it becomes a navigation property. If it's a scalar property, it becomes a single property. It's important to note that only *mapped* class members become an entity type property. The name of the property in the EM will be the name of the mapped class property or the name of mapped class field with the "F" prefix removed. As an example, consider the following Aurelius mapping:

```
[Entity]
[Automapping]
TCustomer = class
strict private
  FId: Integer;
  FName: string;
  [Transient]
  FSex: Nullable<TSex>;
  [Column('BIRTHDAY', [])]
  FBirthday: Nullable<TDate>;
  FCountry: TTC_Country;
  [Column('Description', [])]
  FDescription: TBlob;
  [Transient]
  FPhoto: TBlob;
  [Transient]
  FPhoto2: TBlob;
public
  property Id: Integer read FId write FId;
  property Name: string read FName write FName;
  [Column('SEX', [])]
  property Sex: Nullable<TSex> read FSex write FSex;
  property Birthday: Nullable<TDate> read FBirthday write FBirthday;
  property Country: TCountry read FCountry write FCountry;
  property Description: TBlob read FDescription write FDescription;
  [Column('Photo', [TColumnProp.Lazy])]
  property Photo: TBlob read FPhoto write FPhoto;
  property Photo2: TBlob read FPhoto2 write FPhoto2;
end;
```

The above class will become an entity type named "TCustomer", with the following named properties: Id, Name, Birthday, Country, Description, Sex and Photo. Fields like FId and FName are considered because the class is marked with [Automapping] attribute, so all fields are included (with "F" prefix removed). Also, properties Sex and Photo are also included because they are explicitly mapped. Finally, fields FSex, FPhoto and FPhoto2 are not included because they were excluded from Aurelius mapping because of the [Transient] attribute usage. You can prevent the "F" from being removed of field names by using the entity model builder.

All properties will be created as simple properties, except "Country" which will be a navigation property associated with the "TCountry" that is also an entity type created from TCountry class.

Entity Sets Permissions

By default, all entity sets defined in the XData model will be published by the server with full permissions. This means the clients can list, retrieve, modify, delete, create entities using the entity sets (see Requesting Data and Data Modification for more information).

You can change those permissions individually by each entity set, by choosing which operations can be performed in each entity set.

This is accomplished using the *SetEntitySetPermissions* method of the TXDataServerModule object, declared as following:

```
procedure SetEntitySetPermissions(const EntitySetName: string;
  Permissions: TEntitySetPermissions);
```

The *EntitySetName* parameter specifies the name of the entity set (note that it's not necessarily the name of the class), or '*' for all entity sets and *Permissions* parameter defines the allowed operations in that entity set. Valid values are (declared in unit `XData.Module.Base`):

```
type
  TEntitySetPermission = (List, Get, Insert, Modify, Delete);
  TEntitySetPermissions = set of TEntitySetPermission;

const
  EntitySetPermissionsAll =
[Low(TEntitySetPermission)..High(TEntitySetPermission)];
  EntitySetPermissionsRead = [TEntitySetPermission.List,
TEntitySetPermission.Get];
  EntitySetPermissionsWrite = [TEntitySetPermission.Insert,
TEntitySetPermission.Modify,
    TEntitySetPermission.Delete];
```

So you can use constants *EntitySetPermissionsAll* (all operations), *EntitySetPermissionsRead* (read-only operations), *EntitySetPermissionsWrite* (write-only operations), or any combination of *TEntitySetPermission* enumerated value, which mean:

TEntitySetPermission values

Name	Description
TEntitySetPermission.List	Allows clients to query entities from the entity set using query criteria.
TEntitySetPermission.Get	Allows clients to retrieve a single entity and its subproperties and associated entities.
TEntitySetPermission.Insert	Allows clients to create a new entity in the entity set.
TEntitySetPermission.Modify	Allows clients to modify an entity, by either method (PUT, PATCH, etc.) or even modifying properties or collection properties.
TEntitySetPermission.Delete	Allows clients to delete an entity from the entity set.

Examples

Allows all operations in entity set Country, except deleting an entity:

```
Module.SetEntitySetPermissions('Country',
  EntitySetPermissionsAll - [TEntitySetPermission.Delete]);
```

Allows read-only operatiosn in City entity set:

```
Module.SetEntitySetPermissions('City', EntitySetPermissionsRead);
```

For the History entity set, allows only querying entities, retrieve a single entity, or create a new one:

```
Module.SetEntitySetPermissions('History',
  [TEntitySetPermission.List, TEntitySetPermission.Get, TEntitySetPermission.Inse
rt]);
```

Allows all entities to be read-only by default (list and get only). You can override this setting by setting permissions for a specific entity set:

```
Module.SetEntitySetPermissions('*', EntitySetPermissionsRead);
```

URL Conventions

This chapter describes what URL addresses are made available by the XData Server based on the entity model. It explains the rules for constructing URL address that identify data and metadata exposed by the server.

When you create the XData server, the root URL must be specified. Following the root URL part, additional URL constructions might be added, like path segments, query or fragment parts. In this manual, all examples consider the server root URL to be *http://server:2001/tms/xdata/*.

The URL used by the XData server for entity resources has three significant parts: root URL, resource path and query options, according to the example below.

```
http://server:2001/tms/xdata/Customer?$top=2&$orderby=Name
\_____________________________/\______/ \__________________/
              |                  |               |
          Root URL       resource path   query options
```

For service operations, the URL uses the following format:

```
http://server:2001/tms/xdata/MathService/Multiply
\_____________________________/\__________/\______/
              |                    |          |
          Root URL             service   operation
```

The following topics describe in details the parts of the URL as understood by the XData Server.

Resource Path

This topic defines the rules for resource path construction according to XData URL conventions. Resources exposed by the XData server are addressable by corresponding resource path URL components to enable interaction of the client with that resource aspect. To illustrate the concept, some examples for resources might be: customers, a single customer, orders related to a single customer, and so forth. Examples of addressable aspects of these resources as exposed by the data model might be: collections of entities, a single entity, properties, and so on.

The following topics explain several different ways to access data through an URL.

Addressing Entity Sets

An entity set can be addressed by using the name of entity set after the server root. As a general rule, the name of the entity set is the name of the base entity type of all entities belonging to that entity set (see Aurelius Equivalence to Entity Model for more information). So, for example, the entity set "Customer" will contain all entities of type "Customer" and its descendants. The collection of all Customer entities, for example, can be accessed with the following URL:

```
http://server:2001/tms/xdata/Customer
```

A query part can be appended to the URL address of an entity set, so entities in an entity set can be filtered, ordered, etc., for example:

```
http://server:2001/tms/xdata/Customer?$order=Name
```

There is a specific chapter about all the query options you can use in the query part of the URL.

XData uses JSON Format to represent the entity set resources available in such addresses. To know more about the format specification, please refer to the topic that explains JSON format representation of a collection of entities.

You can suffix the address with *$count* segment to return the number of entities:

```
http://server:2001/tms/xdata/Customer/$count
```

By default the name of entity set is the name of the associated Aurelius class name without the leading "T". Thus, if the class name is *TCustomer*, the entity set path will be "Customer". You can change that name explicitly by adding *URIPath* attribute to the class:

```
uses {...}, XData.Service.Common;

type
  [URIPath('Customers')]
  TCustomer = class
```

When the attribute is applied, the entity set for TCustomer entities will be accessible in the following address (considering server base url is *server:2001/tms/xdata*):

```
http://server:2001/tms/xdata/Customers
```

Addressing Single Entity

A single entity in an entity set can be addressed by passing the entity key between parenthesis after the entity set name. The following example represents a specified *Customer* object with id (key) equals to 3.

```
http://server:2001/tms/xdata/Customer(3)
```

The literals in key predicates can also represent other types than integer, such as strings or guids. The following example retrieves the employee with id equals to "XYZ".

```
http://server:2001/tms/xdata/Employee('XYZ')
```

If you have entity types with compound keys (not recommended, actually), you can access it by naming each value and separate them using commas. The order is not significant. The next example represents the URL address of a *Person* entity identified by a compounded key made up of two properties, *LastName* and *FirstName*, which values are "Doe" and "John", respectively.

```
http://server:2001/tms/xdata/Person('Doe','John')
http://server:2001/tms/xdata/Person(LastName='Doe',FirstName='John')
```

XData uses JSON Format to represent the entity resources available in such URL addresses. To know more about the format specification, please refer to the topic that explains JSON format representation of a single entity.

Key as segments

If property *TXDataServerModule.EnableKeyAsSegment* is True (same property is available in TXDataServer), single entities can also be addresses by the name of the entity followed by a slash and the id, as the following examples:

```
http://server:2001/tms/xdata/Customer/3
http://server:2001/tms/xdata/Employee/XYZ
```

Compound keys can be passed one after order, each also separated by slashes:

```
http://server:2001/tms/xdata/Person/Doe/John
```

Addressing Navigation Properties

From a single entitiy, you can access associated entities through navigation properties. For example, if a customer type has a country associated to it, you can access it using the name of navigation property. The following example retrieves the country object associated with customer 3:

```
http://server:2001/tms/xdata/Customer(3)/Country
```

You can only have a maximum of one level of nested navigation property. The next address, for example, trying to request the country associated with a customer that is associated to an order, is **not valid**. To request such country, you should use the address in previous example (*Customer()/Country*) or access the country directly using *Country(<key>)*.

```
http://server:2001/tms/xdata/Order(41)/Customer/Country (invalid address)
```

Note that it's also possible to retrieve collection of entities if the navigation property is a collection. For example, the following URL addresses the collection of order items for a specific order:

```
http://server:2001/tms/xdata/Order(41)/OrderItems
```

For such navigation properties that represents collections (many-valued associations), it's not possible to use key predicates (to identify a single entity in the collection), or query options (to filter or order the collection).

XData uses JSON Format to represent the entity resources available in such URL addresses. To know more about the format specification, please refer to the topic that explains JSON format representation of a single entity.

Addressing Individual Properties

It's also possible to directly access a simple property. In the example below, the URL represents the *Name* property of customer 3. Note there is a specific behavior for streams (blob) properties.

```
http://server:2001/tms/xdata/Customer(3)/Name
```

When accessing a simple property resource, it contains a JSON representation of the property value. You can also alternatively access the raw value of property using *$value* suffix:

```
http://server:2001/tms/xdata/Customer(3)/Name/$value
```

The difference between the two address is that the latter (*$value*) contains the property value in raw format (plain text), while the former, contains the property value wrapped in JSON format.

The only exception is a property address that represents a named stream (blob). In this case, the resource will contain the raw value of the blob, without needing to add the "$value" suffix. As an example, the following URL address contains the raw binary data representing the photo of a customer:

```
http://server:2001/tms/xdata/Customer(3)/Photo
```

XData uses JSON Format to represent the individual property resources available in such URL addresses. To know more about the format specification, please refer to the topic that explains JSON format representation of an individual property.

Addressing Streams (Blobs)

You can access a stream (blob) property directly using the property name. In the example below, the URL represents the *Photo* property of customer 3.

```
http://server:2001/tms/xdata/Customer(3)/Photo
```

The stream property resource contains the raw (binary) value of the blob content. Content-type is not specified. Unlike addressing individual properties, stream properties do not support the *$value* suffix, since the stream property resource already provides the raw value of the property.

Counting Entities

You can append "$count" path segment to a url that retrieves an entity set or a navigation property that returns a list of entities. That will retrieve the number of entities in that resource, in text/plain format, instead of the entities themselves.

For example, the following url will return the total number of *Customer* entities:

```
http://server:2001/tms/xdata/Customer/$count
```

While the following url will return the number of Customer entities with name is equal to "John":

```
http://server:2001/tms/xdata/Customer/$count?$filter=Name eq 'John'
```

You can also use *$count* with associated entities:

```
http://server:2001/tms/xdata/Order(41)/OrderItems/$count
```

Model Metadata

Any XData module provides the *$model* URL that returns the metadata for the API model. It lists the entities, service operations, parameters, etc. For now the model metadata is used by XData tools (like clients) and is subject to modification.

```
http://server:2001/tms/xdata/$model
```

Query Options

When addressing resources, it's possible to include a query string in the URL to perform extra operations on the resource being addressed, like filtering or ordering entities. Query options start with *?* character and can be provided in the format *name=value*, separated by the *&* character. Many query options can be applied in a single URL. For example, the URL below retrieves the first 10 customers which country is USA, ordered by name:

```
http://server:2001/tms/xdata/Customer?$filter=Country/Name eq
'USA'&$orderby=Name&$top=10
```

The following table lists all the query options that can be used in a query string:

Query option	Description
$filter	Allows filtering the entities in an entity set by a specified condition.
$orderby	Specifies the order of retrieved entities, by one or more properties or expression.
$top	Specifies the maximum number of entities to be returned by the server.
$skip	Specifies the number of entities that the server should skip before returning the requested entities.
$inlinecount	Includes the total number of entities (without paging) in the JSON response.
$expand	Expands associated objects in by forcing the server to include them inline in JSON response.

$filter

The *$filter* query option can be used in a URI query string to specify a predicate by which entities will be filtered from an entity set. The $filter option only applies when addressing entity sets. The filter format is the following:

> *$filter=<boolean expression>*

where <boolean expression> is a boolean expression with the predicate. Example:

```
http://server:2001/tms/xdata/Customer?$filter=Name eq 'John'
```

Properties can be accessed through their name, as in the previous example. The properties available are the ones from the entity type of the specified entity set. In the previous example, the entity set being retrieved contains instances of entity type *Customer*, which means properties of Customer are directly accessible in the boolean expression.

To access a property of an associated entity (navigation property), you can use slashes to build the path to the property name. The following example retrieves all customers which country is USA:

```
http://server:2001/tms/xdata/Customer?$filter=Country/Name eq 'USA'
```

You can use logical operators and parenthesis to define multiple expressions and change evaluation order:

```
http://server:2001/tms/xdata/Customer?$filter=(Name eq 'John' or Name eq 'Jack') and Country/Name eq 'USA'
```

The previous examples use string literals in the expressions. String literals are always defined using single quotes. You can have literals of other types in the expression. Since the filter query option is part of URL, you must always format the literals according to the rules defined in Literals in URI section.

The following table lists the operators supported in filter expressions, and the order of precedence from highest to lowest. Operators in the same category have equal precedence.

Category	**Expression**	**Description**
Grouping	(x)	Enclosing parenthesis - expressions are evaluated with higher precedence
Primary	Name/ SubName	Slash, allowing access to a sub property (navigation property)
Unary	-x	Negate x
Unary	not x	Logical not applied to x
Multiplicative	x mul y	Multiplies x by y
Multiplicative	x div y	Divides x by y
Additive	x add y	Adds x and y
Additive	x sub y	Subtracts y from x
Relational	x lt y	Evaluates true if x is lower than y
Relational	x gt y	Evaluates true is x is greater than y
Relational	x le y	Evaluates true if x is lower than or equal to y
Relational	x ge y	Evaluates true if x is greater than or equal to y
Equality	x eq y	Evaluates true if x is equal to y
Equality	x ne y	Evaluates true is x is not equal to y

Category	Expression	Description
AND Condition	x and y	Evaluates true is both x and y are true
OR Condition	x or y	Evaluates true is either x or y are true

$orderby

The *$orderby* query option can be used in a URL query string to determine which values are used to order the entities in the entity set. The $orderby option only applies for URI addressing entity sets. The format is the following:

> *$orderby=<expression> [asc/desc]*
> *$orderby=<expression> [asc/desc],<expression> [asc/desc],...,<expression> [asc/desc]*

Where <expression> must contain the value by which the entities must be order, which is mostly the name of a property of sub property. Example:

```
http://server:2001/tms/xdata/Customer?$orderby=Name
```

The asc/desc identifier is optional and indicates if the entities must be sorted by the specified expression values in ascending or descending order. If not specified, ascending order is used. You can use slashes to access associations (navigation properties) and sort by values of such associations. As an example, the following URI will retrieve all invoices sorted by the name of the customer's country associated with that invoice, in descending order:

```
http://server:2001/tms/xdata/Invoice?$orderby=Customer/Country/Name desc
```

You can also order by multiple values, separated by comma. The following example lists all customers by last name and for customers with the same last name, by first name:

```
http://server:2001/tms/xdata/Customer?$orderby=LastName,FirstName
```

$top

The *$top* query option can be used in a URI query string to specify the maximum number of entities that will be returned by the server. The $top option only applies for URI addressing entity sets. The format is the following:

> *$top=<integer>*

The returned entities are always the first entities in the result set. Usually $top is used together with $orderby query option so that the order by which the entities are returned are known. It is also often used together with $skip query option to perform paging of results. In the following example, the server will return the first 10 invoice entities ordered by Date, even if there are more than 10 invoices in the server:

```
http://server:2001/tms/xdata/Order?$orderby=Date&$top=10
```

$skip

The *$skip* query option can be used in a URI query string to specify the number of entities that will be skipped by the server before returning the results. The $skip option only applies for URI addressing entity sets. The format is the following:

> *$skip=<integer>*

The skipped entities are always the first entities in the result set. Usually $skip is used together with $orderby query option so that the order by which the entities are returned are known. It is also often used together with $top query option to perform paging of results. In the following URI example, the server will return invoice entities ordered by Date in descending order, buy skipping the 10 first invoices and starting in the 11th one:

```
http://server:2001/tms/xdata/Order?$orderby=Date desc&$skip=10
```

$inlinecount

The *$inlinecount* query option allows including the total number of entities in the JSON response representing a collection of objects. The counting of entities ignores paging query options like $top and $skip. It's useful when paging results, so you can get a single page but also retrieve the total number of entities so the client can know in a single request how many entities are in total even when requesting only a page.

The $inline option only applies for URI addressing entity sets. The format is the following:

> *$inlinecount=none|allpages*

When using "none" (the default), no info is included. When using "allpages", the total number of entities for all pages will be included in the JSON response, in a property named *$xdata.count*.

In the following example, the server will return the first 10 invoice entities ordered by Date, and ask for the total number of entities available.

```
http://server:2001/tms/xdata/Order?$orderby=Date&$top=10&$inlinecount=allpages
```

The JSON result would be something like this (supposing the total number of entities is 142):

```
{
  "@xdata.count": 142,
  "value": [
    <_list_of_Order_objects_here_>
  ]
}
```

$expand

The *$expand* query option allows fine-grain control about how associations will be represented in JSON response sent by the server or how blobs will be represented.

Expanding associations

By default, the server will represent any associated object in a JSON response using the reference convention. For example, suppose a request for a *Customer* object:

`http://server:2001/tms/xdata/Customer(3)`

will return a JSON response like this:

```
{
  "$id": 1,
  "@xdata.type": "XData.Default.Customer",
  "Id": 3,
  "Name": "Bill",
  "Country@xdata.ref": "Country(10)"
}
```

Note the associated Country is represented as a reference (Country with Id = 10). This is very handy when using Delphi client (TXDataClient) or even non-Delphi client where you just want to modify the country reference without needing to retrieve the whole object.

However, in some cases you might want to have the full Country object inline. With the $expand query option you can explicitly indicate which associated properties you want to be inline:

`http://server:2001/tms/xdata/Customer(3)?$expand=Country`

Response will be:

```
{
  "$id": 1,
  "@xdata.type": "XData.Default.Customer",
  "Id": 3,
  "Name": "Bill",
  "Country": {
    "$id": 2,
    "@xdata.type": "XData.Default.Country",
    "Id": 10,
    "Name": "Germany",
  }
}
```

$expand applies to both single association and lists. It also forces any server-side proxy to be loaded, so even if Country is lazy-loaded, $expand will retrieve the entity inline. XData optimizes this retrieval process by forcing the proxy to be loaded eagerly in a single SQL statement (except for lists).

Expanding blob properties

You can also use $expand to ask for the server to bring blob content inline in object representation. By default, blob properties will be represented as a proxy reference:

```
"Data@xdata.proxy": "Customer(55)/Data"
```

If you want the blob content to be retrieved together with the object, you can ask to $expand the property:

```
http://server:2001/tms/xdata/Invoice(3)?$expand=Data
```

Then the blob content will be inline as a base64 string:

```
"Data": "T0RhdGE"
```

Expanding multiple properties

You can expand multiple properties by separating them using comma. The following example expands properties *Customer*, *Seller* and *Products*:

```
http://server:2001/tms/xdata/Invoice(10)?$expand=Customer,Seller,Products
```

You can also expand subproperties (deeper levels) by using slash to specify subproperties. The following example expands property Customer and also the *Country* property of such customer (and Seller) property:

```
http://server:2001/tms/xdata/Invoice(10)?$expand=Customer/Country,Seller
```

Built-in Functions

In addition to operators, a set of functions is also defined for use with the $filter or $orderby system query options. The following sections describe the available functions.

Upper

The *upper* function returns the input parameter string value with all characters converted to uppercase.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=upper(Name) eq 'PAUL'
```

Lower

The *lower* function returns the input parameter string value with all characters converted to lowercase.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=lower(Name) eq 'paul'
```

Length

The *length* function returns the number of characters in the parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=length(Name) eq 15
```

Substring

The *substring* function returns a substring of the first parameter string value identified by selecting *M* characters starting at the *Nth* character (where *N* is the second parameter integer value and *M* is the third parameter integer value). N parameter is 1-based, thus N = 1 means the first character.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=substring(CompanyName, 2, 4) eq 'oogl'
```

Position

The *position* function returns the 1-based character position of the first occurrence of the first parameter value in the second parameter value. If the string in first parameter is not found in the string in second parameter, it returns zero.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=position('jr', Name) gt 0
```

Concat

The *concat* function returns the string concatenation of the two string input parameters passed to it.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=concat(concat(Name, ' '), LastName) eq
'PAUL SMITH'
```

Contains

The *contains* function returns True if the first string parameter contains the string specified by the second string parameter. False otherwise.

The following example will return all customers which Name contains the string "Walker":

```
http://server:2001/tms/xdata/Customer?$filter=contains(Name, 'Walker')
```

StartsWith

The *startswith* function returns True if the first string parameter starts with the string specified by the second string parameter. False otherwise.

The following example will return all customers which Name starts with the string "Paul":

```
http://server:2001/tms/xdata/Customer?$filter=startswith(Name, 'Paul')
```

EndsWith

The *endswith* function returns True if the first string parameter ends with the string specified by the second string parameter. False otherwise.

The following example will return all customers which Name ends with the string "Smith":

```
http://server:2001/tms/xdata/Customer?$filter=endswith(Name, 'Smith')
```

Year

The *year* function returns the year component of a single date/time parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=year(BirthDate) eq 1971
```

Month

The *month* function returns the month component of a single date/time parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=month(BirthDate) eq 12
```

Day

The *day* function returns the day component of a single date/time parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=day(BirthDate) eq 31
```

Hour

The *hour* function returns the hour component of a single date/time parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=hour(BirthDate) eq 14
```

Minute

The *minute* function returns the minute component of a single date/time parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=minute(BirthDate) eq 45
```

Second

The *second* function returns the second component of a single date/time parameter value.

Example:

```
http://server:2001/tms/xdata/Customer?$filter=second(BirthDate) eq 30
```

Literals in URI

When building an URI to perform a request to the server, you sometimes need to include literal values in it. You might need to use literals, for example, when addresing a single entity, where you need to pass the key value in the URI. Or when you use query options like $filter, where you usually include boolean expressions that compare property values to literal values. The following table indicates how to build literal values of several different data types.

Data Type	Examples	Description
null	null	Null value
Boolean	true false	Boolean values
DateTime	2013-12-25 2013-12-25T12:12 2013-12-25T12:12:20.125	Date time value must be in the ISO 8601 format (YYYY-MM-DDTdd:mm:ss.zzz). Parts of time can be omitted, for example, you can omit the milliseconds part (zzz) and seconds part (ss), or the full time and provide only the date part.
Float	3.14 1.2e-5	Float number values. Exponential format is also supported.
Guid	E314E4B3-ECE5-4BD5-9D41-65B7E74F7CC8	Guid value must not have enclosing brackets, must have the hyphens separating the five guid blocks (8 char, 4 char, 4 char, 4 char, 12 char). Each guid block is composed by hexadecimal digits.
Integer	1234	Integer values
String	'John'	Strings must be enclosed in single quotes.
Enumerated	csActive TCustomerStatus.csActive	Values of enumerated types can be directly referred by their name, without any quotes. To avoid ambiguity, you can prefix the enumeration name with the type name and a dot.

Custom Functions

Besides the built-in functions you can use in $filter and $orderby query options, you can also register your own custom functions. Such functions will then be translated into Aurelius' LINQ "SQL functions" that also need to be previously registered for the query to work.

For example, to register a function named "unaccent":

```
uses {...}, XData.Query.Parser;

TQueryParser.AddMethod('unaccent', TQueryMethod.Create('unaccent', 1));
```

The numeric parameter (1) indicates the number of parameters the function receives.

Then to use the function from query API:

```
http://server:2001/tms/xdata/Customer?$filter=unaccent(Name) eq 'Andre'
```

Requesting Data

XData server support requests for data via HTTP GET requests. You can follow URL conventions to determine the correct URL to be requested according to data needed.

The resource path of the URL specifies the target of the request (for example, collection of entities, entity, associated entity (navigation property), etc.). Additional query operators, such as filter, sort, page are specified through query options. The format of returned data depends on the data being requested but usually follows the specified XData JSON Format.

The following topics describe the types of data requests defined by XData and related information.

Querying Collections

XData server supports querying collections of entities. The target collection is specified through a URL that addresses an entity set (or an URL that addresses navigation properties representing a collection of items, like items of an order for example), and query operations such as filter, sort, paging are specified as system query options provided as query options. The names of all system query options are prefixed with a dollar ($) character.

For example, to query all the existing *Country* objects in the server, you perform a GET request to the respective entity set URL:

```
GET http://server:2001/tms/xdata/Country HTTP/1.1
```

You can add query options to the URL to filter the collection of entities. The following GET request returns the first 10 customers which country name is equals to "USA", ordered by customer name:

```
GET http://server:2001/tms/xdata/Customer?$filter=Country/Name eq
'USA'&$orderby=Name&$top=10 HTTP/1.1
```

The following example retrieves the items of order with id equals to 10.

```
GET http://server:2001/tms/xdata/Order(10)/Items HTTP/1.1
```

NOTE

You can only use query options on entity sets. When addressing navigation properties that represent a collection (like the previous example), query options are **not** available.

Requesting Associated Entities

To request associated entities according to a particular relationship, the client issues a GET request to the source entity's request URL, followed by a forward slash and the name of the navigation property representing the relationship, according to URL conventions for addressing associated entities.

If the navigation property does not exist on the entity indicated by the request URL, the service returns *404 Not Found*.

If the association terminates on a collection (many-valued association), then behavior is similar as described in topic "Querying Collections", without the exception that query options (filtering, ordering, etc.) cannot be applied.

If the association terminates on a single entity (single-valued association), then behavior is the same as described in topic "Requesting Single Entities". If no entity is related, the service returns *204 No Content*.

Examples

Retrieving the customer associated with order 10 (single entity):

```
GET http://server:2001/tms/xdata/Order(10)/Customer HTTP/1.1
```

Retrieving items associated with order 10 (collection of entities):

```
GET http://server:2001/tms/xdata/Order(10)/Items HTTP/1.1
```

Requesting Individual Properties

To retrieve an individual property, the client issues a GET request to the property URL. The property URL is the entity read URL with "/" and the property name appended, according to URL conventions for addressing simple properties. For example:

```
GET http://server:2001/tms/xdata/Product(1)/Name HTTP/1.1
```

The result format following the JSON format specification for individual properties, i.e., a JSON object with a name/value pair which name is "value" and the value contains the actual property value. The result for the above request might be, for example:

```
{
  "value": "Silver Hammer XYZ"
}
```

If the property has the null value, the service responds with *204 No Content*.

If the property doesn't exist the service responds with *404 Not Found*.

You can alternatively request the raw value of the property. To retrieve the raw value of a primitive type property, the client sends a GET request to the property value URL, which is the property value URL suffixed with "$value":

```
GET http://server:2001/tms/xdata/Product(1)/Name/$value HTTP/1.1
```

The Content-Type of the response is text/plain and the response content is the property value in plain text, formatted according to the JSON format specification for property values. The response content of the above request might something like the following:

```
Silver Hammer XYZ
```

A *$value* request for a property that is null results in a *204 No Content* response.

If the property doesn't exist the service responds with *404 Not Found*.

Requesting Streams (Blobs)

To retrieve the content of a stream (blob) property, the client issues a GET request to a URL that addresses a stream (blob) resource. For example:

```
GET http://server:2001/tms/xdata/Customer(1)/Photo HTTP/1.1
```

The server will provide the binary blob content in response body, without specifying the content type. It's up to your client application to determine the type of content and process it accordingly. Note that this behavior is different from when you request individual properties, which in this case provide a JSON representation of the property value.

If the property has the null value, or the stream content is empty, the service responds with *204 No Content*, otherwise a sucessful *200 OK* response is provided.

If the property doesn't exist the service responds with *404 Not Found*.

HTTP Request Headers

XData defines semantics around the following HTTP request and response headers. Additional headers may be specified, but have no unique semantics defined in XData.

The following are the available request headers.

xdata-expand-level

Clients can optionally include this header in the request to define the maximum depth for which the associated entities will be expanded (serialized inline) in an entity JSON representation.

```
xdata-expand-level: 3
```

An associated entity (navigation property) can be represented as an association reference, or inline object.

If *XData-ExpandLevel* is notpresent, the value 0 (zero) is assumed, which means all direct associated entities will be represented as references. When the header is present and value is higher, then all associated entities will be serialized inline, until the depth level specified by the

header. The higher the expand level, the bigger the response payload will be, since more objects will be serialized. But if client will need those objects anyway, this will minimize the need of further server requests to retrieve associated objects. Proxies are not affected by this.

xdata-put-mode

Overrides the value of TXDataServerModule.PutMode property for the request. Please refer to the property documentation for more info. Example:

```
xdata-put-mode: update
```

Valid values are "update" and "merge".

xdata-serialize-instance-ref

Overrides the value of TXDataServerModule.SerializeInstanceRef property for the request. Please refer to the property documentation for more info. Example:

```
xdata-serialize-instance-ref: ifrecursive
```

Valid values are "always" and "ifrecursive".

xdata-serialize-instance-type

Overrides the value of TXDataServerModule.SerializeInstanceType property for the request. Please refer to the property documentation for more info. Example:

```
xdata-serialize-instance-type: ifneeded
```

Valid values are "always" and "ifneeded".

Requesting Single Entities

To retrieve a single entity, the client makes a GET request to the read URL of an entity.

The read URL can be obtained by following the URL conventions, either that addresses a single entity, or associated entities. The read URL can be also obtained from a response payload in JSON format containing information about associated entities (through navigation properties of the entity represented in the payload).

If no entity exists with the key values specified in the request URL, the service responds with *404 Not Found*.

Examples

Retrieving a customer with id equals to 3:

```
GET /tms/xdata/Customer(3) HTTP/1.1
Host: server:2001
```

Retrieving the customer associated with order 10:

```
GET /tms/xdata/Order(10)/Customer HTTP/1.1
Host: server:2001
```

In both examples above, the response content might be:

```
{
  "$id": 1,
  "@xdata.type": "XData.Default.Customer",
  "Id": 55,
  "Name": "Joseph",
  "Birthday": "1980-05-20",
  "Sex": "tsMale",
  "Picture": null
}
```

Data Modification

XData Server supports *Create*, *Update*, and *Delete* operations for some or all exposed resources (entities, individual properties, etc.). You can follow URL conventions to determine the correct URL address of the resource path in order to perform the data modification operation.

Since XData Server follows REST/JSON architecture, such data modification operations are performed by sending HTTP requests to the server using POST, PUT, PATCH or DELETE methods, and when applicable, providing data in message payload using JSON format.

The following topics explains the several different methods of performing data modification in a XData Server.

Create an Entity

To create an entity in a collection, clients must perform a POST request to that collection's URL, which can be either the entity set URL address or a navigation property URL address that represents a collection. The POST body must contain a single valid JSON representation of the entity. The entity representation can be full (all possible properties) or partial (a subset of all available properties for the entity type).

Properties in entity representation must be valid properties or navigation properties of the entity type as specified in the entity model. Additional property values beyond those specified in the entity type should not be sent in the request body. The request will fail if unable to persist all property values specified in the request. Missing properties will be set to their default value. If a missing property is required and does not have a default value in the database server, the request will fail.

The following example creates a new *Country* entity in the server.

```
POST /tms/xdata/Country HTTP/1.1
Host: server:2001

{
  "@xdata.type": "XData.Default.Country",
  "Name": "Brazil"
}
```

When creating an entity, the "xdata.type" annotation is not required. If present, it indicates the type of the entity which should be created by the server. It can be useful when creating derived entities, for example, even though you POST to an URL representing an entity set of base types (for example, *Animal*), the entity being created might be a type inherited from the base type.

In the following example the request will create a *Cat* entity in entity set *Animal*. It's a valid request. If the provided xdata.type annotation indicates a type that is different or not inherited from the type of entity set being POST'ed, the request will fail.

```
POST /tms/xdata/Animal HTTP/1.1
Host: server:2001

{
  "@xdata.type": "XData.Default.Cat",
  "Name": "Wilbur",
  "CatBreed": "Persian"
}
```

If "xdata.type" is not present, the created entity will have the same type as the base entity type of the collection addressed by the URL. For example, if client POST to the url "Cat" without "xdata.type" annotation present in payload, an entity of type *Cat* wil be created.

The entity representation in request payload must not contain associated entities as inline content. If values for navigation properties should be specified, then they must be represented as an association reference. For single-valued (single entity) navigation properties, this will update the relationship (set the value of property to reference the associated object represented in the association reference) after entity is created.

The following example creates a new *City* entity which is associated with a *Country* object of id 2.

```
POST /tms/xdata/City HTTP/1.1
Host: server:2001

{
  "$id": 1,
  "Name": "Frankfurt",
  "Country@xdata.ref": "Country(2)"
}
```

Proxy values can be present in the entity payload (for example, when you are sending back a modified JSON representation previously retrieved from the server, it's possible that proxy values are present), but they are ignored by the server and navigation properties with proxy values that were not modified.

When perform POST request to a navigation property URL that represents a collection property, for example, *Order(1)/Items* which represent items of a specific order, the newly created entity will be automatically associated with (added to) that collection.

On successful update, the server returns with a *201 Created success* response and the JSON representation of the updated entity in the message payload. The response will also contain a *Location* header that contains the URL of the created entity. Note that the returned entity might be different from the one sent by the client, since some properties might have been updated automatically by XData Server and/or the database server.

Update an Entity

To update an entity, clients must perform a PATCH or PUT request to a single entity URL address, providing the JSON representation of the entity in message payload of the request. The entity representation can be full (all possible properties) or partial (a subset of all available properties for the entity type).

PATCH method should be preferred for updating an entity as it provides more resiliency between clients and services by directly modifying only those values specified by the client. The semantics of PATCH, as defined in [RFC5789], are to merge the content in the request payload with the entity's current state, applying the update only to those components specified in the request body. Thus, properties provided in the payload will replace the value of the corresponding property in the entity or complex type. Missing properties of the containing entity will not be altered.

In the following example, only properties *Name* and *DogBreed* of resource *Dog(1)* will be updated in the server.

```
PATCH /tms/xdata/Dog(1) HTTP/1.1
Host: server:2001

{
  "@xdata.type": "XData.Default.Dog",
  "Id": 1,
  "Name": "Willy",
  "DogBreed": "Yorkshire"
}
```

PUT method can also be used, but you should be aware of the potential for data-loss for missing properties in the message payload. When using PUT, all property values will be modified with those specified in the request body. Missing properties will be set to their default values.

In the following example, properties *Name* and *DogBreed* will be updated, and any other existing property of *Dog* will be set to its default value (strings will be blank, associations will be nil, etc.). If one of missing properties is required and not present, the server might raise an error indicating a required property should have a value.

```
PUT /tms/xdata/Dog(1) HTTP/1.1
Host: server:2001

{
  "@xdata.type": "XData.Default.Dog",
  "Id": 1,
  "Name": "Willy",
  "DogBreed": "Yorkshire"
}
```

Key and other non-updatable properties can be omitted from the request. If the request contains a value for one of these properties the server will ignore that value when applying the update. That's the reason why *Id* property is not updated in the above examples.

The entity must not contain associated entities as inline content. If values for navigation properties should be specified, then they must be represented as an association reference. For single-valued (single entity) navigation properties, this will update the relationship (set the value of property to reference the associated object represented in the association reference).

The following example updates the *Country* association of a *City* object to the country with id equals to 2.

```
PATCH /tms/xdata/City(1) HTTP/1.1
Host: server:2001

{
  "$id": 1,
  "@xdata.type": "XData.Default.City",
  "Country@xdata.ref": "Country(2)"
}
```

Proxy values can be present in the entity payload (for example, when you are sending back a JSON representation previously retrieved from the server, it's possible that proxy values are present), but they are ignored by the server and navigation properties with proxy values are not modified.

For update operations (PUT and PATCH), the "xdata.type" annotation is not required. Since you are updating an existing server resource, the type of resource is already known. Actually, including "xdata.type" annotation will force the server to perform a type check, if the type declared in the request doesn't match the type of server resource, an error occurs. So if you include xdata.type annotation in payload, be sure it matches the type of server resource you are trying to update.

On successful update, the server returns with a success response and the JSON representation of the updated entity in the message payload. Note that the returned entity might be different from the one sent by the client, since some properties might have been updated automatically by XData Server and/or the database server.

Delete an Entity

To delete an entity, clients must perform a DELETE request to a single entity URL address. The request body must be empty. The entity represented by that address will be removed from server.

The following example will delete the entity resource at address */tms/xdata/Dog(1)* - which is a *Dog* entity with id equals to 1:

```
DELETE /tms/xdata/Dog(1) HTTP/1.1
Host: server:2001
```

Note that depending on cascades configured in the server, associated entities might also be removed. If the entity being removed is associated by another entity that can't be removed in a cascade operation, a constraint enforcement will fail and the server will return an error without actually removing the entity.

On successful delete, the server responds with a *204 No Content* response and no content in message payload.

Managing Streams (Blobs)

To update the content of a stream (blob) property, clients must perform a PATCH or PUT request to a stream property URL address, providing stream content in message body. You don't need to specify content-type and if you do, it will be ignored. Using either PATCH or PUT method results in same server behavior.

The request is an example of how to update the content of *Photo* property of the customer identified by id equals to 3.

```
PUT /tms/xdata/Customer(1)/Photo HTTP/1.1
Host: server:2001

<binary photo content>
```

You can also use DELETE method to clear the blob content (remove any data):

```
DELETE /tms/xdata/Customer(1)/Photo HTTP/1.1
Host: server:2001
```

On successful UPDATE or DELETE calls, the server will respond with a *204 No Content* status code.

TXDataClient

The *TXDataClient* object (declared in unit `XData.Client`) allows you to send and receive objects to/from a XData server in a high-level, easy-to-use, strong-typed way. From any platform, any development environment, any language, you can always access XData just by using HTTP and JSON, but if you are coding from Delphi client, TXDataClient makes it much easier to write client applications that communicate with XData server.

To start using a TXDataClient, you just need to instantiate it and set the *Uri* property to point to root URL of the XData server:

```
uses {...}, XData.Client;

{...}

Client := TXDataClient.Create;
Client.Uri := 'http://server:2001/tms/xdata';
// <use client>
Client.Free;
```

The following topics explain how to use TXDataClient in details.

Invoking Service Operations

You can easily invoke service operations from a Delphi application using the TXDataClient class. Even though XData implements service operations using standards HTTP and JSON, which allows you to easily invoke service operations using HTTP from any client or platform, the TXDataClient makes it even easier by providing strong typing and direct method calls that in turn perform the HTTP requests under the hood.

Another advantage is that you don't need to deal building or finding out the endpoint URL (routing), with binding parameters or even create client-side proxy classes (when you use Aurelius entities). The same service interface you defined for the server can be used in the client. You can share the units containing the service interfaces between client and server and avoid code duplication.

To invoke service operations, you just need to:

1. Retrieve an interface using *Service<I>* method.
2. Call methods from the interface.

Here is an example of invoking the method *Sum* of interface *IMyService*, declaring in the topic "defining service interface":

```
uses
  {...}
  MyServiceInterface,
  XData.Client;

var
  Client: TXDataClient;
  MyService: IMyService;
  SumResult: double;
begin
  // Instantiate TXDataClient
  Client := TXDataClient.Create;
  // Set server Uri
  Client.Uri := 'http://server:2001/tms/xdata';
  // Retrieve IMyService inteface
  MyService := Client.Service<IMyService>;
  // call inteface methods
  SumResult := MyService.Sum(5, 10);
end;
```

Note that the client won't destroy any object passed as parameters, and will only destroy entities created by it (that were returned from the server), but no regular objects (like *TList<T>* or *TStream*). See "Memory Management" topic for detailed information.

Client Memory Management

Since method operations can deal with several types of objects, either Aurelius entities, plain old Delphi objects or even normal classes like *TList<T>* or *TStream*, it's important to know exactly how XData handles the lifetime of those objects, in order to avoid memory leaks or access violation exceptions due to releasing the same object multiple times.

This is the behavior or TXDataClient when it comes to receiving/sending objects (there is a separated topic for server-side memory management).

- Any object sent to the server (passed as a parameter) is **not destroyed**. You must handle the lifetime of those objects yourself.
- Any object of type *TStream* or *TList<T>* returned from the server is **not destroyed**. You must handle the lifetime of those objects yourself.
- Any other object returned from the server which type is not the ones mentioned in the previous item **is automatically destroyed** by default.

So consider the example below:

```
var
  Customer: TCustomer;
  Invoices: TList<TInvoice>;

{...}
Invoices := Client.Service<ISomeService>.DoSomething(Customer);
Customer.Free;
Invoices.Free;
```

Customer object is being passed as parameter. It will not be destroyed by the client and you must destroy it yourself. This is the same if you call Post, Put or Delete methods.

Items object (TList<T>) is being returned from the function. You must destroy the list yourself, it's not destroyed by the client. It's the same behavior for List method.

The *TInvoice* objects that are inside the list will be destroyed automatically by the client. You must not destroy them. Also, the same behavior for Get and List methods - entities are also destroyed by the client.

Alternatively, you can disable automatic management of entity instances at the client side, by using the *ReturnedInstancesOwnership* property:

```
Client.ReturnedInstancesOwnership := TInstanceOwnership.None;
```

The code above will prevent the client from destroying the object instances. You can also retrieve the list of all objects created by the client (that are supposed to be destroyed automatically) using property *ReturnedEntities*, in case you need to destroy them manually:

```
for Entity in Client.ReturnedEntities do {...}
```

Working With CRUD Endpoints

The following topics describe how to use TXDataClient to deal with TMS Aurelius CRUD Endpoints.

Requesting a Single Entity

To request a single entity, use the *Get* generic method passing the Id of the object as parameter:

```
Customer := Client.Get<TCustomer>(10);
State := Client.Get<TState>('FL');
```

The Id parameter is of type *TValue*, which has implicit conversions from some types like integer and string in the examples above. If there is no implicit conversion from the type of the id, you can use an overloaded method where you pass the type of Id parameter:

```
var
  InvoiceId: TGuid;
begin
  { ... get invoice Id }
  Invoice := Client.Get<TInvoice, TGuid>(InvoiceId);
end;
```

You can use the non-generic version of Get in case you only know the entity type at runtime (it returns a TObject and you need to typecast it to the desired type):

```
Customer := TCustomer(Client.Get(TCustomer, 10));
```

Requesting an Entity List

Use the *List* method to query and retrieve a list of entities from the server:

```
var
  Fishes: TList<TFish>;
begin
  Fishes := Client.List<TFish>;
```

The *TXDataClient.List<T>* function will always create and retrieve an object of type *TList<T>*. By default you must manually destroy that list object later, as explained in memory management topic.

Optionally you can provide a query string to send to the server to perform filtering, order, etc., using the XData query options syntax:

```
Customers := Client.List<TCustomer>('$filter=(Name eq ''Paul'') or (Birthday lt
1940-08-01)&$orderby=Name desc');
```

Use the non-generic version in case you only know the type of the entity class at runtime. In this case, the function will create and return an object of type *TList<TObject>*:

```
var
  Fishes: TList<TObject>;
begin
  Fishes := XClient.List(TFish);
```

You also can use *Count* method to retrieve only the total number of entities without needing to retrieve the full entity list:

```
var
  TotalFishes: Integer;
  TotalCustomers: Integer;
begin
  TotalFishes := Client.Count(TFish);
  TotalCustomers := Client.Count(TCustomer, '$filter=(Name eq ''Paul'') or
(Birthday lt 1940-08-01)&$orderby=Name desc');
end;
```

Creating Entities

Use *TXDataClient.Post* to create a new object in the server.

```
C := TCountry.Create;
try
  C.Name := 'Germany';
  Client.Post(C);
finally
  C.Free;
end;
```

Pay attention to client memory management to learn which objects you need to manually destroy. In this case, the client won't destroy the *TCountry* object automatically so you need to destroy it yourself.

The client makes sure that after a successful Post call, the Id of the object is properly set (if generated by the server).

Updating Entities

Use *TXDataClient.Put* to update an existing object in the server.

```
Customer := Client.Get<TCustomer>(10);
Customer.City := 'London'; // change city
Client.Put(Customer); // send changes
```

Pay attention to client memory management to learn which objects you need to manually destroy. Client won't destroy objects passed to Put method. In the above example, though, the object doesn't need to be destroyed because it was previously retrieved with Get, and in this case (for objects retrieved from the server), the client will manage and destroy it.

Removing Entities

Use *TXDataClient.Delete* to delete an object from the server. The parameter must be the object itself:

```
Customer := Client.Get<TCustomer>(10);
Client.Delete(Customer); // delete customer
```

Pay attention to client memory management to learn which objects you need to manually destroy. Client won't destroy objects passed to Delete method. In the above example, though, the object doesn't need to be destroyed because it was previously retrieved with Get, and in this case (for objects retrieved from the server), the client will manage and destroy it.

Using the Query Builder

XData allows you to easily query entities using a full query syntax, either by directly sending HTTP requests to entity set endpoints, or using the `List` method of `TXDataClient`.

For example, to query for customers which name is "Paul" or birthday date is lower then August 1st, 1940, ordered by name in descending order, you can write a code like this:

```
Customers := Client.List<TCustomer>('$filter=(Name eq ''Paul'') or (Birthday lt
1940-08-01)&$orderby=Name desc');
```

Alternatively to manually writing the raw query string yourself, you can use the XData Query Builder. The above code equivalent would be something like this:

```
uses {...}, XData.QueryBuilder, Aurelius.Criteria.Linq;

Customers := Client.List<TCustomer>(
  CreateQuery
    .From(TCustomer)
    .Filter(
      (Linq['Name'] = 'Paul')
      or (Linq['Birthday'] < EncodeDate(1940, 8, 1))
    )
    .OrderBy('Name', False)
    .QueryString
  );
```

Filter and FilterRaw

The `Filter` method receives an Aurelius criteria expression to later convert it to the syntax of XData `$filter` query parameter. Please refer to Aurelius criteria documentation to learn more about how to build such queries. A quick example:

```
  CreateQuery.From(TCustomer)
    .Filter(Linq['Name'] = 'Paul')
    .QueryString
```

Will result in `$filter=Name eq Paul`. You can also write the raw query string directly using `FilterRaw` method:

```
  CreateQuery.From(TCustomer)
    .FilterRaw('Name eq Paul')
    .QueryString
```

OrderBy and OrderByRaw

Method `OrderBy` receives either a property name in string format, or an Aurelius projection. A second optional boolean parameter indicates if the order must be ascending (`True` , the default) or descending (`False`). For example:

```
CreateQuery.From(TCustomer)
  .OrderBy('Name')
  .OrderBy('Id', False)
  .QueryString
```

Results in `$orderby=Name,Id desc` . The overload using Aurelius project allows for more complex expressions, like:

```
CreateQuery.From(TCustomer)
  .OrderBy(Linq['Birthday'].Year)
  .QueryString
```

Which results in `$orderby=year(Birthday)` . You can also write the raw order by expression directly using `OrderByRaw` method:

```
CreateQuery.From(TCustomer)
  .OrderByRaw('year(Birthday)')
  .QueryString
```

Top and Skip

Use Top and Skip methods to specify the values of `$top` and `$skip` query options:

```
CreateQuery.Top(10).Skip(30)
  .QueryString
```

Results in `$top=10&$skip=30` .

Expand

Specifies the properties to be added to `$expand` query option:

```
CreateQuery.From(TInvoice)
  .Expand('Customer')
  .Expand('Product')
  .QueryString
```

Results in `$expand=Customer,Product` .

Subproperties

If you need to refer to a subproperty in either `Filter`, `OrderBy` or `Expand` methods, just separate the property names using dot (`.`):

```
  CreateQuery.From(TCustomer)
    .Filter(Linq['Country.Name'] = 'Germany')
    .QueryString
```

Results in `$filter=Country/Name eq 'Germany'`.

From

If your query refers to property names, you need to use the `From` method to specify the base type being queried. This way the query builder will validate the property names and check their types to build the query string properly. There are two ways to do so: passing the class of the object being queries, or the entity type name:

```
  CreateQuery.From(TCustomer) {...}
  CreateQuery.From('Customer') {...}
```

Note that you can also specify the name of an instance type, ie., an object that is not necessarily an Aurelius entity, but any Delphi object that you might be passing as a DTO parameter.

When you pass a class name, the query builder will validate agains the names of field and properties of the class, not the final JSON value. For example, suppose you have a class mapped like this:

```
  TCustomerDTO = class
  strict private
    FId: Integer;
    [JsonProperty('the_name')]
    FName: string;
{...}
```

The following query will **work ok**:

```
  CreateQuery.From('Customer').Filter(Linq['the_name'] = 'Paul')
```

While the following query will **fail**:

```
  CreateQuery.From(TCustomerDTO).Filter(Linq['the_name'] = 'Paul')
```

Because `the_name` is not a valid property name for `TCustomerDTO` class. The correct query should be:

```
  CreateQuery.From(TCustomerDTO).Filter(Linq['Name'] = 'Paul')
```

Which will then result in the query string `$filter=the_name eq 'Paul'`.

Client and Multi-Model

When you create the TXDataClient object, it uses the default entity model to retrieve the available entity types and service operations that can be retrieved/invoked from the server. When your server has multiple models, though, you need to specify the model you are using when accessing the server. This is useful for the client to know which service interface contracts it can invoke, and of course, the classes of entities it can retrieve from the server. To do that, you pass the instance of the desired model to the client constructor:

```
Client := TXDataClient.Create(TXDataAureliusModel.Get('Security'));
```

See topic "Multiple servers and models" for more information.

Authentication Settings

For the HTTP communication, TXDataClient uses under the hood the Sparkle THttpClient. Such object is accessible through the *TXDataClient.HttpClient* property. You can use all properties and events of THttpClient class there, and the most common is the OnSendingRequest event, which you can use to set custom headers for the request. One common usage is to set the *Authorization* header with credentials, for example, a JSON Web Token retrieved from the server:

```
XDataClient.HttpClient.OnSendingRequest :=
  procedure(Req: THttpRequest)
  begin
    Req.Headers.SetValue('Authorization', 'Bearer ' + vToken);
  end;
```

Legacy Basic authentication

TXDataClient class provides you with the following properties for accessing servers protected with basic authentication.

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

Defines the UserName and Password to be used to connect to the server. These properties are empty by default, meaning the client won't send any basic authentication info. This is equivalent to set the Authorization header with property basic authentication value.

Ignoring Unknown Properties

TMS XData allows you work with the entity and DTO classes at client-side. Your client application can be compiled with the same class used in the server, and when a response is received from the server, the class will be deserialized at client-side.

However, it might happen that your server and client classes get out of sync. Suppose you have a class *TCustomer* both server and client-side. The server serializes the TCustomer, and client deserializes it. At some point, you update your server adding a new property *TCustomer.Foo*. The

server then sends the JSON with an additional Foo property, but the client was not updated and it doesn't recognize such property, because it was compiled with an old version of TCustomer class.

By default, the client will raise an exception saying Foo property is not known. This is the safest approach since if the client ignore the property, it might at some point send the TCustomer back to the server without Foo, and such property might get cleared in an update, for example.

On the other hand, this will require you to keep your clientes updated and in sync with the server to work. If you don't want that behavior, you can simply tell the client to ignore properties unknown by the client. To do this, use the *IgnoreUnknownProperties* property from TXDataClient:

```
XDataClient1.IgnoreUnknownProperties := True;
```

JSON Format

XData server uses JSON format in message payloads when receiving and sending HTTP messages to represent several different structures like entities, collection of entities or individual properties. Although JSON specification is very simple and describes completely how to use JSON format, the meaning of each JSON structure (especially name/value pairs) depend on the application and server behavior.

The following topics describes how each different structure is represented in JSON format by XData, and additional useful info about it.

Entity and Object Representation

Any Aurelius entity or simple Delphi object is serialized as a JSON object. Each property is represented as a name/value pair within the object.

The name of the properties in JSON for a simple Delphi object will be the field names of the object class, with the leading "F" removed from the name, if it exists.

The name of the properties in JSON for an Aurelius entity will be the same as property names defined in the XData Model according to Aurelius Equivalence to Entity Model.

A Delphi object in payload will always have all properties, unless you explicitly change this behavior by using JSON attributes to customize the serialization.

An Aurelius entity in a payload may be a complete entity, with all existing properties for that entity type, or a partial entity update (for partial update operations), which do not list all properties of the entity.

The following text illustrates how XData represents a simple *Customer* object (*TCustomer* class).

```
{
  "$id": 1,
  "Id": 55,
  "Name": "Joseph",
  "Birthday": "1980-05-20",
  "Sex": "tsMale",
  "Picture": null
}
```

In above JSON representation, Customer entity type contains simple properties *Id* (integer), *Name* (string), *Birthday* (date), *Sex* (enumeration) and *Picture* (blob). The property values are represented as direct JSON values. Note that XData also includes some metadata information in the JSON object, like "$id" name which represents the object reference id. In some cases, XData might include the "xdata.type" annotation which is needed for it to work properly with polymorphism. The following topics describe more specific details about how entities and its properties are represented in XData.

Property Values

Simple (scalar) properties of an entity/object are represented in as name/value pairs in the JSON object. The JSON name contains the property name, and value contains a JSON value that can be either a JSON string, number, boolean or null, depending on the property type. The format of most types are very straightforward (a string property is represented as a string JSON, an integer property as integer JSON, and so on), but a few types have some specific representation. The following table explains the JSON representation of the most common property types.

Data Type	**Examples**	**Description**
<null values>	null	Represented as JSON null literal.
Binary	"T0RhdGE"	Represented as JSON string, whose content must be the binary value encoding as Base64.
Boolean	true false	Represented as JSON true or false literals.
DateTime	"2013-12-25" "2013-12-25T12:12" "2013-12-25T12:12:20.050"	Represented as JSON string, whose content must be the date/time in ISO 8601 format (YYYY-MM-DDTdd:mm:ss.zzz). The time part can be completely omitted. If time part is present, hour and minutes are required, and seconds and milliseconds parts can also optionally be omitted.
Enum types	"tsMale" "Yellow"	Represented as JSON string, whose content is the name corresponding to the ordinal value of the enumerated property.
Float	3.14 1.2e-5	Represented as JSON number.
Guid	"E314E4B3-ECE5-4BD5-9D41-65B7E74F7CC8"	Represented as JSON string, whose content must be the string representation of the GUID, must not have enclosing brackets and must have the hyphens separating the five guid blocks (8 char, 4 char, 4 char, 4 char, 12 char). Each guid block is composed by hexadecimal digits.
Integer	1234	Represented as a JSON number.
String	"John"	Represented as JSON string, using JSON string escaping rules.

Object References

XData provides the concept of object referencing in JSON. This is useful to indicate which objects are the same object instance, and also to avoid circular references.

When serializing objects, XData attributes an "instance id" for that object and serializes it. The instance id is serialized as the first property of the object, with the name "$id". If during serialization the serializer finds another reference to the same object, it won't serialize the object again, but instead, will create a "instance reference" that refers to the instance id of the object previously serialized.

For example, consider you have *Product* entity type which has a *Category* property that points to a *Category* entity type. Suppose you have a list of two Product entities "Ball" and "Doll" that point to the same "Toys" category. This is how such list would be serialized:

```
[
  {
    "$id": 1,
    "@xdata.type": "XData.Default.Product",
    "Id": 10,
    "Name": "Ball",
    "Category": {
      "$id": 2,
      "@xdata.type": "XData.Default.Category",
      "Id": 5,
      "Name": "Toys"
    }
  },
  {
    "$id": 2,
    "@xdata.type": "XData.Default.Product",
    "Id": 12,
    "Name": "Doll",
    "Category": {
      "$ref": 2,
    }
  }
]
```

The TXDataClient deserializer also uses such notation to avoid duplicating objects and using the same object instance for multiple references. When the deserializer finds a JSON object with a "$ref" property it will try to find the object instance with the same "$id" value as the "$ref" property value. Then it will use the reference to that object instead of creating a new one.

In the JSON example above, when creating the second Product instance and setting the Category property, the deserializer will not create a new Category object. Instead, it will use the same Category object created in the first product. Thus, the Category property of both Product objects will point to the same Category object instance.

If the deserializer can't find an instance pointed by the "$ref" property, an error will be raised. All other properties of a JSON object containing a "$ref" property will be ignored.

Although the XData serializers adds an "$id" property to all objects, such property is not required in XData notation, thus the deserializer won't raise an error if an object doesn't have an instance id. But if present, this property must have the very first property (name/value pair) in the JSON object.

The rule of when object reference ($ref) will be used depend on the TXDataServerModule.SerializeInstanceRef property. By default, it's always used whenever an instance appears again. But it can be configured to only appear for recursive ocurrences (thus avoiding an infinite loop). Please refer to the property documentation for more info. When used only for recursive ocurrences, the example above will not include $ref, and the Category object will be repeated in the response:

```
[
  {
    "$id": 1,
    "@xdata.type": "XData.Default.Product",
    "Id": 10,
    "Name": "Ball",
    "Category": {
      "$id": 2,
      "@xdata.type": "XData.Default.Category",
      "Id": 5,
      "Name": "Toys"
    }
  },
  {
    "$id": 2,
    "@xdata.type": "XData.Default.Product",
    "Id": 12,
    "Name": "Doll",
    "Category": {
      "$id": 2,
      "@xdata.type": "XData.Default.Category",
      "Id": 5,
      "Name": "Toys"
    }
  }
]
```

Annotation "xdata.type"

All JSON objects in XData that represent an object might contain a metadata property named "@xdata.type". If present it must appear before any regular (non-metadata) property, otherwise an error is raised while deserializing the object. This property indicates the entity type of the JSON object. This is used by the deserializer to know which class to be used to instantiate the object. The value of this property is the name of the object class, or Aurelius entity type, prefixed by "XData.Default".

An example of an object of type "Customer":

```
{
  "$id": 1,
  "@xdata.type": "XData.Default.Customer",
  "Id": 55,
  "Name": "Joseph"
}
```

When sending requests to the server, "@xdata.type" annotation is optional. If not present, XData will try to infer the entity type from the context - for example, if your service operation is expecting a parameter of type *TCustomer* then it will deserialize the JSON as a TCustomer instance, if xdata.type annotation is missing. Thus, this annotation is mostly used when dealing with polymorphism, so you be sure that the entity will be deserialized as the correct type.

The presence of xdata.type annotation in server responses depend on the configuration of TXDataServerModule.SerializeInstanceType property. By default, it always send entity/object representations with the xdata.type, but the server can be configured to only include the annotation for derived types. Please refer to the property documentation for more info.

Representing Associated Objects

There are several ways to represent an associated entity in a JSON object.

An *associated object* is any object property that references a regular Delphi object.

An *associated entity* is any object property that references a TMS Aurelius entity through a navigation property (association).

For example, you might have an Aurelius entity type *Customer* with a navigation property *Country*, which target is the *Country* entity type. So the Country object is associated with the Customer object through the Country property, and might be represented in JSON notation. The equivalent Aurelius class would be something like this (parts of code removed):

```
TCustomer = class
  { code stripped }
  property Country: TCountry;
end;
```

Here we describe the different ways to represent an associated entity/object in JSON format. Please note that this is not related to representing object references, they are different concepts.

XData server responses will often use entity references when responding to resource requests. This behavior might be affected by the XData-ExpandLevel request header. Some other classes might also make use of such header transparently, like TXDataClient which automatically sets an expand level to 3 (thus all non-proxy entities will be serialized inline until the 3rd level in object tree).

Note that all the following options are only available for associated entities. For associated objects, only inline representation is allowed.

Entity Reference

This only applies to associated entities.

This is the most common way to represent an associated entity and should be preferred over any other method, if possible. It's somehow related to reference to an object instance, in a programming language. It's represented by annotating the navigation property name with "xdata.ref" sufix. The value of such property must be the canonical id of the associated object:

```
"Country@xdata.ref": "Country(10)"
```

In the simplified example above, the property *Country* is associated to the Country object with id equals to 10.

Clients can use that value to retrieve further info about the associated object by using the rules described in "canonical id" topic, performing an extra request to the server. Also, in update/insert operations, clients can also use this format to tell the server how to update the association. In the example above, if you send such object to the server in an update operation, the customer object will have its Country property updated to point to the *Country(10)* object (country with id equals to 10).

Entity/Object Inline

In some specific situations, the associated entity might be represented inline (for associated objects, this is always the case). That would be represented as a JSON object representing the whole object inline:

```
{
  "Country": {
    "$id": 2,
    "@xdata.type": "XData.Default.Country",
    "Id": 10,
    "Name": "Germany",
  }
}
```

Such representation would usually be used by client applications that want to provide associated entities that still do not have an id, thus can't be represented using the association reference format. Or can be returned by the server, if the expand level is increased. You can also explicitly ask for this format using the $expand query option.

To illustrate how association references are different from object references, you can have an entity represented inline, but using an object reference. For example, if the country object represented in the previous code (country of id 10 represented inline) was already present in the JSON tree before, it could be represented as an object reference:

```
{
  "Country": {
    "$ref": 2
  }
}
```

Proxy Info

This only applies to associated entities.

For performance reasons, the server might provide an object representation using proxy info instead of an association reference. The format is similar to the association reference, but the annotation is "xdata.proxy". The value must contain an URL (relative to server base URL) that clients can use to retrieve the associated object using a GET request:

```
"Country@xdata.proxy": "$proxy?id=52&classname='Customer'&classmember='Country'"
```

Association reference and proxy info are very similar, but they have two differences:

1. An association reference value has a very specific format (canonical id). Thus, association references, in addition to be a valid relative URL to retrieve the associated object, it is also in a specific format so that entity type and id can be parsed from the value, allowing you to safely know the entity type and id of the associated object without performing a GET request. Proxy info, in turn, is just a relative URL to retrieve the object, but the URL format is not standard and no additional metadata info can be safely extracted from the URL.
2. On data modification requests (insert/update) operations, association references are used to update the value of the navigation property (the object associated with the main object). Proxy info, in turn, are completely ignored by the server and do not cause any modification in the navigation property.

Blob Representation

In general, binary values are represented in JSON notation as base64 value, following the rules for serializing simple property values. The following JSON name/value pair is an example of a binary property Data representation:

```
"Data": "T0RhdGE"
```

When the property type in Aurelius is declared as TBlob type, though, the name/value pair in JSON notation might be declared using the "xdata.proxy" annotation after the property name, which allows clients to load the blob content in a lazy (deferred) way. The value of such name/value pair is the URL used to retrieve the blob content:

```
"Data@xdata.proxy": "Customer(55)/Data"
```

If the property is a *TBlob* type, the property will always be represented like that, unless:

1. The object to which the property belongs is a transient object (in the example above, if the customer object didn't have an id). In this case, the blob content will be represented inline as base64 string.

2. The blob content is somehow available to the server when the object is retrieved from database, and the blob content is empty. In this case, the property will be serialized normally (without annotation), and value is "null":

```
"Data": null
```

3. If the *$expand* query option was used to explicit load the content of the blob property. In this case, the blob content will be represented inline as base64 string.

Including Or Excluding Properties

This is how XData decides what properties to serialize/deserialize in JSON. When we mention "serialize" here it also implies "deserialize" (meaning that the JSON property will be recognized as valid and associated field/property will be updated from the JSON value).

- For entities, all class members (fields or properites) that are mapped using Aurelius will be serialized. Any transient field or property will not be serialized.
- For objects, all class fields will be serialized. No property will be serialized.

You can override this default and choose, for both your entity and object classes, what property will be included in JSON. The way to do that is different for entities and objects.

Aurelius Entities

For entities you can use attributes *XDataProperty* and *XDataExcludeProperty* (declared in unit `XData.Model.Attributes`). They have no arguments, and when put in a class member (field or property) it imply indicates that such property will be added or removed from the JSON.

In the following example, even though *FBirthday* is the field being mapped to the database, the final JSON will not include it, but instead will have three properties: *Year*, *Month* and *Day*.

```
uses {...}, XData.Model.Attributes;

  [Entity, Automapping]
  TCustomer = class
  private
    FId: Integer;
    FName: string;
    [XDataExcludeProperty]
    FBirthday: TDateTime;
    function GetDay: Integer;
    function GetMonth: Integer;
    function GetYear: Integer;
    procedure SetDay(const Value: Integer);
    procedure SetMonth(const Value: Integer);
    procedure SetYear(const Value: Integer);
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
    property Birthday: TDateTime read FBirthday write FBirthday;
    [XDataProperty]
    property Year: Integer read GetYear write SetYear;
    [XDataProperty]
    property Month: Integer read GetMonth write SetMonth;
    [XDataProperty]
    property Day: Integer read GetDay write SetDay;
  end;
```

PODO (Plain Old Delphi Objects)

JsonProperty and JsonIgnore attributes

For regular objects, you can use similar approach, but using attributes *JsonProperty* and *JsonIgnore* (declared in unit `Bcl.Json.Attributes`). One small difference is that JsonProperty can optionally receive a string which is the name of the property in final JSON.

In the following example, final Json will have properties *Id*, *PersonName*, *Birthday* and *YearOfBirth*. Field *FTransient* will not be serialized because of attribute JsonIgnore. Field *FName* will be serialized with a different name (*PersonName*) and property *YearOfBirth* will also be serialized because of the presence of attribute JsonProperty.

```
uses {...}, Bcl.Json.Attributes

  TDTOPerson = class
  private
    FId: Integer;
    [JsonProperty('PersonName')]
    FName: string;
    FBirthday: TDateTime;
    [JsonIgnore]
    FTransient: string;
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
    [JsonProperty]
    property YearOfBirth: Integer read GetYearOfBirth;
    property Birthday: TDateTime read FBirthday write FBirthday;
    property Transient: string read FTransient write FTransient;
  end;
```

JsonInclude attribute

You can also use *JsonInclude* attribute to specify which properties will be serialized to JSON based on their values. JsonInclude attribute should be added to the class type and receive a parameter of type *TInclusionMode*:

```
[JsonInclude(TInclusionMode.NonDefault)]
TDTOAddress = class
private
  {...}
```

Valid TInclusionMode values are:

- *TInclusionMode.Always*: Always serialize all DTO fields/properties (default behavior).
- *TInclusionMode.NonDefault*: Only serialize the fields/property if the value is a non-default value (not "empty", so to speak). Here is the list of checked types and what is considered their default value:
    - Object types: Nil pointer;
    - String types: Empty string;
    - Numeric types: Zero;
    - Enumerated Types: First enumerated value (ordinal valueof zero);
    - Set Types: Empty set;
    - Array Types: Empty array.

JsonEnumValues attribute

When serializing an enumerated value, by default it's the enumerated name value that will be serialized. For example:

```
type
  TMyEnum = (myFirst, mySecond, myThird);
```

A property of type *TMyEnum* could be serialized as following:

```
"MyEnumProp": "myFirst"
```

You can change that value using the *JsonEnumValues* property, passing the new values in a comma-separated string:

```
type
  [JsonEnumValues('first,second,third')]
  TMyEnum = (myFirst, mySecond, myThird);
```

That will generate the following JSON:

```
"MyEnumProp": "first"
```

JsonNamingStrategy attribute

If you have a general rule for naming the properties in the final JSON, you can use *JsonNamingStrategy* attribute instead of using a JsonProperty attribute for every single field/property you want to define a name. You add this attribute to the class informing the naming strategy to be used:

```
[JsonNamingStrategy(TCamelCaseNamingStrategy)]
TMySimpleClass = class
```

Here is the list of available naming strategies. The JSON examples are based on a class with the following field and property:

```
FFirstName: string;
property LastName: string;
```

- *TDefaultNamingStrategy*: This is the default strategy to be used in you don't specify one. It will keep the property name as-is, field names will have leading "F" removed.

```
{
  "FirstName": "Joe",
  "LastName": "Smith"
}
```

- *TCamelCaseNamingStrategy*: Names will be camel case, with first letter in lower case. Field names will have leading "F" removed before converting.

```
{
  "firstName": "Joe",
  "lastName": "Smith"
}
```

- *TSnakeCaseNamingStrategy*: Names will be snake case: all lower case with words separated by underlines. Field names will have leading "F" removed before converting.

```
{
  "first_name": "Joe",
  "last_name": "Smith"
}
```

- *TIdentityNamingStrategy*: Property and field names will be kept-as is.

```
{
  "FFirstName": "Joe",
  "LastName": "Smith"
}
```

- *TIdentityCamelCaseNamingStrategy*: Same as TCamelCaseNamingStrategy, but no leading "F" will be removed from field name.

```
{
  "fFirstName": "Joe",
  "lastName": "Smith"
}
```

- *TIdentitySnakeCaseNamingStrategy*: Same as TSnakeCaseNamingStrategy, but no leading "F" will be removed from field name.

```
{
  "ffirst_name": "Joe",
  "last_name": "Smith"
}
```

Customizing JSON Serialization

In addition to including and excluding properties from the JSON serialization, you can also modify the way a field/property is serialized as JSON. XData has its default serialization behavior for the primitive types, but you can modify it using the *JsonConverter* attribute.

Creating the converter

The converter must inherit from *TCustomJsonConverter* class and override methods *ReadJson* and *WriteJson*, which will read and write the serialized JSON value. Here is a small example:

```
uses {...}, Bcl.Json.Converters, Bcl.Json.Reader, Bcl.Json.Writer, System.Rtti;

type
  TSampleJsonConverter = class(TCustomJsonConverter)
  protected
    procedure ReadJson(const Reader: TJsonReader; var Value: TValue); override;
    procedure WriteJson(const Writer: TJsonWriter; const Value: TValue);
override;
  end;

{...}

procedure TSampleJsonConverter.ReadJson(const Reader: TJsonReader; var Value: TVa
lue);
var
  S: string;
begin
  S := Reader.ReadString;
  if SameText(S, 'one') then
    Value := 1
  else
  if SameText(S, 'two') then
    Value := 2
  else
    Value := StrToInt(S);
end;

procedure TSampleJsonConverter.WriteJson(const Writer: TJsonWriter; const Value:
TValue);
begin
  case Value.AsOrdinal of
    1: Writer.WriteString('one');
    2: Writer.WriteString('two');
  else
    Writer.WriteString(IntToStr(Value.AsOrdinal));
  end;
end;
```

The converter above can be applied to an integer property like this:

```
uses {...}, Bcl.Json.Attributes;

{...}
type
  TFoo = class
  private
    [JsonConverter(TSampleJsonConverter)]
    FProp: Integer;
```

With the setup above, the following behavior will apply:

- If FProp = 1, then it will be serialized as:

```
"FProp": "one"
```

- If FProp = 2, then it will be serialized as:

```
"FProp": "two"
```

- For other FProp values, it will be serialized normally as integer:

```
"FProp": 5
```

The converter also handles deserialization properly, i.e., if it reads value "one", it will set FProp as 1, and so on.

Collection of Objects

A collection of objects is represented as JSON array where each element is the representation of an entity or the representation of an entity reference, or representation of any simple object type supported by XData. An empty collection is represented as an empty JSON array.

Example of collection of entities with objects inline:

```
[
  {
    "$id": 1,
    "@xdata.type": "XData.Default.Country",
    "Id": 10,
    "Name": "Germany",
  },
  {
    "$id": 2,
    "@xdata.type": "XData.Default.Country",
    "Id": 13,
    "Name": "USA",
  }
]
```

When the server is responding to a request to an entity set resource address or a navigation property that returns an entity collection, or any service operation that returns a list of arbitrary objects, it wraps the collection in a JSON object with a name/value pair named "value":

```
{
  "value": [
    {
      "$id": 1,
      "@xdata.type": "XData.Default.Country",
      "Id": 10,
      "Name": "Germany",
    },
    {
      "$id": 2,
      "@xdata.type": "XData.Default.Country",
      "Id": 13,
      "Name": "USA",
    }
  ]
}
```

Individual Properties

When performing requests to an URL that represents an individual property, such property is represented as a JSON object (except for blob properties).

The property is represented as an object with a single name/value pair, whose name is "value" and whose value is represented according to the XData JSON Format notation for property values.

For example, when requesting the *Name* property of a *TCustomer* object (such as from address "*http://server:2001/tms/xdata/Customer(3)/Name*"), result might be:

```
{
  "value": "John Doe"
}
```

Note that when the URL is suffixed with "$value" segment or the property represents a blob property, then the content is the raw value of the property (or binary value), not a JSON representation.

Error Response

When an error happens in the XData server while processing a client request, the server might provide information about the error in the HTTP message body as a single JSON object, with a single name/value pair named "error" which value is also a JSON object. Such inline JSON object might contain the following name/value pairs:

- "*code*", which provides a JSON string representing the error code for the error raised by the server;
- "*message*", which provides a JSON string with a human-readable text describing the error.

Example:

```
{
  "error": {
    "code": "EntityNotFound",
    "message": "Requested entity does not exist."
  }
}
```

Canonical Id

The canonical id is a string representation that completely defines and identifies an Aurelius entity. It's used by XData in several places in JSON format, like when using association references.

The format of canonical id is "<entityset>(<id>)", where <entityset> is the name of the entity set which contains the entity, and <id> is the id entity in URL literal representation. The entity set must be the one which related entity type is the exact type of the entity being identifies. What this means is that if an entity belongs to more than one entity set, the type of the entity set must match the type of entity. For example, when dealing with inheritance between entity types, an entity of type "Dog" might belong to entity sets "Dog", "Mammal" and "Animal". In this case, the entity set must be "Dog".

Here are some examples of canonical id's:

```
"Invoice(5)"
"Customer('John')"
"InvoiceItem(15)"
```

It's important to note that the canonical id also represents the URI of the associated entity, relative to the server base URI. In other words, if you append the canonical id to the server base URI, you will end up with the URI of the associated entity, which you can use to retrieve, update or delete the entity (depending on the HTTP method used). For example, suppose the server base URI of the above entities is "http://myserver:2001/tms/xdata", then the resource URI of those entities are:

```
http://server:2001/tms/xdata/Invoice(5)
http://server:2001/tms/xdata/Customer('John')
http://server:2001/tms/xdata/InvoiceItem(15)
```

Design-Time Components

TMS XData provides several components for design-time usage. The main purpose is to provide a RAD experience, by just dropping components in the form and configuring them, allowing setting up servers with almost no line of code.

Even though you can use the components at runtime, creating them from code, that would usually be not necessary, as the components are just **wrappers** for the existing non-component classes, like TXDataServerModule or IDBConnectionPool.

Since TMS XData is based on TMS Sparkle components, you might want to read about TMS Sparkle design-time component usage, to learn more details about the architecture, like the common features and middleware system usage.

General usage is:

1. **Drop a dispatcher component in the form (for example, *TSparkeHttpSysDispatcher*).**

2. **Drop a TXDataServer component in the form.**

3. **Associated the TXDataServer component to the dispatcher through the *Dispatcher* property.**

4. **Specify the *BaseUrl* property of the server (for example, *http://+:2001/tms/xdata*).**

5. **Set *Active* property of dispatcher component to true.**

From now on you can already create server-side business logic using service operations.

Optionally, if you wan to use TMS Aurelius and want to publish TMS Aurelius CRUD Endpoints, follow the next steps:

6. **Drop a TAureliusConnection in the form and configure it to connect to your database** (you might need to drop additional database-access components, like *TFDConnection* if you want to use FireDac).

7. **Drop a TXDataConnectionPool component and associated it to TAureliusConnection through the *Connection* property.**

8. **Associated the TXDataServer component to the pool through the *Pool* property.**

Now your XData server is able to publish database data automatically and you can also use the TXDataOperationContext to use Aurelius to retrieve data from database.

You can also use the TAureliusConnection component to automatically generate Aurelius classes from the database, right-clicking the component and choosing "Generate entities from the database".

TXDataServer Component

TXDataServer component wraps the TXDataServerModule module. Basically all properties in the component has a direct relation with the properties of the TXDataServerModule, in case of doubt refer to TXDataServerModule reference to learn more details about each property.

Properties

Name	Description
Pool: TXDataConnectionPool	Contains a reference to a TXDataConnectionPool component. This will be used as the connection pool for the XData server database operations.
ModelName: string	Specifies the name of the model to be used to create the TXDataServerModule instance.
DefaultExpandLevel: Integer PutMode: TXDataPutMode PostMode: TXDataPostMode FlushMode: TXDataFlushMode ProxyLoadDepth: Integer ProxyListLoadDepth: Integer SerializeInstanceRef: TInstanceRefSerialization SerializeInstanceType: TInstanceTypeSerialization UnknownMemberHandling: TUnknownMemberHandling	All those property values will be used to set the a property with same name in the TXDataServerModule instance when it's created. Refer to TXDataServerModule topic to learn the purpose of each property.
DefaultEntitySetPermissions: TEntitySetPermissions	Specifies the default permissions for all entity sets. By default no permissions are provided, which means entity publish will not be available. This is **different behavior** than when creating TXDataServerModule directly, since it automatically publishes all entities.
EntitySetPermissions: TEntitySetPermissionItems	A collection where you can specify entity set permissions for an entity set in particular. This will override the default entity set permissions.
EnableEntityKeyAsSegment: Boolean	When True, it's possible to address single entities by using the URL format "*/entityset/id*" - in addition to the default "*/entityset(id)*". Default is False.
SwaggerOptions: TXDataSwaggerOptions SwaggerUIOptions: TXDataSwaggerUIOptions	Provide access to configure Swagger and SwaggerUI behavior. See more information at OpenAPI/Swagger Support.

Events

Name	Description
OnModuleCreate: TXDataModuleEvent	Fired when the TXDataServerModule instance is created.

Name	Description
OnGetPoolInterface: TGetPoolInterfaceEvent	Fired when the IDBConnectionPool interface is created by the TXDataConnectionPool component.
OnEntityInserting OnEntityModifying OnEntityDeleting OnEntityGet OnEntityList OnModuleException OnManagerCreate	These are wrappers around the events described in the server-side events chapter. Please refer to that chapter to know more about the events and how to use them. Arguments (event-handler parameters) are exactly the same.

TXDataModuleEvent

```
TXDataModuleEvent = procedure(Sender: TObject; Module: TXDataServerModule) of obj
ect;
```

Module parameter is the newly created TXDataServerModule instance (*OnModuleCreate* event).

TGetPoolInterfaceEvent

```
TGetPoolInterfaceEvent = procedure(Sender: TObject; var Pool: IDBConnectionPool)
of object;
```

Pool parameter is the newly created IDBConnectionPool interface (*OnGetPoolInterface* event). You can override that value by creating your own interface and passing it in the Pool variable.

TXDataConnectionPool Component

TXDataConnectionPool component creates IDBConnectionPool instances using a TAureliusConnection to create the IDBConnection interfaces.

Properties

Name	Description
Connection: TAureliusConnection	Contains a reference to a TAureliusConnection component, used to create IDBConnection interfaces used by the pool.
Size: Integer	The size of the pool to be created.

Events

Name	Description
OnPoolInterfaceCreate: TPoolInterfaceEvent	Fired when the IDBConnectionPool interface is created by the TXDataConnectionPool component.

TPoolInterfaceEvent

```
TPoolInterfaceEvent = procedure(Sender: TObject; var Pool: IDBConnectionPool) of object;
```

Pool parameter is the newly created IDBConnectionPool interface (*OnPoolInterfaceCreate* event). You can override that value by creating your own interface and passing it in the Pool variable.

XData Model

All the data exposed by the XData server is specified in the XData Model represented by the class *TXDataAureliusModel* (declared in unit `XData.Aurelius.Model`). The XData Model describes all the types (classes) and properties provided by the server, among other info.

The XData Model describes all information the server provides, which are used to create endpoints and to describe types and data received or returned by the server. This allows additional flexibility, like for example, having a property name in XData to be different than the property name in the mapped Aurelius class, or manually adding a service operation without relying on *ServiceContract/ServiceImplementation* attributes.

The XData model in summary, describes the service operations available, the Aurelius entities published from the CRUD endpoints, and the types it understands.

The following topics explain in more details what the XData model is about.

Entity Model Concepts

This topic describes the main concepts used in the XData Model:

Services/Contracts

Services (specified by contracts) are a set of service operations, actions that are executed based on a client request and HTTP method. The service contracts are defined by service interfaces (contracts) and implemented by the server. For more information, see service operations.

Enum Type

Enum types are named scalar, primitive types that contain a list of name/value pairs which indicates its valid values. For example, the enum type "TSex" might have two name/value pairs: First is name "Male", with value 0, and second is name "Female", with value 1. Simple properties which types are enum types can only receive such values.

Concepts related to Aurelius CRUD Endpoints

Entity Set (the CRUD Endpoint itself)

An entity set is a logical container for instances of an entity type (Aurelius entities) and instances of any type derived from that entity type. It can also be though as a collection of entities of a specified entity type. Thus, each entity set is associated with a entity type. Entity sets could be though as equivalent of a table in a database, but it's not exactly the same, since conceptually you might have different entity sets for the same entity type (although it's not often used). By default the name of the entity set is the name of the entity type associated to it so an entity set "Customer" will provide list of entities of type "Customer". It might be confused that both have same names, but they have different concepts.

Entity Type

The Aurelius entity type relates to an entity in the same way as a class relates to an object in object-oriented programming. In other words, an entity type is the "class definition" used by XData to retrieve info about an entity it receives or provides. An entity type contain a list of properties, what of those properties are considered the primary key for the entity, etc.. The entity type is related to a Delphi class, and as such it also supports inheritance (an entity type can be inherited from another entity type). By default the name of entity type is the name of the class, with the "T" prefix removed (if any). For example, there might be an entity type named "Customer", related to class TCustomer, with simples properties "Name", "City" and navigation property "Country".

Simple Property

A simple property is a property of a scalar type, belonging to an entity type. A simple property can be of any primitive type as integer, boolean, string, etc., or enum types, and can have several facets indicating for example if the property is nullable, or its maximum length. For example, an entity type "TCustomer" can have a simple property "Name" of type string, and a simple property "Sex" of enum type "TSex".

Navigation Property

A navigation property is a property that represents a relationship between two entity types. It might relate to a single entity or a collection of other entities. It's the equivalent of an association in Aurelius mapping, or a foreign key in a database definition. A navigation property has a target property which points to another entity type. For example, an entity type "TOrder" might have a navigation property named "Customer" which target type is the "TCustomer" entity type.

Using TXDataModelBuilder

XData server uses XData model to specify the data it will publish through the Rest/Json interface. To work properly, the server module (TXDataServerModule object) needs a XData model, represented by a *TXDataAureliusModel* to work properly.

However, you can create the *TXDataServerModule* without explicitly providing an entity model. In this case, the server will create one internally, based on the default *TMappingExplorer*, using the conventions of service operations and Aurelius CRUD endpoints. That's the most straightforward and common way to use it, and you will rarely need to create an XData model yourself.

In the case you want to build your own model and create a server based on it (for example, when you don't want classes and fields to have their "T" and "F" prefix removed from their names), you can do it using the *TXDataModelBuilder*.

The following example creates a model based on the default mapping explorer, however it sets properties *UseOriginalClassNames* and *UseOriginalFieldNames* to true to avoid the model builder to remove the "T" prefix from class names when naming entity types, and remove "F" prefix from field names when naming properties and navigation properties.

```
uses
  {...}, XData.Aurelius.ModelBuilder, XData.Aurelius.Model,
  XData.Server.Module, Aurelius.Mapping.Explorer;

var
  Builder: TXDataModelBuilder;
  Model: TXDataAureliusModel;
  Explorer: TMappingExplorer;
  Module: TXDataServerModule;
begin
  // model uses a TMappingExplorer instance with Aurelius mapping
  // this example doesn't explain how to obtain one, please refer to Aurelius
documentation
  Explorer := TMappingExplorer.DefaultInstance;
  Model := TXDataAureliusModel.Create(Explorer);
  try
    Builder := TXDataModelBuilder.Create(Model);
    try
      Builder.UseOriginalClassNames := true;
      Builder.UseOriginalFieldNames := true;
      Builder.Build;
    finally
      Builder.Free;
    end;
  except
    Model.Free; // destroy the model in case an error occurs
    raise;
  end;

  // create the module using the created model.
  Module := TXDataServerModule.Create(MyServerUrl, MyConnectionPool, Model);
end;
```

When adding service operations to the model, the model builder will only take into consideration those service interfaces that belong to the same model of the TMappingExplorer instance you are using. For example, if your TMappingExplorer represents the model "Sample", only service contracts marked for the "Sample" model will be included in the entity model.

You can use the methods and properties of TXDataModelBuilder to better define your XData Model:

TXDataModelBuilder class

Declared in Unit `XData.Aurelius.ModelBuilder`.

Properties

Name	Description
UseOriginalClassNames: Boolean	By default XData removes the leading "T" of the class name to define the entity set name. So a class TCustomer will correspond to the entity set "Customer". If you don't want this behavior and wants the entity set name to be exactly the name of the class, set this property to true.
UseOriginalFieldNames: Boolean	By default XData removes the leading "F" of a mapped field name to define the property name of an entity type. So a field named "FName" will correspond to a property "Name" in JSON responses. If you don't want this behavior and wants the property name to be exactly the name of the field, set this property to true.
UseOriginalContractNames: Boolean	By default XData removes the leading "I" of the interface name to define the url segment of a service operation. So an interface ITestService will correspond to a service URL which path segment will beign with "TestService". If you don't want this behavior and wants the service operation base URL to be exactly the name of the interface, set this property to true.

Methods

Name	Description
function AddEntitySet(AClass: TClass): TXDataEntitySet	Creates a new entity set (CRUD endpoint) based on the class defined by AClass. The class **must** be an Aurelius entity. This will automatically create the entity type associated with the entity set.
procedure AddService<T>	Adds a new service operation contract to the model. Example: `Model.AddService<IMyService>;`
procedure RemoveEntitySet(AClass: TClass)	Use this to remove an entity set already created by the model. This is useful when you want to use the automatic creation of entity sets, based on Aurelius entities, but then you want to remove a few entity sets from the model.

Multiple servers and models

You can create multiple XData servers (TXDataServerModule instances), at different addresses. This makes more sense if you use different entity models for each of those XData modules. It's very easy to do so in XData using a way similar to Aurelius multi-model design. When defining different models with different classes (and service operations), you can easily create different XData modules in a single server application.

To specify the different models in each server module, you just need to provide an instance of a *TXDataAureliusModel* that represents the desired entity model. And you can easily retrieve the correct instance by using the *TXDataAureliusModel.Get* class method, passing the model name to the function. The entity model will then only consider entity types and service operations that belongs to that model.

For example, the following code creates two different TXDataServerModule modules, using two different entity models:

```
XDataSampleModule := TXDataServerModule.Create('http://server:2001/tms/xdata/
sample',
  SampleConnectionPool, TXDataAureliusModel.Get('Sample'));

XDataSecurityModule := TXDataServerModule.Create('http://server:2001/tms/xdata/
security',
  SecurityConnectionPool, TXDataAureliusModel.Get('Security'));

HttpServer.AddModule(XDataSampleModule);
HttpServer.AddModule(XDataSecurityModule);
```

Note that this approach will filter all classes (entity types) belonging that specific model (for example, *XDataSampleModule* will only publish entity types which classes are marked with *Model('Sample')* attribute:

```
[Model('Sample')]
```

It will filter both service operations and Aurelius CRUD Endpoints according to the model name. So XDataSampleModule will only have services operations which service interfaces are marked with the *Model('Sample')* attribute.

Server-Side Events

TXDataServerModule published several events that you can use to implement additional server-side logic, customize XData behavior, among other tasks. You can use events, for example, to:

- Implement custom authorization, refusing or accepting an operation based on the user credentials or any other context information;
- Restrict/change the data returned to the client by TMS Aurelius CRUD Endpoints, by adding more filters to a query made by the user, for example;
- Implement additional server-side logic, for example, performing extra operations after a resource is saved.

Events in XData are available in the *Events* property of the TXDataServerModule object. Such property refers to a *TXDataModuleEvents* (declared in unit `XData.Module.Events`) object with several subproperties, each to them related to an event.

Read Using Events for more detailed info. You can also see real-world usage of events by checking the Authentication Example using JSON Web Token (JWT).

Events in TXDataModuleEvents

General-purpose events

Name	Description
OnModuleException	Occurs when an exception is raised during request processing. You can use it to provide custom error-handling.
OnManagerCreate	Occurs when a TObjectManager instance is created to be used during request processing. You can use it to customize settings for the TObjectManager.

Events of TMS Aurelius CRUD Endpoints

Name	Description
OnEntityGet	Occurs after an entity is retrieved, right before being sent to the client.
OnEntityList	Occurs when the client queries an entity collection.
OnEntityInserting	Occurs right before an entity creation.
OnEntityInserted	Occurs right after an entity creation.
OnEntityModifying	Occurs right before an entity update.
OnEntityModified	Occurs right after an entity update.
OnEntityDeleting	Occurs right before an entity delete.
OnEntityDeleted	Occurs right after an entity delete.

Using Events

Events in XData are available in the *Events* property of the TXDataServerModule object. Such property refers to a *TXDataModuleEvents* (declared in unit `XData.Module.Events`) object with several subproperties, each to them related to an event.

For example, to access the *OnEntityInserting* event:

```
uses {...}, XData.Server.Module, XData.Module.Events;

// Module is an instance of TXDataServerModule object
Module.Events.OnEntityInserting.Subscribe(
  procedure(Args: TEntityInsertingArgs)
  begin
    // Use Args.Entity to retrieve the entity being inserted
  end
);
```

In a less direct way, using method reference instead of anonymous method:

```
uses {...}, XData.Server.Module, XData.Module.Events;

procedure TSomeClass.MyEntityInsertingProc(Args: TEntityInsertingArgs);
begin
  // Use Args.Entity to retrieve the entity being inserted
end;

procedure TSomeClass.RegisterMyEventListeners(Module: TXDataServerModule);
var
  Events: TXDataModuleEvents;
begin
  Events := Module.Events;
  Events.OnEntityInserting.Subscribe(MyEntityInsertingProc);
end;
```

Listeners are method references that receive a single object as a parameter. Such object has several properties containing relevant information about the event, and differ for each event type. Names of event properties, method reference type and arguments follow a standard. The event property is named "On<event>", method reference type is "T<event>Proc" and parameter object is "T<event>Args". For example, for the "EntityInserting" event, the respective names will be "OnEntityInserting", "TEntityInsertingProc" and "TEntityInsertingArgs".

All events in XData are multicast events, which means you can add several events handlers (listeners) to the same event. When an event occurs, all listeners will be notified. This allows you to add a listener in a safe way, without worrying if it will replace an existing listener that might have been set by other part of the application.

It's always safe to set the events before adding the module and running the server.

OnEntityGet Event

Occurs after an entity is retrieved, right before being sent to the client. This event is also fired when the client requests part of that entity, for example, individual properties of the entity, blob data, or associated entities.

Example:

```
Module.Events.OnEntityGet.Subscribe(
  procedure(Args: TEntityGetArgs)
  begin
    // code here
  end
);
```

TEntityGetArgs Properties:

Name	Description
Entity: TObject	The retrieved entity.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityList Event

Occurs when the client queries an entity collection. This event is fired after the *TCriteria* object is built based on the client request, and right before the criteria is actually executed to retrieve the entities and sent to the client.

Example:

```
Module.Events.OnEntityList.Subscribe(
  procedure(Args: TEntityListArgs)
  begin
    // code here
  end
);
```

TEntityListArgs Properties:

Name	Description
Criteria: TCriteria	The Aurelius criteria built based on client request that will be used to retrieve the collections. You can modify the request here, adding extra filters, orders, etc., before it's executed and results are sent to the client.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityInserting Event

Occurs right before an entity creation. This event happens in the middle of a database transaction, right before the entity is about to be created in the database.

Example:

```
Module.Events.OnEntityInserting.Subscribe(
  procedure(Args: TEntityInsertingArgs)
  begin
    // code here
  end
);
```

TEntityInsertingArgs Properties:

Name	Description
Entity: TObject	The entity being inserted.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityInserted Event

Occurs after an entity is created. Note that, unlike OnEntityInserting Event, this event happens **after** the transaction is committed. There is no way to rollback the insertion of the entity, and any database operation here will be performed with no active transaction (unless you begin one manually).

Example:

```
Module.Events.OnEntityInserted.Subscribe(
  procedure(Args: TEntityInsertedArgs)
  begin
    // code here
  end
);
```

TEntityInsertedArgs Properties:

Name	Description
Entity: TObject	The entity which was created (inserted).
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityModifying Event

Occurs right before an entity update. This event happens in the middle of a database transaction, right before the entity is about to be updated in the database.

Example:

```
Module.Events.OnEntityModifying.Subscribe(
  procedure(Args: TEntityModifyingArgs)
  begin
    // code here
  end
);
```

TEntityModifyingArgs Properties:

Name	Description
Entity: TObject	The entity being modified.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityModified Event

Occurs right after an entity update. Note that, unlike OnEntityModifying Event, this event happens **after** the transaction is committed. There is no way to rollback the update of the entity, and any database operation here will be performed with no active transaction (unless you begin one manually).

Example:

```
Module.Events.OnEntityModified.Subscribe(
  procedure(Args: TEntityModifiedArgs)
  begin
    // code here
  end
);
```

TEntityModifiedArgs Properties:

Name	Description
Entity: TObject	The entity which was modified.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityDeleting Event

Occurs right before an entity delete. This event happens in the middle of a database transaction, right before the entity is about to be deleted in the database.

Example:

```
Module.Events.OnEntityDeleting.Subscribe(
  procedure(Args: TEntityDeletingArgs)
  begin
    // code here
  end
);
```

TEntityDeletingArgs Properties:

Name	Description
Entity: TObject	The entity being deleted.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnEntityDeleted Event

Occurs after an entity is deleted. Note that, unlike OnEntityDeleting event, this event happens **after** the transaction is committed. There is no way to rollback the deletion of the entity, and any database operation here will be performed with no active transaction (unless you begin one manually).

Example:

```
Module.Events.OnEntityDeleted.Subscribe(
  procedure(Args: TEntityDeletedArgs)
  begin
    // code here
  end
);
```

TEntityDeletedArgs Properties:

Name	Description
Entity: TObject	The entity which has been deleted.
Handler: TXDataBaseRequestHandler	The XData request processor object.

OnModuleException Event

Occurs when an unexpected exception is raised during server request processing. By default, when that happens XData will send a response to the client with the property HTTP status code (usually *500* but other codes might be provided as well) and a JSON response with details of the error (a JSON object with properties *ErrorCode* and *ErrorMessage*). You can use this event to change that behavior when a specific exception occurs.

Example:

```
Module.Events.OnModuleException.Subscribe(
  procedure(Args: TModuleExceptionArgs)
  begin
    // code here, for example:
    if Args.Exception is EInvalidJsonProperty then
      Args.StatusCode := 400;
  end
);
```

TModuleExceptionArgs Properties:

Name	Description
Exception: Exception	The Exception object raised while processing the requested.
StatusCode: Integer	The HTTP status code to be returned to the client. You can change this property to tell XData to send a different status code.
ErrorCode: string	The value of the ErrorCode property in the JSON response to be sent to the client. You can modify this value.
ErrorMessage: string	The value of the ErrorMessage property in the JSON response to be sent to the client. You can modify this value.
Action: TModuleExceptionAction	The action to be performed: `TModuleExceptionAction = (SendError, RaiseException, Ignore)` Default value is *SendError* which means XData will send the HTTP response to the client with the specified *StatusCode* and with a JSON response that includes *ErrorCode* and *ErrorMessage*. You can optionally use *RaiseException*, which means re-raising the original exception and let it propagate through the code. This gives an opportunity for some Sparkle middleware to catch the raise exception. If that doesn't happen, the exception will be handled by the Sparkle dispatcher. The third option is simply choose *Ignore*. Use this option if you want to send a custom HTTP response yourself. In this case XData will simply do nothing and finish processing request silently.

OnManagerCreate Event

Occurs when a TObjectManager instance is internally created by XData during request processing. You can use this event to initialize settings in the object manager, or change some behavior. For example, you can use this event to enable filters.

Example:

```
Module.Events.OnManagerCreate.Subscribe(
  procedure(Args: TManagerCreateArgs)
  begin
    // code here, for example:
    Args.Manager.EnableFilter('Multitenant')
      .SetParam('tenantId', 123);
  end
);
```

TManagerCreateArgs Properties:

Name	Description
Manager: TObjectManager	The TObjectManager instance that has just been created by XData.

Authentication Example using JSON Web Token (JWT)

Please refer to Authentication and Authorization guide for more information about how to secure your XData API.

Authentication and Authorization

Authentication and Authorization mechanisms in XData are available through the built-in auth mechanisms provided in TMS Sparkle, the underlying HTTP framework which XData is based on.

The authentication/authorization mechanism in Sparkle is generic and can be used for any types of HTTP server, not only XData. But this topic illustrates how to better use specific XData features like server-side events and attributed-based authorization to make it even easier to secure your REST API.

In this guide we will present the concept of JSON Web Token, then how to authenticate requests (make sure there is a "user" doing requests) and finally how to authorize requests (make sure such user has permissions to perform specific operations).

NOTE
Even though we are using JWT as an example, the authentication/authorization mechanism is generic. You can use other type of token/authentication mechanism, and the authorization mechanism you use will be exactly the same regardless what token type you use. Holger Flick's book TMS Hands-on With Delphi shows a good example of authentication/authorization using a different approach than JWT.

JSON Web Token (JWT)

From Wikipedia:

> JSON Web Token (JWT) is a JSON-based open standard (RFC 7519) for passing claims between parties in web application environment.

That doesn't say much if we are just learning about it. There is plenty of information out there, so here I'm going directly to the point in a very summarized practical way.

A JWT is a string with this format:

```
aaaaaaaaaaa.bbbbbbbbbb.ccccccccc
```

It's just three sections in string separated by dots. Each section is a text encoded using base64-url:

```
<base64url-encoded header>.<base64url-encoded claims>.<base64url-encoded signature>
```

So a real JWT looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoidG1zdXNlciIsImlzcyI6IlRNUyBYRGF0YSBTZXJ2ZXIiLCJhZG
```

If we decode each part of the JWT separately (remember, we have three parts separated by dots), this is what we will have from part one (spaces and returns added to make it more readable). It's the **header**:

```
{
  "alg":"HS256",
  "typ":"JWT"
}
```

And this is part two decoded, which is the **payload** or **claims set**:

```
{
  "name":"tmsuser",
  "iss":"TMS XData Server",
  "admin":true
}
```

Finally the third part is the **signature**. It makes no sense to decode it here since it's just a bunch of bytes that represent the hash of the header, the payload, and a secret that only the generator of the JWT knows.

The **payload** is the JSON object that "matters", it's the actualy content that end-user applications will read to perform actions. The **header** contains meta information the token, mostly the hashing algorithm using to generate the **signature**, also present in the token. So, we could say that a JWT is just an alternative way to represent a JSON object, but with a signature attached to it.

What does it has to do with authentication and authorization? Well, you can think of the JWT as a "session" or "context" for an user accessing your server. The JSON object in the payload will contain arbitrary information that you are going to put in there, like permissions, user name, etc.. This token will be generated by your server upon some event (for example, an user "login"), and then the client will resend the token to the server whenever he wants to perform any operation. This would be the basic workflow:

1. Client performs "login" in the server by passing regular user credentials (user name and password for example).
2. The server validates the credentials, generate a JWT with relevant info, using the *secret*, and sends the JWT back to the client.
3. The client sends the JWT in next requests, passing the JWT again to the server.
4. When processing each request, the server checks if the JWT signature is valid. If it is, then it can trust that the JSON Object in payload is valid and process actions accordingly.

Since only the server has the secret, there is no way the client can change the payload, adding "false" information to it for example. When the server receives the modified JWT, the signature will not match and the token will be rejected by the server.

For more detailed information on JSON Web Tokens (JWT) you can refer to https://jwt.io, the Wikipedia article or just the official specification. It's also worth mentioning that for handling JWT internally, either to create or validate the tokens, TMS XData uses under the hood the open source Delphi JOSE and JWT library.

Authentication

Enough of theory, now we will show you how to do authentication using JWT in TMS XData. This is just a suggestion, and it's up to you to define with more details how your system will work. In this example we will create a login service, add the middleware and use server-side events and attributes to implement authorization.

User Login and JWT Generation

We're going to create a service operation to allow users to perform login. Our service contract will look like this:

```
[ServiceContract]
ILoginService = interface(IInvokable)
['{BAD477A2-86EC-45B9-A1B1-C896C58DD5E0}']
  function Login(const UserName, Password: string): string;
end;
```

Clients will send user name and password, and receive the token. Delphi applications can invoke this method using the TXDataClient, or invoke it using regular HTTP, performing a POST request passing user name and password parameters in the body request in JSON format.

> **WARNING**
> It's worth noting that in production code you should always use a secure connection (HTTPS) in your server to protect such requests.

The implementation of such service operation would be something like this:

```
uses {...}, Bcl.Jose.Core.JWT, Bcl.Jose.Core.Builder;

function TLoginService.Login(const User, Password: string): string;
var
  JWT: TJWT;
  Scopes: string;
begin
  { check if UserName and Password are valid, retrieve User data from database,
    add relevant claims to JWT and return it. In this simplified example,
    we are doing a simple check for password }
  if User <> Password then
    raise EXDataHttpUnauthorized.Create('Invalid password');
  JWT := TJWT.Create;
  try
    JWT.Claims.SetClaimOfType<string>('user', User);
    if User = 'admin' then
      JWT.Claims.SetClaimOfType<Boolean>('admin', True);
```

```
    Scopes := 'reader';
    if (User = 'admin') or (User = 'writer') then
      Scopes := Scopes + ' writer';
    JWT.Claims.SetClaimOfType<string>('scope', Scopes);
    JWT.Claims.Issuer := 'XData Server';
    Result := TJOSE.SHA256CompactToken('secret', JWT);
  finally
    JWT.Free;
  end;
end;
```

Now, users can simply login to the server by performing a request like this (some headers removed):

```
POST /loginservice/login HTTP/1.1
content-type: application/json

{
  "UserName": "writer",
  "Password": "writer"
}
```

and the response will be a JSON object containing the JSON Web Token (some headers removed and JWT modified):

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "value":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoid3JpdGVyIiwic2NvcGUiOiJyZWFkZXIgd3JpdGVyIiwiaXNzIjoiWERhdGEgU2VydmVyIn0.QdRTt6gOl3tb1Zg0aAJ74bepQwqm0Rd735FKToPEyFY"
}
```

For further requests, clients just need to add that token in the request using the authorization header by indicating it's a bearer token. For example:

```
GET /artist?$orderby=Name HTTP/1.1
content-type: application/json
authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoid3JpdGVyIiwic2NvcGUiOiJyZWFkZXIgd3JpdGVyIiwiaXNzIjoiWERhdGEgU2VydmVyIn0.QdRTt6gOl3tb1Zg0aAJ74bepQwqm0Rd735FKToPEyFY
```

NOTE

This authentication mechanism is a suggestion and is totally independent from the rest of this guide. The JWT token could be provided in any other way: a different service, different server, different sign-in mechanism (not necessarily username/password), or even a 3rd party token provider.

Implementing JWT Authentication with TJwtMiddleware

The second step is to add a JWT middleware to your server.

At design-time, right-click the TXDataServer component, choose the option to manage the middleware list, and add a JWT Middleware to it. The middleware has an `OnGetSecret` event that you need to handle to pass to it the JWT secret our server will use to validate the signature of the tokens it will receive.

If you are using the XData server module, all you need to do is to add a `TJwtMiddleware` and inform the secret in the constructor:

```
uses {...}, Sparkle.Middleware.Jwt;

{...}
  Module.AddMiddleware(TJwtMiddleware.Create('secret'));
```

That's it. What this will do? It will automatically check for the token in the authorization header. If it does exist and signature is valid, it will create the `IUserIdentity` interface, set its Claims based on the claims in the JWT, and set such interface to the User property of THttpRequest object.

WARNING
Regardless if the token exists or not and the User property is set or not, the middleware will forward the processing of the request to your server. It's up to you to check if user is present in the request or not. If you want the token to prevent non-authenticated requests to be processed, set its `ForbidAnonymousAccess` to true.

If the token is present and is invalid, it will return an to the client immediately and your server code will not be executed.

Authorization

Now that we know how to check for authenticated requests, it's time to authorize the requests - in other words, check if the authenticated client/user has permission to perform specific operations.

Attribute-based Authorization

The easiest way to authorize your API is to simply add authorization attributes to parts of the code you want to specify permissions.

Remember that a XData server has two mechanism for publishing endpoints: service operations and automatic CRUD endpoints. Each of them has different ways to be authorized.

You must use unit `XData.Security.Attributes` to use authorization attributes.

Authorize Attribute

This attribute is can be used in service operations. Just add an `Authorize` attribute to any method in your service contract interface, and such method will only be invoked if the request is authenticated.

```
[ServiceContract]
IMyService = interface(IInvokable)
['{80A69E6E-CA89-41B5-A854-DFC412503FEA}']

  function NonRestricted: string;

  [Authorize]
  function Restricted: string;
end;
```

In the example above, the endpoint represented by the method `NonRestricted` will be publicly accessible, while the method `Restricted` will only be invoked if the request is authenticated. Otherwise, a `403 Forbidden` response will be returned.

You can also apply the `Authorize` attribute at the interface level:

```
[ServiceContract]
[Authorize]
IMyService = interface(IInvokable)
['{80A69E6E-CA89-41B5-A854-DFC412503FEA}']

  function Restricted: string;
  function AlsoRestricted: string;
end;
```

In this case, the attribute rule will be applied to all methods of the interface. In the example above, both `Restricted` and `AlsoRestricted` methods will only be invoked if request is authenticated.

AuthorizeScopes Attribute

You can use `AuthorizeScope` attribute in service operations if you want to allow them to be invoked only if the specified scopes are present in user claims.

It will check for a user claim with name `scope`, and check its values. The scope values in `scope` claim must be separated by spaces. For example, the scope claim might contain `editor` or `reader writer`. In the last example, it has two scopes: `reader` and `writer`.

```
  [AuthorizeScopes('admin')]
  procedure ResetAll;
```

In the example above, the `ResetAll` method can only be invoked if the `admin` scope is present in the `scope` claim.

You can specify optional scopes that will allow a method to be invoked, by separating scopes with comma:

```
[AuthorizeScopes('admin,writer')]
procedure ModifyEverything;
```

In the previous example, `ModifyEverything` can be invoked if the `scope` claim contain **either** `admin` scope, **or** `writer` scope.

You can add multiple `AuthorizeScopes` attributes, which will end up as two requirements that **must** be met to allow method to be invoked:

```
[AuthorizeScopes('publisher')]
[AuthorizeScopes('editor')]
procedure PublishAndModify;
```

For method `PublishAndModify` to be invoked, **both** scopes `publisher` **and** `editor` must be present in the `scope` claim.

> **NOTE**
> Just like `Authorize` attribute and any other authorization attribute, you can apply `AuthorizeScopes` attribute at both method and interface level. If you apply to both, then all requirements set by the authorization attributes added to interface and method will have to be met for the method to be invoked.

AuthorizeClaims Attribute

If you want to check for an arbitrary user claim, the use `AuthorizeClaims` attribute:

```
[AuthorizeClaims('admin')]
procedure OnlyForAdmins;
```

The `OnlyForAdmins` method will only be executed if the claim 'admin' is present in user claims.

You can also check for a specific claim value, for example:

```
[AuthorizeClaims('user', 'john')]
procedure MethodForJohn;
```

In this case, `MethodForJohn` will only be executed if claim `user` is present and its value is `john`.

As already described above, `AuthorizeClaims` attribute can be used multiple times and can be applied at both method and interface level.

EntityAuthorize Attribute

You can also protect automatically created CRUD endpoints using authorization attributes. Since those endpoints are based on existing Aurelius entities, you should apply those attributes to the entity class itself.

NOTE
Attributes for automatic CRUD endpoints are analogous to the ones you use in service operations. The different is they have a prefix `Entity` in the name, and receive an extra parameter of type `TEntitySetPermissions` indicating to which CRUD operations the attribute applies to.

The simplest one is the `EntityAuthorize` attribute:

```
[Entity, Automapping]
[EntityAuthorize(EntitySetPermissionsWrite)]
TCustomer = class
```

In the previous example, to invoke endpoints that modify the `TCustomer` entity (like `POST`, `PUT`, `DELETE`), the request must be authenticated.

WARNING
The rules are applied by entity set permission. In the previous example, the read permissions (`GET` a list of customer or a specific customer) are not specified, and **thus they are not protected**. User don't need to be authenticated to execute them.

EntityAuthorizeScopes Attribute

Similarly to AuthorizeScopes, you can restrict access to CRUD endpoints depending on the existing scopes using `EntityAuthorizeScopes`:

```
[Entity, Automapping]
[EntityAuthorizeScopes('reader', EntitySetPermissionsRead)]
[EntityAuthorizeScopes('writer', EntitySetPermissionsWrite)]
TArtist = class
```

In the previous example, read operations will be allowed **if and only if** the scope `reader` is present. At the same time, the `writer` scope must be present to perform write operations.

That means that to perform **both** read and write operations, the scope claim must have **both** `reader` and `writer` values: `reader writer`.

One alternative approach is the following:

```
[Entity, Automapping]
[EntityAuthorizeScopes('reader,writer', EntitySetPermissionsRead)]
[EntityAuthorizeScopes('writer', EntitySetPermissionsWrite)]
TArtist = class
```

In the case above, if the scope just have `writer`, then it will be able to perform both read and write operations, since read permissions are allowed if either `reader` or `writer` are present in `scope` claim. Either approach is fine, it's up to you to decide what's best for your application.

EntityAuthorizeClaims Attribute

Finally, similarly to `AuthorizeClaims` attribute, you can use `EntityAuthorizeClaims` to allow certain operations only if a claim exists and/or has a specific value:

```
[Entity, Automapping]
[EntityAuthorizeClaims('user', 'john', [TEntitySetPermissions.Delete])]
TArtist = class
```

In the example above, only users with claim `user` equals `john` will be able to delete artists.

Manual Authorization in Service Operations

Finally, in addition to authorization attributes, you can always add specific code that authorizes (or forbids) specific operations based on user identity and claims. All you have to do is check for the request User property and take actions based on it.

For example, suppose you have a service operation *DoSomething* that does an arbitrary action. You don't want to allow anonymous requests (not authenticated) to perform such action. And you will only execute such action is authenticated user is an administrator. This is how you would implement it:

```
uses {...}, Sparkle.Security, XData.Sys.Exceptions;

procedure TMyService.DoSomething;
var
  User: IUserIdentity;
begin
  User := TXDataOperationContext.Current.Request.User;
  if User = nil then
    raise EXDataHttpUnauthorized.Create('User not authenticated');
  if not (User.Claims.Exists('admin') and User.Claims['admin'].AsBoolean) then
    raise EXDataHttpForbidden.Create('Not enough privileges');

  // if code reachs here, user is authenticated and is administrator
  // execute the action
end;
```

Using Server-Side Events

You can also use server-side events to protect the entity sets published by XData, and add custom code to it. For example, you can use the `OnEntityDeleting` event to forbid non-admin users from deleting resources. The event handler implementation would be pretty much the same as the code above (Module refers to a `TXDataServerModule` object):

```
Module.Events.OnEntityDeleting.Subscribe(procedure(Args: TEntityDeletingArgs)
  var User: IUserIdentity;
  begin
    User := TXDataOperationContext.Current.Request.User;
    if User = nil then
      raise EXDataHttpUnauthorized.Create('User not authenticated');
    if not User.Claims.Exists('admin') then
      raise EXDataHttpForbidden.Create('Not enough privileges');
  end
);
```

That applies to all entities. Of course, if you want to restrict the code to some entities, you can check the *Args.Entity* property to verify the class of object being deleted and perform actions accordingly.

Finally, another nice example for authorization and server-side events: suppose that every entity in your application has a property named "Protected" which means only admin users can see those entities. You can use a code similar to the one above to refuse requests that try to modify, create or retrieve a protected entity if the requesting user is not admin.

But what about complex queries? In this case you can use the `OnEntityList event`, which will provide you with the Aurelius criteria that will be used to retrieve the entities:

```
Module.Events.OnEntityList.Subscribe(procedure(Args: TEntityListArgs)
  var
    User: IUserIdentity;
    IsAdmin: Boolean;
  begin
    User := Args.Handler.Request.User;
    IsAdmin := (User <> nil) and User.Claims.Exists('admin');
    if not IsAdmin then
      Args.Criteria.Add(not Linq['Protected']));
  end
);
```

The code above simply checks if the requesting user has elevated privileges. If it does not, then it adds an extra condition to the criteria (whatever the criteria is) which filters only the entities that are not protected. So non-admin users will not see the protected entities in the server response.

Using Authentication Credentials with TXDataClient

If you are using TXDataClient from a Delphi application to access the XData server, you can simply use the OnSendingRequest event to add authentication credentials (the token you retrieved from the server):

```
var
  Login: ILoginService;
  JwtToken: string;
begin
   JwtToken := Client.Service<ILoginService>.Login(edtUser.Text,
edtPassword.Text);
   Client.HttpClient.OnSendingRequest := procedure(Req: THttpRequest)
     begin
       Req.Headers.SetValue('Authorization','Bearer ' + JwtToken);
     end;
end;
```

Other Tasks

This chapter explains basic tasks you can do with XData in code. It basically explains how to setup and start the server and how to do some basic configuration using the available classes and interfaces. The following topics describe the most common XData programming tasks and main classes and interfaces.

Creating an XData Server

TMS XData Server is based on the TMS Sparkle framework. The actual XData Server is a Sparkle server module that you add to a 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 XData Server, just create and add a XData Server Module (*TXDataServerModule* class, declared in unit `XData.Server.Module`) to the Sparkle HTTP Server. The following code illustrates how to create and run the server. Note that the code that creates the XData server module is not displayed here. You should refer to the "XData Server Module" topic to learn about how to create the module.

```
uses
  {...},
  Sparkle.HttpSys.Server, XData.Server.Module;

function CreateXDataServerModule(const BaseUrl: string): TXDataServerModule;
begin
  // Create and return the TXDataServerModule here,
  // using the BaseUrl as the server address
end;

var
  Module: TXDataServerModule;
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;
  Module := CreateXDataServerModule('http://server:2001/tms/xdata');
  Server.AddModule(Module);
  Server.Start;
  ReadLn;
  Server.Stop;
  Server.Free;
end;
```

The code above will create and start an XData server that will receive and respond to HTTP requests at the address "http://server:2001/tms/xdata".

TXDataServerModule

To create an XData server, you need to add a *TXDataServerModule* object to the Sparkle HTTP Server. As with any Sparkle module, it will have a base (root) URL associated with it, and it will respond to requests sent to any URL which matches the root URL of the module.

The TXDataServerModule is the main XData class, because it is the one which receives and handles all HTTP requests. So in a way, that is the class that implements the XData server. Although it is a very important class, its usage is very simple. You need to create the class using one of the overloaded constructors (TXDataServerModule is declared in unit `XData.Server.Module`):

```
constructor Create(const ABaseUrl: string); overload;
constructor Create(const ABaseUrl: string; AConnectionPool: IDBConnectionPool); o
verload;
constructor Create(const ABaseUrl: string; AConnectionPool: IDBConnectionPool;
  AModel: TXDataAureliusModel); overload;
constructor Create(const ABaseUrl: string; AConnection: IDBConnection); overload;
constructor Create(const ABaseUrl: string; AConnection: IDBConnection;
  AModel: TXDataAureliusModel); overload;
```

In summary, you must provide the base URL and optionally an IDBConnectionPool interface so that the server can retrieve IDBConnection interfaces to operate with the database (if database connectivity is desired). For example:

```
XDataServerModule := TXDataServerModule.Create('http://server:2001/tms/xdata',
  TDBConnectionPool.Create(50,
    TDBConnectionFactory.Create(
      function: IDBConnection
      var
        MyDataModule: TMyDataModule;
      begin
        MyDataModule := TMyDataModule.Create(nil);
        Result := TFireDacConnectionAdapter.Create(MyDataModule.FDConnection1, My
DataModule);
      end
)));
```

The example above creates the server with the root URL "http://server:2001/tms/xdata" providing a connection pool of a maximum of 50 simultaneous connections. The database connection used will be a FireDac TFDConnection component named *FDConnection1*, declared in a data module named *TMyDataModule*. That is all you need to set up for your XData server to run and to expose your Aurelius objects.

The first overloaded constructor, used in the previous example, takes just two parameters: the base url and the connection pool. The other overloaded versions are just variations that provide different settings.

If you have multiple Aurelius mapping models in your application, you can optionally use an entity model different from default, or explicitly build an entity model and provide it in the constructor for specific mapping. This allows for better control which classes and mapping settings are available from the XData server. If the model is not provided, XData uses the default entity model which in turn uses the default Aurelius model. Note that the model passed to the constructor will not be owned by the server module and must be destroyed manually. For example, the following code creates a module using the "Sample" model:

```
XDataServerModule := TXDataServerModule.Create('http://server:2001/tms/xdata',
  MyConnectionPool, TXDataAureliusModel.Get('Sample'));
```

There are also versions of the *Create* constructor that receive an *IDBConnection* interface instead of an IDBConnectionPool. Those are easy-to-use variations that internally create a connection pool with a single connection (no simultaneous connections available). It is an easy approach for testing and debug purposes, but should not be used in production environments because performance might not be ideal.

Properties

Name	Description
UserName: string Password: string	TXDataServerModule provides these properties to specify UserName and Password required by the server using Basic Authentication. By default, these values are empty which means no authentication is performed and any client can access server resources and operations. When basic authentication is used, be sure to use HTTP secure (HTTPS) if you do not want your user name or password to be retrieved by middle-man attacks. If you do not use it, both user name and password are transmitted in plain text in HTTP requests. These properties provide a very limited basic authentication mechanism. For a more advanced variant, you should use the TMS Sparkle built-in Basic Authentication mechanism.
AccessControlAllowOrigin: string	Specifies the accepted client hosts for which CORS will be enabled. If you want to accept any client connection, set this property value to '*'. This will enable CORS in the server including proper responses to preflighted requests.
DefaultExpandLevel: integer	Defines the minimum level that associated entities will be expanded (included inline) in JSON responses. Default value is 0 meaning that all associated entities will be represented as references unless specified otherwise. Clients can override this value by using xdata-expand-level header.
Events: TXDataModuleEvents	Container for server-side events.

Name	Description
PutMode: TXDataPutMode	Defines how PUT will be implemented at server side with Aurelius: Either *TXDataPutMode.Update* or *TXDataPutMode.Merge* method (default). You will rarely need to change this property unless to ensure backward compatibility with older versions. This property value can be overridden in a specific request by using xdata-put-mode HTTP Request header.
SerializeInstanceRef: TInstanceRefSerialization	Controls how instance reference ($ref) will appear in JSON response. See below for options. This property value can be overridden in a specific request by using xdata-serialize-instance-ref HTTP Request header.
SerializeInstanceType: TInstanceTypeSerialization	Controls whenever the entity/object type appears in the JSON response (property annotation "xdata.type"). See below for options. This property value can be overridden in a specific request by using xdata-serialize-instance-type HTTP Request header.
UnknownMemberHandling: TUnknownMemberHandling	Defines server behavior when receiving JSON from the client with a property that is not known for that request. For example, JSON representing an entity with a property that does not belong to that entity. See below for options.
RoutingPrecedence: TRoutingPrecedence	Specifies which route to use when there is a conflict between URL endpoints defined by automatic CRUD endpoints and service operations . See below for options.
EnableEntityKeyAsSegment: Boolean	When True, it's possible to address single entities by using the URL format "*/entityset/id*" - in addition to the default "*/entityset(id)*". Default is False.
SwaggerOptions: TSwaggerOptions SwaggerUIOptions: TSwaggerUIOptions	Provide access to configure Swagger and SwaggerUI behavior. See more information at OpenAPI/Swagger Support.

TInstanceRefSerialization

- *TInstanceRefSerialization.Always*: $ref is always used if the same instance appears again in the JSON tree. This mode is more optimized for use with TXDataClient, and is the default option.
- *TInstanceRefSerialization.IfRecursive*: $ref is only used if the instance appears as an associated object of itself. If the instance appears in a non-recursive way, $ref is not used and the instance is fully serialized inline instead. This mode is more suited for JavaScript and other non-Delphi clients so those clients do not need to resolve the $ref objects.

TInstanceTypeSerialization

- *TInstanceTypeSerialization.Always*: The xdata.type annotation always appears in JSON responses.
- *TInstanceTypeSerialization.IfNeeded*: The xdata.type annotation is only present if the entity/object type is a descendant of the expected type. For example, suppose a GET request is performed in url Customers. For any entity in JSON that is of type Customer, the annotation will not present. If the entity in JSON is a descendant of Customer (e.g., DerivedCustomer) then xdata.type apppears. Basically, it means that xdata.type is implicit when absent and is of the expected type of the request/specification.

TUnknownMemberHandling

- *TUnknownMemberHandling.Error*: An *InvalidJsonProperty* error is raised if JSON contains an invalid property. This is the default behavior.
- *TUnknownMemberHandling.Ignore*: Invalid properties in JSON will be ignored and the request will be processed.

TRoutingPrecedence

- *TRoutingPrecedence.Crud*: The URL from automatic CRUD endpoint will be used if there is a URL conflict with service operation endpoints. This is the default behavior.
- *TRoutingPrecedence.Service*: The URL from service operation will be used if there is a URL conflict with automatic CRUD endpoints.

Methods

Name	Description
procedure SetEntitySetPermissions(const EntitySetName: string; Permissions: TEntitySetPermissions)	Specify the permissions for a specified entity set.

IDBConnectionPool Interface

The *IDBConnectionPool* interface is used by the XData server module to retrieve an IDBConnection interface used to connect to a database. As client requests arrive, the XData server needs an *IDBConnection* interface to connect to the database server and execute the SQL statements. It achieves this by trying to acquire an IDBConnection interface from the IDBConnectionPool interface. Thus, providing an IDBConnectionPool interface to the XData server module is mandatory for this to work.

The IDBConnectionPool interface is declared in the unit `Aurelius.Drivers.Interfaces` and contains a single *GetConnection* method:

```
IDBConnectionPool = interface
  function GetConnection: IDBConnection;
end;
```

The easiest way to create an IDBConnectionPool interface is by using the *TDBConnectionPool* class which implements the IDBConnectionPool interface. To instantiate it, you need to call the *Create* constructor passing an IDBConnectionFactory interface which will be used by the connection pool to create a new connection if needed. Alternatively, you can pass an anonymous method that creates and returns a new IDBConnection interface each time it is called.

Also, you need to specify the maximum number of IDBConnection interfaces (database connections) available in the pool. Whenever a client request arrives, the XData server acquires an IDBConnection from the pool, does all the database processing with it, and then returns it to the pool. If many simultaneous requests arrive, the pool might run out of available connections when the number of acquired IDBConnection interfaces reaches the maximum number specified. If that happens, the client request will wait until a new connection is available in the pool. TDBConnectionPool class is declared in the unit `XData.Aurelius.ConnectionPool`.

The following code illustrates how to create an IDBConnectionPool interface. It uses a function named *CreateMyConnectionFactory* which is not shown in this example. To learn how to create such interface, please refer to IDBConnectionFactory topic.

```
uses
  {...}, Aurelius.Drivers.Interfaces, XData.Aurelius.ConnectionPool;

var
  ConnectionPool: IDBConnectionPool;
  ConnectionFactory: IDBConnectionFactory;
begin
  ConnectionFactory := CreateMyConnectionFactory; //  Creates
IDBConnectionFactory
  ConnectionPool := TDBConnectionPool.Create(
    50, // maximum of 50 connections available in the pool
        // Define a number that best fits your needs
    ConnectionFactory
  );

  // Use the IDBConnectionPool interface to create the XData server module
end;
```

Alternatively, you can just provide an anonymous method that creates the IDBConnection instead of providing the IDBConnectionFactory interface:

```
uses
  {...}, Aurelius.Drivers.Interfaces, XData.Aurelius.ConnectionPool;

var
  ConnectionPool: IDBConnectionPool;
begin
  ConnectionPool := TDBConnectionPool.Create(
    50, // maximum of 50 connections available in the pool
    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 IDBConnectionPool interface to create the XData server module
end;
```

If you do not need a pooling mechanism but just want one database connection to be created for each time someone asks for a connection from the pool, you can use the TDBConnectionFactory class. It also implements the IDBConnectionPool interface:

```
var
  ConnectionPool: IDBConnectionPool;
begin
  ConnectionPool := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        MyDataModule: TMyDataModule;
      begin
        // Creates a datamodule which contains a
        // TSQLConnection 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 := TDBExpressConnectionAdapter.Create(
          MyDataModule.SQLConnection1, MyDataModule);
      end
    ));
end;
```

IDBConnectionFactory Interface

The *IDBConnectionFactory* interface is used to create an IDBConnectionPool interface used by the XData module. As client requests arrive, the XData server needs to retrieve an IDBConnection interface from the IDBConnectionPool so it can perform operations on the database. The connection pool creates a new IDBConnection by calling *IDBConnectionFactory.CreateConnection* method.

The IDBConnectionFactory interface is declared in unit `Aurelius.Drivers.Interfaces`, and it contains a single *CreateConnection* method:

```
IDBConnectionFactory = interface
  function CreateConnection: IDBConnection;
end;
```

The easiest way to create such an interface is using the *TDBConnectionFactory* class which implements the IDBConnectionFactory interface. To create a TDBConnectionFactory object, you just need to pass an anonymous method that creates and returns a new IDBConnection interface each time it is called. The TDBConnectionFactory class is declared in the unit `Aurelius.Drivers.Base`.

The IDBConnection interface is part of the TMS Aurelius library used by XData. You can refer to the TMS Aurelius documentation to learn how to create the IDBConnection interface.

In the following example, the factory will create an IDBConnection pointing to a Microsoft SQL Server database using a dbExpress connection. You can connect to many other database servers (Oracle, Firebird, MySQL, etc.) using many different database-access components (FireDac, dbExpress, UniDac, ADO, etc.). Please refer to Database Connectivity topic on TMS Aurelius

documentation to learn about all those options. Regardless of what you use, the structure of the following code will be the same. What will change only is the content of the *function: IDBConnection*.

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

var
  ConnectionFactory: IDBConnectionFactory;
begin
  ConnectionFactory := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        conn: 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
        conn := TSQLConnection.Create(nil);
        conn.DriverName := 'MSSQL';
        conn.GetDriverFunc := 'getSQLDriverMSSQL';
        conn.VendorLib := 'sqlncli10.dll';
        conn.LibraryName := 'dbxmss.dll';
        conn.Params.Values['HostName'] := 'server';
        conn.Params.Values['Database'] := 'master';
        conn.Params.Values['User_Name'] := 'sa';
        conn.Params.Values['Password'] := 'password';
        conn.LoginPrompt := false;
        Result := TDBExpressConnectionAdapter.Create(SQLConn, true);
      end
    ));

  // Use the ConnectionFactory interface to create an IDBConnectionPool interface
end;
```

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

```
var
  ConnectionFactory: IDBConnectionFactory;
begin
  ConnectionFactory := TDBConnectionFactory.Create(
      function: IDBConnection
      var
        MyDataModule: TMyDataModule;
      begin
        // Creates a datamodule which contains a
        // TSQLConnection 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 := TDBExpressConnectionAdapter.Create(
          MyDataModule.SQLConnection1, MyDataModule);
      end
    ));

  // Use the ConnectionFactory interface to create an IDBConnectionPool interface
end;
```

OpenAPI/Swagger Support

The XData server can optionally provide a JSON file containing the OpenAPI Specification (OAS, formerly Swagger) for your whole server API. This opens up a lot of possibilities; usage of several tools of the OpenAPI ecosystem is possible. The main one is Swagger UI: it is web front-end to describe and test your API.

Enabling support for OpenAPI (Swagger)

To enable OpenAPI support in your server, just set the property *SwaggerOptions.Enabled* to true in your TXDataServer component, either in object inspector or from code:

```
XDataServer1.SwaggerOptions.Enabled := True;
```

Alternatively, if you are using *TXDataServerModule* instead, you can just use the unit `XData.OpenAPI.Service` and call the method *RegisterOpenAPIService* anywhere in your server application:

```
uses {...}, XData.OpenAPI.Service;

{...}
  RegisterOpenAPIService;
```

Retrieving the OpenAPI Specification (OAS) file

The OAS file is available through a GET request to the URL "*/openapi/swagger.json*" relative to your server root URL. For example, if your server root is *http://server:2001/tms/xdata/*, then you will be able to access the file from this URL:

```
GET http://server:2001/tms/xdata/openapi/swagger.json
```

Excluding methods from Swagger

You can flag some service contract methods to be excluded from Swagger document, by simply adding the *[SwaggerExclude]* attribute to the method:

```
[ServiceContract]
[Route('arith')]
IArithmeticService = interface(IInvokable)
  ['{9E343ABD-D97C-4411-86BF-AD2E13BE71F3}']
  [HttpGet] function Add(A, B: Double): Double;
  [HttpGet] function Subtract(A, B: Double): Double;
  [SwaggerExclude]
  [HttpGet] function NotThisFunctionPlease: Integer;
end;
```

In the example above, methods *Add* and *Subtract* will be added to the documentation, but method *NotThisFunctionPlease* will not appear.

Enabling Swagger UI

XData server can optionally publish and endpoint to provide a SwaggerUI web-based interface that works as both a full documentation of your API as well a test interface. Swagger UI is a web-based front-end to dynamically document and test your API. Quoting their website:

> "Swagger UI allows anyone - be it your development team or your end consumers - to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your Swagger specification, with the visual documentation making it easy for back end implementation and client side consumption."

To enable SwaggerUI support in your server, just set the property *SwaggerUIOptions.Enabled* to true in your TXDataServer component, either in object inspector or from code:

```
XDataServer1.SwaggerUIOptions.Enabled := True;
```

Alternatively, if you are using *TXDataServerModule* instead, you can just use the unit `XData.SwaggeUI.Service` and call the method *RegisterSwaggerUIService* anywhere in your server application:

```
uses {...}, XData.SwaggerUI.Service;

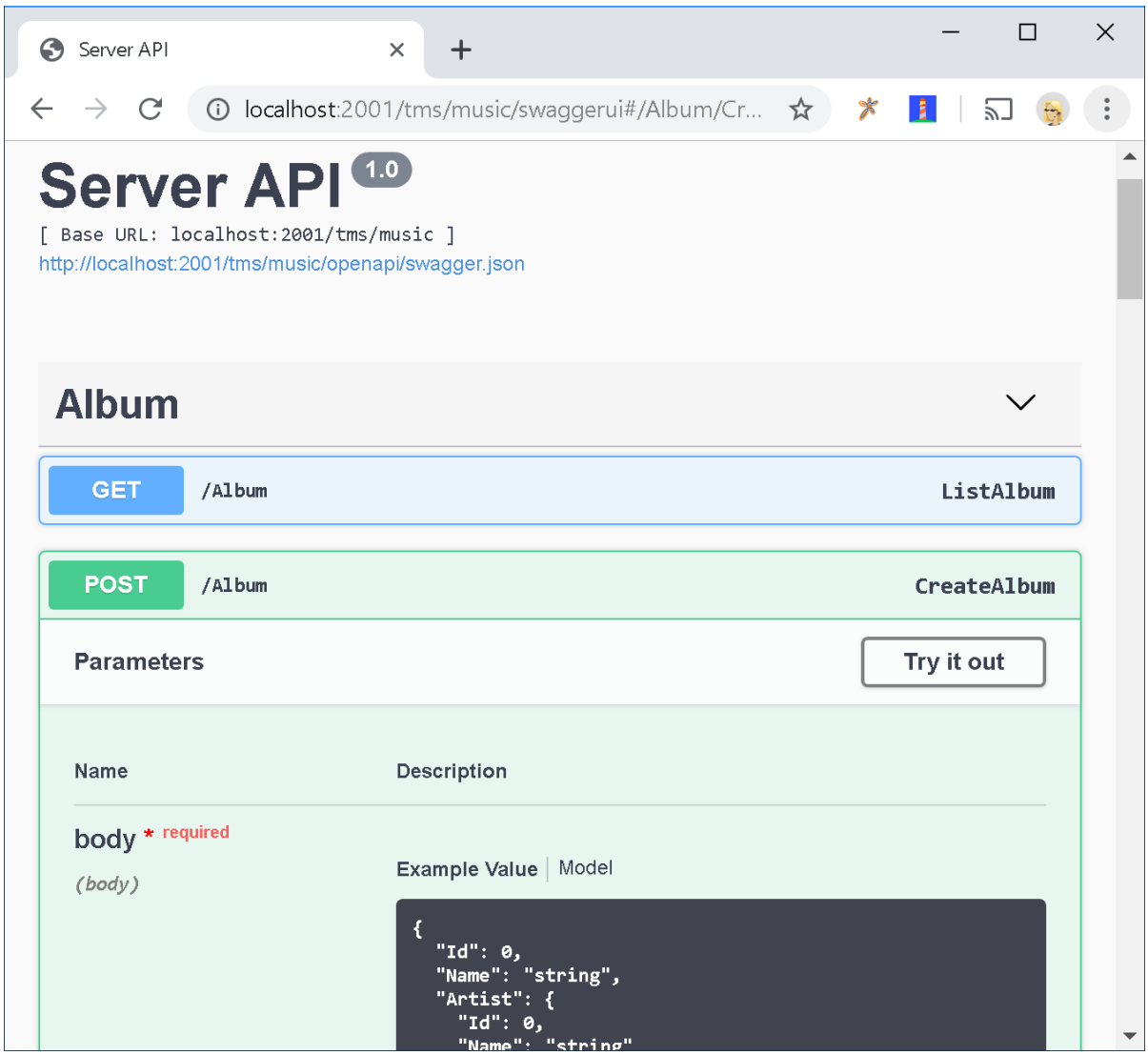
{...}
  RegisterSwaggerUIService;
```

Using SwaggerUI

Once enabled, SwaggerUI is available in the address "*/swaggerui*" relative to your server base path. Just open your browser and go to the address:

`http://server:2001/tms/xdata/swaggerui`

It will display an interface like this:



Configuring Swagger (OpenAPI Specification)

For the Swagger specification, you can use *SwaggerOptions* property of either TXDataServer or TXDataServerModule to configure the specification returned by the server:

```
XDataModule.SwaggerOptions.AuthMode := TSwaggerAuthMode.Jwt;
```

Only available property is *AuthMode* and options are *TSwaggerAuthMode.Jwt* or *None*. When Jwt is enabled, a new security definition named *jwt* requiring an Authorization header will be added to all requests.

You can also configure the specification by passing parameters in the query part of the URL, and they can be one of the following:

Name	Description
ExcludeEntities	Boolean parameter. Default is False. When True, the specification will not contain the automatic CRUD operations on entities provided automatically by XData (entity resources). Example: `/openapi/swagger.json?ExcludeEntities=True`
ExcludeOperations	Boolean parameter. Default is False. When True, the specification will not contain any service operations. Example: `/openapi/swagger.json?ExcludeOperations=True`
AuthMode	String parameter. Options are "none" and "jwt". When jwt is specified, a new security definition named *jwt* requiring an Authorization header will be added to all requests. Example: `/openapi/swagger.json?authmode=jwt`

Configuring SwaggerUI

The SwaggerUI interface can also be configured using the *SwaggerUIOptions* property of either TXDataServer or TXDataServerModule. Available properties are:

Name	Description
ShowFilter	Boolean parameter. Default is False. When True, the SwaggerUI will display a filter edit box to search for API operations.
DocExpansion	Enumerated parameter. Specifies the default expand level of the listed API operations. Valid values are: `TSwaggerUIExpansion = (List, None, Full)` Default value is *List*.

Examples:

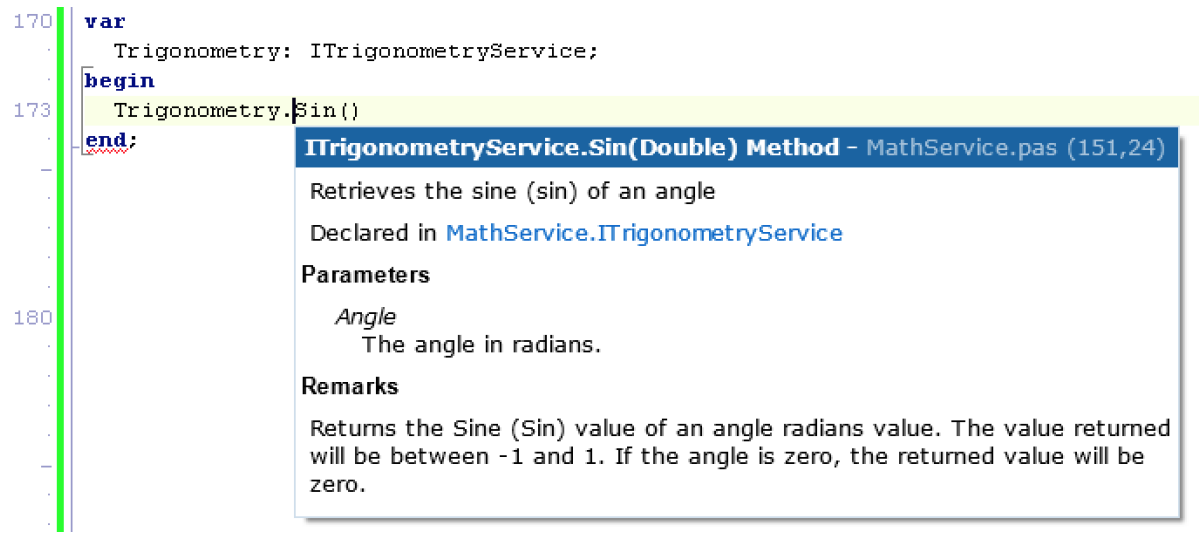
```
XDataModule.SwaggerUIOptions.Filter := True;
XDataModule.SwaggerUIOptions.DocExpansion := TSwaggerUIExpansion.None;
```

Using XML Documentation with Swagger

A good (if not the best) way to document your source code is to use XML Documentation Comments. In the interfaces and methods that build your service contract, you can simply add specific XML tags and content, like this:

```
/// <summary>
///   Retrieves the sine (sin) of an angle
/// </summary>
/// <remarks>
///  Returns the Sine (Sin) value of an angle radians value.
///  The value returned will be between -1 and 1.
///  If the angle is zero, the returned value will be zero.
/// </remarks>
/// <param name="Angle">
///   The angle in radians.
/// </param>
function Sin(Angle: Double): Double;
```

And Delphi IDE will automatically use it for Help Insight, showing you information about the method on-the-fly. For example, if some developer is trying to use the *Sin* method of your API, information will be conveniently displayed:

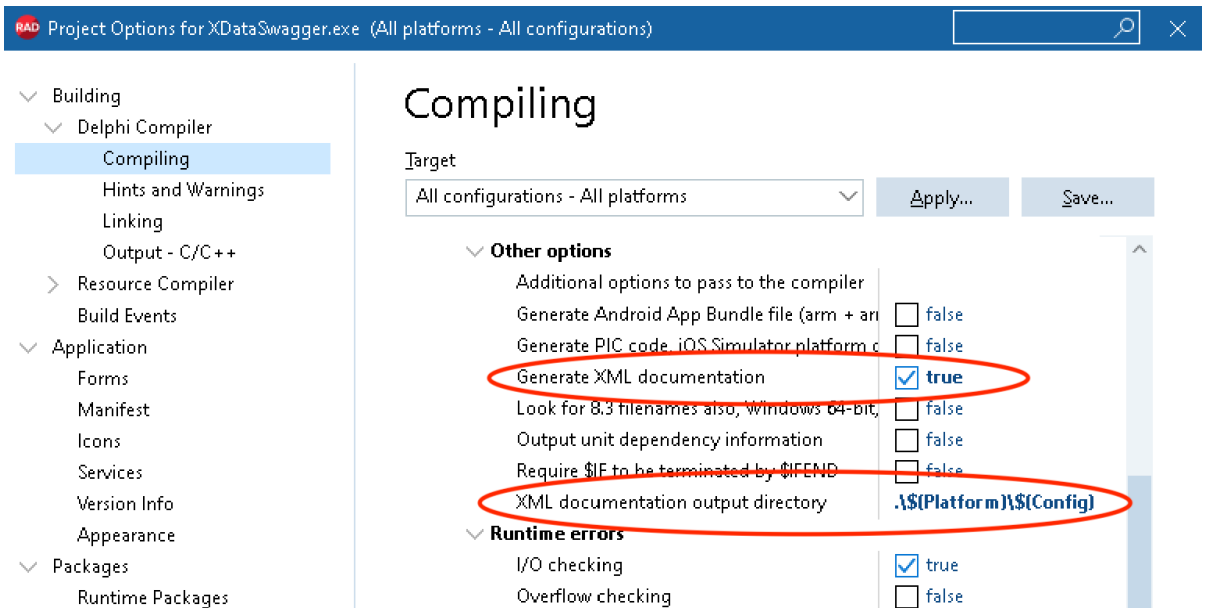


The good news is that, with XData, you can use such XML comments in the Swagger document and Swagger-UI that are generated automatically by XData, improving even more your REST API documentation. Since the API endpoints are generated directly from the interfaced and methods, XData knows exactly the meaning of each documentation and can map it accordingly to Swagger.

Enabling generation of XML documentation files

XData needs to read the XML files with the comments to import it into the Swagger document. You need to tell Delphi compiler to generate the XML documentation files.

In your project options (Delphi menu Project | Options, or Shift+Ctrl+F11), go to "Building, Delphi Compiler, Compiling" (for Delphi 10.4.1 Sydney. Previous Delphi version might differ), then enable option "*Generate XML documentation*". You might also want to explicitly set the directory where the XML files will be generated, in option "*XML documentation output directory*". It's recommended to use ".*$(Platform)**$(Config)*", this way the XML files will be in the same folder as the executable.



Importing XML documentation in Swagger

XML documentation usage in Swagger can be enabled with a single line of code:

```
uses {...}, XData.Aurelius.ModelBuilder;

...

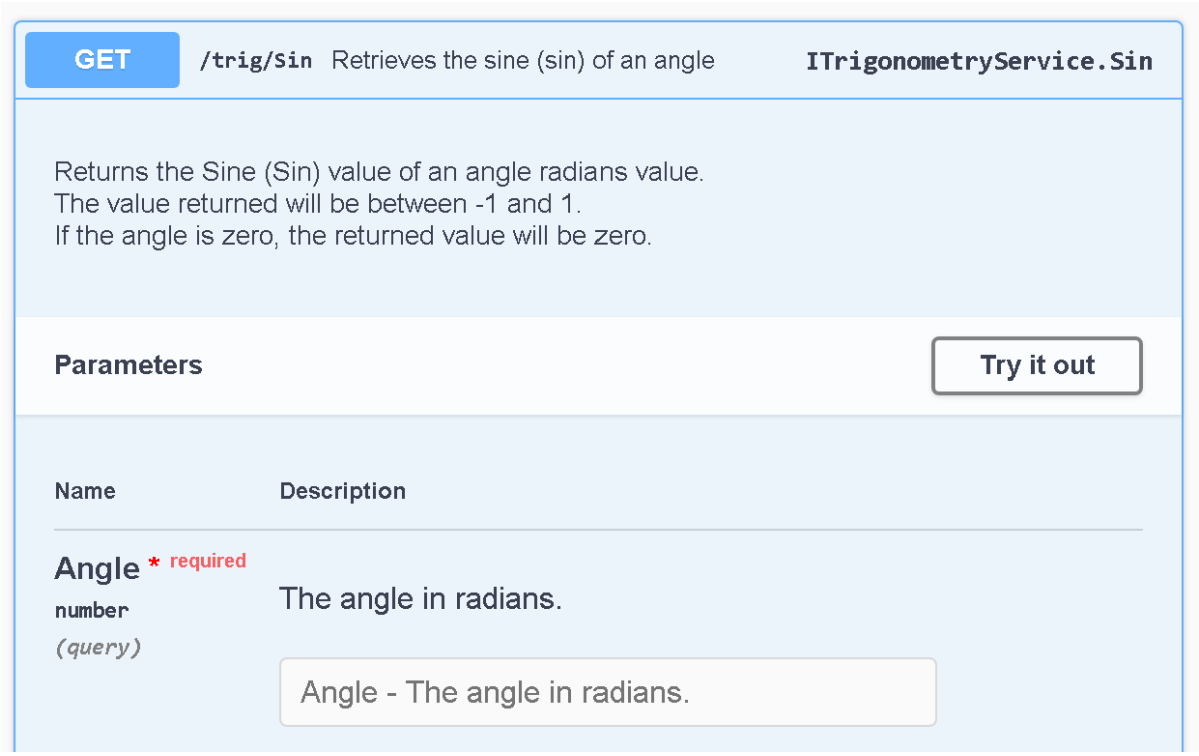
  TXDataModelBuilder.LoadXmlDoc(XDataServer.Model);
```

Add the line at the beginning of your application, be it in your dpr file, or *initialization* section of some unit, or in the *OnCreate* event of your TDataModule that contains your XData/Sparkle components. In the first parameter you must provide the XData model you want to import XML files to. In the example above, *XDataServer* is a TXDataServer component. If you are using multi-model design, just provide the proper model there.

LoadXmlDoc will try to find the XML files in the *same directory as your application is located*. If the XML files are in a different folder, you can specify it explicitly using a second parameter:

```
TXDataModelBuilder.LoadXmlDoc(XDataServer.Model, 'C:\MyXMLFiles');
```

Once you do that, if you check your Swagger documentation via Swagger-UI, you will see something like this:



Note how the content of *summary* tag goes directly at the right of the *GET* button, as a summary for the endpoint.

The content of *remarks* tag is the detailed description of the endpoint.

And the content of each *param* tag explains each respective parameter. Sweet!

Using different documentation for Help Insight and Swagger

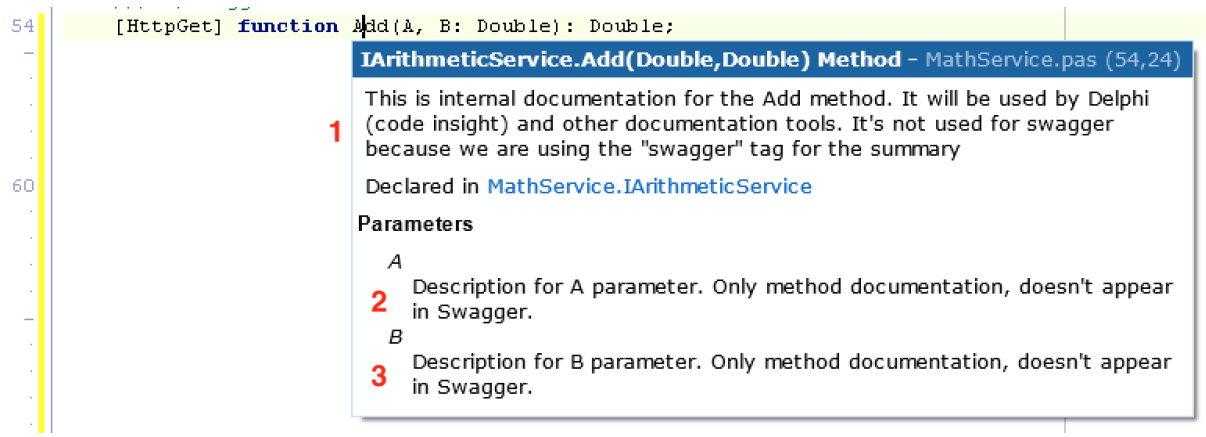
Reusing the same XML comments is nice as you don't repeate yourself. Document your code just once, and the same documentation is used for documenting your Delphi interfaces (Delphi developments) and your REST API (API consumer development).

But, if for some reason you want to use different documentation content for Delphi developers and for REST API users, that's also possible. You can use the specific *swagger* tag to fill specific parts of the documentation. You then use the *name* attribute to differentiate between *swagger* tags.

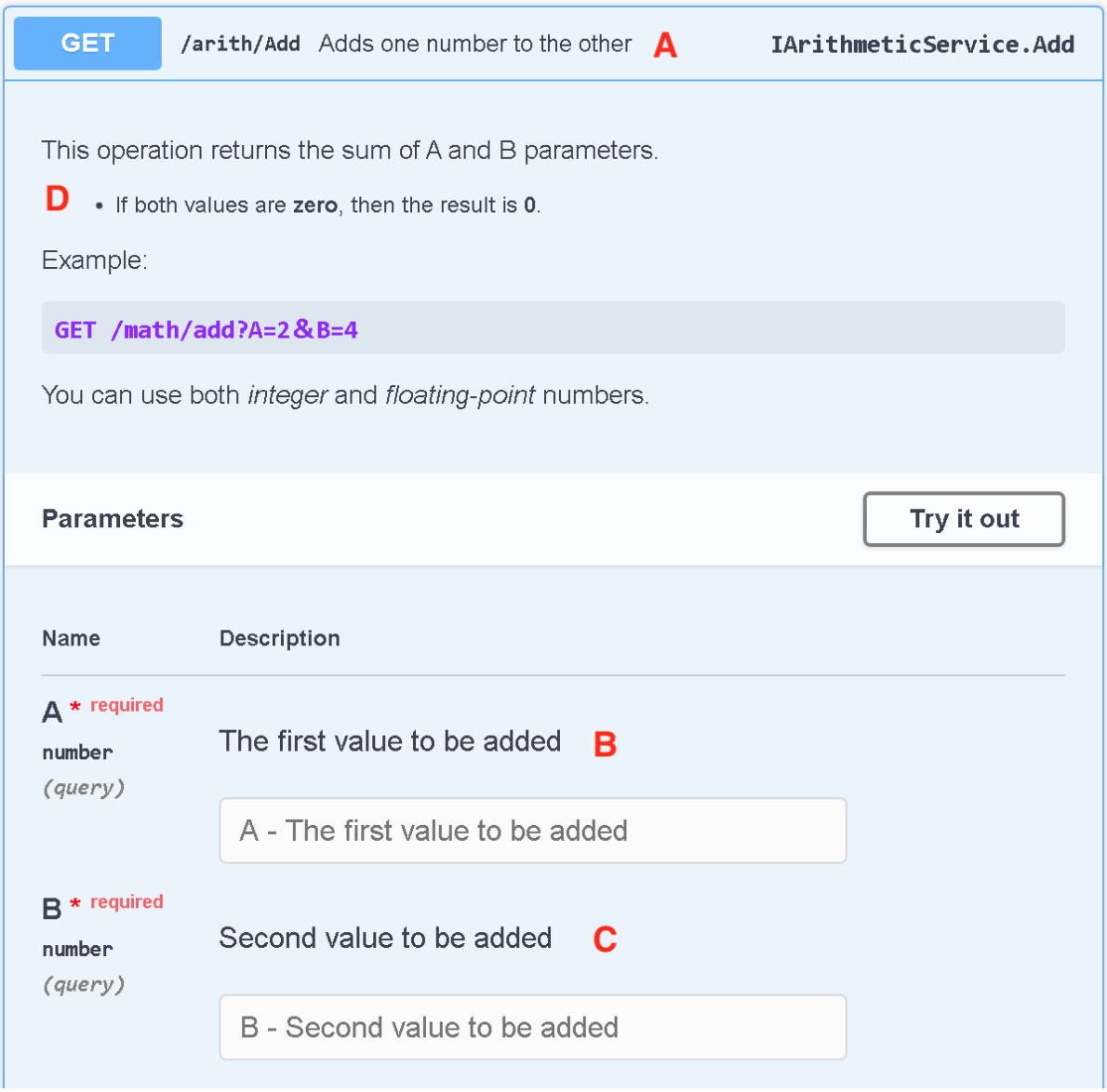
For example, suppose the following documentation:

```
/// <summary>
///   This is internal documentation for the Add method. It will be used by Delphi (code ins
/// 1 and other documentation tools. It's not used for swagger because we are using
///   the "swagger" tag for the summary
/// </summary>
/// <param name="A">
/// 2 Description for A parameter. Only method documentation, doesn't appear in Swagger.
/// </param>
/// <param name="B">
/// 3 Description for B parameter. Only method documentation, doesn't appear in Swagger.
/// </param>
///
/// <swagger>
/// A Adds one number to the other
/// </swagger>
/// <swagger name="param-A">
/// B The first value to be added
/// </swagger>
/// <swagger name="param-B">
/// C Second value to be added
/// </swagger>
/// <swagger name="remarks">
/// This operation returns the sum of A and B parameters.
/// D
/// - If both values are **zero**, then the result is **0**.
///
/// Example:
///
///     GET /math/add?A=2&#65286;B=4
///
/// You can use both *integer* and *floating-point* numbers.
/// </swagger>
[HttpGet] function Add(A, B: Double): Double;
```

Note that tags *summary* (1) and *param* (2 and 3) are the regular XML documentation tags. They will be used for Help Insight:



And *swagger* tags with no name attribute (A), or name *param-A* (B), *param-B* (C) and *remarks* (D) will be used exclusively for Swagger documentation:



Customizing tags

You can also customize the tags in Swagger. Endpoints are grouped together inside a tag, which can have a name and description.

By default, the name of the tag will be path segment of the interface service. But you can change it using *swagger* tag with *tag-name* attribute.

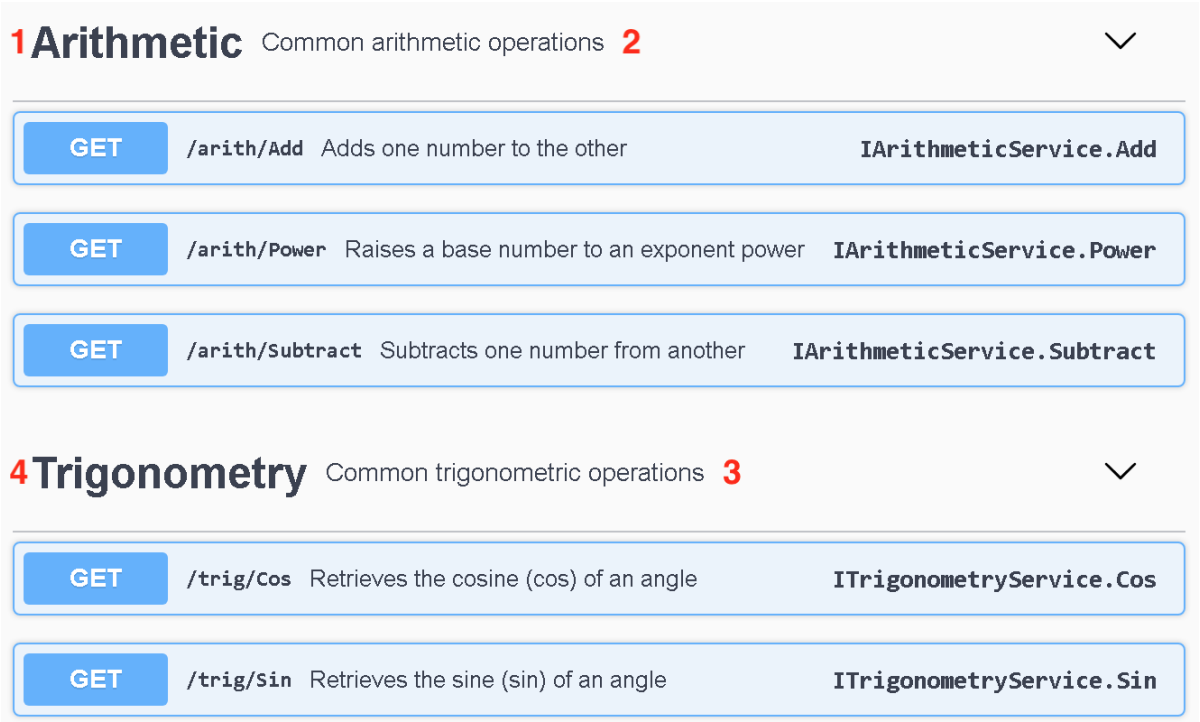
The description of the tag by default is empty, but you can define it using the regular *summary* tag, or optionally using the *swagger* tag with *tag-description* attribute.

Consider the following documentation for both *IArithmenticService* and *ITrigonometryService*:

```
type
  /// <summary>
  ///  An interface that provides arithmentic functions. This documentation
  ///  doesn not appear in Swagger, because we use <c>swagger</c> tags
  /// </summary>
  /// <swagger name="tag-name">Arithmetic</swagger>  <-- 1
  /// <swagger name="tag-description">
  ///  Common arithmetic operations  <-- 2
  /// </swagger>
  [ServiceContract]
  [Route('arith')]
  IArithmeticService = interface(IInvokable)
    ['{9E343ABD-D97C-4411-86BF-AD2E13BE71F3}']
    XML Documentation
    [HttpGet] function Add(A, B: Double): Double;
    XML Documentation
    [HttpGet] function Subtract(A, B: Double): Double;
    XML Documentation
    [HttpGet] function Power(Base, Exponent: Double): Double;
  end;

  /// Common trigonometric operations  <-- 3
  /// <swagger name="tag-name">Trigonometry</swagger>  <-- 4
  [ServiceContract]
  [Route('trig')]
  ITrigonometryService = interface(IInvokable)
    ['{B9BB0872-A63A-4D06-BF21-3DB1F8A6699A}']
    XML Documentation
    [HttpGet] function Sin(Angle: Double): Double;
    XML Documentation
    [HttpGet] function Cos(Angle: Double): Double;
  end;
```

The above tags will generate the following output in Swagger UI:

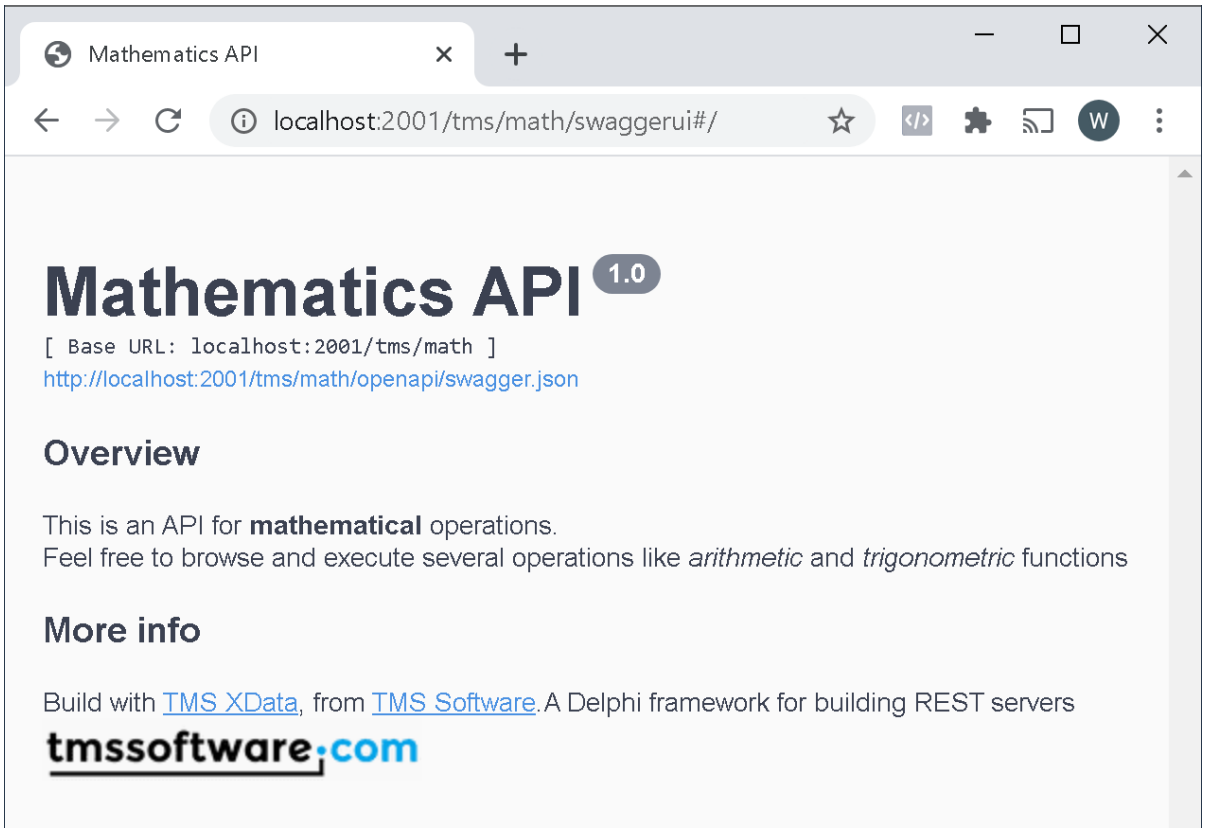


Customizing document header and description

XData also takes the model name, version and description into account to build the final document. You can configure such settings directly by changing some properties of the XData model (remember that *XDataServer* in this example is a TXDataServer component):

```
  XDataServer.Model.Title := 'Mathematics API';
  XDataServer.Model.Version := '1.0';
  XDataServer.Model.Description :=
    '### Overview'#13#10 +
    'This is an API for **mathematical** operations. '#13#10 +
    'Feel free to browse and execute several operations like *arithmetic* ' +
    'and *trigonometric* functions'#13#10#13#10 +
    '### More info'#13#10 +
    'Build with [TMS XData](https://www.tmssoftware.com/site/xdata.asp), from ' +
    '[TMS Software](https://www.tmssoftware.com).' +
    'A Delphi framework for building REST servers'#13#10 +
    '[![TMS Software](https://download.tmssoftware.com/business/tms-logo-
small.png)]' +
    '(https://www.tmssoftware.com)';
```

Note that you can use Markdown syntax to format the final text. Here is what it will look like:



Web Applications with TMS Web Core

TMS Web Core is the TMS Software framework for building web applications using Delphi. It allows you to create pure HTML/JS Single-Page-Applications that runs in your browser.

Even though the web application generated by TMS Web Core can run 100% stand alone in the browser, in many scenarios it needs data to work with, and such data needs to be retrieved from a server (and eventually sent back for modifications). Usually this data communication is done through a REST API Server - the web client performs requests using HTTP(S), and send/receive data in JSON format.

TMS XData is the ideal back-end solution for TMS Web Core. It not only allows you to create REST API servers from an existing database in an matter of minutes, but is also provides the most complete and high-level client-side framework for TMS Web Core, including Delphi design-time support with visual components, a *TXDataWebClient* component that abstracts low-level HTTP/JSON requests in a very easy and high-level way of use, and a dataset-like optional approach that simply feels home to Delphi users but still uses modern REST/JSON requests behind the scenes.

The following topics cover in details how to use TMS XData Web-Client Framework and make use of TMS XData servers from TMS Web Core applications.

Setting Up the Connection with TXDataWebConnection

TXDataWebConnection is the first (and key) component you need to use to connect to XData server from TMS Web Core applications. Simply drop the component in the form set its *URL* property to the root URL of XData server:

```
XDataWebConnection1.URL := 'http://localhost:2001/tms/music';
```

TIP
Even though the examples above and below will show setting properties from code, since TXDataWebConnection is available at design-time, you can set most of the properties described here at design-time in object inspector, including testing the connection.

Then you perform the connection setting *Connected* property to true:

```
DataWebConnection1.Connected := True;
```

It's as simple as that. However, for web applications, you must be aware that **all** connections are performed **asynchronously**. This means that you can't be sure when the connection will be performed, and any code after Connected is set to true is not guaranteed to work if it expects the connection to be established. In the following code, for example, the second line will probably not work because the connection is probably not yet finished:

```
XDataWebConnection1.Connected := True;
// The following line will NOT work because connection
// is still being established asynchronously
PerformSomeRequestToXDataServer();
```

To make sure the component is connected and you can start performing requests to XData server, you should use *OnConnect* and *OnError* events:

From TMS Web Core 1.6 and on, you can also use the `OpenAsync` method using the `await` keyword. This will ensure the next line will be executed after `OpenAsync` is complete, even though it's executed asynchronously:

```
await(XDataWebConnection1.OpenAsync);
// The following line will be executed
// after the connection asynchronously established
PerformSomeRequestToXDataServer();
```

OnConnect and OnError events

When connecting, either one of those two events will be fired upon request complete. Use the OnConnect event to be sure the connection was performed and start communicating with XData server:

```
procedure TForm1.ConnectButtonClick(Sender: TObject);
begin
  XDataWebConnection1.URL := 'http://localhost:2001/tms/music';
  XDataWebConnection1.OnConnect := XDataWebConnection1Connect;
  XDataWebConnection1.OnError := XDataWebConnection1Error;
  XDataWebConnection1.Connected := True;
end;

procedure TForm1.XDataWebConnection1Connect(Sender: TObject);
begin
  WriteLn('XData server connected succesfully!');
  PerformSomeRequest;
end;

procedure TForm1.XDataWebConnection1Error(Error: TXDataWebConnectionError);
begin
  WriteLn('XData server connection failed with error: ' + Error.ErrorMessage);
end;
```

Open method

As an alternative to events, you can connect using *Open* method, which you pass two parameters: a callback for success and a callback for error.

```
procedure TForm1.ConnectButtonClick(Sender: TObject);

  procedure OnConnect;
  begin
    WriteLn('XData server connected succesfully!');
    PerformSomeRequest;
  end;

  procedure OnError(Error: TXDataWebConnectionError);
  begin
    WriteLn('XData server connection failed with error: ' + Error.ErrorMessage);
  end;

begin
  XDataWebConnection1.URL := 'http://localhost:2001/tms/music';
  XDataWebConnection1.Open(@OnConnect, @OnError);
end;
```

As stated previously, `OpenAsync` is the equivalent to `Open` that can be used with `await`:

```
procedure TForm1.ConnectButtonClick(Sender: TObject);
begin
  XDataWebConnection1.URL := 'http://localhost:2001/tms/music';
  try
    await(XDataWebConnection1.OpenAsync);
    WriteLn('XData server connected succesfully!');
    PerformSomeRequest;
  except
    on Error: Exception do
      WriteLn('XData server connection failed with error: ' +
Error.ErrorMessage);
  end;
end;
```

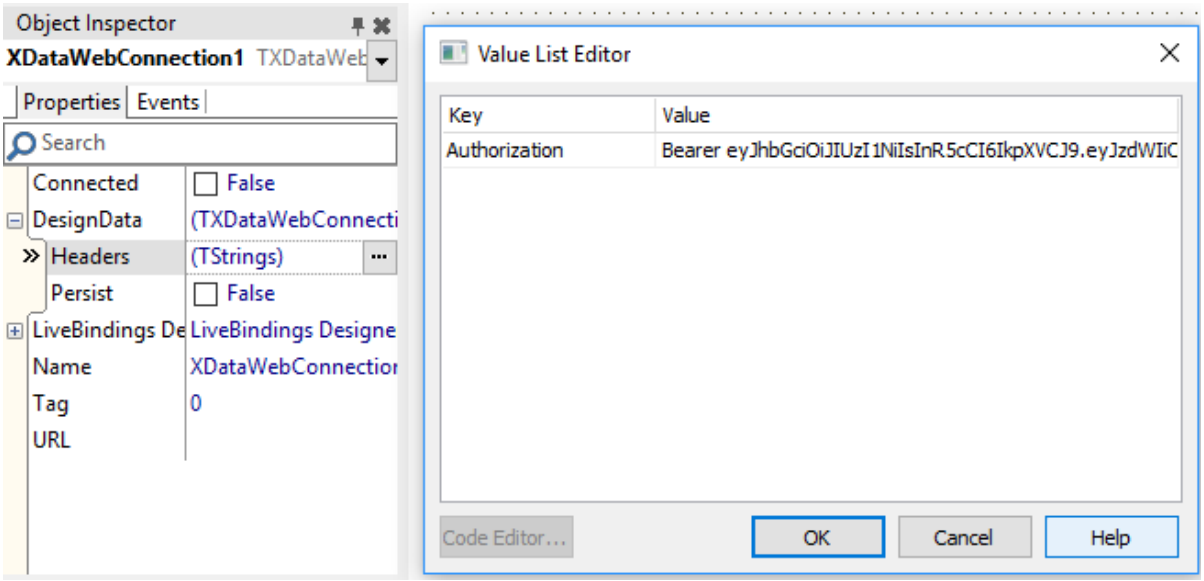
OnRequest event

OnRequest event is called before every request about to be sent to the XData server. This is an useful event to modify all requests at once. For example, it's often used to add authentication information the request, like a authorization token with a JWT header:

```
procedure TForm1.XDataWebConnection1Request(Request: TXDataWebConnectionRequest);
begin
  Request.Request.Headers.SetValue('Authorization', 'Bearer ' + LocalJWTToken);
end;
```

DesignData property

DesignData property is intended to be used just at design-time, as an opportunity for you to add custom headers to be sent to the server. Analogously to the OnRequest event, it's useful to add authorization header to the request so you can connect to XData server at design-time. To use it just click the ellipsis button in the *DesignData.Headers* property and add the headers you need.



Those headers will by default **not be saved** in the DFM, meaning you will lose that information if you close/open the project, or even the form unit. This is for security purposes. In the exceptional case you want information to be saved (and thus also loaded at runtime), you can set the *Persist* property to True.

Using TXDataWebClient

After setting up the connection, you can use *TXDataWebClient* component to communicate with the TMS XData Server from a TMS Web Core application. Note that the TXDataWebConnection **must be previously connected**. Performing operations with TXDataWebClient won't automatically setup the connection.

TXDataWebClient perform operations similar to the ones performed by TXDataClient, used in Delphi desktop and mobile applications. It means you can retrieve single and multipe entities (GET), insert (POST), update (PUT), delete (DELETE) and also invoke service operations.

However, there are several differences. The first and main one is that all requests are performed asynchronously. This means that when you call TXDataWebClient methods, you won't have a function result provided immediately to you, but instead you have to use event or callback to receive the result. Here is, for example, how to retrieve an entity (GET request) from server, more specifically retrieve an object artist, from entity set "Artist", with id 1:

```
procedure TForm1.GetArtistWithId1;
begin
  XDataWebClient1.Connection := XDataWebConnection1;
  XDataWebClient1.OnLoad := XDataWebClient1Load;
  XDataWebClient1.Get('Artist', 1);
end;

procedure TForm1.XDataWebClient1Load(Response: TXDataClientResponse);
var
  Artist: TJSObject;
begin
  // Both lines below are equivalent.
  Artist := TJSObject(Response.Result);
  Artist := Response.ResultAsObject;

  // Use Artist object
end;
```

First, associate the TXDataWebClient component to an existing TXDataWebConnection component which will has the connection settings. This can be also done at design-time.

Second, set the *OnLoad* event of the component and add the code there to be executed when request is completed. Also can be done at design-time.

Finally, execute the method that perform the operation. In this case, *Get* method, passing the entity set name (Artist) and the id (1).

When the request is complete, the OnLoad event will be fired, and the result of the request (if it does return one) will be available in the *Response.Result* property. That property is of type *JSValue*, which can be any valid value. You will have to interpret the result depending on the request, if you are retrieving a single entity, that would be a *TJSObject*. If you are retrieving a list of objects, then the value can be a *TJSArray*, for example. You can alternatively use *ResultAsObject* or *ResultAsArray*.

TXDataWebClient component is very lightweight, so you can use as many components as you want. That means that you can drop one TXDataWebClient component in the form for each different request you want to perform, so that you will have one OnLoad event handler separated for each request. This is the more RAD and straightforward way to use it. In the case you want to use a single web client component for many requests, you can differentiate each request by the request id.

New async/await mechanism:

As of TMS Web Core 1.6, you can also use the `Async` version of all methods. It's the same name but with the `Async` suffix, and you can then use the `await` keyword so you don't need to work with callbacks:

```
procedure TForm1.GetArtistWithId1;
var
  Response: TXDataClientResponse;
  Artist: TJSObject;
begin
  XDataWebClient1.Connection := XDataWebConnection1;
  Response := await(XDataWebClient1.GetAsync('Artist', 1));

  // Both lines below are equivalent.
  Artist := TJSObject(Response.Result);
  Artist := Response.ResultAsObject;

  // Use Artist object
end;
```

Using RequestId

Each response in the OnLoad event will have a request id. By default the request id is the name of the performed operation, in lowercase. So it will be "get" for get requests, "list" for list requests, etc.:

```
procedure TForm1.XDataWebClient1Load(Response: TXDataClientResponse);
var
  Artist: TJSObject;
begin
  if Response.RequestId = 'get' then
  begin
    Artist := TJSObject(Response.Result);
  end;
end;
```

If you want to change the default request id, you can pass a different one as an extra parameter to the request, and check for it in the OnLoad event:

```
XDataWebClient1.Get('Artist', 1, 'get artist');
```

Handling errors

If you don't specify anything, all request errors that might happen will fire the OnError event of the associated TXDataWebConnection component. This way you can have a centralized place to handle all request errors for that XData server.

But if you want to add error-handling code that is specific to a TXDataWebClient connection, TXDataWebClient also provides an *OnError* event:

```
procedure TForm1.XDataWebClient1Error(Error: TXDataClientError);
begin
  WriteLn('Error on request: ' + Error.ErrorMessage);
end;
```

When you use the `Async` methods, all you have to do is wrap the code in a try..except block:

```
  try
    Resonse := await(XDataWebClient1.GetAsync('Artist', 1));
  except
    on E: Exception do ; // do something with E
  end;
```

Using callbacks

Alternatively to using OnLoad and OnError events, you can use the request methods passing callback as parameter. You can pass a callback for successful response, and optionally an extra callback for error response (if you don't pass the error callback, it will fallback to the OnError event).

```
procedure TForm1.GetArtistWithId1;

  procedure OnSuccess(Response: TXDataClientResponse);
  var
    Artist: TJSObject;
  begin
    Artist := TJSObject(Response.Result);
    // Use Artist object
  end;

  procedure OnError(Error: TXDataClientError);
  begin
    WriteLn('Error on request: ' + Error.ErrorMessage);
  end;

begin
  XDataWebClient1.Get('Artist', 1, @OnSuccess, @OnError);
end;
```

Available request methods

The following is a list of available request methods in TXDataWebClient. Remember that the method signatures in this list include only the required parameters. You can always also use either *RequestId* parameter or the callback parameters, as previously explained. Also, remember that all those methods have an "async/await version" that you can use, just by suffixing the method name with the `Async` suffix: `GetAsync`, `PostAsync`, etc.

Name	Description
procedure **Get**(const EntitySet: string; Id: JSValue)	Retrieves a single entity from the server (GET request), from the specified entity set and with the specified id. Result will be a TJSObject value.

Name	Description
procedure **Get**(const EntitySet, QueryString: string; Id: JSValue)	Retrieves a single entity from the server (GET request), from the specified entity set and with the specified id. Optionally you can provide a QueryString parameter that must contain query options to added to the query part of the request. For get requests you would mostly use this with $expand query option. Result will be a TJSObject value.
procedure **List**(const EntitySet: string; const Query: string = '')	Retrieves a collection of entities from the specified entity set in the server (GET request). You can provide a QueryString parameter that must contain query options to added to the query part of the request, like $filter, $orderby, etc.. Result will be a TJSArray value.
procedure **Post**(const EntitySet: string; Entity: TJSObject)	Inserts a new entity in the entity set specified by the EntitySet parameter. The entity to be inserted is provided in the Entity parameter.
procedure **Put**(const EntitySet: string; Entity: TJObject)	Updates an existing entity in the specified entity set. You don't need to provide an id separately since the Entity parameter should already contain all the entity properties, including the correct id.
procedure **Delete**(const EntitySet: string; Entity: TJObject)	Deletes an entity from the entity set. The id of the entity will be retrieved from the Entity parameter. Since this is a remove operation, only the id properties are relevant, all the other properties will be ignored.
procedure **RawInvoke**(const OperationId: string; Args: array of JSValue)	Invokes a service operation in the server. The OperationId identifies the operation and Args contain the list of parameters. More info below.

Invoking service operations

You can invoke service operations using *RawInvoke* methods. Since you can't use the service contract interfaces in TMS Web Core yet, the way to invoke is different from TXDataClient. The key parameter here is *OperationId*, which identifies the service operation to be invoked. By default, it's the interface name plus dot plus method name.

For example, if in your server you have a service contract which is an interface named "IMyService" which contains a method "Hello", that receives no parameter, you invoke it this way:

```
XDataWebClient1.RawInvoke('IMyService.Hello', []);
```

If the service operation provides a result, you can get it the same way as described above: either using *OnLoad* event, or using callbacks:

```
procedure TForm2.WebButton1Click(Sender: TObject);

  procedure OnResult(Response: TXDataClientResponse);
  var
    GreetResult: string;
  begin
    GreetResult := string(TJSObject(Response.Result)['value']);
  end;

begin
  Client.RawInvoke('IMyService.Greet', ['My name'], @OnResult);
end;
```

Since we're in the web/JavaScript world, you must know in more details how results are returned by service operations in JSON format.

As the example above also illustrates, you can pass the operation parameters using an array of JSValue values. They can be of any type, inculding *TJSObject* and *TJSArray* values.

Just as the methods for the CRUD endpoints, you also have a `RawInvokeAsync` version that you can use using `await` mechanism:

```
procedure TForm2.WebButton1Click(Sender: TObject);
var
  Response: TXDataClientResponse
  GreetResult: string;

begin
  Response := await(Client.RawInvokeAsync('IMyService.Greet', ['My name']);
  GreetResult := string(TJSObject(Response.Result)['value']);
end;
```

Other properties

- property *ReferenceSolvingMode: TReferenceSolvingMode*
  Specifies how $ref occurrences in server JSON response will be handled.
  - **rsAll**: Will replace all $ref occurrences by the instance of the referred object. This is default behavior and will allow dealing with objects easier and in a more similar way as the desktop/mobile TXDataClient. It adds a small overhead to solve the references.
  - **rsNone**: $ref occurrences will not be processed and stay as-id.

Using TXDataWebDataset

In addition to TXDataWebClient, you have the option to use *TXDataWebDataset* to communicate with TMS XData servers from web applications. It's even higher level of abstraction, at client side you will mostly work with the dataset as you are used to in traditional Delphi applications, and XData Web Client framework will translate it into REST/JSON requests.

Setting it up is simple:

1. Associate a TXDataWebConnection to the dataset through the *Connection* property (you can do it at design-time as well). Note that the TXDataWebConnection **must be previously connected**. Performing operations with TXDataWebDataset won't automatically setup the connection.

```
XDataWebDataset1.Connection := XDataWebConnection1;
```

2. Set the value for *EntitySetName* property, with the name of the entity set in XData server you want to manipulate data from. You can also set this at design-time, and object inspector will automatically give you a list of available entity set names.

```
XDataWebDataset1.EntitySetName := 'artist';
```

3. Optionally: specify the persistent fields at design-time. As with any dataset, you can simply use the dataset fields editor and add the desired fields. TXDataWebDataset will automatically retrieve the available fields from XData server metadata. If you don't, as with any dataset in Delphi, the default fields will be created.

And your dataset is set up. You can the use it in several ways, as explained below.

Loading data automatically

Once the dataset is configured, you just need to call *Load* method to retrieve data:

```
XDataWebDataset1.Load;
```

This will perform a GET request in XData server to retrieve the list of entities from the specific entity set. Always remember that such requests are **asynchronous**, and that's why you use *Load* method instead of *Open*. *Load* will actually perform the request, and when it's finished, it will provide data to the dataset and only then, call *Open* method. Which in turn will fire *AfterOpen* event. If you want to know when the request is finished and data is available in the dataset, use the *AfterOpen* event.

> **NOTE**
> If you already have data in the dataset and want `Load` method to fully update existing data, make sure the dataset is closed before calling `Load` method.

To filter out results, you can (and should) use the *QueryString* property, where you can put any query option you need, including $filter and $top, which you should be use to filter out the results server-side and avoiding retrieving all the objects to the client. Minimize the number of data sent from the server to the client!

```
XDataWebDataset1.QueryString := '$filter=startswith(Name, ''John'')&$top=50';
XDataWebDataSet1.Load;
```

Paging results

The above example uses a raw query string that includes "&$top=50" to retrieve only 50 records. You can do paging that way, by building the query string accordingly. But TXDataWebDataset provides additional high-level properties for paging results in an easier way. Simply use *QueryTop* and *QuerySkip* property to define the page size and how many records to skip, respectively:

```
XDataWebDataset1.QueryTop := 50; // page size of 50
XDataWebDataset1.QuerySkip := 100; // skip first 2 pages
XDataWebDataset1.QueryString := '$filter=startswith(Name, ''John'')';
XDataWebDataSet1.Load;
```

The datase also provides a property *ServerRecordCount* which might contain the total number of records in server, regardless of page size. By default, this information is not retrieved by the dataset, since it requires more processing at server side. To enable it, set *ServerRecordCountMode*:

```
XDataWebDataset1.ServerRecordCountMode := smInlineCount;
```

When data is loaded from the dataset (for example in the *AfterOpen* event), you can read *ServerRecordCount* property:

```
procedure TForm4.XDataWebDataSet1AfterOpen(DataSet: TDataSet);
begin
  TotalRecords := XDataWebDataset1.ServerRecordCount;
end;
```

Loading data manually

Alternatively you can simply retrieve data from the server "manually" using TXDataWebClient (or even using raw HTTP requests, if you are bold enough) and provide the retrieved data to the dataset using *SetJsonData*. Since the asynchronous request was already handled by you, in this case where data is already available, and you can simply call *Open* after setting data:

```
procedure TForm1.LoadWithXDataClient;

  procedure OnSuccess(Response: TXDataClientResponse);
  begin
    XDataWebDataset1.SetJsonData(Response.Result);
    XDataWebDataset1.Open;
  end;

begin
  XDataWebClient1.List('artist', '$filter=startswith(Name, ''New'')',
@OnSuccess);
end;
```

Modifying data

When using regular dataset operations to modify records (*Insert, Append, Edit, Delete, Post*), data will only be modified in memory, client-side. The underlying data (the object associated with the current row) will have its properties modified, or the object will be removed from the list, or a new object will be created. You can then use those modified objects as you want - manually send changes to the server, for example.

But TXDataWebDataset you can have the modifications to be automatically and transparently sent to the server. You just need to call *ApplyUpdates*:

```
XDataWebDataset1.ApplyUpdates;
```

This will take all the cached modifications (all objects modified, deleted, inserted) and will perform the proper requests to the XData server entity set to apply the client modifications in the server.

Other properties

Name	Description
SubPropsDepth: Integer	Allows automatic loading of subproperty fields. When adding persistent fields at design-time or when opening the dataset without persistent fields, one TField for each subproperty will be created. By increasing SubpropsDepth to 1 or more, dataset will also automatically include subproperty fields for each property in each association, up to the level indicated by SubpropsDepth. For example, if SubpropsDepth is 1, and the entity type has an association field named "Customer", the dataset will also create fields like "Customer.Name", "Customer.Birthday", etc.. Default is 0 (zero).
CurrentData: JSValue	Provides the current value associated with the current row. Even though CurrentData is of type JSValue for forward compatibility, for now CurrentData will always be a TJSObject (an object).
EnumAsIntegers: Boolean	This property is for backward compatibility. When True, fields representing enumerated type properties will be created as TIntegerField instances. Default is False, meaning a TStringField will be created for the enumerated type property. In XData, the JSON representing an entity will accept and retrieve enumerated values as strings.

Solving Errors

In the process of solving errors in web applications, it's important to **always check the web browser console**, available in the web browser built-in developer tools. Each browser has its own mechanism to open such tools, in Chrome and Firefox, for example, you can open it by pressing F12 key.

The console gives you detailed information about the error, the call stack, and more information you might need to understand what's going on (the HTTP(S) requests the browser has performed, for example).

You should also have in mind that sometimes the web application doesn't even show a visible error message. Your web application might misbehave, or do not open, and no clear indication is given of what's going on. Then, whenever such things happen or you think your application is not behaving as it should, **check the web browser console**.

Here we will see common errors that might happen with web applications that connect to XData servers.

Error connecting to XData server

This is the most common error when starting a Web Core application using XData. The full error message you might get is the following:

```
XDataConnectionError: Error connecting to XData server |
fMessage::XDataConnectionError:
Error connecting to XData server fHelpContext::0
```

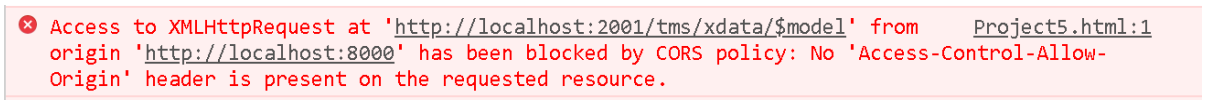
And it will look like this:



As stated above, the first thing you should is open the browser console to check for more details - the reason for the connection to fail. There are two common reasons for that:

CORS issue

The reason for the error might be related to CORS if in the browser console you see a message like this:



This is caused because your web application files is being served from one host (*localhost:8000* in the example above), and the API the app is trying to access is in a different host (*localhost:2001* in the example).

To solve it, you have two options:

1. Modify either your web app or API server URL so both are in the same host. In the example above, run your web application at address *localhost:2001*, or change your XData API server URL to *localhost:8000/tms/xdata*.
2. Add CORS middleware to your XData server.

HTTPS/HTTP issue

One second common reason for wrong connection is a mix of HTTPS and HTTP connections. This usually happens when you deploy your Web Core application to a server that provides the files through HTTPS (using an SSL certificate), but your XData server is still at HTTP. Web browsers do not allow a web page served through HTTPS to perform Javascript requests using HTTP.

The solution in this case is:

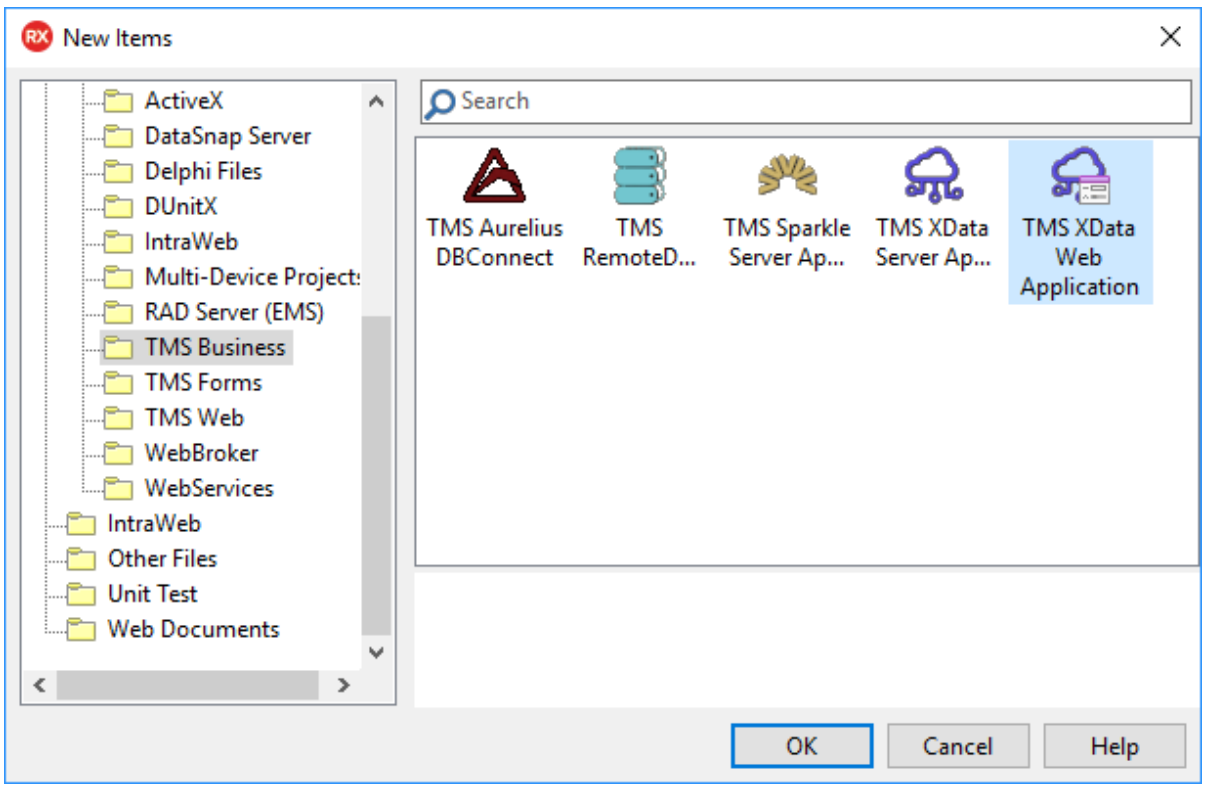
1. Use HTTPS also in your XData server. It's very easy to associate your SSL certificate to your XData server. If you don't have an SSL certificate and don't want to buy one, you can use a free Let's Encrypt certificate in your XData server.
2. Revert your web app back to an HTTP server, so both are served through HTTP. Obviously, for production environments, this is not a recommended option.

XData Web Application Wizard

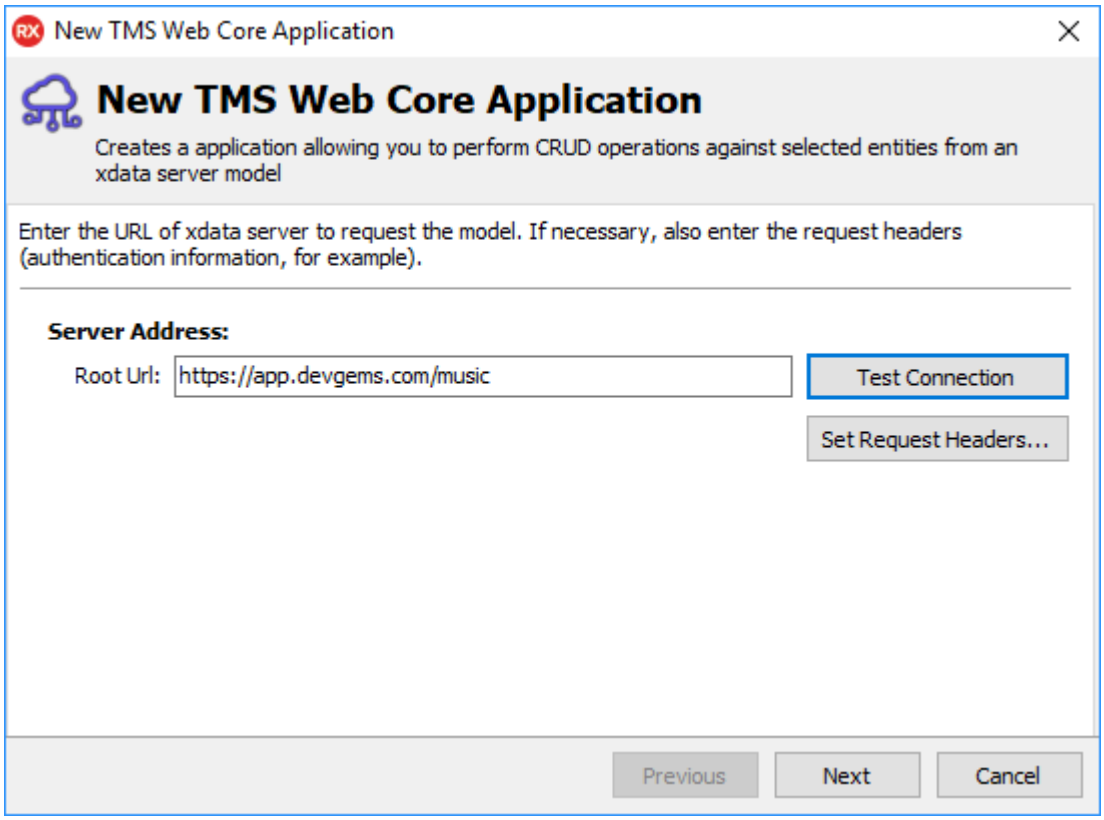
TMS XData provides a great wizard to scaffold a full TMS Web Core web client application.

This wizard will connect to an existing XData server and extract meta information it with the list of entities published by the server. With that information, it will create a responsive, Boostrap-based TMS Web Core application that will serve as a front-end for listing entities published by the server. The listing features include data filtering, ordering and paging.

To launch the wizard, go to Delphi menu *File* > *New* > *Other...*, then from the *New Items* dialog choose *Delphi Projects* > *TMS Business*, and select wizard "*TMS XData Web Application*".

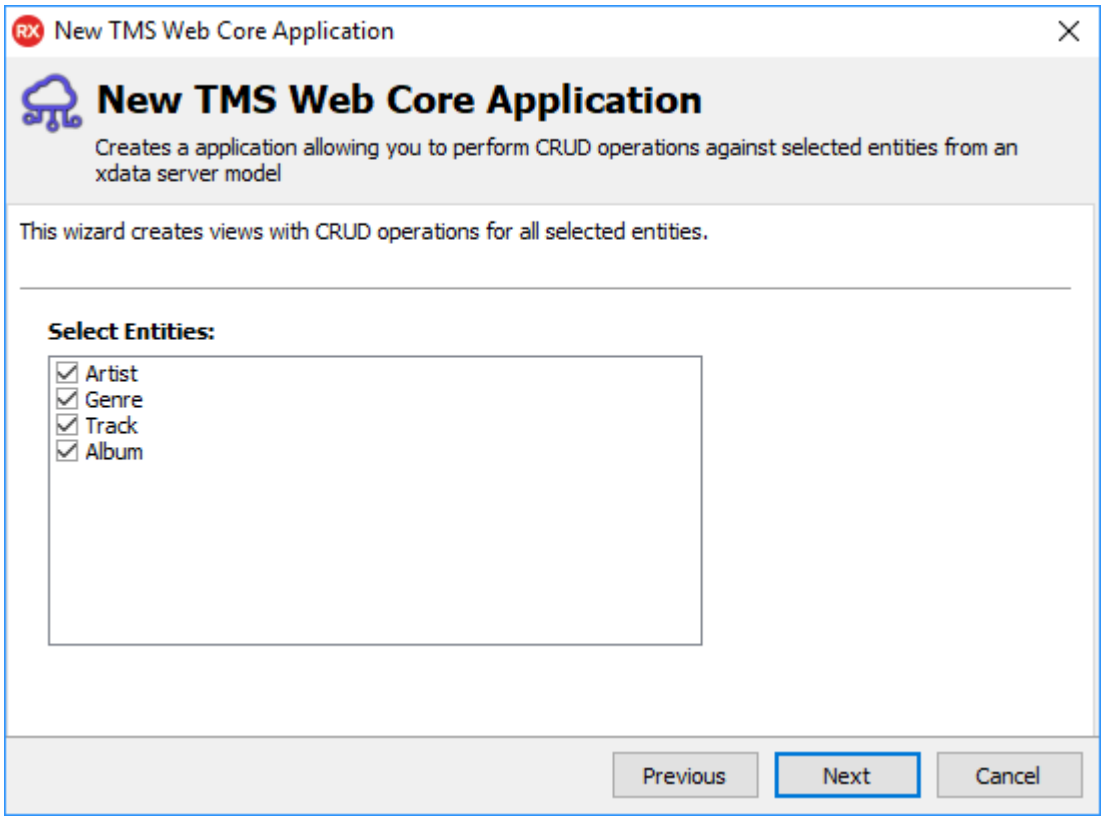


Click Ok and the wizard will be launched:

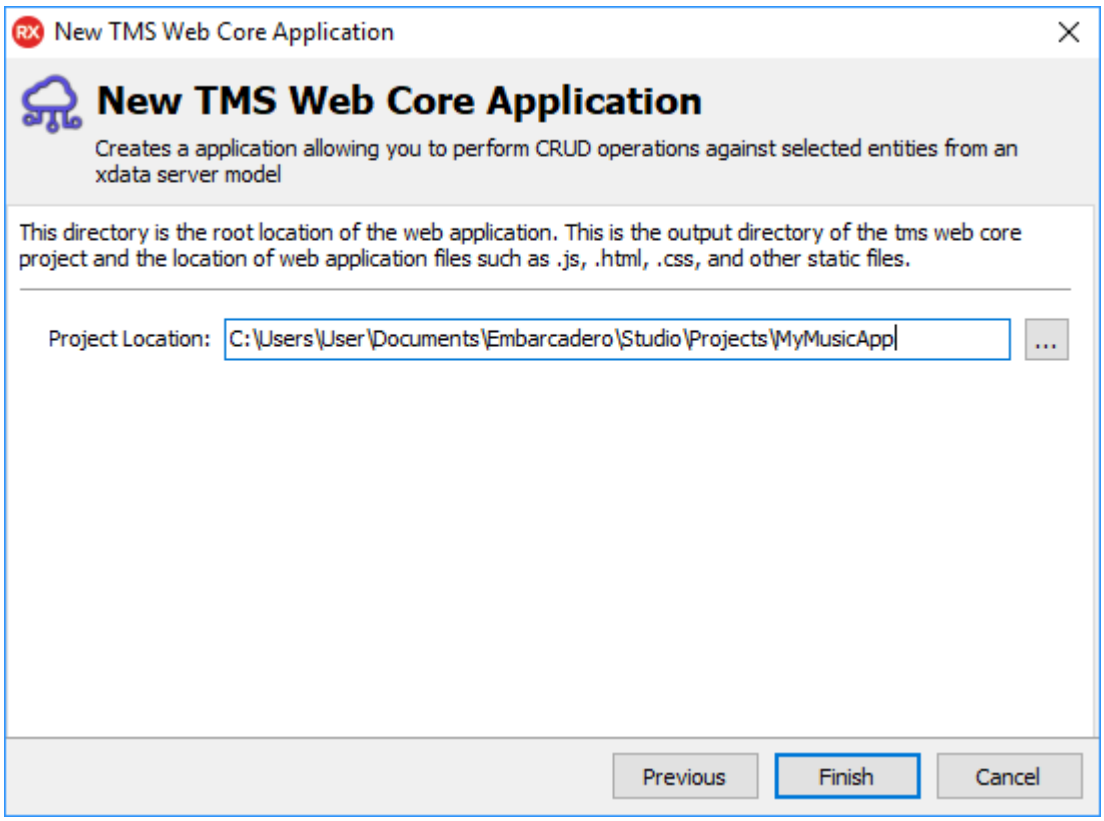


In this first page you must provide the URL address of a running XData server. You can click the "Test Connection" button to check if the connection can be established. If the server requires authentication or any extra information sent by the client, you can use the "Set Request Headers..." button to add more HTTP headers to the client request (for example, adding a JWT token to an *Authorization* header).

Once the server URL is provided and connecting, click *Next* to go to next page.

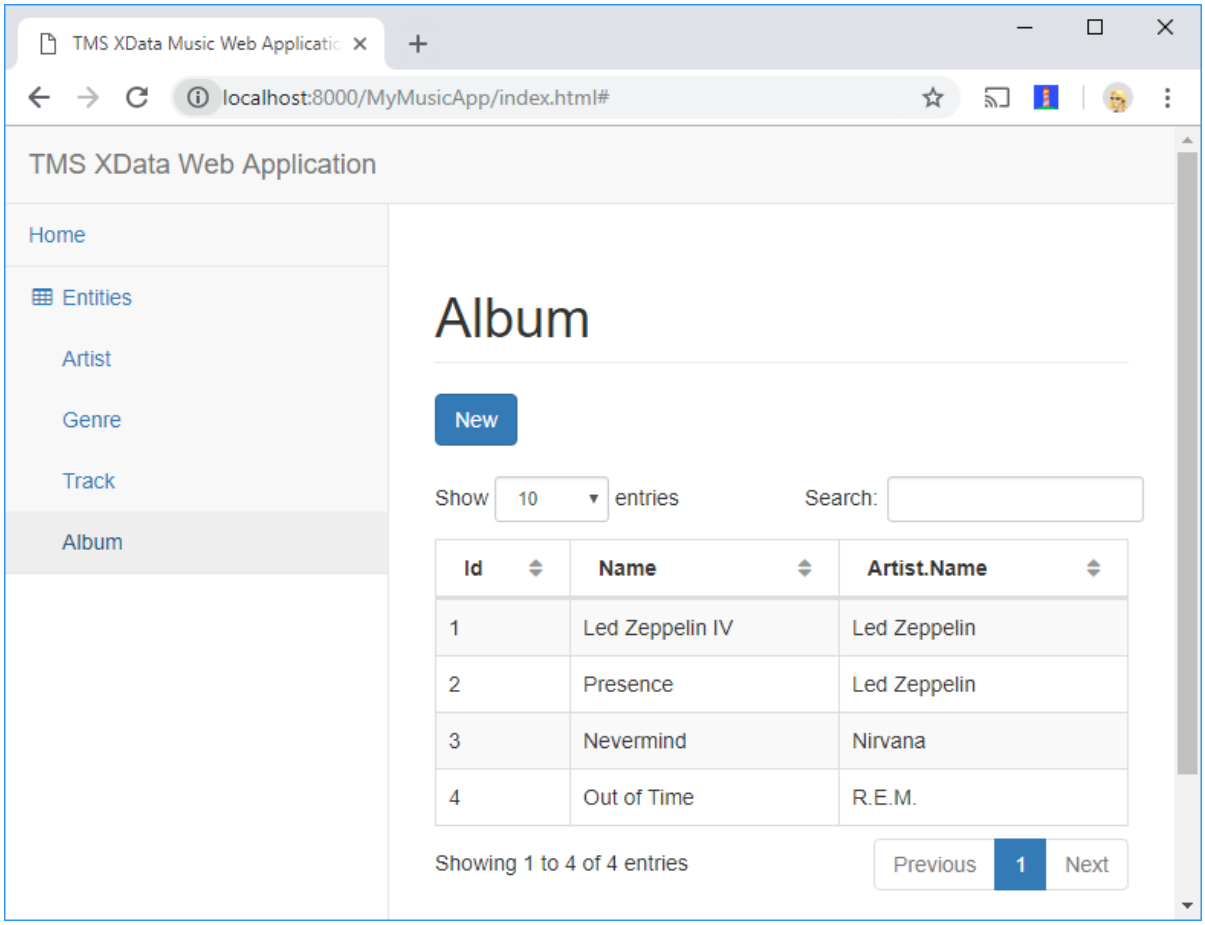


This will list all entities published by the XData server. You can then select the ones you want to generate a listing page for. Unselected entities will not have an entry in the menu nor will have a listing page. Select the entities you want and click *Next*.



The final wizard page will ask you for a directory where the source code of the web application will be generated. Choose the output folder you want and click *Finish*. The application source code will be generated in the specified folder, and the project will be open in Delphi IDE.

You can now compile and run the application, and of course, modify it as you want. This is a easy and fast way to start coding with TMS Web Core and TMS XData backend. Here is a screenshot of the generated application running in the browser, using the settings above:



Extra Resources

There are additional quality resources about TMS Web Core and TMS XData available, in different formats (video training courses and books):

Online Training Course: Introduction to TMS Web Core
By Wagner Landgraf
https://courses.landgraf.dev/p/web-applications-with-delphi-tms-web-core

Book: TMS Software Hands-on With Delphi
(Cross-plataform Mult-tiered Database Applications: Web and Desktop Clients, REST/JSON Server and Reporting, Book 1)
By Dr. Holger Flick
https://www.amazon.com/dp/B088BJLLWG/

Book: TMS WEB Core: Web Application Development with Delphi
By Dr. Holger Flick
https://www.amazon.com/dp/B086G6XDGW/

Book: TMS WEB Core: Webanwendungen mit Delphi entwickeln (German)
By Dr. Holger Flick
https://www.amazon.de/dp/1090700822/

About

This documentation is for TMS XData.

In this section:

What's New

Copyright Notice

Getting Support

Breaking Changes

What's New

Version 5.5 (Mar-2022)

- **New: Validation attributes now also work for DTO objects**.
- **New: TXDataSwaggerUIOptions.DisplayOperationId property.** Breaking change: this property `False` by default, which hides the operation id from Swagger UI.
- **New: SwaggerExclude attribute now applies to DTO and entity properties**.
- **Fixed:** Serialization of polymorphic lists now correctly output the class name of the list item if different from the base class.

Version 5.4 (Feb-2022)

- **New:** `TXDataConnectionPool.CleanupTimeout` **property.** When different than zero, the connection pool will release the connections that hasn't been used for the at least the time specified by the `CleanupTimeout` property.
- **New:** `TXDataMultiDBConnectionPool` **component allows for easy creation of XData modules connecting to multiple tenant databases.**
- **New: OnModuleException now includes a property named** `Errors` **in the** `Args` **parameter.** This property, if filled, will add a list of errors in the JSON response to the client, in addition to the existing single error message.
- **New: Client-side exception** `EXDataClientRequestException` **now also includes a property** `Errors` to provide a list of errors provided by the server (if available).
- **New: TXDataHttpHandler.SetStatusCode** allows setting a custom HTTP status code from XData service operations.
- **New: Now it's possible to close the HTTP response inside a XData service operation.** In this case, XData will not touch the HTTP response anymore.
- **New:** `Consumes` **attribute allows receiving parameters as** `x-www-form-urlencoded` **type.**
- **New:** `JsonManaged` **attribute allows flagging which associated objects in the DTO should not be automatically destroyed by XData serializer/deserializer.**
- **New: Swagger now supports and generates documentation from comments in DTO classes and properties.**
- **New: You can specify which DTO properties will be flagged as "required" in Swagger**, using the following XML comment:

```
/// <swagger name="required">true</swagger>
```

- **Improved:** List and dictionary JSON serializers now automatically take `OwnsObjects` property into account, and do not try to destroy the items if such property is set to true.
- **Improved:** Better error message when database connection pool was not properly configured in XData server.
- **Improved:** TXDataConnectionPool doesn't require the Connection property to be set, if a pool interface is provided via OnGetPoolInterface event.
- **Improved:** JSON date-time deserialization now accepts dates in format `YYYYMMDD` and time in format `hhnnss`.
- **Improved**: Swagger demo updated to show how to work with DTO documentation.
- **Fixed:** `TXDataClient` was sending float parameters via URL query with wrong formatting depending on local settings (regression).
- **Fixed:** Music web demo was not compiling with latest Pas2JS/Web Core versions.
- **Fixed:** `TJsonInclude.NonDefault` option was not working with nullable types.
- **Breaking change**: Internal method `SendErrorResponse` and interface `IErrorWriter` were modified. If you use anyb of them, your existing code won't compile and you will have to adapt to the new signature.

Version 5.3 (Sep-2021)

- **New:** Delphi 11 support.
- **New:** Web Core application demo showing master-detail usage.
- **New:** Web Core application demo showing blob/memo usage.
- **Fixed:** `OnManagerCreate` event raising errors if Delphi memory manager is configured to completely clean memory upon destruction (FastMM in debug mode, for example).

Version 5.2 (Jun-2021)

- **New: XData Query Builder**. You can now **build XData queries using fluent interface using the new XData Query Builder**, and then get the final query string to be used in your HTTP requests or queries using TXDataClient. For example:

```
  Customers := Client.List<TCustomer>(
   CreateQuery
     .From(TCustomer)
     .Filter(
       (Linq['Name'] = 'Paul')
       or (Linq['Birthday'] < EncodeDate(1940, 8, 1))
     )
     .OrderBy('Name', False)
     .Take(10).Skip(50)
     .Expand('Country')
     .QueryString
  );
```

- **New: Mechanism to accept XData query options in service operations.** You can now easily receive and process XData query syntax like `$filter`, `$orderby`, `$top` and `$skip` in service operations, even for DTOs, and create Aurelius criteria from it:

```
type  IMyService = interface(IInvokable)
    [HttpGet] function List(Query: TXDataQuery): TList<TCustomer>;

{...}

function TMyService.List(Query: TXDataQuery): TList<TCustomer>;
begin
  Result := TXDataOperationContext.Current
    .CreateCriteria<TCustomer>(Query).List;
end;
```

- **New: Service operations can now receive DTOs (Delphi objects) from the query string of the request URL.** You can now have service like this:

```
    [HttpGet] function FindByIdOrName(Customer: TCustomerDTO): TList<TCusto
mer>;
```

And have your `Customer` parameter be received from a request URL like this: `/CustomerService/FindByIdOrName?Id=10&Name='Paul'`.

- **Improved: Nullable types are now supported in plain Delphi objects** (in addition to existing support in Aurelius entities). When using plain Delphi objects as DTOs, like when receiving or returning them in service operations, they are now properly serialized/deserialized as JSON.
- **Improved:** Internal `TJsonWriter` class is now 50% faster.
- **Improved:** Web Core stub for `TXDataClientResponse` was missing some properties.
- **Fixed:** Error "Could not convert variant array of type" when retrieving entities which Id is an association to another entity.
- **Fixed:** XData enum values were ignoring `JsonEnumValues` attribute in Swagger Schema and `$filter` queries.

- **Fixed:** Missing properties in XData entities created from Aurelius classes with `AbstractEntity` attribute.
- **Fixed:** Swagger support was being enabled automatically in Linux projects. This was causing an "AmbiguousActionError" exception when Swagger was also enabled in `TXDataServer` component.
- **Fixed:** SerializeInstanceType being ignored in entity property endpoints (e.g., Customer/City).
- **Fixed:** "DevTools failed to load source map" error message now removed from web browser console log when executing TMS XData Music application.

Version 5.2.1

- **Fixed:** Query Builder wrongly creating enumeration literals around quotes (`'tsMale'`).
- **Fixed**: Query Builder not finding properties from inherited classes when using `From` method passing class types.

Version 5.1 (Mar-2021)

- **Improved:** All compilation warnings removed when using XData from TMS Web Core applications.
- **Fixed:** Compilation error in Multitenancy demo.

Version 5.0 (Mar-2021)

- **New: Attribute-based Authorization allows you to secure your REST API in a declarative way, adding attributes to methods and entities you want to be protected by specific permissions.** It's as simple as adding authorization attributes to methods and entities:

```
  [ServiceContract]
  IMyService = interface(IInvokable)

    [AuthorizeScopes('admin,writer')]
    procedure ModifyEverything;

    [Authorize]
    function Restricted: string;
  end;

  [Entity, Automapping]
  [EntityAuthorizeScopes('reader', EntitySetPermissionsRead)]
  [EntityAuthorizeScopes('writer', EntitySetPermissionsWrite)]
  TArtist = class
```

- **New: async/await methods in XData web client makes it even easier to build TMS Web Core applications using XData as backend - the asynchronous requests to server can be coded as if they were synchronous**:

```
var
  Response: TXDataClientResponse
  GreetResult: string;
 begin
  await(XDataConnection.OpenAsync);
  Response := await(Client.RawInvokeAsync('IMyService.Greet', ['My name']));
  GreetResult := string(TJSObject(Response.Result)['value']);
end;
```

- **New: RoutingPrecedence property in XData server allows you to override automatic CRUD endpoints behavior using service operations, by using the same routing URL.**
- **New: Swagger demo projects now include a client-side application at `xdata\demos\Swagger\Client` that shows how Swagger can be helpful at client side.**
- **New: Demo `multitenancy` shows how to create multitenant servers using XData in an easy and straightforward way with the new `TMultiTenantConnectionPool` implementation.**
- **New: OnManagerCreate event which is fired whenever a new TObjectManager instance is created by XData.**
- **Fixed:** Deserialization of a JSON empty object would result in nil object, instead of an empty (instantiated) object with default property values.
- **Fixed:** List serialization failed when list to be serialized was nil in some situations.
- **Fixed:** Batch operations not working in an optimized way when using connection interfaces coming from XData connection pool.
- **Fixed:** `TXDataWebDataset` was not bringing ancestor persistent fields at design-time for entity classes in a inheritance hierarchy.

- **Fixed:** XML documentation tags for SwaggerUI were being ignored for service contracts with an empty route (the "default" endpoint).
- **Fixed:** SwaggerUI was sometimes displaying empty tags. This happened with Automatic CRUD Endpoints, when no endpoint were set for an existing entity. Still, the tag for the entity appeared.
- **Fixed:** Error "Cannot deserialize value from Url format" when invoking service operations passing double values in URL - for example: `?a=1.6`).
- **Fixed:** Typo in `EInvalidParamBinding` exception message: "Invalid binding in param".

Version 4.17 (Sep-2020)

- **New: Full support to use XML documentation in Swagger documents.** You can now **describe your REST API documentation using XML documentation syntax**, and even using the **same content for both**. Write your documentation once, use it in several places - for your Delphi developer, for your API consumer.
- **New: SwaggerExclude attribute allows you to exclude specific endpoints from the final Swagger API documentation.**
- **New: Swagger demo shows how to create Swagger documentation, provide SwaggerUI interface to your end user, and use XML documentation to enrich your documentation content.**
- **Improved:** Service contract routing now allows replacing the root URL. Requests like "GET /" or "POST /" can be now be configured to be routed to your own service implementation.
- **Improved:** Swagger documents now take entity set permissions into account, omitting Aurelius CRUD endpoints which are not available due to the entity set permissions configuration.
- **Improved:** The endpoints for Swagger-UI ("/swaggerui") and Swagger.json ("/openapi/swagger.json") don't appear anymore in the Swagger document itself. This avoids pollution of the Swagger document, focusing only on your own API endpoints.
- **Improved:** Endpoints are now sorted alphabetically in Swagger documents, making it much easier to be visualized in Swagger-UI for example.
- **Improved:** Parameters and function results of type TDate, in service contracts, will now show in Swagger document as "date" instead of "date-time".

Version 4.16 (Aug-2020)

- **New: FireDAC-SQL demo shows how to create a REST resource (GET, POST, PUT, DELETE) using FireDAC to access the database directly via SQL, without using TMS Aurelius.**
- **Improved:** EXDataClientException now contains property Info holding the JSON response of the error.

- **Improved:** XData Web Request (THttpRequest) now provides Timeout property.
- **Improved:** Compatibility with TMS Web Core 1.5.
- **Fixed:** Service operations returning objects and also with "out" params were returning wrong JSON.
- **Fixed:** When loading objects using TXDataClient, proxied objects were not being loaded if they were located at a depth level higher than the MaxEagerDepth configuration (default to 3).
- **Fixed:** SwaggerUI was not working in SQLiteConsoleServer demo.

Version 4.15 (Jun-2020)

- **Improved:** XData trial binary compatibility with TMS Web Core 1.4.2 trial.
- **Fixed:** Wrong serialization of objects that indirectly inherited from List<T>. For example, TList2 = class(TList1), TList1 = class(TList<TSomeClass>). Serialization of TList2 was wrong.

Version 4.14 (Jun-2020)

- **Improved:** Web Core client components updated to support the newest TMS Web Core 1.4.

Version 4.13 (May-2020)

- **New: Flexible URL routing mechanism using Route attribute, allows multiple segments in path routing, and flexible parameter binding.** For a single method, users can route it to an URL like '*orders/approved/{year}/{month}*'. The service operation will be routed from the invoked URL, and the values passed in URL will be automatically bound to method parameters.
- **New: Delphi 10.4 Sydney support.** Full compatibility with the recently released new Delphi version.
- **New: XData server now returns a xdata-version header in each request.** This will make it easier to tell the available features of the XData server being connected to. Current version returned is 2.
- **New: TXDataClient.ForceBlobLoading property for backward compatibility with old XData servers that could not receive eager blobs as @xdata.proxy.**
- **Improved: Significant performance increase when sending entities to XData server that contain blobs.** For example, consider a Customer with Photo (blob) field you retrieved using TXDataClient. In previous versions, calling Put to update the customer would load the Photo content from the server, even if it was not modified. Now it's optimized and it will send to the server just a proxy information. The server is also now smart enough to identify those proxies and proceed properly, even when the blob is not lazy.

- **Improved: Blob serialization was not flagging blob as loaded, causing a lazy blob to be loaded multiple times.** Suppose there is a property TCustomer.Photo of type TBlob. Such blob is lazy and not loaded. During JSON serialization, the content of the blob is loaded, but property TCustomer.Photo was not being marked as load. Thus, in a second serialization, or when the app tried to read property content, the blob would load again.
- **Improved: It's now possible to send blobs in JSON using format "Photo@xdata.proxy": "Customers(1)/Photo", even for blobs not flagged as lazy. In previous versions, this was only possible for lazy blobs.** Sending such values will simply not modify the blob value in the server.
- **Improved: Ambiguity between enumeration and member name solved in $filter expressions.** Suppose a filter expression like "$filter=Pending eq 2". There might be a situation where Pending would be both an enumeration name and an entity property. Now, if that happens, Pending will be considered as an entity property. Otherwise, as an enumeration.
- **Improved: Enumeration values can be prefixed by the enum type in $filter expressions.** To avoid ambiguous naming, enumeration values in filter expressions can now be prefixed by the enum type. For example: "$filter=Pending eq TStatus.Pending". TStatus.Pending is an enumeration value, Pending is a property value.
- **Fixed:** Memory leak when building a TXDataAureliusModel object using a TXDataModelBuilder raises an error.
- **Fixed:** TXDataClient 404 error not raising exception when server methods did not return objects (procedures, not functions).
- **Fixed:** Value endpoints (/$value) were not returning correct string values when the content had double quotes.
- **Fixed:** Serialization of TArray<byte> and TBytes was not working correctly when the application/server was compiled using runtime packages.
- **Fixed:** Error "Requested action has ambiguous implementation" in SwaggeUI endpoint when starting/stopping XData server multiple times.
- **Fixed:** Delphi IDE splash screen showing multiple XData icons.
- **Fixed:** XData Web App Generator creating wrong persistent fields when XData server had GUID field types.

Version 4.12 (Apr-2020)

- **Improved:** UriPathSegment now can be applied to entities (for entity set naming).
- **Fixed:** TXDataWebClient not raising OnError events when an error happened during RawInvoke call.
- **Fixed:** XData Web Application generator updated to work with latest Web Core 1.4.
- **Fixed:** XData Music Demo updated to work with latest Web Core 1.4.

Version 4.11 (Apr-2020)

- **New:** Enumerated literals now supported in $filter query string.
- **Fixed:** Wrong TXDataWebDataset date/time conversion from JSON (regression). It was causing "Invalid date time" errors and also was displaying wrong date time fields, by incorrectly adding time zone offset to the date.
- **Fixed:** TXDataClient.IgnoreUnknownProperties had no effect when using service operations.

Version 4.10 (Mar-2020)

- **New: OnEntityDeleted, OnEntityInserted, OnEntityModified are new server-side events that fire after automatic CRUD endpoint operations are performed, in addition to existing OnEntityDeleting, OnEntityModifying and OnEntityInserting events, which are performed before such operations.**
- **New: JsonConverter attribute allows for custom JSON serialization/deserialization of specific PODO and entity properties.** You can for example serialize an integer as an string, a date/time as a string in a specific format, etc.
- **New: Support for TStream param mixed with other params.** You can now have a service operation that receives a TStream parameter to also receive more parameters, as long as they are not defined to be in the request body (using FromBody parameter). This way you can use the TStream to process the raw body of the request, but still receive other parameters through the URL (either using FromQuery or FromPath attributes).
- **New: Demo project showing how to use multiple models (multi-model design) in XData.**
- **New: BasicAuth demo showing how to use Basic Authentication in both server and client sides.**
- **New: Demo project showing how to use XData as an Apache module.**
- **Improved:** Swagger doesn't include definition names for lists anymore (interface gets less cluttered).
- **Fixed:** Swagger specification was wrong for entity sets (in automatic CRUD endpoint). It was indicating an array of entities, but the correct format was an object with "value" property and then array of entities, according to the specification.
- **Fixed:** Sending an object with a blob property set to null was not modifying the blob in the database.
- **Fixed:** Generated XData Web Application was being created with title "TMS XData Music Web Application". The word "Music" was removed from the title.

Version 4.9 (Nov-2019)

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

- **New: TXDataOperationContext.CreateManager and AddManager methods.** You can now add more Aurelius managers (TObjectManager) to the XData context. This makes it easier to create other Aurelius managers in addition to the default one, and at the same time return entities from that new manager, without worrying about evicting those objects to destroy them.
- **New: TXDataClient.IgnoreUnknownPropeties property.** You can now set this property to true to force the client to ignore properties sent by the server that the client don't know. Until now, the client would raise an exception saying the property is unknown.
- **Improved:** $expand now appears as an optional parameter in SwaggerUI for endpoints that return Aurelius entities.
- **Improved:** SwaggerUI was displaying parameters as required even when they had default values in service operations.
- **Improved:** TXDataWebDataset compatibility with TMS Web Core 1.3. There is a breaking change between TMS Web Core 1.3 and 1.2 regarding dataset, which means this XData version won't work with previous TMS Web Core versions (1.2 and lower).
- **Fixed:** TXDataClient and TXDataWebClient were building request URLs with double slashes (//) in some requests. This didn't cause any errors but was wrong anyway.
- **Fixed:** TMS XData Music Demo failed to compile in TMS Web Core 1.3 (fixed in 4.9.1).
- **Fixed:** TMS XData Web Application wizard was generating code that didn't compile (fixed in 4.9.1).

Version 4.8 (Oct-2019)

- **Improved:** TXDataWebClient was raising some exceptions as Exception class. It's now using EXDataClientException.
- **Fixed:** SQLiteConsoleServer demo (the "Music server" used in several demos, like the TMS XData/Web Core Music Web Application) was not correctly importing Tracks into Albums.

Version 4.7 (Sep-2019)

- **New: $expand query option now also applies for blobs.** In previous versions, clients could request associations to be inline in JSON response by using the $expand query option in format "*$expand=Customer*". Now you can also ask for blob properties to be inline (as base64 string) in the JSON response, by providing the name(s) of blob properties in the $expand query option. For example, "*$expand=Photo*".
- **Improved:** TXDataWebClient.Post now updates the key (id) properties of the object with the generated value returned by the server. When calling TXDataWebClient.Post you pass a Javascript object instance to the method, which will be serialized as JSON and sent to the server to be inserted. If the server generates a new id for the object, then the Javascript object instance passed to Post will have its id property automatically updated with the id generated by the server. This has a good side effect on using

TXDataWebDataset as well: when automatically applying updates using TXDataWebDataset, the id fields will automatically be filled with the server-side generated value.

- **Improved:** Polymorphism when deserializing JSON objects allows deserializing both PODO and XData/Aurelius entities using the same method. For example, suppose a service operation (server-side method) that receives a parameter of type "TObject". Clients can now send either a PODO object or a Aurelius entity object to that method, and either will be serialized correctly. In previous versions it would consider everything as PODO and wrongly serialize the Aurelius entity.
- **Fixed:** Multiple calls to TXDataWebDataset.ApplyUpdates could cause some errors. That would happen if in the second call to ApplyUpdates, the modifications of the first call were not yet sent to the server, causing the same delta to be processed twice.

Version 4.6 (Jul-2019)

- **New: HttpPatch attribute to specify service operations responding to PATCH HTTP method.**
- **New: macOS 64 support in Delphi Rio 10.3.2.**
- **Fixed:** Swagger not including all service operation actions when the actions endpoints were the same.

Version 4.5 (Jun-2019)

- **New: FromQuery, FromPath and FromBody attributes allow higher control in parameter binding and URL routing for service operations.**
- **New: Support for SwaggerUI provides a built-in web-based environment to test the API from the browser.**
- **New: SwaggerOptions and SwaggerUIOptions properties** allow to easily enable and configure Swagger support for the API.
- **New: EnableEntityKeyAsSegment property** allows single-entity URL endpoints to follow the format "entity/id" - for example, *customer/10*, in addition to existing default format *customer(10)*.
- **Improved:** String and enumeration literals now can be sent in URL without being enclosed by single quotes (e.g., CustomerByName?Name=John instead of CustomerByName?Name='John').
- **Fixed:** TXDataWebClient invoking service operations with default parameters was requiring all parameters to be passed, including the ones with default values.

Version 4.4 (Mar-2019)

- **Fixed:** Index out of range error in TMS XData Web Application Wizard when XData server has no entities.

- **Fixed:** Generated XData Web Application not supporting filtering by subproperties (e.g., Invoice/Customer/Name eq 'John').

Version 4.3 (Jan-2019)

- **New: XData Server wizards generate XData server applications using the new design-time components instead of pure code.** The generated application is now way easier to maintain and evolve, given it uses the RAD approach with design-time components. There is now wizard for server as VCL application. Next versions will include console application, Apache module or Windows service.
- **New: TXDataWebDataset.EnumAsInteger property** controls how enumerated type properties will be represented in the dataset. This is a breaking change.
- **Improved:** All demos refactored, now using the new non-visual components: TXDataServer, TAureliusConnection, TXDataConnectionPool and TSparkleHttpSysDispatcher, making them easier to understand for beginners and easier to change them.
- **Improved:** TXDataWebClient.OnLoad method now provides the newly created entity in the Args.Result property, when using Post method.
- **Fixed:** TXDataWebClient.OnRequest event not being called when executing service operations with RawInvoke method.
- **Fixed:** Workaround a bug in Delphi Rio causing serialization of TJSONNumber values to be always serialized as null (TJSONNumber.Null always returning true).
- **Fixed:** Type mismatch when passing an empty array to a service operation that receives the dynamic array as var/out parameter.
- **Fixed:** XData Service wizard did not appear if there was no open project in IDE.
- **Fixed:** Inoffensive memory leak in TSparkleHttpSysDispatcher when the dispatcher was never activated.
- **Fixed:** TXDataWebDataset now creates TFloatField field for XData properties of type Currency, instead of TCurrencyField (unsupported in TMS Web Core clients).
- **Fixed:** XData Music Demo was issuing JSON Web Tokens with wrong expiration date/time.

Version 4.2 (Dec-2018)

- **Improved:** TXDataWebConnection now "disconnects" (cleans model object, requiring a new connection) when URL is modified.
- **Fixed:** Workaround for a bug in Delphi Rio causing JWT tokens to be always rejected due to expiration (TJSONNumber.Null always returns true).
- **Fixed:** TXDataWebClient.RawInvoke did not allow to receive a full JSON object as a single parameter.
- **Fixed:** XData Web App Application Wizard incorrectly created TExtendedField fields for Int64 types on the server.

- **Fixed:** Typo in the generated XData service code: "form the client" should be "from the client".

Version 4.1 (Nov-2018)

- **New: XData Web Application Wizard to generate GUI to perform CRUD operations on entities.** You can now generate a web application that allows users to insert, modify and/or delete entities.
- **New: Support for Delphi 10.3 Rio.**
- **Fixed:** TTime fields not correctly supported in TXDataWebDataset.
- **Fixed:** XData Service Wizard created new service units in the first project of a project group. New services will be created in the currently active project now.

Version 4.0 (Nov-2018)

- **New: XData Web App Generator wizard.** This wizard creates a complete TMS Web Core client app based on a responsive Bootstrap template. The generated application should be based on an XData server and will provide a user-interface for listing XData server entities, including searching, ordering and pagination.
- **New: Design-time support with a new set of components for XData: TXDataServer and TXDataConnectionPool components.**
- **New: XData Service Operation wizard.** Makes it very easy to create new service operations (server-side business logic).
- **Improved:** TXDataWebDataset.Load now automatically connects the TXDataWebConnection if it is not connected.
- **Improved:** TXDataServerModule.SetEntitySetPermissions supports '*' as the entity set name which will be considered the default permissions for all entity sets.
- **Improved:** TXDataClientResponse.ResultAsArray and ResultAsObject for quick access to Result as a Json array or object.
- **Fixed:** TXDataWebClient.RawInvoke did incorrectly require parameters of type "out".
- **Fixed:** TXDataWebDataset did not set correct datetime values for the entity properties.

Version 3.2 (Sep-2018)

- **New: TXDataWebDataset properties QueryTop, QuerySkip, ServerRecordCount, ServerRecordCountMode.** This makes it easier to retrieve paged results from the server (QueryTop, QuerySkip) and retrieve the total number of records on the server (ServerRecordCount, ServerRecordCountMode). This number does not take the page number and page size into account.

- **New: TJwtMiddleware.AllowExpiredToken and ForbidAnonymousAccess properties.** This makes it easier to reject requests with expired tokens or requests without tokens (anonymous). Just set these properties. This is a breaking change as the middleware now rejects expired tokens by default. Please refer to the TMS Sparkle documentation for more information.
- **New: TCorsMiddleware middleware** makes it straightforward to add CORS support to any Sparkle module. Please refer to the TMS Sparkle documentation for more info.
- **Improved:** XData Music Demo includes options for ordering, paging and filtering (search) of listed records.
- **Improved:** "New TMS XData Server" Wizard adds commented code to include CORS and Compress middleware. Uncomment the generated code to easily enable the features.
- **Improved:** URIPathSegment attribute with empty string can be inside a service contract; it supports creating service operations at the "root" of the server.
- **Fixed:** TXDataWebDataset now correctly creates TStringField for GUID properties (TGuidField is not supported in TMS Web Core).
- **Fixed:** Removed memory leaks in desktop client of the JWT Auth demo.
- **Fixed:** "New TMS XData Server" wizard updated with option to use new TMS Aurelius native database drivers.

Version 3.1 (Jul-2018)

- **New: XData Music Web Application demo.** A complete web application example using a TMS XData server with TMS Web Core as web front-end. Full source code is provided, and the online version of the demo is available at https://app.devgems.com/xdata/music/app.
- **New: TXDataWebClient.ReferenceSolvingMode** allows automatic solving of $ref occurrences in JSON response, which is now the default behavior.
- **New: Demos XData Web-Client Framework**, showing use of TXDataWebClient, TXDataWebDataset and integration with FNC Grid.
- **New: TXDataWebConnection.OnResponse** event intercepts all successful HTTP requests made by the XData web client framework (3.1.1).
- **Fixed:** Servers with Int64 type in model were not supported when used from TMS Web client (3.1.1).

Version 3.0 (Jul-2018)

- **New:** Full-featured TMS XData Web-Client Framework provides RAD, design-time and high-level components for building web applications using TMS Web Core and TMS XData. Buildling a REST API-based web application has never been as easy. This includes dataset-like usage that feels home to Delphi developers.

- **New:** TXDataWebClient.RawInvoke allows low-level invoking of service operations from TMS Web Core applications.
- **New:** TXDataWebConnection.OnRequest and TXDataWebClient.OnRequest events allow intercepting and modifying outgoing requests (for example, to add authentication headers).
- **New:** TXDataWebDataset.CurrentData provides the underlying JSON object of the current web dataset record.
- **New:** TXDataWebConnection.OnError event which is fired whenever an error happens while performing requests to XData server from TMS Web Core apps.
- **New:** TXDataWebClient request methods (List, Get, Post, RawInvoke, etc.) now have overloaded methods that accept callback functions.
- **New:** TXDataWebConnection.Open method allows for providing callbacks for connection successful and connection error.
- **New:** TXDataWebConnection.DesignData allows for adding custom headers at design-time (for example authorization) to properly connect to XData server.
- **Improved:** Smooth design-time experience with TXDataWebDataset. Manually loading field defs is not necessary anymore - just connect a dataset to a web connection, provide the URL and fields will be loaded automatically at both design-time or runtime. Design-time menu option "Load Field Defs" has been removed.
- **Improved:** Editing TXDataWebDataset.EntitySetName at design-time in object inspector now provides a combobox with the list of available names retrieved from the server.
- **Improved:** Dataset fields now supported in TXDataWebDataset. They will contain data (list of objects) for associated lists (many-valued associations).
- **Improved:** TXDataWebDataset.SetJsonData now accepts non-array values (to edit a single object, for example).
- **Improved:** TXDataWebClient.Get overloaded method allows for passing an additional query string (to use $expand option for example).
- **Improved:** TXDataServerModule.Create overload requires just BaseUrl - no database connection pool is needed. Useful for XData servers that do not connect to a database or have a specific database usage.
- **Improved:** TMS Web Core-only TXDataConnection and TXDataDataset components have been renamed to TXDataWebConnection and TXDataWebDataset. This is a breaking change.
- **Fixed:** Exceptions that were not inherited from EXDataHttpException were wrongly reporting an error code (regression).
- **Fixed:** Memory leaks when using mobile/Linux compilers (when using automatic reference counting).

Version 2.9 (May-2018)

- **New: OnModuleException event** allows for a more flexible custom error-handling processing of exception raised during XData processing request.
- **Fixed:** OnEntityModifying event was being fired before merge operation thus not providing correct information about the object in manager and previous state. Breaking change that can be disabled by using *_FixEntityModifyingOnUpsert := False*.
- **Fixed:** Deserialization of array of objects was keeping the same instance for all objects in the array.

Version 2.8 (Feb-2018)

- **New: TMS RADical WEB enabled!** The TMS framework for front-end Web development; XData users now have a high-level framework to write Web clients, by using the new TXDataConnection, TXDataDataset and TXDataWebClient components to access XData servers.
- **New: TXDataClient.Count method.** Allows retrieving the number of entities in a specific resource endpoint, optionally using a query filter:

```
TotalCustomersFromLondon := XDataClient.Count('$filter=City eq ''London''');
```

- **New: TQueryParser.AddMethod allows registering custom SQL Functions to be called from XData query API.** Using TMS Aurelius you can add custom SQL functions to use from LINQ. You can now also register such a method in XData so clients can use such functions from the XData API:

```
TQueryParser.AddMethod('unaccent', TQueryMethod.Create('unaccent', 1));
```

Then use it from query API:

```
http://server:2001/tms/xdata/Customer?$filter=unaccent(Name) eq 'Andre'
```

- **New: /$model** built-in URL returns metadata for the whole XData API.
- **New: MultiTentant Demo** included in distribution shows how to use XData with multi-tentant applications (multiple databases being accessed by the same server). The demo shows how to specify the tentant by different URL or by using a custom HTTP header.
- **Improved:** A connection was being retrieved from the pool even when the context manager was not being used by a service operation.
- **Fixed:** Service operations invoked via GET method using enumerated type parameters are now supported.

Version 2.7 (Oct-2017)

- **New: OpenAPI/Swagger support**! XData servers provide a JSON file containing the OpenAPI Specification (OAS, formerly Swagger) for your whole server API. This opens up a lot of possibilities, usage of several tools of the OpenAPI ecosystem is now possible. Main one is the Swagger UI, a web front-end to describe and test your API.
- **New: Several new types supported in service operations.** Lots of new Delphi types can now be (de)serialized to JSON, meaning that these types can be used in service operations, either as input parameters, or as function results, or even as properties of PODO objects sent/received. The new supported types include:
  - **Generic arrays**: values of type TArray<T>, where T can be any supported type;
  - **Generics lists of primitive types**: values of type TList<T>, where T can by any supported type. In previous versions only lists of objects were supported;
  - **Sets**: values of type "set of T" where T is an enumerated type;
  - **TStrings** type.
- **New: Support for default parameter values in service operations.** You can now specify default values as parameters in service operations, using the [XDefault] attribute, and make them not required when invoking the service operations from non-Delphi clients (default parameters were already supported in Delphi clients).
- **New: JsonInclude attribute in PODO classes.** You can configure how properties/fields with default values will be serialized in PODO classes. This attribute makes it possible to have a smaller JSON representation, by removing properties which value is null, zero, empty string, etc. Usage example: [JsonInclude(TInclusionMode.NonDefault)].
- **New: JsonNamingStrategy attribute.** Allows you to automatically define a strategy for naming the JSON properties based on the field/property names of the class. You can use several predefined strategies: default, identity, camelCase, snake_case, identityCamelCase, identity_snake_case.
- **New: JsonEnumValues attribute.** You can use this attribute to specify different values when serializing enumerated values.
- **New: TXDataRequestHandler.ManagedObjects property.** This property provides more flexibility when it comes to automatic memory management at server side, allowing you to add objects to that collection and don't worrying about destroyed them.
- **Fixed:** Error when using params of type TDate (TDateTime was not affected) in service operations (Regression).
- **Fixed:** Design-time wizard icon not showing correctly in Delphi 10.2 Tokyo.

Version 2.6 (Jul-2017)

- **New: XDataProperty and XDataExcludeProperty attributes.** When converting Aurelius entities to/form JSON, by default XData serializes all (and only) fields/properties that are mapped using Aurelius attributes. You can now include other (transient) properties or fields in JSON by using XDataProperty attribute. You also have the option to exclude an existing mapped member by using XDataExcludeProperty attribute.
- **New: JsonProperty and JsonIgnore attributes.** When serializing regular Delphi objects (DTO) to JSON, XData includes all class fields by default. You can use JsonProperty attribute to add properties to the JSON (and/or change their name in JSON object) and use JsonIgnore attribute to exclude fields from JSON.
- **New: Support for passing parameters by reference in service operations.** You can now declare and use service operations (methods) that receive parameters by reference. A server method like *Swap(var A, B: Integer)* is now supported for both server-side and client-side (using TXDataClient or regular HTTP requests).
- **New: Filter functions startswith, endswith and contains.** You can now perform $filter queries with those functions to search for substrings in entity fields. For example: *?$filter=contains(Name, 'Walker') or startswith(Name, 'John')*.
- **New: Filter function concat.** You can now use *concat* function when performing queries to concatenate two or more strings. For example: *?$filter=concat(concat(FirstName, ' '), LastName) = 'James Smith'*.
- **New: TXDataSeverModule.UnknownMemberHandling property.** You can now optionally tell the server to ignore unknown JSON properties sent by the client (instead of raising an InvalidJsonProperty error).
- **Fixed:** Using TStream as a parameter for service operations was causing "JSON converter not found" error.

Version 2.5 (May-2017)

- **New: Linux support using Delphi 10.2 Tokyo and later.** Using XData on Linux doesn't require any specific change in your XData-related code. All your Linux specific code relates to TMS Sparkle - creating an Apache module and configuring TMS Sparkle to respond to requests. Once you do that, you just add the XData module to the Sparkle dispatcher as usual.
- **Fixed:** Location header of POST responses now uses host of the client request, not the host configured in XData module.

Previous Versions

Version 2.4 (Mar-2017)

- New: Delphi 10.2 Tokyo Support.

- Fixed: Server-side stack overflow when trying to load a proxy property sent by the client, if the parent entity of that proxy was merged into the current Object Manager.

Version 2.3 (Jan-2017)

- New: Demo project showing authentication using JWT (JSON Web Token).
- Fixed: Sporadic server error loading proxied lists sent by the client.
- Fixed: JWT encoding not working on XE6.
- Fixed: XData Server Wizard generating corrupted source code when TMS Component Pack was installed.

Version 2.2 (Aug-2016)

- New: Url convention now allows $count path segment to retrieve number of entities in a resource.
- Fixed: RefCount property was wrongly being serialized in PODO objects from mobile clients.
- Fixed: TXDataClient.Service<T> failing for Android clients when compiling in Release config.

Version 2.1 (Jul-2016)

- New: $expand query option allows clients to have full control on how associated entities appear in JSON response.
- New: Support for entities that have associations in ID (primary key containing foreign keys).
- New: Support for Variant-type parameters in service operations when using GET HTTP method.
- New: **Breaking change:** TXDataServerModule.PutMode property controls how PUT will behave at server-side.
- New: TXDataServerModule.SerializeInstanceRef property controls how instances of same object will be represented in JSON response.
- New: TXDataServerModule.SerializeInstanceType property controls how xdata type metadata will appear in JSON response.
- New: Support for Nullable<T> values in PODO classes.
- Improved: Errors on query syntax now return http code 400 instead of 500.
- Fixed: JSON Proxies (@xdata.proxy) sent by the client were not being solved when reading such properties at server-side.
- Fixed: Association references (@xdata.ref) were not being solved when receiving entities in service operations parameters.

Version 2.0 (May-2016)

- New: Service operations can now receive and return any type of object. This increases flexibility significantly as you can use any type of object for structure input/output parameters, or to send/receive DTO classes, etc.
- New: Server-Side Events allow subscribing listeners events and perform additional server-side logic
- New: JWT (Json Web Token) authentication and Basic authentication, thanks to new TMS Sparkle.
- New: Authentication and authorization mechanism, based on TMS Sparkle.
- New: Delphi 10.1 Berlin support.
- New: Service operations can now receive and return TJSONAncestor objects (Delphi XE6 and up only). This allows full control over the JSON request and response.
- New: Service operations can now receive and return TCriteriaResult objects, making it easy to return Aurelius query results that use projections.
- New: $inlinecount query option allow retrieving the total number of entities when using paged results.
- New: Method TXDataModelBuilder.RemoveEntitySet for more flexibility when building XData models.
- New: TXDataServerModule.SetEntitySetPermissions allows configuring what operations are available in each resource type.
- Improved: All server-side operation (entity CRUD, service operation execution) are now performed in database transactions.
- Improved: TXDataOperationContext.Current now also available in entity resources.

Version 1.6 (Feb-2016)

- New: Design-time wizard to create a XData Server with a few clicks.
- Fixed: Service operation using enumerated types as parameters or function results not working properly.
- Fixed: EntitySet requests were not taking xdata-expandlevel header into consideration. This is a breaking change.

Version 1.5.1 (Sep-2015)

- New: Delphi 10 Seattle support.

Version 1.5 (Aug-2015)

- New: Several built-in functions available to increase flexibility when querying objects in REST requests. New available functions are Upper, Lower, Length, Substring, Position, Year, Month, Day, Hour, Minute and Second.

- New: TXDataClient.HttpClient property provides low level access to the Http client and allows using OnSendingRequest events to customize requests.
- New: TXDataOperationContext Request and Response properties gives full control for service implementations to customize the processing of client requests.
- New: TXDataServerModule.DefaultExpandLevel allows defining the expand level of JSON responses when it's not defined by client request.
- Fixed: TXDataClient Get, Delete, Patch and Put operations were broken when using entities with composite id.
- Fixed: POST requests not working correctly with entities with composite key.
- Fixed: Data Modification requests sending content-type header with parameter (for example, ";charset=UTF8") were causing errors.

Version 1.4 (Jun-2015)

- New: HttpMethod attribute allows specifying the HTTP method a service operation should respond to.
- New: URIPathSegment attribute allows specifying a different name for operation/service to be used in URL.
- Fixed: GET requests with query order was sometimes causing the same column to appear multiple times in a "ORDER BY" SQL clause.

Version 1.3.1 (Apr-2015)

- New: Delphi XE8 support.

Version 1.3 (Mar-2015)

- New: Support for CORS (Cross-origin resource sharing) preflighted requests.

Version 1.2 (Dec-2014)

- New: TXDataClient methods Get, Post, Put, Delete and List allows easy and high-level access to XData server objects from Delphi clients.
- New: Android and iOS support for XData client objects.
- New: Server support for "*x-http-method-override*" header (allowing clients to tunnel HTTP methods to server through POST requests).
- Fixed: Issues with floating-point literals in Query URL with non-English server systems.
- Fixed: Service operations returning nil entities should respond with status code 404.

Version 1.1 (Oct-2014)

- New: Service Operations allow adding custom business logic to your server using interfaces and methods.
- New: Multi-Model design makes it easy to create multiple servers with different mapping, types and service operations.
- New: UserName and Password properties in TXDataServerModule for basic authentication protection.
- Fixed: Malfunctioning with $skip and $top query options.

Version 1.0.1 (Sep-2014)

- New: Delphi XE7 support.

Version 1.0 (Aug-2014)

- First public release.

Licensing and 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 a 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 the TMS Software web site. Any other way of distribution requires a written authorization from 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 the component distributions, if available.
- Search the TMS support forum and the TMS newsgroups to see if your question has not been already answered.
- Make sure you installed 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 are using, and please also let us know which OS you use.
- For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee a fast reply.

Send your 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.

Version 4.3

TXDataWebDataset now creates string fields (TStringField) for XData entity properties that are enumerated types. In previous versions, integer fields were created. This is actually a bug fix. Integer fields are used for enumerated types when using TMS Aurelius directly in desktop/ mobile applications, because an enumerated type in Delphi is, in the end, an ordinal (integer) type.

For TMS Web Core, though, there are no entities, but a direct manipulation of the JSON returned by XData. And in XData JSON representation, an enumerated type is represented as a string (the enum name). For modifying or inserting such property in an entity, user need to provide the string value.

This is a bug fix but breaks existing applications. If by any chance you need to keep the enum fields as integers, set TXDataWebDataset.EnumAsIntegers property to true.

Version 3.0

This version is the first "official" (non-beta) support for TMS Web Core, and it has renamed two key components:

- *TXDataWebDataset* (formely TXDataDataset)
- *TXDataWebConnection* (formely TXDataConnection)

If you have used a previous version and used the above components in TMS Web Core applications, you might get an error "TXDataConnection not found" when opening the form asking for Cancel or Ignore. Just click Cancel, close the project, and open the offending .pas/.dfm in your preferred text editor (not Delphi - it can be Notepad, for example).

Then replace any occurrence of *TXDataDataset* with *TXDataWebDataset*, and any occurrence of *TXDataConnection* with *TXDataWebConnection*, in both .pas and .dfm files. Save the files and this will fix the issue.

Version 2.9

OnEntityModifying event was being fired before merge operation thus not providing correct information about the object in manager and previous state. In case you want to go back to the previous behavior, set *_FixEntityModifyingOnUpsert* global variable to false.

```
uses XData.Server.Module;

_FixEntityModifyingOnUpsert := False;
```

Version 2.1

- *TXDataServerModule.PutMode* property controls how PUT will behave at server-side.

Version 1.5.2

This version fixes a bug that the header xdata-expandlevel was being ignored when returning entity sets. Even though it's a bug fix, this is a breaking change as the server changed its behavior.

Version 1.1

a. Example provided in Building The Entity Model topic has changed to illustrate how to correctly build the model in version 1.1 (*TXDataModelBuilder* constructor and *Build* method changed the signature).

b. Model is not owned by the TXDataServerModule class and must be now destroyed explicitly. So in the following code:

```
XDataServerModule := TXDataServerModule.Create(MyUrl, MyConnectionPool, MyModel);
```

now you need to destroy the model when the server is shutdown:

```
MyModel.Free;
```

Note this is not a critical issue because if there is a leak, it will only happen when the whole server application is shutdown. Also, with the new multiple model feature, it's very rare that you would need to create your own model explicitly.

Version 2.1 - Breaking Changes

Version 2.1 introduces the TXDataServerModule.PutMode property which defines how PUT requests will be implemented by the server. The two options are "Update" and "Merge". Basically, the server receives the object to be updated in JSON, deserializes it to an object, and uses the TMS Aurelius TObjectManager and either the Update or the Merge method:

```
// TXDataPutMode.Update
Manager.Update(ReceivedObject);
Manager.Flush;
```

```
// TXDataPutMode.Merge
Manager.Merge(ReceivedObject);
Manager.Flush;
```

The breaking change here is that until the version prior to 2.1, the one and only behavior was *Update*. Now, the default has changed to *Merge*. We're not aware of any serious issue with this change, but the behavior has changed in some specific cases - for the better. Suppose you have a JSON request with the following object:

```
{
  "$id": 1,
  "@xdata.type": "XData.Default.Product",
  "Id": 10,
  "Name": "Ball",
  "Category1": {
    "$id": 2,
    "@xdata.type": "XData.Default.Category",
    "Id": 5,
    "Name": "Toys"
  },
  "Category2": {
    "$id": 2,
    "@xdata.type": "XData.Default.Category",
    "Id": 5,
    "Name": "Toys"
  }
}
```

In the example above, the client is sending a Product entity which has two category properties: Category1 and Category2. Both properties are pointing to the same "Toys" category (id = 5) so the PUT request is intending to set both product categories to "Toys".

With the previous behavior, the server would raise an error, because there are two different instances of the same category in the product. With the new Merge behavior, the PUT request will process just fine.

In any case, if you want to go back to the previous behavior you can just set *TXDataServerModule.PutMode* property to *TXDataPutMode.Update*.