Dotmim.Sync

Release 0.9.5

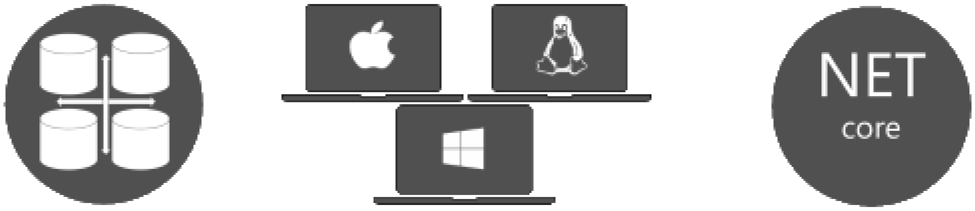
Aug 16, 2023

DotMim.Sync (**DMS**) is a straightforward framework for syncing relational databases, developed on top of **.Net Standard 2.0**, available and ready to use within **Xamarin**, **MAUI**, **.NET Core 3.1**, **.NET 6 & 7** and so on :)

Available for syncing **SQL Server**, **MySQL**, **MariaDB**, **PostgreSQL** and **Sqlite** databases.

Note: The source code is available on Github.

This framework is still in beta. There is no support other than me and the time I can put on it. Don't be afraid to reach me out, but expect delay sometimes :)



CHAPTER 1

Starting from scratch

Here is the easiest way to create a first sync, from scratch :

- Create a **.NET Core 3.1** or **.NET 6 / .NET 7** console application.
- Add the nugets packages DotMim.Sync.SqlServer and DotMim.Sync.Sqlite
- If you don't have any hub database for testing purpose, use this one : AdventureWorks lightweight script for SQL Server
- If you want to test **MySql**, use this script : AdventureWorks lightweight script for MySQL Server

Add this code:

```
// Sql Server provider, the "server" or "hub".
SqlSyncProvider serverProvider = new SqlSyncProvider(
   @"Data Source=.;Initial Catalog=AdventureWorks;Integrated Security=true;");

// Sqlite Client provider acting as the "client"
SqliteSyncProvider clientProvider = new SqliteSyncProvider("advworks.db");

// Tables involved in the sync process:
var setup = new SyncSetup("ProductCategory", "ProductDescription", "ProductModel",
 "Product", "ProductModelProductDescription", "Address", "Customer",
 "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail" );

// Sync agent
SyncAgent agent = new SyncAgent(clientProvider, serverProvider);

do
{
   var result = await agent.SynchronizeAsync(setup);
   Console.WriteLine(result);

} while (Console.ReadKey().Key != ConsoleKey.Escape);
```

And here is the result you should have, after a few seconds:

```
Synchronization done.
      Total changes  uploaded: 0
      Total changes  downloaded: 3514
      Total changes  applied on client: 3514
      Total changes  applied on server: 0
      Total changes  failed to apply on client: 0
      Total changes  failed to apply on server: 0
      Total resolved conflicts: 0
      Total duration :00.00:02.125
```

You're done !

Now try to update a row in your client or server database, then hit enter again. You should see something like that:

```
Synchronization done.
      Total changes  uploaded: 0
      Total changes  downloaded: 1
      Total changes  applied on client: 1
      Total changes  applied on server: 0
      Total changes  failed to apply on client: 0
      Total changes  failed to apply on server: 0
      Total resolved conflicts: 0
      Total duration :00.00:00.030
```

Yes it's blazing fast !

CHAPTER 2

Need Help

Feel free to ping me: @sebpertus

2.1 Overview

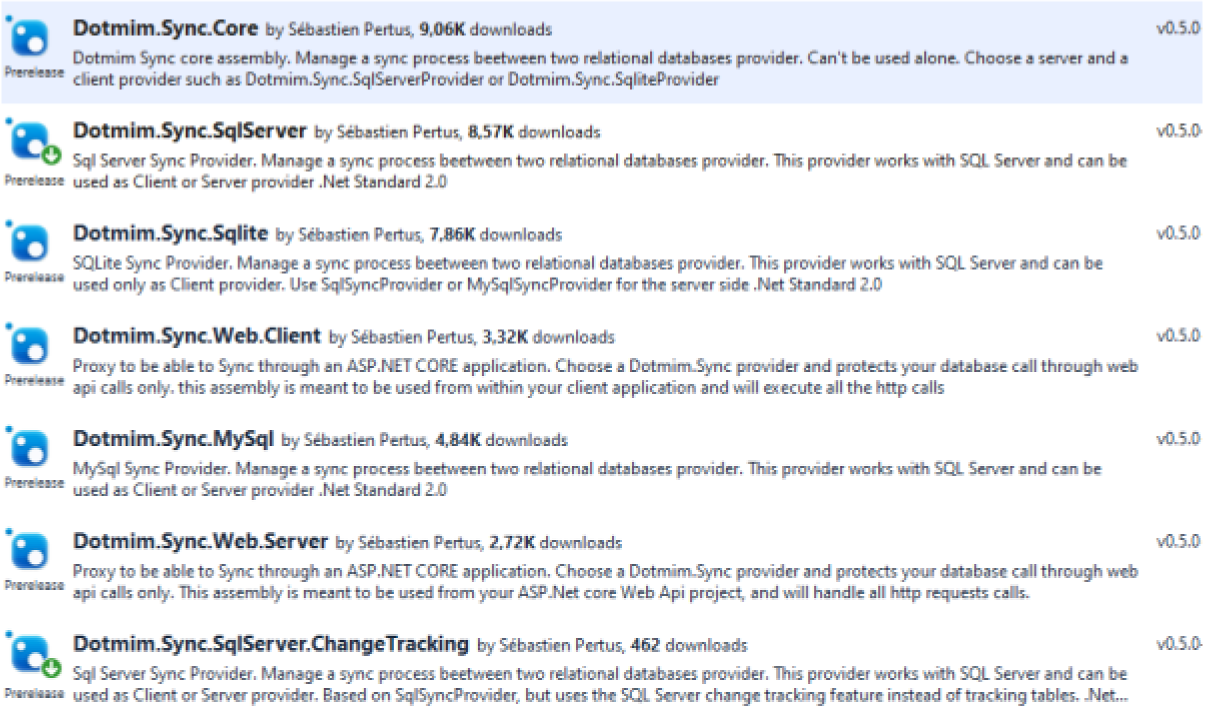
Dotmim.Sync (**DMS**) is the easiest way to handle a full **synchronization** between one server database and multiples clients databases.
Dotmim.Sync is cross-platforms, multi-databases and based on **.Net Standard 2.0**.
Choose either **SQL Server**, **SQLite**, **MySQL**, **MariaDB** and (hopefully, I hope soon. . .) Oracle or PostgreSQL !

For simplicity, we can say **DMS** framework.

No need to handle any configuration file, or code generation code or whatever.
Just a few lines of code, with the list of tables you want to synchronize then call `SynchronizeAsync()` and you're done !

2.1.1 Nuget packages

Basically, **DMS** is working with *sync database providers*, that are available through nuget, from the **Visual Studio** interface:



Dotmim.Sync.Core by Sébastien Pertus, **9,06K** downloads v0.5.0
Prerelease
Dotmim Sync core assembly. Manage a sync process beetween two relational databases provider. Can't be used alone. Choose a server and a client provider such as Dotmim.Sync.SqlServerProvider or Dotmim.Sync.SqliteProvider

Dotmim.Sync.SqlServer by Sébastien Pertus, **8,57K** downloads v0.5.0
Prerelease
Sql Server Sync Provider. Manage a sync process beetween two relational databases provider. This provider works with SQL Server and can be used as Client or Server provider .Net Standard 2.0

Dotmim.Sync.Sqlite by Sébastien Pertus, **7,86K** downloads v0.5.0
Prerelease
SQLite Sync Provider. Manage a sync process beetween two relational databases provider. This provider works with SQL Server and can be used only as Client provider. Use SqlSyncProvider or MySqlSyncProvider for the server side .Net Standard 2.0

Dotmim.Sync.Web.Client by Sébastien Pertus, **3,32K** downloads v0.5.0
Prerelease
Proxy to be able to Sync through an ASP.NET CORE application. Choose a Dotmim.Sync provider and protects your database call through web api calls only. this assembly is meant to be used from within your client application and will execute all the http calls

Dotmim.Sync.MySql by Sébastien Pertus, **4,84K** downloads v0.5.0
Prerelease
MySql Sync Provider. Manage a sync process beetween two relational databases provider. This provider works with SQL Server and can be used as Client or Server provider .Net Standard 2.0

Dotmim.Sync.Web.Server by Sébastien Pertus, **2,72K** downloads v0.5.0
Prerelease
Proxy to be able to Sync through an ASP.NET CORE application. Choose a Dotmim.Sync provider and protects your database call through web api calls only. This assembly is meant to be used from your ASP.Net core Web Api project, and will handle all http requests calls.

Dotmim.Sync.SqlServer.ChangeTracking by Sébastien Pertus, **462** downloads v0.5.0
Prerelease
Sql Server Sync Provider. Manage a sync process beetween two relational databases provider. This provider works with SQL Server and can be used as Client or Server provider. Based on SqlSyncProvider, but uses the SQL Server change tracking feature instead of tracking tables. .Net...

Obviously, you can add them through your command line, assuming you are developing with **Visual Studio Code**, **Rider** or even **Notepad** :)

```
# Adding the package required to synchronize a SQL Server database:
dotnet add package Dotmim.Sync.SqlServer
# Adding the package required to synchronize a SQL Server database, using Change Tracking feature:
dotnet add package Dotmim.Sync.SqlServer.ChangeTracking
# Adding the package required to synchronize a MySQL database:
dotnet add package Dotmim.Sync.MySql
# Adding the package required to synchronize a MariaDB database:
dotnet add package Dotmim.Sync.MariaDB
# Adding the package required to synchronize a SQLite database:
dotnet add package Dotmim.Sync.Sqlite
```

For instance, if you need to synchronize two **MySql** databases, the only package you need to install, on both Server and Client side, is `Dotmim.Sync.MySql`.

On the other side, if you need to synchronize a SQL server database, with multiple SQLite client databases, install `Dotmim.Sync.SqlServer` (or `Dotmim.Sync.SqlServer.ChangeTracking`) on the server side and then install `Dotmim.Sync.Sqlite` on each client.

Note: The package `Dotmim.Sync.Core` is the core framework, and is used by all the providers. You don't have to explicitely add it to your projects, since it's always part of the provider you've just installed.

The last two packages available, `Dotmim.Sync.Web.Client` and `Dotmim.Sync.Web.Server` are used for a specific scenario, where you server database is not accessible directly, but instead is available and exposed through a **Web Api**, built with **ASP.Net Core** or **ASP.NET**.

All packages are available through **nuget.org**:

Dotmim.Sync.Core : https://www.nuget.org/packages/Dotmim.Sync.Core
Dotmim.Sync.SqlServer : https://www.nuget.org/packages/Dotmim.Sync.SqlServer
Dotmim.Sync.SqlSyncChangeTrackingProvider :
https://www.nuget.org/packages/Dotmim.Sync.SqlServer.ChangeTracking
Dotmim.Sync.Sqlite : https://www.nuget.org/packages/Dotmim.Sync.Sqlite
Dotmim.Sync.MySql : https://www.nuget.org/packages/Dotmim.Sync.MySql
Dotmim.Sync.MariaDB : https://www.nuget.org/packages/Dotmim.Sync.MariaDB
Dotmim.Sync.Web.Server : https://www.nuget.org/packages/Dotmim.Sync.Web.Server
Dotmim.Sync.Web.Client : ttps://www.nuget.org/packages/Dotmim.Sync.Web.Client

2.1.2 Tutorial: First sync

First sync

This tutorial will describe all the steps required to create a first sync between two relational databases:

- If you don't have any databases ready for testing, you can use:
  - For **SQL Server** : AdventureWorks for SQL Server
  - For **MySQL** : AdventureWorks for MySQL
- The script is ready to execute in SQL Server (or MySQL Workbench). It contains :
  - A lightweight AdvenureWorks database, acting as the Server database (called AdventureWorks)
  - An empty database, acting as the Client database (called Client)

Hint: You will find this sample here : HelloSync sample

You can see this sample as well, live, hosted on dotnetfiddle : Hello Sync On dotnetfiddle

Warning: In the code sample below, we are using a special provider called `SqlSyncChangeTrackingProvider`. This provider is using the **CHANGE_TRACKING** feature from **SQL SERVER**.

Before running this code, use this SQL statement on your server database to enable the *Change Tracking*:

```
ALTER DATABASE AdventureWorks SET CHANGE_TRACKING=ON
(CHANGE_RETENTION=2 DAYS,AUTO_CLEANUP=ON)
```

Otherwise, if you don't want to use the *Change Tracking* feature, just change `SqlSyncChangeTrackingProvider` to `SqlSyncProvider`

```
// First provider on the server side, is using the Sql change tracking feature.
var serverProvider = new SqlSyncChangeTrackingProvider(serverConnectionString);

// IF you want to try with a MySql Database, use the [MySqlSyncProvider] instead
// var serverProvider = new MySqlSyncProvider(serverConnectionString);

// Second provider on the client side, is the [SqliteSyncProvider] used for SQLite␣
↪databases
// relying on triggers and tracking tables to create the sync environment
```

(continues on next page)

(continued from previous page)

```
var clientProvider = new SqliteSyncProvider(clientConnectionString);

// Tables involved in the sync process:
var setup = new SyncSetup("ProductCategory", "ProductModel", "Product",
    "Address", "Customer", "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail");

// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, serverProvider);

do
{
    // Launch the sync process
    var s1 = await agent.SynchronizeAsync(setup);
    // Write results
    Console.WriteLine(s1);

} while (Console.ReadKey().Key != ConsoleKey.Escape);

Console.WriteLine("End");
```

And here is the result you should have, after a few seconds:

```
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 3514
        Total changes  applied on client: 3514
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 0
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:02.125
```

It took **2 seconds** on my machine to make a full synchronization between the **Server** and the **Client**.

Second sync

This first sample took **2 seconds** to make a *full* sync between a **Server** and a **Client**.

It's a little bit long (I'm kidding… no), because, under the hood, the `Dotmim.Sync` framework, on the **first sync only**, will have to:

- Get the schema from the **Server** side and create all the tables on the **Client** side, if needed. (yes, you don't need a client database with an existing schema)
- Create on both side all the required stuff to be able to manage a full sync process, creating *tracking* tables, stored procedures, triggers and so on … be careful, `Dotmim.Sync` could be a little bit intrusive if you're not using the `SqlSyncChangeTrackingProvider` provider :)
- Then eventually launch the first sync, and get the **2752** items from the **Server**, and apply them on the **Client**.

Now everything is configured and the first sync is successfull.

We can add **101** items in the *ProductCategory* table (on the server side, *Adventureworks*):

```
Insert into ProductCategory (ProductCategoryID, Name)
Select newid(), SUBSTRING(CONVERT(varchar(255), NEWID()), 0, 7)
Go 100
```

From the same console application (indeed, we have a *do while* loop), same code, just hit *enter* to relaunch the synchronization and see the results:

```
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 100
        Total changes  applied on client: 100
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 0
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.059
```

Boom, it's . . . . fast, isn't it ?

2.2 How does it work

Basically, **DMS** architecture is composed of several business objects:

- **Providers** : A provider is in charge of the communication with the local database. You can choose various providers, like `SQL`, `MySQL`, `MariaDB` or `Sqlite`. Each provider can work on both side of the sync architecture : Server side or Client side.
- **Orchestrators** : An orchestrator is agnostic to the underlying database. it communicates with the database through a provider. A provider is always required when you're creating a new orchestrator. We have two kind of orchestrator : *local* and *remote* (or let's say *client side* and *server side* orchestrators)
- **SyncAgent**: There is only one sync agent. This object is responsible of the correct *flow* between two orchestrators. The sync agent will:
  - Create a local orchestrator with a typed provider.
  - Create a remote orchestrator with a typed provider.
  - Synchronize client and server, using all the methods from the orchestrators.

2.2.1 Overview

Here is the big picture of the components used in a simple synchronization, over **TCP**:

If we take a close look to the HelloSync sample:

```
var serverProvider = new MySqlSyncProvider(serverConnectionString);
var clientProvider = new SqliteSyncProvider(clientConnectionString);

var setup = new SyncSetup("ProductCategory", "ProductModel", "Product");

var agent = new SyncAgent(clientProvider, serverProvider);

var result = await agent.SynchronizeAsync(setup);

Console.WriteLine(result);
```

There is no mention of any `Orchestrators` here.

It's basically because the `SyncAgent` instance will create them under the hood, for simplicity.

We can rewrite this code, this way:

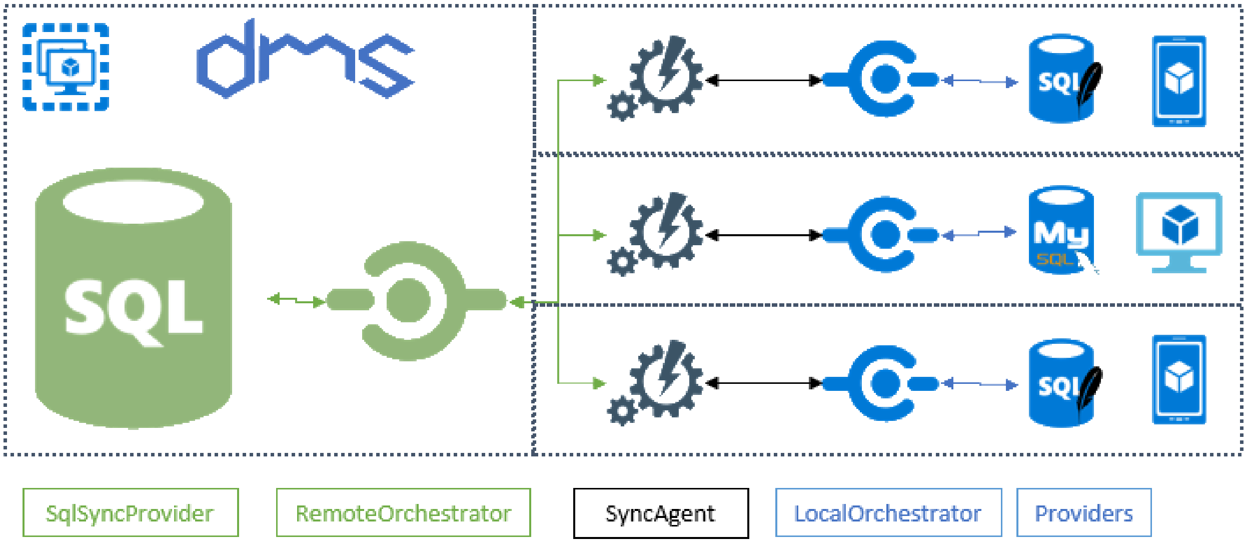
```csharp
// Create 2 providers, one for MySql, one for Sqlite.
var serverProvider = new MySqlSyncProvider(serverConnectionString);
var clientProvider = new SqliteSyncProvider(clientConnectionString);

// Setup and options define the tables and some useful options.
var setup = new SyncSetup("ProductCategory", "ProductModel", "Product");

// Define a local orchestrator, using the Sqlite provider
// and a remote orchestrator, using the MySql provider.
var localOrchestrator = new LocalOrchestrator(clientProvider);
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

// Create the agent with existing orchestrators
var agent = new SyncAgent(localOrchestrator, remoteOrchestrator);

// Launch the sync
var result = await agent.SynchronizeAsync(setup);

Console.WriteLine(result);
```

As you can see here, all the components are declared:

- Each provider : One Sqlite and One MySql
- Each orchestrator : a local orchestrator coupled with the Sqlite provider and a remote orchestrator coupled with the MySql provider
- One sync agent : The sync agent instance needs of course both orchestrators to be able to launch the sync process.

2.2.2 Multiple clients overview

Of course, a real scenario will involve more clients databases. Each client will have its own provider, depending on the local database type. And each client will have a sync agent, responsible of the sync process:

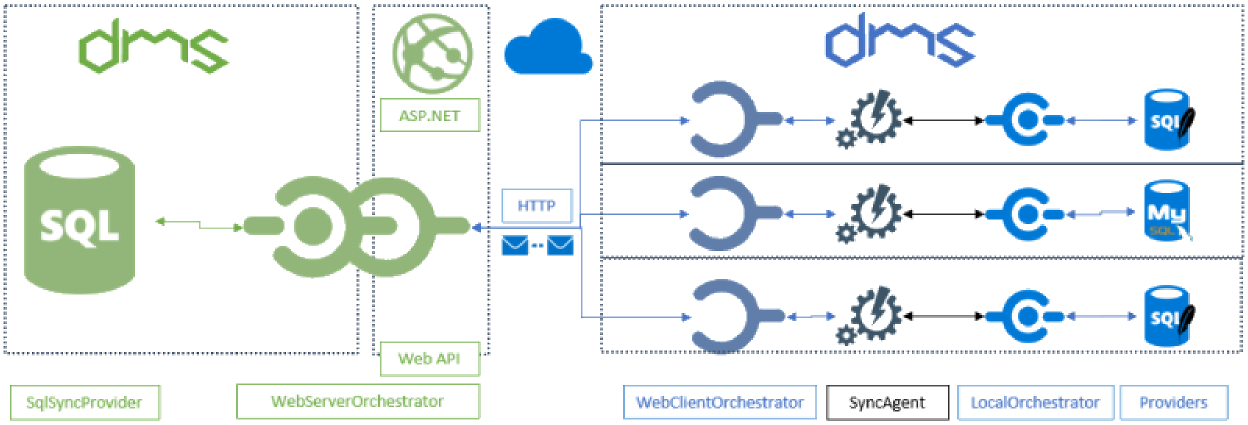


2.2.3 Sync over HTTP

In a real world scenario, you may want to protect your hub database (the *server side* database), if your clients are not part of your local network, like mobile devices which will communicate only through an http connection. In this particular scenario, the sync agent will not be able to use a simple RemoteOrchestrator, since this one works only on a tcp network. Here is coming a new orchestrator in the game. Or shoud I say *two* new orchestrators:

- The `WebRemoteOrchestrator`: This orchestrator will run locally, and will act "*as*" a orchestrator from the sync agent, but under the hood will generate an http request with a payload containing all the required information
- The `WebServerAgent`: On the opposite side, this web server agent is hosted through an exposed web api, and will get the incoming request from the `WebRemoteOrchestrator` and will then call the server provider correctly.

Here is the big picture of this more advanced scenario:



You can read more on the web architecture and how to implement it, here: Asp.net Core Web Api sync proxy

2.3 Synchronization types

You have one main method to launch a synchronization, with several optional parameters:

```
SynchronizeAsync();
SynchronizeAsync(IProgress<ProgressArgs> progress);
SynchronizeAsync(CancellationToken cancellationToken);
SynchronizeAsync(SyncType syncType);
SynchronizeAsync(SyncType syncType, CancellationToken cancellationToken);
```

You can use the `CancellationToken` object whenever you want to rollback an "*in progress*" synchronization. And since we have an async synchronization, you can pass an `IProgress<ProgressArgs>` object to have feedback during the sync process.

Note: The progression system is explained in the next chapter Progress

let's see now a straightforward sample illustrating the use of the `SyncType` argument.

Hint: You will find the sample used for this chapter, here : SyncType sample

```
SqlSyncProvider serverProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"AdventureWorks"));
SqlSyncProvider clientProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"Client"));

var setup = new SyncSetup("ProductCategory", "ProductModel", "Product", "Address",
↪"Customer",
        "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail");

SyncAgent agent = new SyncAgent(clientProvider, serverProvider);

var syncContext = await agent.SynchronizeAsync(setup);

Console.WriteLine(syncContext);
```

Here is the result, after the **first initial** synchronization:

```
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 2752
        Total changes  applied: 2752
        Total resolved conflicts: 0
        Total duration :0:0:4.720
```

As you can see, the client has downloaded 2752 lines from the server.

Obviously if we made a new sync, without making any changes neither on the server nor the client, the result will be :

```
SqlSyncProvider serverProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"AdventureWorks"));
SqlSyncProvider clientProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"Client"));

SyncAgent agent = new SyncAgent(clientProvider, serverProvider);

var syncContext = await agent.SynchronizeAsync();

Console.WriteLine(syncContext);
```

Note: Since you've made a first sync before, the setup is already saved in the databases. So far, no need to pass the argument anymore now.

```
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 0
        Total changes  applied: 0
        Total resolved conflicts: 0
        Total duration :0:0:0.382
```

Ok make sense !

2.3.1 SyncType

The `SyncType` enumeration allows you to **reinitialize** a client database (already synchronized or not).
For various reason, you may want to re-download the whole database schema and rows from the server (bug, out of sync, and so on . . .)

`SyncType` is mainly an enumeration used when calling the `SynchronizeAsync()` method:

```
public enum SyncType
{
        /// <summary>
        /// Normal synchronization
        /// </summary>
        Normal,

        /// <summary>
        /// Reinitialize the whole sync database, applying all rows from the server␣
↪to the client
        /// </summary>
        Reinitialize,

        /// <summary>
        /// Reinitialize the whole sync database, applying all rows from the server␣
↪to the client,
        /// after tried a client upload
        /// </summary>
        ReinitializeWithUpload
}
```

- `SyncType.Normal`: Default value, represents a normal sync process.
- `SyncType.Reinitialize`: Marks the client to be resynchronized. Be careful, any changes on the client will be overwritten by this value.
- `SyncType.ReinitializeWithUpload`: Like *Reinitialize* this value will launch a process to resynchronize the whole client database, except that the client will *try* to send its local changes before making the resync process.

From the sample we saw before, here is the different behaviors with each `SyncType` enumeration value:

First of all, for demo purpose, we are updating a row on the **client**:

```
-- initial value is 'The Bike Store'
UPDATE Client.dbo.Customer SET CompanyName='The New Bike Store' WHERE CustomerId = 1
```

SyncType.Normal

Let’s see what happens, now that we have updated a row on the client side, with a *normal* sync:

```
SqlSyncProvider serverProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"AdventureWorks"));
SqlSyncProvider clientProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"Client"));

var syncContext = await agent.SynchronizeAsync();
```

(continues on next page)

(continued from previous page)

```

Console.WriteLine(syncContext);
```

```
Synchronization done.
                Total changes  uploaded: 1
                Total changes  downloaded: 0
                Total changes  applied: 0
                Total resolved conflicts: 0
                Total duration :0:0:1.382
```

The default behavior is what we were waiting for: Uploading the modified row to the server.

SyncType.Reinitialize

The `SyncType.Reinitialize` mode will **reinitialize** the whole client database.

Every rows on the client will be deleted and downloaded again from the server, even if some of them are not synced correctly.

Use this mode with caution, since you could lost some "*out of sync client*" rows.

```
SqlSyncProvider serverProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"AdventureWorks"));
SqlSyncProvider clientProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"Client"));

var syncContext = await agent.SynchronizeAsync(SyncType.Reinitialize);

Console.WriteLine(syncContext);
```

```
Synchronization done.
                Total changes  uploaded: 0
                Total changes  downloaded: 2752
                Total changes  applied: 2752
                Total resolved conflicts: 0
                Total duration :0:0:1.872
```

As you can see, the `SyncType.Reinitialize` value has marked the client database to be fully resynchronized.

The modified row on the client has not been sent to the server and then has been restored to the initial value sent by the server row.

SyncType.ReinitializeWithUpload

`ReinitializeWithUpload` will do the same job as `Reinitialize` except it will send any changes available from the client, before making the reinitialize phase.

```
SqlSyncProvider serverProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"AdventureWorks"));
SqlSyncProvider clientProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"Client"));

var syncResult = await agent.SynchronizeAsync(SyncType.ReinitializeWithUpload);

Console.WriteLine(syncResult);
```

```
Synchronization done.
                Total changes  uploaded: 1
                Total changes  downloaded: 2752
                Total changes  applied: 2752
                Total resolved conflicts: 0
                Total duration :0:0:1.923
```

In this case, as you can see, the `SyncType.ReinitializeWithUpload` value has marked the client database to be fully resynchronized, but the edited row has been sent correctly to the server.

2.3.2 Forcing operations on the client from server side

Warning: This part covers some concept explained later in the next chapters:

- Progression : Using interceptors.
- HTTP architecture : Using ASP.Net Web API

This technic applies if you do not have access to the client machine, allowing you to *force* operations from the client side.

It could be useful to *override* a normal synchronization, for example, with a reinitialization for a particular client, from the server side.

Note: Forcing a reinitialization from the server is a good practice if you have an **HTTP** architecture.

Here are the operation action you can use to force the client in a particular situation:

```
public enum SyncOperation
{
    /// <summary>
    /// Normal synchronization
    /// </summary>
    Normal = 0,

    /// <summary>
    /// Reinitialize the whole sync database, applying all rows from the server to␣
↪the client
    /// </summary>
    Reinitialize = 1,

    /// <summary>
    /// Reinitialize the whole sync database,
                /// applying all rows from the server to the client, after trying a␣
↪client upload
    /// </summary>
    ReinitializeWithUpload = 2,

    /// <summary>
    /// Drop all the sync metadatas even tracking tables and scope infos and make a␣
↪full sync again
```

(continues on next page)

(continued from previous page)

```
    /// </summary>
    DropAllAndSync = 4,

    /// <summary>
    /// Drop all the sync metadatas even tracking tables and scope infos and exit
    /// </summary>
    DropAllAndExit = 8,

    /// <summary>
    /// Deprovision stored procedures & triggers and sync again
    /// </summary>
    DeprovisionAndSync = 16,

    /// <summary>
    /// Exit a Sync session without syncing
    /// </summary>
    AbortSync = 32,
}
```

Hint: Use the client scope id to identify the current client trying to sync.

```
[HttpPost]
public async Task Post()
{
        // Get the current scope name
        var scopeName = this.HttpContext.GetScopeName();

        // Get the current client scope id
        var clientScopeId = this.HttpContext.GetClientScopeId();

        // override sync type to force a reinitialization from a particular client
        if (clientScopeId == OneParticularClientScopeIdToReinitialize)
        {
                webServerAgentRemoteOrchestrator.OnGettingOperation(operationArgs=>
                {
                                // this operation will be applied for the current sync
                                operationArgs.Operation = SyncOperation.Reinitialize;
                });
        }

        // handle request
        await webServerAgent.HandleRequestAsync(this.HttpContext);
}
```

2.3.3 SyncDirection

The *SyncType* enumeration allows you to synchronize **all** the tables.
Another way to synchronize your tables is to set a direction on each of them, through the *SyncDirection* enumeration.
This options is not global to all the tables, but should be set on each table.

You can specify three types of direction: **Bidirectional**, **UploadOnly** or **DownloadOnly**.

You can use the SyncDirection enumeration for each table in the SyncSetup object.

Note: Bidirectional is the default value for all tables added.

Since, we need to specify the direction on each table, the SyncDirection option is available on each SetupTable:

```
var syncSetup = new SyncSetup("SalesLT.ProductCategory", "SalesLT.ProductModel",
↪"SalesLT.Product",
                "SalesLT.Address", "SalesLT.Customer", "SalesLT.CustomerAddress");

syncSetup.Tables["Customer"].SyncDirection = SyncDirection.DownloadOnly;
syncSetup.Tables["CustomerAddress"].SyncDirection = SyncDirection.DownloadOnly;
syncSetup.Tables["Address"].SyncDirection = SyncDirection.DownloadOnly;

var agent = new SyncAgent(clientProvider, serverProvider);
```

SyncDirection.Bidirectional

This mode is the default one. Both server and client will upload and download their rows.

Using this mode, all your tables are fully synchronized with the server.

SyncDirection.DownloadOnly

This mode allows you to specify some tables to be only downloaded from the server to the client.

Using this mode, your server will not receive any rows from any clients, on the configured tables with the download only option.

SyncDirection.UploadOnly

This mode allows you to specify some tables to be uploaded from the client to the server only.

Using this mode, your server will not send any rows to any clients, but clients will sent their own modified rows to the server.

2.4 Scopes

2.4.1 What is a scope ?

Scopes are objects containing the list of the tables to synchronize.
They are stored in the *scope_info* table, on each side (**Server** data source / All **Client** data sources).

Note: You can change the default table name **scope_info** using the ScopeInfoTableName property from SyncOptions:

```
var options = new SyncOptions();
options.ScopeInfoTableName = "table_information";
```

Basically, a **scope** is defining a **setup** and its database **schema**, and is identified by a **unique name**.

For example, you can define multiples scopes representing:

- A first scope called **“products”** : Contains the *Product* , *ProductCategory* and *ProductModel* tables.
- A second scope called **“customers”**: Contains the *Customer* and related *SalesOrderHeader* tables.
- A third scope with the default name (default **“DefaultScope”**), containing all the tables to sync.

The *scope_info* table (and the scope record) is automatically created by DMS when you start a synchronization. You can also create it manually if you want to, using the *CreateScopeInfoTableAsync* method.

Example:

```
var serverProvider = new SqlSyncChangeTrackingProvider(serverConnectionString);
var clientProvider = new SqliteSyncProvider(clientConnectionString);

var setup = new SyncSetup("ProductCategory", "ProductModel", "Product",
    "Address", "Customer", "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail"␣
↪);

var agent = new SyncAgent(clientProvider, serverProvider);
var s1 = await agent.SynchronizeAsync(setup);
```

Here is the *scope_info* table, once the sync is complete:

sync_scope_name	sync_scope_setup	sync_scope_schema
DefaultScope	{ “t” : [{“ProductCategory”, “Product”, “ProductModel”, “Address”, “Customer”, . . . }] }	{ “t” : [{“Prod. . . ”}] }

Note: The **sync_scope_setup** column is a serialized json object, containing all the tables and options you defined from your `SyncSetup` instance.

The corresponding object in `DMS` is the `ScopeInfo` class. It basically contains all fields from the *scope_info* table.

2.4.2 Methods & Properties

You can access a `SyncScope` stored on the server or client, using a `LocalOrchestrator` or `RemoteOrchestrator` instance (directly from a `SyncAgent` instance or by creating a new instance directly)

Properties

Once a first scope sync has been done, you will have, on both sides, a *scope_info* table, containing:

- A **scope name**: Defines a user friendly name (unique) for your scope. Default name is `DefaultScope`.
- A **setup**, serialized: Contains all the tables and options you defined from your `SyncSetup` instance.
- A **schema**, serialized: Contains all the tables, filters, parameters and so on, for this scope.

- The **last cleanup timestamp**: This timestamp is used to cleanup the tracking tables. It's the last time the cleanup was done.
- The **version**: DMS database version.

The corresponding .NET objet is the `ScopeInfo` class:

```
public class ScopeInfo
{
    public string Name { get; set; }
    public SyncSet Schema { get; set; }
    public SyncSetup Setup { get; set; }
    public string Version { get; set; }
    public long? LastCleanupTimestamp { get; set; }
    public string Properties { get; set; }
}
```

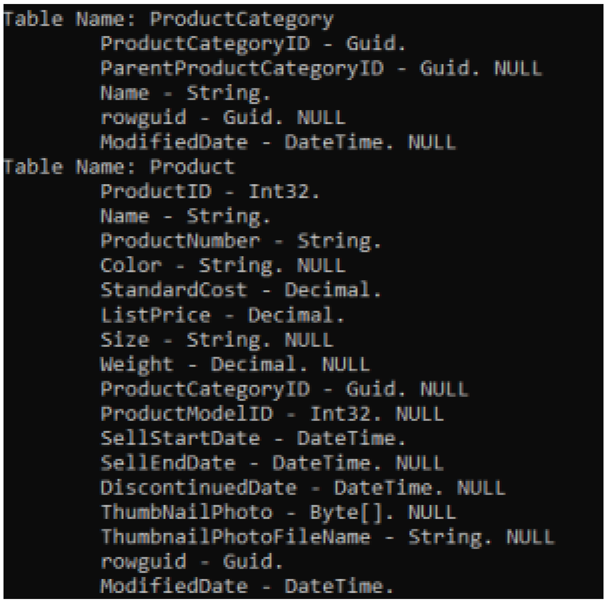
GetScopeInfoAsync

Get the scope info from the database. This method will return a `ScopeInfo` object:

Note: If the **scope_info** table is not existing, it is created.

If no scope record is found, an empty scope will be created with empty schema and setup properties.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var scopeInfo = await localOrchestrator.GetScopeInfoAsync();

if (scopeInfo.Schema == null)
    return;

foreach (var schemaTable in scopeInfo.Schema.Tables)
{
    Console.WriteLine($"Table Name: {schemaTable.TableName}");

    foreach (var column in schemaTable.Columns)
        Console.WriteLine($"\t{column}. {(column.AllowDBNull ? "NULL": "")}");
}
```



```
Table Name: ProductCategory
        ProductCategoryID - Guid.
        ParentProductCategoryID - Guid. NULL
        Name - String.
        rowguid - Guid. NULL
        ModifiedDate - DateTime. NULL
Table Name: Product
        ProductID - Int32.
        Name - String.
        ProductNumber - String.
        Color - String. NULL
        StandardCost - Decimal.
        ListPrice - Decimal.
        Size - String. NULL
        Weight - Decimal. NULL
        ProductCategoryID - Guid. NULL
        ProductModelID - Int32. NULL
        SellStartDate - DateTime.
        SellEndDate - DateTime. NULL
        DiscontinuedDate - DateTime. NULL
        ThumbNailPhoto - Byte[]. NULL
        ThumbnailPhotoFileName - String. NULL
        rowguid - Guid.
        ModifiedDate - DateTime.
```

On `RemoteOrchestrator`, you can use a `SyncSetup` argument to get a `ScopeInfo` object containing the **Setup** and **Schema** properties filled.

```
var remoteOrchestrator = new RemoteOrchestrator(clientProvider);
var setup = new SyncSetup("Product, ProductCategory");
var scopeInfo = await remoteOrchestrator.GetScopeInfoAsync(setup);

foreach (var schemaTable in scopeInfo.Schema.Tables)
{
    Console.WriteLine($"Table Name: {schemaTable.TableName}");

    foreach (var column in schemaTable.Columns)
        Console.WriteLine($"\t{column}. {(column.AllowDBNull ? "NULL": "")}");
}
```

GetAllScopeInfosAsync

Get all scope infos from a data source.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var scopeInfo = await localOrchestrator.GetAllScopeInfosAsync();
```

SaveScopeInfoAsync

Save a scope info to the local data source.

```
var scopeInfo = await localOrchestrator.GetScopeInfoAsync();
scopeInfo.Setup = setup;
scopeInfo.Schema = schema;
scopeInfo.ScopeName = "v1";
await localOrchestrator.SaveScopeInfoAsync(scopeInfo);
```

DeleteScopeInfoAsync

Delete a scope info from the local data source.

```
var scopeInfo = await localOrchestrator.GetScopeInfoAsync("v0");
await localOrchestrator.DeleteScopeInfoAsync(scopeInfo);
```

CreateScopeInfoTableAsync

Create a scope info table in local data source.

```
await localOrchestrator.CreateScopeInfoTableAsync();
```

ExistScopeInfoTableAsync

Check if a scope_info table exists in the local data source

```
var exists = await localOrchestrator.ExistScopeInfoTableAsync();
```

2.4.3 Multi Scopes

In some scenario, you may want to sync some tables at one time, and some others tables at another time.

These kind of scenarios are possible using the **multi scopes** sync architecture.

How does it work ?

To be able to create a multi scopes scenario, you just have to:

- Create two `SyncSetup` instances with your tables / filters and options.
- Sync your databases calling `SynchronizeAsync` with a different scope name for each setup.
- Or call `ProvisionAsync` with your scope name.

Example

Here is a full example, where we sync separately the *Product* table, then the *Customer* table:

```
// Create 2 Sql Sync providers
var serverProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(serverDbName));
var clientProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(clientDbName));

// Create 2 setup
var setupProducts = new SyncSetup("ProductCategory", "ProductModel", "Product");
var setupCustomers = new SyncSetup("Address", "Customer", "CustomerAddress",
            "SalesOrderHeader", "SalesOrderDetail");

// Create an agent
var agent = new SyncAgent(clientProvider, serverProvider);
```

(continues on next page)

(continued from previous page)

```
// Using the Progress pattern to handle progession during the synchronization
var progress = new SynchronousProgress<ProgressArgs>(s =>
    Console.WriteLine($"{s.Context.SyncStage}:\t{s.Message}");
);

Console.WriteLine("Hit 1 for sync Products. Hit 2 for sync customers and sales");
var k = Console.ReadKey().Key;

if (k == ConsoleKey.D1)
{
    Console.WriteLine("Sync Products:");
    var s1 = await agent.SynchronizeAsync("products", setupProducts, progress);
    Console.WriteLine(s1);
}
else
{
    Console.WriteLine("Sync Customers and Sales:");
    var s1 = await agent.SynchronizeAsync("customers", setupCustomers, progress);
    Console.WriteLine(s1);
}
```

Once you have made the 2 syncs, your local *syns_scope* table should looks like that:

sync_scope_name	sync_scope_schema	sync_scope_setup
products	{ "t" : [{. . . . . . }] }	{ "t" : [{. . . . . . }] }
customers	{ "t" : [{. . . . . . }] }	{ "t" : [{. . . . . . }] }

2.5 ScopeInfoClients

2.5.1 What is a scope client ?

We saw that a **scope** is a set of tables and is stored in the *scope_info* table.

A **scope client** is a the association of one scope with a filter, and is stored in the *scope_info_client* table.

A scope client record contains:

- A scope (think "**FROM**" in a database) : Set of tables defined in the scope_info table
- A list of filter parameters (think "**WHERE**" in a database) : The filter definition is stored in the scope. We are talking here about the values of theses filter.

Let's imagine you are synchronizing some **Products** and **ProductCategories**, where you want only the products of the category **"Books"**. You will have to define a scope client with the following parameters:

- **Scope** : *Product*, *ProductCategory* tables.
- **Filter parameters values** : `ProductCategoryID = "Books"`

DMS will automatically create:

- The scope in **scope_info** with the 2 tables *Product*, *ProductCategory*.
- The filter parameter value `ProductCategoryID = 'Books'` in the **scope_info_client** table.

2.5.2 Methods & Properties

You can access the scope client information, as a `ScopeInfoClient` instance, using a `LocalOrchestrator` or `RemoteOrchestrator` instance (directly from a `SyncAgent` instance or by creating a new instance directly)

Properties

Once a first scope sync has been done, you will have, on both sides, a *scope_info_client* table, containing:

- A **scope name**: Defines a user friendly name (unique) for your scope. Default name is `DefaultScope`. References the *scope_info* table.
- A **scope info client id**: Defines a unique id for the scope client. Think this Id as the uniqure representation of the client database.
- A **scope info hash**: Defines the hash of the JSON property `scope_parameters`.
- A **scope info parameters**: Defines the parameters for this scope info client. This is a JSON property, containing the list of filter parameters values, and is, combined with **scope_name**, unique.
- A **scope info timestamp**: Defines the last time the scope info client has been updated.
- A **scope info server timestamp**: Defines the last time the scope info client has been updated on the server side.
- A **scope last sync date**: Defines the last time the scope has been synchronized, as a datetime.
- A **scope last sync duration**: Defines the last time the scope has been synchronized, as a duration.
- A **scope errors**: Defines the last errors happened during last sync. Point directly to a BatchInfo directory containing the errors (as JSON files).
- A **scope properties**: Defines additionnal properties.

Here is a small example to see how scope client infos are created:

```
var setup = new SyncSetup("ProductCategory", "Product", "Employee");

setup.Tables[productCategoryTableName].Columns
        .AddRange("ProductCategoryId", "Name", "rowguid", "ModifiedDate");

setup.Filters.Add("ProductCategory", "ProductCategoryId");
setup.Filters.Add("Product", "ProductCategoryId");

var pMount = new SyncParameters(("ProductCategoryId", "MOUNTB"));
var pRoad = new SyncParameters(("ProductCategoryId", "ROADFR"));

var agent = new SyncAgent(client.Provider, Server.Provider);
var r1 = await agent.SynchronizeAsync("v1", setup, pMount);
var r2 = await agent.SynchronizeAsync("v1", setup, pRoad);
```

Once the sync is done, you will have 2 scope clients created:

sync_scope_id	sync_scope_name	sync_scope_parameters	scope_last_sync_timestamp
F02BC17-A478-..	v1	[{pn:ProductCategoryId, v:MOUNTB}]	2000
F02BC17-A478-..	v1	[{pn:ProductCategoryId, v:ROADFR}]	20022

Each scope client is independant, and can be synchronized separately, since they have their own **timestamp** associated with their combo **scope name / scope parameters**.

Note: We have the same scope for both sync, with the same tables / scope name. You'll see that the *scope_info* will contains only one record for the scope v1

The corresponding .NET objet is the `ScopeInfoClient` class:

```
public class ScopeInfoClient
   {
       public Guid Id { get; set; }
       public string Name { get; set; }
       public string Hash { get; set; }
       public long? LastSyncTimestamp { get; set; }
       public long? LastServerSyncTimestamp { get; set; }
       public bool IsNewScope { get; set; }
       public SyncParameters Parameters { get; set; }
       public DateTime? LastSync { get; set; }
       public long LastSyncDuration { get; set; }
       public string Properties { get; set; }
       public string Errors { get; set; }
       public string LastSyncDurationString { get; }
   }
```

GetScopeInfoClientAsync

This method allows to get a scope client information, from a scope name and a list of filter parameters values.

```
var parameters = new SyncParameters(("ProductCategoryId", "MOUNTB"));
var scopeInfoClient = await orchestrator.GetScopeInfoClientAsync("v1", parameters);
```

Note: If the *scope_info_client* does not exists, it will be created, and the a new record is added.

Warning: If you call this method using a `RemoteOrchestrator`, you'll need to pass the clientId parameter

GetAllScopeInfosAsync

Returns all scope clients information, from a scope name.

```
var cAllScopeInfoClients = await agent.LocalOrchestrator.
↪GetAllScopeInfoClientsAsync();

var minServerTimeStamp = cAllScopeInfoClients.Min(sic => sic.LastServerSyncTimestamp);
var minClientTimeStamp = cAllScopeInfoClients.Min(sic => sic.LastSyncTimestamp);
var minLastSync = cAllScopeInfoClients.Min(sic => sic.LastSync);
```

SaveScopeInfoAsync

This method allows you to save and override a scope client information. You should not have to do it, but some scenarios can be done with this method.

```
var cScopeInfoClient = await localOrchestrator.GetScopeInfoClientAsync();

if (cScopeInfoClient.IsNewScope)
{
  cScopeInfoClient.IsNewScope = false;
  cScopeInfoClient.LastSync = DateTime.Now;
  cScopeInfoClient.LastSyncTimestamp = 0;
  cScopeInfoClient.LastServerSyncTimestamp = 0;

  await agent.LocalOrchestrator.SaveScopeInfoClientAsync(cScopeInfoClient);
}
```

2.6 Orchestrators

2.6.1 Overview

An **Orchestrator** is agnostic to the underlying database.

it communicates with the database through a provider. A provider is always required when you're creating a new orchestrator.

We have two kind of orchestrators:

- The **Local Orchestrator** (or let's say client side orchestrator) : `LocalOrchestrator`.
- The **Remote Orchestrator** (or let's say server side orchestrator) : `RemoteOrchestrator`.

We have other orchestrators, that will handle, under the hood, the web sync process:

- The `WebRemoteOrchestrator`: This orchestrator will run locally, and will act "*as*" an orchestrator from the sync agent, but under the hood will generate an http request with a payload containing all the required information, and will send it to the server side.
- The `WebServerAgent`: On the opposite side, this agent is hosted with an ASP.NET WebApi an is exposed by a web api, and will get the incoming request from the `WebRemoteOrchestrator` and will then call the underline `RemoteOrchestrator` correctly.

A set of methods are accessible from both `LocalOrchestrator` or `RemoteOrchestrator` (and for some of them from `WebRemoteOrchestrator`).

- Database builder methods: Create tables, metadatas, tracking tables . . .
- Sync changes methods: (Get changes, get estimated changes count . . .)
- Tracking Tables
- Tables
- Schemas
- Scopes

2.6.2 Builder Methods

You need a `ScopeInfo` instance to be able to create any metadatas (stored proc, tables, triggers or tracking tables) in your data source.

This method runs on any `Orchestrator`, but we are using here a `RemoteOrchestrator` because the client database is empty and getting a table schema from an empty database… well.. :)

```
var provider = new SqlSyncProvider(serverConnectionString);
var options = new SyncOptions();
var setup = new SyncSetup("ProductCategory", "ProductModel", "Product");
var orchestrator = new RemoteOrchestrator(provider, options);

var serverSchema = await orchestrator.GetSchemaAsync(setup);

foreach (var column in serverSchema.Tables["Product"].Columns)
    Console.WriteLine(column);
```

```
ProductID - Int32
Name - String
ProductNumber - String
Color - String
StandardCost - Decimal
ListPrice - Decimal
Size - String
Weight - Decimal
ProductCategoryID - Int32
ProductModelID - Int32
SellStartDate - DateTime
SellEndDate - DateTime
DiscontinuedDate - DateTime
ThumbNailPhoto - Byte[]
ThumbnailPhotoFileName - String
rowguid - Guid
ModifiedDate - DateTime
```

Managing stored procedures

Managing **Stored Procedures** could be done using:

- `LocalOrchestrator.CreateStoredProcedureAsync()` : Create a stored procedure using the `DbStoredProcedureType` enumeration, for one `SetupTable` argument.
- `LocalOrchestrator.ExistStoredProcedureAsync()`: Check if a stored procedure already exists, using the `DbStoredProcedureType` enumeration, for one `SetupTable` argument.
- `LocalOrchestrator.DropStoredProcedureAsync()` : Drop a stored procedure using the `DbStoredProcedureType` enumeration, for one `SetupTable` argument.
- `LocalOrchestrator.CreateStoredProceduresAsync()` : Create all stored procedures needed for one `SetupTable` argument.
- `LocalOrchestrator.DropStoredProceduresAsync()` : Drop all stored procedures created for one `SetupTable` argument.

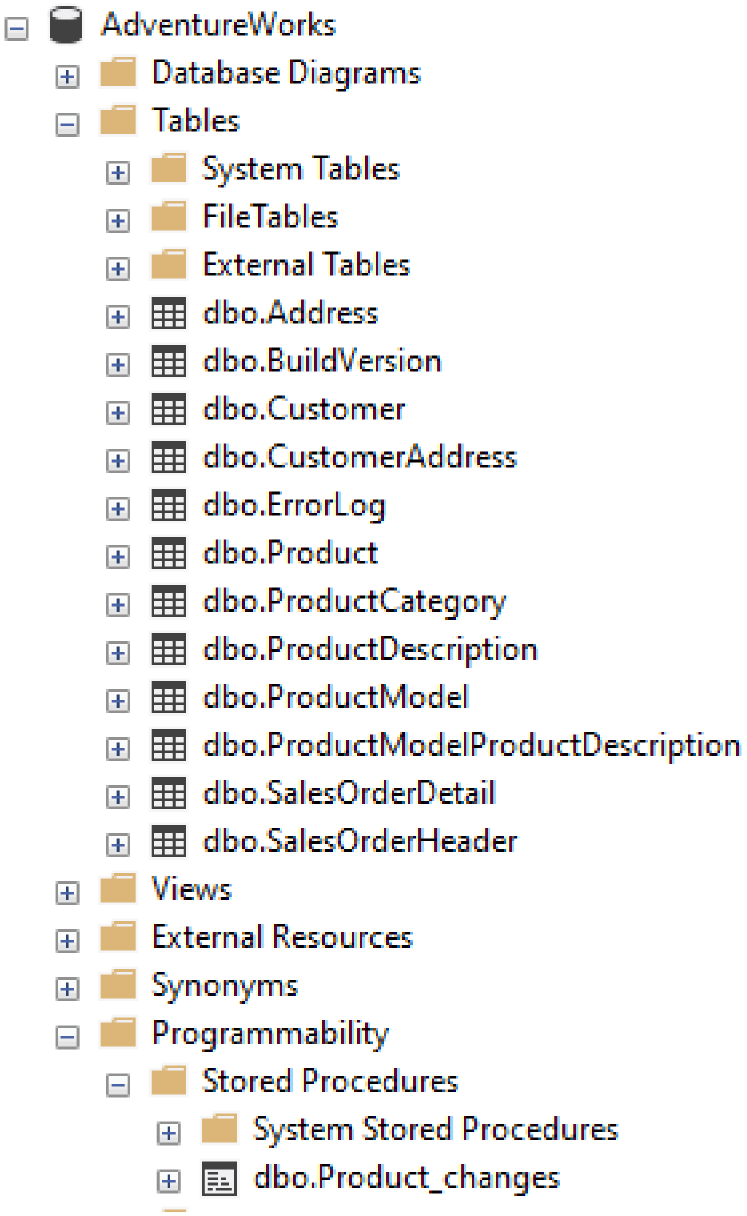
Creating a stored procedure could be done like this:

```
var provider = new SqlSyncProvider(serverConnectionString);
var remoteOrchestrator = new RemoteOrchestrator(provider, options);
var scopeInfo = await remoteOrchestrator.GetScopeInfoAsync(setup);

var spExists = await orchestrator.ExistStoredProcedureAsync(scopeInfo, "Product",␣
↪null,
```

(continues on next page)

(continued from previous page)

```
                        DbStoredProcedureType.SelectChanges);
if (!spExists)
    await orchestrator.CreateStoredProcedureAsync(scopeInfo, "Product", null,
                        DbStoredProcedureType.SelectChanges);
```



AdventureWorks
Database Diagrams
Tables
System Tables
FileTables
External Tables
dbo.Address
dbo.BuildVersion
dbo.Customer
dbo.CustomerAddress
dbo.ErrorLog
dbo.Product
dbo.ProductCategory
dbo.ProductDescription
dbo.ProductModel
dbo.ProductModelProductDescription
dbo.SalesOrderDetail
dbo.SalesOrderHeader
Views
External Resources
Synonyms
Programmability
Stored Procedures
System Stored Procedures
dbo.Product_changes

Be careful, this stored procedure relies on a tracking table for table `Product`, but we did not create it, yet.

Creating a tracking table

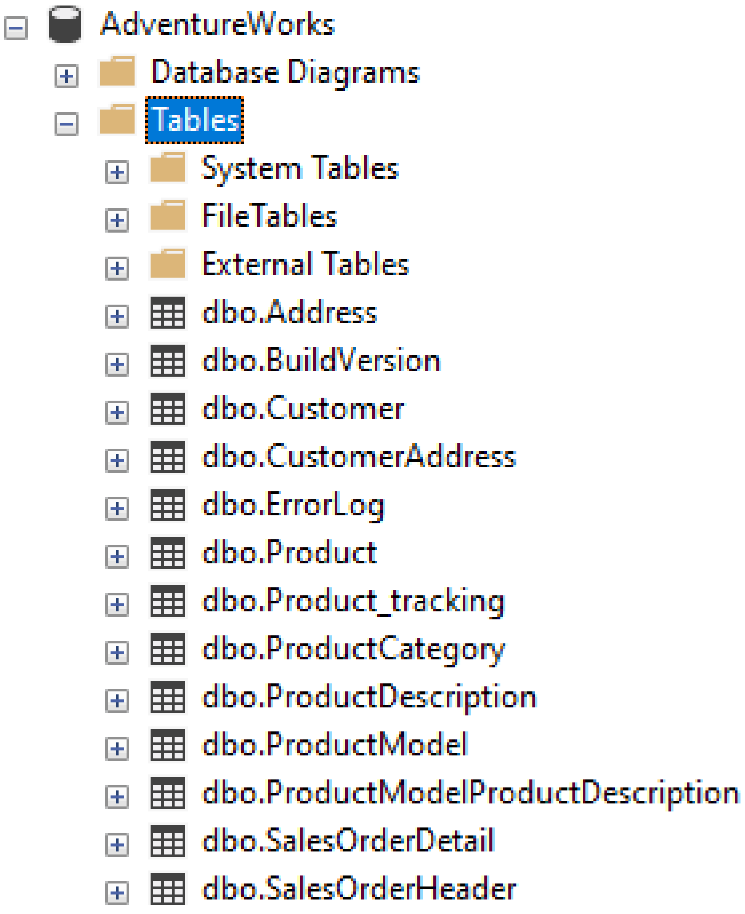
Continuing on the last sample, we can create in the same way, the tracking table for table *Product*:

```
var provider = new SqlSyncProvider(serverConnectionString);
var remoteOrchestrator = new RemoteOrchestrator(provider, options);
```

(continues on next page)

(continued from previous page)

```
var scopeInfo = await remoteOrchestrator.GetScopeInfoAsync(setup);

var spExists = await remoteOrchestrator.ExistTrackingTableAsync(scopeInfo, "Employee
↪");
if (!spExists)
    await remoteOrchestrator.CreateTrackingTableAsync(scopeInfo, "Employee");
```



```
AdventureWorks
    Database Diagrams
    Tables
        System Tables
        FileTables
        External Tables
        dbo.Address
        dbo.BuildVersion
        dbo.Customer
        dbo.CustomerAddress
        dbo.ErrorLog
        dbo.Product
        dbo.Product_tracking
        dbo.ProductCategory
        dbo.ProductDescription
        dbo.ProductModel
        dbo.ProductModelProductDescription
        dbo.SalesOrderDetail
        dbo.SalesOrderHeader
```

Droping a tracking table and a stored procedure

Now we can drop this newly created stored procedure and tracking table:

```
var provider = new SqlSyncProvider(serverConnectionString);
var remoteOrchestrator = new RemoteOrchestrator(provider, options);
var scopeInfo = await remoteOrchestrator.GetScopeInfoAsync(setup);

var trExists = await orchestrator.ExistTrackingTableAsync(scopeInfo, "Employee");
if (trExists)
    await orchestrator.DropTrackingTableAsync(scopeInfo, "Employee");

var spExists = await orchestrator.ExistStoredProcedureAsync(scopeInfo, "Employee",␣
↪null,
                        DbStoredProcedureType.SelectChanges);
if (spExists)
```

(continues on next page)

(continued from previous page)

```
    await orchestrator.DropStoredProcedureAsync(scopeInfo, "Employee", null,
                                DbStoredProcedureType.SelectChanges);
```

2.6.3 LocalOrchestrator

The local orchestrator runs only on the client side. You have access to several useful methods to get the changes to send on the next sync, or even an estimation of these changes.

GetChangesAsync

Get the changes from local datasource, to be sent to the server.

You need to provide a `ScopeInfoClient` instance to be able to get the changes.
Returns an instance of `ClientSyncChanges` containing a reference to the changes serialized on disk.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var cScopeInfoClient = await localOrchestrator.GetScopeInfoClientAsync(scopeName,␣
↪parameters);
var changes = await localOrchestrator.GetChangesAsync(cScopeInfoClient);
```

If you need to load all changes in memory, you can use `LoadTableFromBatchInfoAsync` method:

GetEstimatedChangesCountAsync

Get the estimated changes count from local datasource, to be sent to the server.

You need to provide a `ScopeInfoClient` instance to be able to get the changes.
Returns an instance of `ClientSyncChanges` containing a reference to the changes serialized on disk.
The propery `ClientChangesSelected` (of type `DatabaseChangesSelected`) from the returned `ClientSyncChanges` value, contains the estimated changes count.

> **Warning:** No changes are downloaded, so far the `ClientBatchInfo` property is always **null**.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var cScopeInfoClient = await localOrchestrator.GetScopeInfoClientAsync(scopeName,␣
↪parameters);
var estimatedChanges = await localOrchestrator.
↪GetEstimatedChangesCountAsync(cScopeInfoClient);

Console.WriteLine(estimatedChanges.ClientChangesSelected.TotalChangesSelected);

foreach (var table in changes.ClientChangesSelected.TableChangesSelected)
    Console.WriteLine($"Table: {table.TableName} - Total changes:{table.TotalChanges}
↪");
```

LoadTableFromBatchInfoAsync

Load a table from a batch info. This method is used to load all rows contains in a BatchInfo instance in memory.

You can specify a SyncRowState parameter to get rows with a specific state.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
// Loading all rows for table SalesLT.SalesOrderDetail, with a state fo Deleted:
var sodTable = await localOrchestrator.LoadTableFromBatchInfoAsync(
            scopeName, batchInfo, "SalesOrderDetail", "SalesLT", SyncRowState.
↪Deleted);

foreach (var orderDetail in sodTable.Rows)
    Console.WriteLine(orderDetail["TotalLine"]);
```

LoadBatchInfosAsync

Load all batch infos for a given scope name. The batch infos are loaded from the tmp directory set from SyncOptions.BatchDirectory.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var batchInfos = await localOrchestrator.LoadBatchInfosAsync();

foreach (var batchInfo in batchInfos)
    Console.WriteLine(batchInfo.RowsCount);
```

LoadTablesFromBatchInfoAsync

Load all tables from a batch info. This method is used to load all tables contains in a BatchInfo instance in memory.

Each file contained in the BatchInfo instance is loaded in memory, and returned as a SyncTable instance.

> **Warning:** this method returns an IAsyncEnumerable<SyncTable>. You need to iterate on it using the async keyword to get all tables.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var batchInfos = await localOrchestrator.LoadBatchInfosAsync();

foreach (var batchInfo in batchInfos)
{
    var allTables = localOrchestrator.LoadTablesFromBatchInfoAsync(batchInfo);

    // Enumerate all rows from each table
    await foreach (var table in allTables)
        foreach (var row in table.Rows)
            Console.WriteLine(row);
}
```

SaveTableToBatchPartInfoAsync

Save a batch info to a batch part files.

```
// TODO
```

GetSchemaAsync

Get the schema from the local datasource.

Be careful:

- `GetScopeInfo()` returns a ScopeInfo object, which contains the schema of the local database, saved in the *scope_info* table.
- `GetSchema()` returns a SyncSet object, which contains the schema of the local database, read on the fly.

Internally **DMS** is using GetSchema whenever it's needed, and eventually saved the schema in the *scope_info* table.

Using `GetSchema()` will not save the schema anywhere.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var setup = new SyncSetup("ProductCategory", "Product");
var schema = await localOrchestrator.GetSchemaAsync(setup);
```

ProvisionAsync

Provision the local datasource with the tracking tables, stored procedures, triggers and even tables needed for the sync process.

You need a `ScopeInfo` instance to be able to provision the local database.

If you do not specify the `provision` argument, a default value `SyncProvision.Table | SyncProvision.StoredProcedures | SyncProvision.Triggers | SyncProvision.TrackingTable` is used.

Usually, the ScopeInfo instance is retrieved from your server database, using a `RemoteOrchestrator` or a `WebRemoteOrchestrator` instance.

```
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);
var sScopeInfo = await remoteOrchestrator.GetScopeInfoAsync();
var cScopeInfo = await localOrchestrator.ProvisionAsync(sScopeInfo);
```

If you have already done a first sync (or a first provision) of your client database, you can use the `GetScopeInfoAsync` method to get the ScopeInfo instance from your client database instead of your server database.

Provision an already provisioned local database can be useful if you want to overwrite / recreate everything.

> **Warning:** Be careful, the client database may not contains a ScopeInfo instance if you have not done a first sync.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var cScopeInfo = await localOrchestrator.GetScopeInfoAsync();
if (cScopeInfo != null)
    cScopeInfo = await localOrchestrator.ProvisionAsync(cScopeInfo, overwrite:true);
```

More . . .

Check the Provision & Deprovision section for more details about the provision process.

DeprovisionAsync

Deprovision the local datasource. This will drop tracking tables, stored procedures or triggers created by the sync process.

Note: By default, **DMS** will never deprovision a table, if not explicitly set with the **provision** argument.

Same behavior applies to the *scope_info* and *scope_info_client* tables.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
await localOrchestrator.DeprovisionAsync(SyncProvision.StoredProcedures |␣
↪SyncProvision.Triggers);
```

If you do not have any scope info locally (the *scope_info* table does not exists anymore, or is empty), you still can try to deprovision your local database using a simple `SyncSetup` instance:

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var setup = new SyncSetup("ProductCategory", "Product");
await localOrchestrator.DeprovisionAsync(setup,
            SyncProvision.StoredProcedures | SyncProvision.Triggers);
```

More . . .

Check the Provision & Deprovision section for more details about the provision process.

DropAllAsync

Drop all DMS metadatas from your local database, except tables. Everythin is dropped: **tracking tables**, **stored procedures**, **triggers**, **scope info tables**, etc.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
await localOrchestrator.DropAllAsync();
```

DeleteMetadatasAsync

Delete all DMS metadatas from the tracking tables, in your local database.

This operation is automatically managed by DMS on the client side. You should not have to use it manually, except on specific scenario.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
await localOrchestrator.DeleteMetadatasAsync();
```

More . . .

Check the Metadatas section for more details about the metadatas deletion process.

ResetTableAsync

Delete all rows from a **table** and the corresponding **tracking table**.

This method is used internall

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var scopeInfo = await localOrchestrator.GetScopeInfoAsync();
await localOrchestrator.ResetTableAsync(scopeInfo, "ProductCategory");
```

Warning: Be careful, this method will delete all rows from your table !!

EnableConstraintsAsync & DisableConstraintsAsync

Enable or **Disable** all constraints on your local database.

Useful if you want to apply rows without having to check any constraints.

This method is used internally by **DMS** when you are using the `SyncOptions.DisableConstraintsOnApplyChanges` option.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);

using var sqlConnection = new SqlConnection(clientProvider.ConnectionString);

sqlConnection.Open();
using var sqlTransaction = sqlConnection.BeginTransaction();

var scopeInfo = await localOrchestrator.GetScopeInfoAsync(sqlConnection,␣
↪sqlTransaction);
await localOrchestrator.DisableConstraintsAsync(scopeInfo, "ProductCategory", default,
    sqlConnection, sqlTransaction);

// .. Do some random insert in the ProductCategory table
await DoSomeRandomInsertInProductCategoryTableAsync(sqlConnection, sqlTransaction);

await localOrchestrator.EnableConstraintsAsync(scopeInfo, "ProductCategory", default,
    sqlConnection, sqlTransaction);

sqlTransaction.Commit();
sqlConnection.Close();
```

GetLocalTimestampAsync

Get the local timestamp from the local database.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var ts = await localOrchestrator.GetLocalTimestampAsync();
```

2.6.4 RemoteOrchestrator

The remote orchestrator runs only on the server side. You have access to several useful methods to get the changes to send on the next sync, or even an estimation of these changes.

TCP mode

If you have a TCP connection between your server and your client, you can use the RemoteOrchestrator class within your SyncAgent instance.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

var agent = new SyncAgent(localOrchestrator, remoteOrchestrator);
```

That basically means your client application has a direct access to the server database, through a connection string.

HTTP mode

In the other hand, if you are using the HTTP mode, you must use another remote orchestrator, that can send http requests. This orchestrator is the WebRemoteOrchestrator class.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var remoteOrchestrator = new WebRemoteOrchestrator(serverProvider, "http://
↪localhost:5000");

var agent = new SyncAgent(localOrchestrator, remoteOrchestrator);
```

Note: More on how to handle a basic http scenario can be found in the ASP.NET Web Proxy section.

GetChangesAsync

Get the changes from local datasource, to be sent to a particular client.

You need to provide a ScopeInfoClient instance to be able to get the changes.
Returns an instance of ServerSyncChanges containing a reference to the changes serialized on disk.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var remoteOrchestrator = new RemoteOrchestrator(remoteProvider);
var cScopeInfoClient = await localOrchestrator.GetScopeInfoClientAsync(scopeName,␣
↪parameters);

var changes = await remoteOrchestrator.GetChangesAsync(cScopeInfoClient);
```

If you are not trying to retrieve the changes from a client perspective, but more from a server perspective, you can get a `ScopeInfoClient` instance from the server side as well.
As server is maitaining a reference to all clients, you need to pass the clientId reference to get the correct `ScopeInfoClient` instance.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var remoteOrchestrator = new RemoteOrchestrator(remoteProvider);

// You can load a client scope info from the server database also, if you know the␣
↪clientId
var cScopeInfoClient = await remoteOrchestrator.GetScopeInfoClientAsync(
   clientId, scopeName, parameters);

var changes = await remoteOrchestrator.GetChangesAsync(cScopeInfoClient);
```

If you need to load all changes in memory, you can use *LoadTableFromBatchInfoAsync*.

GetEstimatedChangesCountAsync

Get the estimated changes count from local datasource, to be sent to the server.

You need to provide a `ScopeInfoClient` instance to be able to get the changes.
Returns an instance of `ServerSyncChanges` containing a reference to the changes serialized on disk.
The propery `ServerChangesSelected` (of type `DatabaseChangesSelected`) from the returned `ServerSyncChanges` value, contains the estimated changes count.

Warning: No changes are downloaded, so far the `ServerBatchInfo` property is always **null**.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
var remoteOrchestrator = new RemoteOrchestrator(remoteProvider);
var cScopeInfoClient = await localOrchestrator.GetScopeInfoClientAsync(scopeName,␣
↪parameters);

// You can load a client scope info from the server database also, if you know the␣
↪clientId
// var cScopeInfoClient = await remoteOrchestrator.GetScopeInfoClientAsync(
    clientId, scopeName, parameters);

var estimatedChanges = await remoteOrchestrator.
↪GetEstimatedChangesCountAsync(cScopeInfoClient);

Console.WriteLine(estimatedChanges.ClientChangesSelected.TotalChangesSelected);

foreach (var table in changes.ClientChangesSelected.TableChangesSelected)
    Console.WriteLine($"Table: {table.TableName} - Total changes:{table.TotalChanges}
↪");
```

LoadTableFromBatchInfoAsync

Load a table from a batch info. See *LoadTableFromBatchInfoAsync*.

LoadBatchInfosAsync

Load all batch infos for a given scope name. See *LoadBatchInfosAsync*.

LoadTablesFromBatchInfoAsync

Load all tables from a batch info. See *LoadTablesFromBatchInfoAsync*.

GetSchemaAsync

Get the schema from the local datasource. See *GetSchemaAsync*.

ProvisionAsync

Provision the server datasource with the tracking tables, stored procedures, triggers and even tables needed for the sync process.

You need a `SyncSetup` instance containing all the tables (and optionally the columns list) you want to sync.

```
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);
var setup = new SyncSetup("ProductCategory", "Product");
var sScopeInfo = await remoteOrchestrator.ProvisionAsync(setup);
```

More . . .

Check the Provision & Deprovision section for more details about the provision process.

DeprovisionAsync

Deprovision the server datasource. This will drop tracking tables, stored procedures or triggers created by the sync process.

Note: By default, **DMS** will never deprovision a table, if not explicitly set with the **provision** argument.

Same behavior applies to the *scope_info* and *scope_info_client* tables.

```
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);
await localOrchestrator.DeprovisionAsync(SyncProvision.StoredProcedures | SyncProvision.Triggers);
```

More . . .

Check the Provision & Deprovision section for more details about the provision process.

DropAllAsync

Drop all DMS metadatas from your server database, except tables. Everything is dropped: **tracking tables**, **stored procedures**, **triggers**, **scope info tables**, etc.

```
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);
await remoteOrchestrator.DropAllAsync();
```

DeleteMetadatasAsync

Delete all DMS metadatas from the tracking tables, in your server database.

Warning: A huge difference between LocalOrchestrator and RemoteOrchestrator is that the first one is done automatically after a successfull sync, while the second one is not. You need to call this method manually.

```
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);
await remoteOrchestrator.DeleteMetadatasAsync();
```

More . . .

Check the Metadatas section for more details about the metadatas deletion process.

ResetTableAsync

Delete all rows from a **table** and the corresponding **tracking table**. See *ResetTableAsync*.

EnableConstraintsAsync & DisableConstraintsAsync

Enable or **Disable** all constraints on your local database. See *EnableConstraintsAsync*.

GetLocalTimestampAsync

Get the local timestamp from the local database. see *GetLocalTimestampAsync*.

2.7 Progression

Getting useful information during a sync process could be complex.
You can have a lot of information from an in-going sync, through two kinds of things:

- `IProgress<ProgressArgs>` : A best practice using IProgress<T> to handle progress from within an *awaitable* method.
- `Ìnterceptor<T>` : A more advanced technic to handle a lot of more events from within **DMS**

2.7.1 Overview

During a full synchronization, we have **two distincts** type of progression:

- The **Progression** from the client side.
- The **Progression** from the server side.

We have a lot of progress values raised from both the **server** and the **client** side:

- Each progress value is catched at the end of a method called by the **Orchestrator** instance.
- Each progress value in a sync process corresponds to a specific *stage*, represented by a `SyncStage` enumeration.

```
public enum SyncStage
{
    None = 0,

    BeginSession,
    EndSession,

    ScopeLoading,
    ScopeWriting,

    SnapshotCreating,
    SnapshotApplying,

    SchemaReading,

    Provisioning,
    Deprovisioning,

    ChangesSelecting,
    ChangesApplying,

    Migrating,

    MetadataCleaning,
}
```

To explain how things work, we are starting from a really straightforward sync process example, using the sample from Hello sync sample:

```
var serverProvider = new SqlSyncChangeTrackingProvider(serverConnectionString);
var clientProvider = new SqlSyncProvider(clientConnectionString);

var setup = new SyncSetup("ProductCategory", "ProductModel", "Product",
            "Address", "Customer", "CustomerAddress", "SalesOrderHeader",
            "SalesOrderDetail" );

var agent = new SyncAgent(clientProvider, serverProvider);
do
{
    // Launch the sync process
    var s1 = await agent.SynchronizeAsync(setup);
    // Write results
    Console.WriteLine(s1);

```

(continues on next page)

(continued from previous page)

```
} while (Console.ReadKey().Key != ConsoleKey.Escape);

Console.WriteLine("End");
```

We are going to see how to get useful information, from each stage involved during the sync processus, thanks to `IProgress<T>` and then we will go deeper with the notion of `Interceptor<T>`.

Note: You will find this complete sample here : Progression sample

2.7.2 IProgress<T>

As we said, the progress values are triggered from both side : **Server** side and **Client** side, ordered.

In our sample, we can say that :

- The `RemoteOrchestrator` instance, using the server provider instance, will report all the progress from the server side.
- The `LocalOrchestrator` instance using the client provider instance, will report all the progress from the client side.

Note: A `syncAgent` object is **always** running on the client side of **any** architecture.

Since our main method `SynchronizeAsync()` is marked `async` method, we will use the Progress<T> to be able to report progress value.

So far, the most straightforward way to get feedback from a current sync, is to pass an instance of `IProgress<T>` when calling the method `SynchronizeAsync()`.

Note: `Progress<T>` is **not** synchronous. So far, no guarantee that the progress callbacks will be raised in an ordered way.

That's why you can use a **DMS** progess class called `SynchronousProgress<T>` which is synchronous, using the correct synchronization context.

Here is a quick example used to provide some feedback to the user:

```
var serverProvider = new SqlSyncChangeTrackingProvider(serverConnectionString);
var clientProvider = new SqlSyncProvider(clientConnectionString);

// Tables involved in the sync process:
var setup = new SyncSetup ("ProductCategory", "ProductModel", "Product",
    "Address", "Customer", "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail"␣
↪);

// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, serverProvider);

// Using the IProgress<T> pattern to handle progession dring the synchronization
var progress = new SynchronousProgress<ProgressArgs>(args =>
    Console.WriteLine($"{s.ProgressPercentage:p}:  \t[{s.Source[..Math.Min(4, s.
↪Source.Length)]}] {s.TypeName}: {s.Message}"));
```

(continues on next page)

(continued from previous page)

```
do
{
    // Launch the sync process
    var s1 = await agent.SynchronizeAsync(setup, progress);
    // Write results
    Console.WriteLine(s1);

} while (Console.ReadKey().Key != ConsoleKey.Escape);

Console.WriteLine("End");
```

Here is the result, after the first synchronization, assuming the **Client** database is empty:

```
0,00 %:         [Clie] ProvisionedArgs: Provisioned 9 Tables. Provision:Table,␣
↪TrackingTable, StoredProcedures, Triggers.
55,00 %:        [Adve] TableChangesSelectedArgs: [SalesOrderHeader] [Total]␣
↪Upserts:32. Deletes:0. Total:32.
75,00 %:        [Adve] TableChangesSelectedArgs: [Address] [Total] Upserts:450.␣
↪Deletes:0. Total:450.
75,00 %:        [Adve] TableChangesSelectedArgs: [SalesOrderDetail] [Total]␣
↪Upserts:542. Deletes:0. Total:542.
75,00 %:        [Adve] TableChangesSelectedArgs: [ProductCategory] [Total] Upserts:41.
↪ Deletes:0. Total:41.
75,00 %:        [Adve] TableChangesSelectedArgs: [ProductModel] [Total] Upserts:128.␣
↪Deletes:0. Total:128.
75,00 %:        [Adve] TableChangesSelectedArgs: [CustomerAddress] [Total]␣
↪Upserts:417. Deletes:0. Total:417.
75,00 %:        [Adve] TableChangesSelectedArgs: [ProductDescription] [Total]␣
↪Upserts:762. Deletes:0. Total:762.
75,00 %:        [Adve] TableChangesSelectedArgs: [Product] [Total] Upserts:295.␣
↪Deletes:0. Total:295.
75,00 %:        [Adve] TableChangesSelectedArgs: [Customer] [Total] Upserts:847.␣
↪Deletes:0. Total:847.
75,00 %:        [Adve] DatabaseChangesSelectedArgs: [Total] Upserts:3514. Deletes:0.␣
↪Total:3514. [C:\Temp\DotmimSync\2022_07_17_12iks12xfjrzx]
80,42 %:        [Clie] TableChangesAppliedArgs: [ProductDescription] Changes Modified␣
↪Applied:762. Resolved Conflicts:0.
80,71 %:        [Clie] TableChangesAppliedArgs: [ProductCategory] Changes Modified␣
↪Applied:41. Resolved Conflicts:0.
81,62 %:        [Clie] TableChangesAppliedArgs: [ProductModel] Changes Modified␣
↪Applied:128. Resolved Conflicts:0.
83,72 %:        [Clie] TableChangesAppliedArgs: [Product] Changes Modified␣
↪Applied:295. Resolved Conflicts:0.
86,92 %:        [Clie] TableChangesAppliedArgs: [Address] Changes Modified␣
↪Applied:450. Resolved Conflicts:0.
92,95 %:        [Clie] TableChangesAppliedArgs: [Customer] Changes Modified␣
↪Applied:847. Resolved Conflicts:0.
95,92 %:        [Clie] TableChangesAppliedArgs: [CustomerAddress] Changes Modified␣
↪Applied:417. Resolved Conflicts:0.
96,14 %:        [Clie] TableChangesAppliedArgs: [SalesOrderHeader] Changes Modified␣
↪Applied:32. Resolved Conflicts:0.
100,00 %:       [Clie] TableChangesAppliedArgs: [SalesOrderDetail] Changes Modified␣
↪Applied:542. Resolved Conflicts:0.
100,00 %:       [Clie] DatabaseChangesAppliedArgs: [Total] Applied:3514. Conflicts:0.
100,00 %:       [Clie] SessionEndArgs: [Client] Session Ends. Id:3b69c8ab-cce8-4b94-
↪bf75-db22ea43169d. Scope name:DefaultScope.
```

(continues on next page)

(continued from previous page)

```
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 3514
        Total changes  applied: 3514
        Total resolved conflicts: 0
        Total duration :00.00:02.042
Sync Ended. Press a key to start again, or Escapte to end
```

As you can see, it's a first synchronization, so:

- Session begins
- Server creates all metadatas needed for AdventureWorks database
- Client creates all metadatas needed for Client database
- Server selects all changes to upserts
- Client applies all changes sent from ths server
- Client selects changes to send (nothing, obviously, because the tables have just been created on the client)
- Session ends

You can have more information, depending on your need, and still based on `IProgress<T>`.

Using a `SyncProgressLevel` enumeration affected to the ProgressLevel property of your SyncOptions instance:

```
public enum SyncProgressLevel
{
    /// <summary>
    /// Progress that contain the most detailed messages and the Sql statement␣
↪executed
    /// </summary>
    Sql,

    /// <summary>
    /// Progress that contain the most detailed messages. These messages may contain␣
↪sensitive application data
    /// </summary>
    Trace,

    /// <summary>
    /// Progress that are used for interactive investigation during development
    /// </summary>
    Debug,

    /// <summary>
    /// Progress that track the general flow of the application.
    /// </summary>
    Information,

    /// <summary>
    /// Specifies that a progress output should not write any messages.
    /// </summary>
    None
}
```

Warning: Be careful: The Sql level may contains sensitive data !

```
var syncOptions = new SyncOptions
{
    ProgressLevel = SyncProgressLevel.Debug
};

// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, serverProvider, syncOptions);

var progress = new SynchronousProgress<ProgressArgs>(s =>
{
    Console.WriteLine($"{s.ProgressPercentage:p}:  \t[{s.Source[..Math.Min(4, s.
↪Source.Length)]}] {s.TypeName}: {s.Message}");
});

var s = await agent.SynchronizeAsync(setup, SyncType.Reinitialize, progress);
Console.WriteLine(s);
```

And the details result with a `SyncProgressLevel.Debug` flag:

```
0,00 %:         [Clie] SessionBeginArgs: [Client] Session Begins. Id:f62adec4-21a7-
↪4a35-b86e-d3d7d52bc590. Scope name:DefaultScope.
0,00 %:         [Clie] ClientScopeInfoLoadingArgs: [Client] Client Scope Table␣
↪Loading.
0,00 %:         [Clie] ClientScopeInfoLoadedArgs: [Client] [DefaultScope] [Version 0.
↪9.5] Last sync:17/07/2022 20:06:57 Last sync duration:0:0:2.172.
0,00 %:         [Adve] ServerScopeInfoLoadingArgs: [AdventureWorks] Server Scope␣
↪Table Loading.
0,00 %:         [Adve] ServerScopeInfoLoadedArgs: [AdventureWorks] [DefaultScope]␣
↪[Version 0.9.5] Last cleanup timestamp:0.
0,00 %:         [Adve] OperationArgs: Client Operation returned by server.
10,00 %:        [Clie] LocalTimestampLoadingArgs: [Client] Getting Local Timestamp.
10,00 %:        [Clie] LocalTimestampLoadedArgs: [Client] Local Timestamp␣
↪Loaded:17055.
30,00 %:        [Adve] ServerScopeInfoLoadingArgs: [AdventureWorks] Server Scope␣
↪Table Loading.
30,00 %:        [Adve] ServerScopeInfoLoadedArgs: [AdventureWorks] [DefaultScope]␣
↪[Version 0.9.5] Last cleanup timestamp:0.
30,00 %:        [Adve] DatabaseChangesApplyingArgs: Applying Changes. Total Changes␣
↪To Apply: 0
30,00 %:        [Adve] DatabaseChangesAppliedArgs: [Total] Applied:0. Conflicts:0.
55,00 %:        [Adve] LocalTimestampLoadingArgs: [AdventureWorks] Getting Local␣
↪Timestamp.
55,00 %:        [Adve] LocalTimestampLoadedArgs: [AdventureWorks] Local Timestamp␣
↪Loaded:2000.
55,00 %:        [Adve] DatabaseChangesSelectingArgs: [AdventureWorks] Getting Changes.
↪ [C:\Users\spertus\AppData\Local\Temp\DotmimSync]. Batch size:5000. IsNew:True.
55,00 %:        [Adve] TableChangesSelectingArgs: [Customer] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [Address] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [SalesOrderDetail] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [Product] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [ProductCategory] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [ProductModel] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [SalesOrderHeader] Getting Changes.
55,00 %:        [Adve] TableChangesSelectingArgs: [CustomerAddress] Getting Changes.
```

(continues on next page)

(continued from previous page)

```
55,00 %:        [Adve] TableChangesSelectingArgs: [ProductDescription] Getting␣
↪Changes.
55,00 %:        [Adve] TableChangesSelectedArgs: [ProductCategory] [Total] Upserts:41.
↪ Deletes:0. Total:41.
75,00 %:        [Adve] TableChangesSelectedArgs: [SalesOrderHeader] [Total]␣
↪Upserts:32. Deletes:0. Total:32.
75,00 %:        [Adve] TableChangesSelectedArgs: [ProductModel] [Total] Upserts:128.␣
↪Deletes:0. Total:128.
75,00 %:        [Adve] TableChangesSelectedArgs: [Address] [Total] Upserts:450.␣
↪Deletes:0. Total:450.
75,00 %:        [Adve] TableChangesSelectedArgs: [CustomerAddress] [Total]␣
↪Upserts:417. Deletes:0. Total:417.
75,00 %:        [Adve] TableChangesSelectedArgs: [SalesOrderDetail] [Total]␣
↪Upserts:542. Deletes:0. Total:542.
75,00 %:        [Adve] TableChangesSelectedArgs: [ProductDescription] [Total]␣
↪Upserts:762. Deletes:0. Total:762.
75,00 %:        [Adve] TableChangesSelectedArgs: [Product] [Total] Upserts:295.␣
↪Deletes:0. Total:295.
75,00 %:        [Adve] TableChangesSelectedArgs: [Customer] [Total] Upserts:847.␣
↪Deletes:0. Total:847.
75,00 %:        [Adve] DatabaseChangesSelectedArgs: [Total] Upserts:3514. Deletes:0.␣
↪Total:3514. [C:\Users\spertus\AppData\Local\Temp\DotmimSync\2022_07_17_
↪00fbihwicdj11]
75,00 %:        [Adve] ScopeSavingArgs: [AdventureWorks] Scope Table [ServerHistory]␣
↪Saving.
75,00 %:        [Adve] ScopeSavedArgs: [AdventureWorks] Scope Table [ServerHistory]␣
↪Saved.
75,00 %:        [Clie] DatabaseChangesApplyingArgs: Applying Changes. Total Changes␣
↪To Apply: 3514
75,00 %:        [Clie] TableChangesApplyingArgs: Applying Changes To␣
↪ProductDescription.
75,00 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying␣
↪[ProductDescription] batch rows. State:Modified. Count:762
80,42 %:        [Clie] TableChangesBatchAppliedArgs: [ProductDescription] [Modified]␣
↪Applied:(762) Total:(762/3514).
80,42 %:        [Clie] TableChangesAppliedArgs: [ProductDescription] Changes Modified␣
↪Applied:762. Resolved Conflicts:0.
80,42 %:        [Clie] TableChangesApplyingArgs: Applying Changes To ProductCategory.
80,42 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [ProductCategory]␣
↪batch rows. State:Modified. Count:41
80,71 %:        [Clie] TableChangesBatchAppliedArgs: [ProductCategory] [Modified]␣
↪Applied:(41) Total:(803/3514).
80,71 %:        [Clie] TableChangesAppliedArgs: [ProductCategory] Changes Modified␣
↪Applied:41. Resolved Conflicts:0.
80,71 %:        [Clie] TableChangesApplyingArgs: Applying Changes To ProductModel.
80,71 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [ProductModel]␣
↪batch rows. State:Modified. Count:128
81,62 %:        [Clie] TableChangesBatchAppliedArgs: [ProductModel] [Modified]␣
↪Applied:(128) Total:(931/3514).
81,62 %:        [Clie] TableChangesAppliedArgs: [ProductModel] Changes Modified␣
↪Applied:128. Resolved Conflicts:0.
81,62 %:        [Clie] TableChangesApplyingArgs: Applying Changes To Product.
81,62 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [Product] batch␣
↪rows. State:Modified. Count:295
83,72 %:        [Clie] TableChangesBatchAppliedArgs: [Product] [Modified]␣
↪Applied:(295) Total:(1226/3514).
83,72 %:        [Clie] TableChangesAppliedArgs: [Product] Changes Modified␣
↪Applied:295. Resolved Conflicts:0.
```

(continues on next page)

(continued from previous page)

```
83,72 %:        [Clie] TableChangesApplyingArgs: Applying Changes To Address.
83,72 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [Address] batch␣
↪rows. State:Modified. Count:450
86,92 %:        [Clie] TableChangesBatchAppliedArgs: [Address] [Modified]␣
↪Applied:(450) Total:(1676/3514).
86,92 %:        [Clie] TableChangesAppliedArgs: [Address] Changes Modified␣
↪Applied:450. Resolved Conflicts:0.
86,92 %:        [Clie] TableChangesApplyingArgs: Applying Changes To Customer.
86,92 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [Customer] batch␣
↪rows. State:Modified. Count:847
92,95 %:        [Clie] TableChangesBatchAppliedArgs: [Customer] [Modified]␣
↪Applied:(847) Total:(2523/3514).
92,95 %:        [Clie] TableChangesAppliedArgs: [Customer] Changes Modified␣
↪Applied:847. Resolved Conflicts:0.
92,95 %:        [Clie] TableChangesApplyingArgs: Applying Changes To CustomerAddress.
92,95 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [CustomerAddress]␣
↪batch rows. State:Modified. Count:417
95,92 %:        [Clie] TableChangesBatchAppliedArgs: [CustomerAddress] [Modified]␣
↪Applied:(417) Total:(2940/3514).
95,92 %:        [Clie] TableChangesAppliedArgs: [CustomerAddress] Changes Modified␣
↪Applied:417. Resolved Conflicts:0.
95,92 %:        [Clie] TableChangesApplyingArgs: Applying Changes To SalesOrderHeader.
95,92 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [SalesOrderHeader]␣
↪batch rows. State:Modified. Count:32
96,14 %:        [Clie] TableChangesBatchAppliedArgs: [SalesOrderHeader] [Modified]␣
↪Applied:(32) Total:(2972/3514).
96,14 %:        [Clie] TableChangesAppliedArgs: [SalesOrderHeader] Changes Modified␣
↪Applied:32. Resolved Conflicts:0.
96,14 %:        [Clie] TableChangesApplyingArgs: Applying Changes To SalesOrderDetail.
96,14 %:        [Clie] TableChangesApplyingSyncRowsArgs: Applying [SalesOrderDetail]␣
↪batch rows. State:Modified. Count:542
100,00 %:       [Clie] TableChangesBatchAppliedArgs: [SalesOrderDetail] [Modified]␣
↪Applied:(542) Total:(3514/3514).
100,00 %:       [Clie] TableChangesAppliedArgs: [SalesOrderDetail] Changes Modified␣
↪Applied:542. Resolved Conflicts:0.
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To SalesOrderDetail.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [SalesOrderDetail]␣
↪batch rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To SalesOrderHeader.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [SalesOrderHeader]␣
↪batch rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To CustomerAddress.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [CustomerAddress]␣
↪batch rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To Customer.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [Customer] batch␣
↪rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To Address.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [Address] batch␣
↪rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To Product.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [Product] batch␣
↪rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To ProductModel.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [ProductModel]␣
↪batch rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To ProductCategory.
```

(continues on next page)

(continued from previous page)

```
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying [ProductCategory]␣
↪batch rows. State:Deleted. Count:0
100,00 %:       [Clie] TableChangesApplyingArgs: Applying Changes To␣
↪ProductDescription.
100,00 %:       [Clie] TableChangesApplyingSyncRowsArgs: Applying␣
↪[ProductDescription] batch rows. State:Deleted. Count:0
100,00 %:       [Clie] DatabaseChangesAppliedArgs: [Total] Applied:3514. Conflicts:0.
100,00 %:       [Clie] ClientScopeInfoLoadingArgs: [Client] Client Scope Table␣
↪Loading.
100,00 %:       [Clie] ClientScopeInfoLoadedArgs: [Client] [DefaultScope] [Version 0.
↪9.5] Last sync:17/07/2022 20:06:57 Last sync duration:0:0:2.172.
100,00 %:       [Clie] MetadataCleaningArgs: Cleaning Metadatas.
100,00 %:       [Clie] MetadataCleanedArgs: Tables Cleaned:0. Rows Cleaned:0.
100,00 %:       [Clie] ScopeSavingArgs: [Client] Scope Table [Client] Saving.
100,00 %:       [Clie] ScopeSavedArgs: [Client] Scope Table [Client] Saved.
100,00 %:       [Clie] SessionEndArgs: [Client] Session Ends. Id:f62adec4-21a7-4a35-
↪b86e-d3d7d52bc590. Scope name:DefaultScope.
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 3514
        Total changes  applied: 3514
        Total resolved conflicts: 0
        Total duration :00.00:00.509
Sync Ended. Press a key to start again, or Escapte to end
```

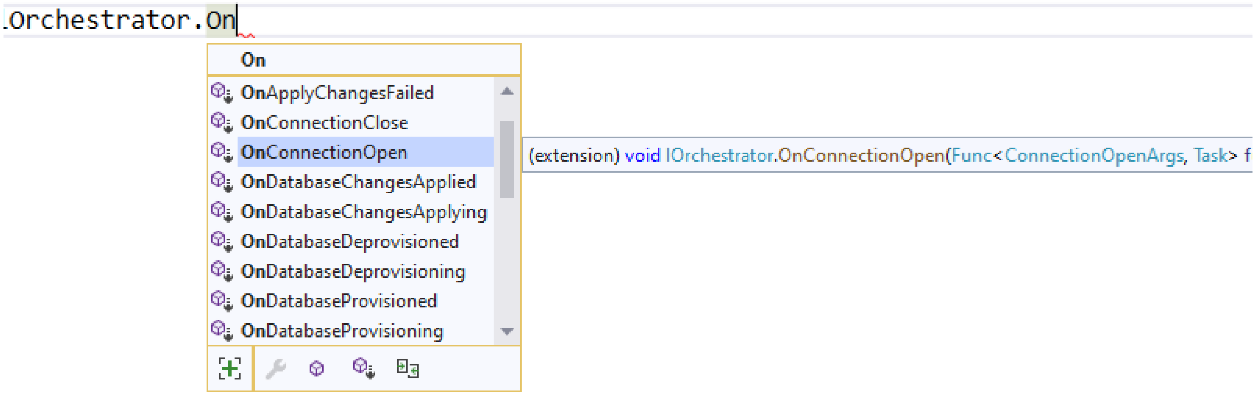
2.8 Interceptors

Ìnterceptor<T> : A more advanced technic to handle a lot of more events from within **DMS**

2.8.1 Overview

The Progress<T> stuff is great, but as we said, it's mainly read only, and the progress is always reported **at the end of a current sync stage**.

So, if you need a more granular control on all the progress values, you can subscribe to an Interceptor<T>.
On each **orchestrator**, you will find a lot of relevant methods to intercept the sync process, encapsulate in a fancy OnMethodAsync() method:



```
.Orchestrator.On
```

(extension) void IOrchestrator.OnConnectionOpen(Func<ConnectionOpenArgs, Task> f

Imagine you have a table that should **never** be synchronized on one particular client (and is part of your `SyncSetup`). You're able to use an interceptor like this:

```
// We are using a cancellation token that will be passed as an argument
// to the SynchronizeAsync() method !
var cts = new CancellationTokenSource();

agent.LocalOrchestrator.OnTableChangesApplying((args) =>
{
    if (args.SchemaTable.TableName == "Table_That_Should_Not_Be_Sync")
        args.Cancel = true;
});
```

Be careful, your table will never be synced !

Intercepting rows

You may want to intercept all the rows that have just been selected from the source (client or server), and are about to be sent to their destination (server or client).
Or even intercept all the rows that are going to be applied on a destination database.
That way, you may be able to modify these rows, to meet your business / requirements rules.

Hint: You will find the sample used for this chapter, here : Spy sample.

`DMS` workload allows you to intecept different kinds of events on different levels:

- Database level
- Table level
- Row level

On each side (client and server), you will have:

- Interceptors during the "_Select_" phase : Getting changes from the database.
- Interceptors during the "_Apply_" phase : Applying Insert / Delete or Update to the database.
- Interceptors for extra workloads like conflict resolution, serialization, converters & so on . . .

On each level you will have:

- A before event: Generally ending by "_ing_" like `OnDatabaseChangesApplying`.
- An after event: Generally ending by "_ied_" like `OnDatabaseChangesApplied`.

2.8.2 Datasource level

We have some interceptors that are not related to a specific table, but to the whole datasource.
They are tight to the connection, the transaction or the command used to get the changes, apply changes or even handle conflicts and errors.

OnConnectionOpen

The `OnConnectionOpen` event is raised when a connection is opened, through the underline provider.

TODO

OnReConnect

The `OnReConnect` event is raised when a connection is re-opened, through the underline provider.

DMS is using a custom retry policy, inspired from Polly to manage a connection retry policy.

```
localOrchestrator.OnReConnect(args => {
    Console.WriteLine($"[Retry] Can't connect to database {args.Connection?.Database}.
↪ " +
    $"Retry N°{args.Retry}. " +
    $"Waiting {args.WaitingTimeSpan.Milliseconds}. Exception:{args.HandledException.
↪Message}.");
});
```

You can customize the retry policy, only on http mode, when using a `WebRemoteOrchestrator` instance.

```
var webRemoteOrchestrator = new WebRemoteOrchestrator(serviceUri);

// limit to 2 retries only
webRemoteOrchestrator.SyncPolicy.RetryCount = 2;
```

```
var webRemoteOrchestrator = new WebRemoteOrchestrator(serviceUri);

// retry for ever (not sure it's a good idea, that being said)
webRemoteOrchestrator.SyncPolicy = SyncPolicy.WaitAndRetryForever(TimeSpan.
↪FromSeconds(1));
```

OnTransactionOpen

The `OnTransactionOpen` event is raised when a transaction is opened, through the underline provider.

TODO

OnConnectionClose

The `OnConnectionClose` event is raised when a connection is closed, through the underline provider.

TODO

OnTransactionCommit

The `OnTransactionCommit` event is raised when a transaction is committed, through the underline provider.

TODO

OnGetCommand

The OnGetCommand interceptor is happening when a command is retrieved from the underline provider (`SqlSyncProvider`, `MySqlSyncProvider`, etc..)

```
agent.RemoteOrchestrator.OnGetCommand(args =>
{
    if (args.Command.CommandType == CommandType.StoredProcedure)
    {
        args.Command.CommandText = args.Command.CommandText.Replace("_filterproducts_
↪", "_default_");
    }
});
```

OnExecuteCommand

The `OnExecuteCommand` interceptor is happening when a command is about to be executed on the client or server.

```
agent.RemoteOrchestrator.OnExecuteCommand(args =>
{
    Console.WriteLine(args.Command.CommandText);
});
```

2.8.3 Selecting changes

Regarding the rows selection from your client or server:

- `OnDatabaseChangesSelecting` : Raised before selecting rows. You have info about the tmp folder and batch size that will be used.
- `OnTableChangesSelecting` : Raised before selecting rows for a particular table : You have info about the current table and the `DbCommand` used to fetch data.

On the other side, once rows are selected, you still can:

- `OnRowsChangesSelected` : Raised once a row is read from the databse, but not yet serialized to disk. Row is still in memory, and connection / reader still opened.
- `OnTableChangesSelected` : Raised once a table changes as been fully read. Changes (all batches for this table) are serialized to disk. Connection / reader are closed.
- `OnDatabaseChangesSelected` : Raised once all changes are grabbed from the local database. Changes are serialized to disk.

OnDatabaseChangesSelecting

Occurs when changes are going to be queried from the underline database.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
localOrchestrator.OnDatabaseChangesSelecting(args => {
    Console.WriteLine($"Getting changes from local database:");
    Console.WriteLine($"Batch directory: {args.BatchDirectory}. Batch size: {args.
↪BatchSize}.
                        Is first sync: {args.IsNew}");
    Console.WriteLine($"From: {args.FromTimestamp}. To: {args.ToTimestamp}.");
}
```

OnTableChangesSelecting

Occurs when changes are going to be queried from the underline database for a particular table.
You have access to the command / connection / transaction that going to be used to query the database.

Note: The `Command` property can be changed here, depending on your needs.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
localOrchestrator.OnTableChangesSelecting(args =>
{
    Console.WriteLine($"Getting changes from local database " +
                      $"for table:{args.SchemaTable.GetFullName()}");

    Console.WriteLine($"{args.Command.CommandText}");
});
```

OnRowsChangesSelected

Occurs when a row is selected from the underline database.
You have access to the `SyncRow` row property, the table schema and the state of the row (Modified, Deleted).
You can change any value from the `SyncRow` property on the fly if needed.

```
var localOrchestrator = new LocalOrchestrator(clientProvider);
localOrchestrator.OnRowsChangesSelected(args =>
{
    Console.WriteLine($"Row read from local database for table:{args.SchemaTable.
↪GetFullName()}");
    Console.WriteLine($"{args.SyncRow}");
});
```

Warning: This event is raised for each row, so be careful with the number of rows you have in your database.

Plus, this event is raised during the reading phase of the database, that means that the connection is still opened.

If you have a lot of rows, you may want to use the `OnTableChangesSelected` event instead, that occurs once the table is fully read, and results are serialized on disk.

OnTableChangesSelected

Occurs when a table is fully selected from the underline database.

```
localOrchestrator.OnTableChangesSelected(args =>
{
    Console.WriteLine($"Table: {args.SchemaTable.GetFullName()} read. " +
                      $"Rows count:{args.BatchInfo.RowsCount}.");" +

    Console.WriteLine($"Directory: {args.BatchInfo.DirectoryName}. " +
                      $"Number of files: {args.BatchPartInfos?.Count()} ");

    Console.WriteLine($"Changes: {args.TableChangesSelected.TotalChanges} " +
                      $"({args.TableChangesSelected.Upserts}/{args.
↪TableChangesSelected.Deletes})");
});
```

Hint: You have access to the serialized rows on disk, in the `BatchInfo` property.

You can iterate through all the files, and read the rows from the files, using the LoadTableFromBatchInfoAsync

OnDatabaseChangesSelected

Occurs when all changes are selected from the underline database.
The `BatchInfo` property is fully filled with all batch files.

```
localOrchestrator.OnDatabaseChangesSelected(args =>
{
    Console.WriteLine($"Directory: {args.BatchInfo.DirectoryName}. "
                      $"Number of files: {args.BatchInfo.BatchPartsInfo?.Count()} ");

    Console.WriteLine($"Total: {args.ChangesSelected.TotalChangesSelected} " +
                        $"({args.ChangesSelected.TotalChangesSelectedUpdates}" +
                        $"/{args.ChangesSelected.TotalChangesSelectedDeletes})");

    foreach (var table in args.ChangesSelected.TableChangesSelected)
        Console.WriteLine($"Table: {table.TableName}. "
                          $"Total: {table.TotalChanges} ({table.Upserts / table.
↪Deletes}");
});
```

Hint: You have access to the serialized rows on disk, in the `BatchInfo` property.

You can iterate through all the files, and read the rows from the files, using the LoadTablesFromBatchInfoAsync

2.8.4 Applying changes

Regarding the rows to apply on your client (or server) database, you can intercept different kind of events:

- `OnDatabaseChangesApplying`: Rows are serialized locally in a batch info folder BUT they are not yet read internally and are not in memory. You can iterate over all the files and see if you have rows to apply.
- `OnTableChangesApplying`: Rows are still on disk and not in memory. This interceptor is called for each table that has rows to apply.
- `OnRowsChangesApplying` : Rows ARE now in memory, in a batch (depending on batch size and provider max batch), and are going to be applied.

On the other side, once rows are applied, you can iterate through different interceptors:

- `OnTableChangesApplied`: Contains a summary of all rows applied on a table for a particular state (DataRowState.Modified or Deleted).
- `OnDatabaseChangesApplied` : Contains a summary of all changes applied on the database level.

OnDatabaseChangesApplying

The `OnDatabaseChangesApplying` interceptor is happening when changes are going to be applied on the client or server.
The changes are not yet loaded in memory. They are all stored locally in a temporary folder.

To be able to load batches from the temporary folder, or save rows, you can use the LoadTablesFromBatchInfoAsync and SaveTableToBatchPartInfoAsync methods

```
localOrchestrator.OnDatabaseChangesApplying(async args =>
{
    foreach (var table in args.ApplyChanges.Schema.Tables)
    {
        // loading in memory all batches containing rows for the current table
        var syncTable = await localOrchestrator.LoadTableFromBatchInfoAsync(
            args.ApplyChanges.BatchInfo, table.TableName, table.SchemaName);

        Console.WriteLine($"Changes for table {table.TableName}. Rows:{syncTable.Rows.
↪Count}");
        foreach (var row in syncTable.Rows)
            Console.WriteLine(row);

        Console.WriteLine();

    }
});
```

OnTableChangesApplying

The `OnTableChangesApplying` is happening right before rows are applied on the client or server.
Like `OnDatabaseChangesApplying` the changes are not yet loaded in memory. They are all stored locally in a temporary folder.
Be careful, this interceptor is called for each state (Modified / Deleted), so be sure to check the state of the rows:
Note that this interceptor is not called if the current tables has no rows to applied.

```
// Just before applying changes locally, at the table level
localOrchestrator.OnTableChangesApplying(async args =>
{
    if (args.BatchPartInfos != null)
    {
        var syncTable = await localOrchestrator.LoadTableFromBatchInfoAsync(
            args.BatchInfo, args.SchemaTable.TableName, args.SchemaTable.SchemaName,␣
↪args.State);

        if (syncTable != null && syncTable.HasRows)
        {
            Console.WriteLine($"- --------------------------------------------");
            Console.WriteLine($"- Applying [{args.State}]
                    changes to Table {args.SchemaTable.GetFullName()}");

            foreach (var row in syncTable.Rows)
                Console.WriteLine(row);
        }

    }
});
```

OnBatchChangesApplying

The `OnBatchChangesApplying` interceptor is happening when a batch for a particular table is about to be applied on the local data source.
The number of rows contained in each batch file is depending on the value you have set in your SyncOptions instance : `SyncOptions.BatchSize` (Default is 2 Mo)
This interceptor is called for each batch file, and for each state (`Modified` / `Deleted`).
That means that if you have **1000** batches, and **2** calls of this interceptor (one for `Modified`, one for `Deleted`), you will fire **2000** times this interceptor.

```
agent.LocalOrchestrator.OnBatchChangesApplying(async args =>
{
    if (args.BatchPartInfo != null)
    {
        Console.WriteLine($"FileName:{args.BatchPartInfo.FileName}. RowsCount:{args.
↪BatchPartInfo.RowsCount} ");
        Console.WriteLine($"Applying rows from this batch part info:");

        var table = await agent.LocalOrchestrator.
↪LoadTableFromBatchPartInfoAsync(args.BatchInfo,
                        args.BatchPartInfo, args.State, args.Connection, args.
↪Transaction);

        foreach (var row in table.Rows)
            Console.WriteLine(row);

    }
});
```

OnRowsChangesApplying

The `OnRowsChangesApplying` interceptor is happening just before applying a batch of rows to the local (client or server) database.

The number of rows to be applied here is depending on:

- The batch size you have set in your SyncOptions instance : `SyncOptions.BatchSize` (Default is 2 Mo)
- The max number of rows to applied in one single instruction : `Provider.BulkBatchMaxLinesCount` (Default is 10 000 rows per instruction)

```
localOrchestrator.OnRowsChangesApplying(async args =>
{
    Console.WriteLine($"- --------------------------------------------");
    Console.WriteLine($"- In memory rows that are going to be Applied");
    foreach (var row in args.SyncRows)
        Console.WriteLine(row);

    Console.WriteLine();
});
```

OnTableChangesApplied

The `OnTableChangesApplied` interceptor is happening when all rows, for a specific table, are applied on the local (client or server) database.

TODO

OnBatchChangesApplying

The `OnBatchChangesApplied` interceptor is happening when a batch for a particular table has been applied.

```
agent.LocalOrchestrator.OnBatchChangesApplied(async args =>
{
    if (args.BatchPartInfo != null)
    {
        Console.WriteLine($"FileName:{args.BatchPartInfo.FileName}. RowsCount:{args.
↪BatchPartInfo.RowsCount} ");
        Console.WriteLine($"Applied rows from this batch part info:");

        var table = await agent.LocalOrchestrator.
↪LoadTableFromBatchPartInfoAsync(args.BatchInfo,
                        args.BatchPartInfo, args.State, args.Connection, args.
↪Transaction);

        foreach (var row in table.Rows)
            Console.WriteLine(row);

    }
});
```

OnDatabaseChangesApplied

The `OnDatabaseChangesApplied` interceptor is happening when all changes are applied on the client or server.

TODO

2.8.5 Snapshots

See how snapshots work in the Snapshots section.

OnSnapshotCreating

The `OnSnapshotCreating` interceptor is happening when a snapshot is going to be created from the server side

TODO

OnSnapshotCreated

The `OnSnapshotCreated` interceptor is happening when a snapshot is created from the server side.

TODO

OnSnapshotApplying

The `OnSnapshotApplying` interceptor is happening when a snapshot is going to be applied on the client side.

TODO

OnSnapshotApplied

The `OnSnapshotApplied` interceptor is happening when a snapshot is applied on the client side.

TODO

2.8.6 Specific

OnProvisioning

The `OnProvisioning` interceptor is happening when the database is being provisioned.

TODO

OnProvisioned

The `OnProvisioned` interceptor is happening when the database is provisioned.

TODO

OnDeprovisioning

The `OnDeprovisioning` interceptor is happening when the database is being deprovisioned.

TODO

OnDeprovisioned

The `OnDeprovisioned` interceptor is happening when the database is deprovisioned.

TODO

OnLocalTimestampLoading

OnLocalTimestampLoaded

OnSchemaLoading

OnSchemaLoaded

OnMetadataCleaning

OnMetadataCleaned

OnApplyChangesConflictOccured

See Conflicts

OnApplyChangesErrorOccured

See Errors

OnSerializingSyncRow

OnDeserializingSyncRow

OnSessionBegin

OnSessionEnd

OnConflictingSetup

OnGettingOperation

The `OnGettingOperation` interceptor is happening when a server receive a request from a client for initiate a synchronization.

From here, you have the option to **override** the operation, using the `SyncOperation` enumeration:

```
public enum SyncOperation
{
    /// <summary>
    /// Normal synchronization
    /// </summary>
    Normal = 0,

    /// <summary>
    /// Reinitialize the whole sync database,
    /// applying all rows from the server to the client
    /// </summary>
    Reinitialize = 1,

    /// <summary>
    /// Reinitialize the whole sync database,
    /// applying all rows from the server to the client, after trying a client upload
    /// </summary>
    ReinitializeWithUpload = 2,

    /// <summary>
    /// Drop all the sync metadatas even tracking tables and
    /// scope infos and make a full sync again
    /// </summary>
    DropAllAndSync = 4,

    /// <summary>
    /// Drop all the sync metadatas even tracking tables and
    /// scope infos and exit
    /// </summary>
    DropAllAndExit = 8,

    /// <summary>
    /// Deprovision stored procedures and triggers and sync again
    /// </summary>
    DeprovisionAndSync = 16,

    /// <summary>
    /// Exit a Sync session without syncing
    /// </summary>
    AbortSync = 32,
}
```

Useful for example to force a ReinitializeWithUpload operation, when you have a conflict on the client side, and you want to force the client to upload all his changes to the server, then reinitialize everything.

Hint: This method is usefull most of the time, from the server side, when using a proxy ASP.NET Core Web API.

```
[HttpPost]
public async Task Post()
{

    var scopeName = context.GetScopeName();
    var clientScopeId = context.GetClientScopeId();

    var webServerAgent = webServerAgents.First(wsa => wsa.ScopeName == scopeName);
```

(continues on next page)

(continued from previous page)

```
    webServerAgent.RemoteOrchestrator.OnGettingOperation(operationArgs =>
    {
        if (scopeName == "all" && clientScopeId == A_PARTICULAR_CLIENT_ID_TO_CHECK)
            operationArgs.SyncOperation = SyncOperation.ReinitializeWithUpload;

    });

    await webServerAgent.HandleRequestAsync(context);
}
```

OnOutdated

The `OnOutdated` interceptor is happening when a client is outdated. You can use this interceptor to force the client to reinitialize its database if it is outdated.

By default, an error is raised, and sync is stopped. This event is raised only on the client side.

```
agent.LocalOrchestrator.OnOutdated(oa =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("local database is too old to synchronize with the server.");
    Console.ResetColor();
    Console.WriteLine("Do you want to synchronize anyway, and potentially lost data ?
↪");
    Console.Write("Enter a value ('r' for reinitialize or 'ru' for reinitialize with␣
↪upload): ");
    var answer = Console.ReadLine();

    if (answer.ToLowerInvariant() == "r")
        oa.Action = OutdatedAction.Reinitialize;
    else if (answer.ToLowerInvariant() == "ru")
        oa.Action = OutdatedAction.ReinitializeWithUpload;

});
```

2.8.7 Web

Some interceptors are specific to web orchestrators `WebRemoteOrchestrator` & `WebServerAgent`.

These orchestrators will let you intercept all the `Requests` and `Responses` that will be generated by `DMS` during a web call.

WebServerAgent

The two first interceptors will intercept basically all requests and responses coming in and out:

- `webServerAgent.OnHttpGettingRequest(args => {})`
- `webServerAgent.OnHttpSendingResponse(args => {})`

Each of them will let you access the *HttpContext*, *SyncContext* and *SessionCache* instances:

```
webServerAgent.OnHttpGettingRequest(args =>
{
    var httpContext = args.HttpContext;
    var syncContext = args.Context;
    var session = args.SessionCache;
});
```

The two last new web server http interceptors will let you intercept all the calls made when server *receives* client changes and when server *sends back* server changes.

- `webServerAgent.OnHttpGettingChanges(args => {});`
- `webServerAgent.OnHttpSendingChanges(args => {});`

Here is a quick example using all of them:

```
webServerAgent.OnHttpGettingRequest(req =>
    Console.WriteLine("Receiving Client Request:" + req.Context.SyncStage +
    ". " + req.HttpContext.Request.Host.Host + "."));

webServerAgent.OnHttpSendingResponse(res =>
    Console.WriteLine("Sending Client Response:" + res.Context.SyncStage +
    ". " + res.HttpContext.Request.Host.Host));

webServerAgent.OnHttpGettingChanges(args
    => Console.WriteLine("Getting Client Changes" + args));
webServerAgent.OnHttpSendingChanges(args
    => Console.WriteLine("Sending Server Changes" + args));

await webServerManager.HandleRequestAsync(context);
```

```
Receiving Client Request:ScopeLoading. localhost.
Sending Client Response:Provisioning. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending All Snapshot Changes. Rows:0
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Getting Client Changes[localhost] Getting All Changes. Rows:0
Sending Server Changes[localhost] Sending Batch  Changes. (1/11). Rows:658
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (2/11). Rows:321
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (3/11). Rows:29
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (4/11). Rows:33
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (5/11). Rows:39
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (6/11). Rows:55
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (7/11). Rows:49
Sending Client Response:ChangesSelecting. localhost
```

(continues on next page)

(continued from previous page)

```
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (8/11). Rows:32
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (9/11). Rows:758
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (10/11). Rows:298
Sending Client Response:ChangesSelecting. localhost
Receiving Client Request:ChangesSelecting. localhost.
Sending Server Changes[localhost] Sending Batch  Changes. (11/11). Rows:1242
Sending Client Response:ChangesSelecting. localhost
Synchronization done.
```

The main differences are that the two first ones will intercept **ALL** requests coming from the client and the two last one will intercept **Only** requests where data are exchanged (but you have more detailed)

WebRemoteOrchestrator

You have pretty much the same `Http` interceptors on the client side. `OnHttpGettingRequest` becomes `OnHttpSendingRequest` and `OnHttpSendingResponse` becomes `OnHttpGettingResponse`:

```
localOrchestrator.OnHttpGettingResponse(req => Console.WriteLine("Receiving Server␣
↪Response"));
localOrchestrator.OnHttpSendingRequest(res =>Console.WriteLine("Sending Client␣
↪Request."));
localOrchestrator.OnHttpGettingChanges(args => Console.WriteLine("Getting Server␣
↪Changes" + args));
localOrchestrator.OnHttpSendingChanges(args => Console.WriteLine("Sending Client␣
↪Changes" + args));
```

```
Sending Client Request.
Receiving Server Response
Sending Client Request.
Receiving Server Response
Sending Client Changes[localhost] Sending All Changes. Rows:0
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (1/11). Rows:658
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (2/11). Rows:321
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (3/11). Rows:29
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (4/11). Rows:33
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (5/11). Rows:39
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (6/11). Rows:55
Sending Client Request.
```

(continues on next page)

(continued from previous page)

```
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (7/11). Rows:49
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (8/11). Rows:32
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (9/11). Rows:758
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (10/11). Rows:298
Sending Client Request.
Receiving Server Response
Getting Server Changes[localhost] Getting Batch Changes. (11/11). Rows:1242
Synchronization done.
```

Example: Hook Bearer token

The idea is to inject the user identifier `UserId` in the `SyncParameters` collection on the server, after having extract this value from a `Bearer` token.

That way the `UserId` is not hard coded or store somewhere on the client application, since this value is generated during the authentication part.

As you can see:

- My `SyncController` is marked with the *[Authorize]* attribute.
- The orchestrator is only called when we know that the user is authenticated.
- We are injecting the `UserId` value coming from the bearer into the `SyncContext.Parameters`.
- Optionally, because we don't want to send back this value to the client, we are removing it when sending the response.

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
public class SyncController : ControllerBase
{
    private WebServerAgent webServerAgent;

    // Injected thanks to Dependency Injection
    public SyncController(WebServerAgent webServerAgent)
        => this.webServerAgent = webServerAgent;

    /// <summary>
    /// This POST handler is mandatory to handle all the sync process
    [HttpPost]
    public async Task Post()
    {
        // If you are using the [Authorize] attribute you don't need to check
        // the User.Identity.IsAuthenticated value
        if (HttpContext.User.Identity.IsAuthenticated)
        {
            // OPTIONAL: -------------------------------------------
            // OPTIONAL: Playing with user coming from bearer token
```

(continues on next page)

(continued from previous page)

```
            // OPTIONAL: -------------------------------------------

            // on each request coming from the client, just inject the User Id␣
↪parameter
            webServerAgent.OnHttpGettingRequest(args =>
            {
                var pUserId = args.Context.Parameters["UserId"];

                if (pUserId == null)
                {
                    var userId = this.HttpContext.User.Claims.FirstOrDefault(
                        x => x.Type == ClaimTypes.NameIdentifier);
                    args.Context.Parameters.Add("UserId", userId);
                }

            });

            // Because we don't want to send back this value, remove it from the␣
↪response
            webServerAgent.OnHttpSendingResponse(args =>
            {
                if (args.Context.Parameters.Contains("UserId"))
                    args.Context.Parameters.Remove("UserId");
            });

            await webServerAgent.HandleRequestAsync(this.HttpContext);
        }
        else
        {
            this.HttpContext.Response.StatusCode = StatusCodes.Status401Unauthorized;
        }
    }

    /// <summary>
    /// This GET handler is optional. It allows you to see the configuration hosted␣
↪on the server
    /// The configuration is shown only if Environmenent == Development
    /// </summary>
    [HttpGet]
    [AllowAnonymous]
    public Task Get() => this.HttpContext.WriteHelloAsync(webServerAgent);
}
```

2.9 Change Tracking

SQL Server provides a great feature that track changes to data in a database: **Change tracking**.
This features enables applications to determine the DML changes (insert, update, and delete operations) that were made to user tables in a database.

Change tracking is supported since **SQL Server 2008** and is available from within **Azure Sql Database**.
If you need, for some reasons, to run your sync from an older version, you will have to fallback on the `SqlSyncProvider`.

Note: If you need more information on this feature, the best place to start is here : Track data changes with SQL Server

A new **Sql** sync provider which uses this **Change Tracking** feature is available with **DMS**:
This provider is called `SqlSyncChangeTrackingProvider`.

The `SqlSyncChangeTrackingProvider` is compatible with all others sync providers: You can have a server database using the `SqlSyncChangeTrackingProvider` and some clients databases using any of the others providers.

What does it mean to use Change Tracking from within your database ?

- No more tracking tables in your database
- No more triggers on your tables in your database
- Metadatas retention managed by SQL Server itself
- Changes tracked by the SQL Engine, way better performances than using triggers and tracking tables

To be able to use `SqlSyncChangeTrackingProvider` on your database, do not forget to activate the **Change Tracking** on your database :

```
ALTER DATABASE AdventureWorks
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 14 DAYS, AUTO_CLEANUP = ON)
```

You don't have to activate **Change Tracking** on each table. It will be enabled by **DMS** on each table part of the sync process.

Once it's done, the code is almost the same:

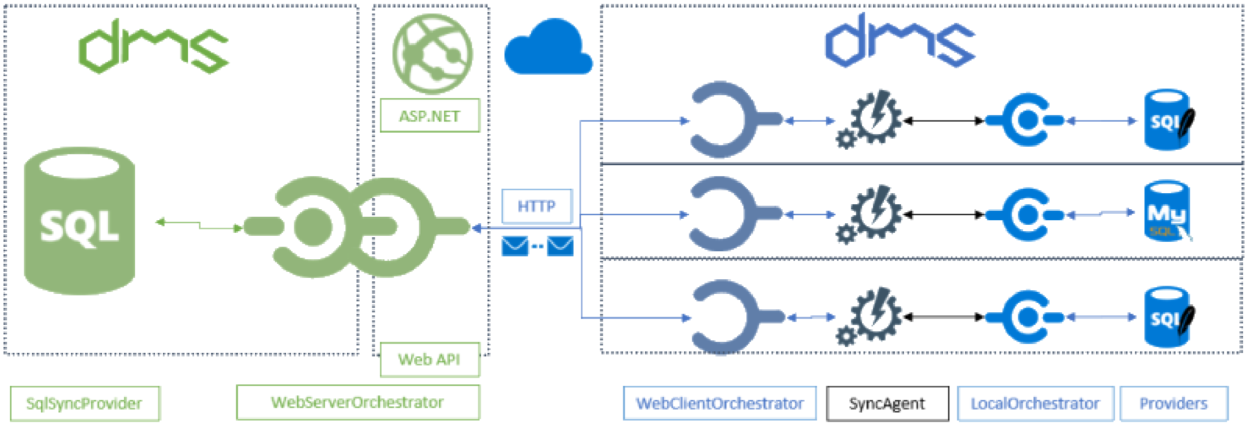
```
var serverProvider = new SqlSyncChangeTrackingProvider("Data Source=....");
var clientProvider = new SqlSyncChangeTrackingProvider("Data Source=....");
```

2.10 ASP.NET Core Web Proxy

Let's say… *in the real world*, you will not have *always* a direct TCP link from your client machine to your enterprise server.

Even though, it's a good practice to protect you database server behing a web api. That's why we will use a *sync web proxy*, and we will expose our server to sync, through a web api.

Here is the overall architecture:



2.10.1 Overview

Hint: You will find the sample used on this chapter, here : Hello web sync sample .

To be able to *proxify* everything, we should:

Server side:

- Create a new **ASP.NET Core Web application**.
- Add the Dotmim.Sync.Web.Server nuget package to the ASP.NET project.
- Add the server provider. As we are using sql server with change tracking, we are adding Dotmim.Sync.SqlSyncChangeTrackingProvider .
- Add the required configuration to the `Startup.cs` file.
- Create a new controller and intercept all requests to handle the synchronisation.

Client side:

- Create any kind of client application (Console, Windows Forms, WPF . . .)
- Add the Dotmim.Sync.Web.Client nuget package to the client application:
- Add the client provider. For example the Dotmim.Sync.SqliteSyncProvider
- Create a new `SyncAgent` using a local orchestrator with the `SqliteSyncProvider` and a remote `WebRemoteOrchestrator` orchestrator.

2.10.2 Server side

Note: We will start from the Hello sync sample sample and will migrate it to the web architecture.

Once your **ASP.NET** application is created, we're adding the specific web server package and our server provider:

- `Dotmim.Sync.Web.Server`: This package will allow us to expose everything we need, through a **.Net core Web API**
- `Dotmim.Sync.SqlServer.ChangeTracking`: This package will allow us to communicate with the SQL Server database.

Once we have added these **DMS** packages to our project, we are configuring the Sync provider in the `Startup` class, thanks to Dependency Injection.

Be careful, some services are required, but not part of **DMS** (like `.AddDistributedMemoryCache()` and `.AddSession()` for instance.)
Do not forget to add the session middleware as well (`app.UseSession();`)

Note: `DMS` uses a lot of http request during one user's sync. That's why Session is mandatory. Do not forget to add it in your configuration.

Having a cache is mandatory to be able to serve multiple requests for one particular session (representing one sync client)

Simple Scope

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));

    // [Required]: Get a connection string to your server data source
    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"SqlConnection"];

    var options = new SyncOptions { };

    // [Required] Tables involved in the sync process:
    var tables = new string[] {"ProductCategory", "ProductModel", "Product",
    "Address", "Customer", "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail" }
↪;

    // [Required]: Add a SqlSyncProvider acting as the server hub.
    services.AddSyncServer<SqlSyncChangeTrackingProvider>(connectionString, tables,␣
↪options);

}


// This method gets called by the runtime. Use this method to configure the HTTP␣
↪request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();
    app.UseSession();
```

(continues on next page)

(continued from previous page)

```
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Once we have correctly configured our sync process, we can create our controller:

- Create a new controller (for example `SyncController`)
- In this newly created controller, inject your `WebServerAgent` instance.
- Use this newly injected instance in the `POST` method, calling the `HandleRequestAsync` method and … **that's all** !
- We can optionally add a `GET` method, to see our configuration from within the web browser. Useful to check if everything is configured correctly.

```
[ApiController]
[Route("api/[controller]")]
public class SyncController : ControllerBase
{
    private WebServerAgent webServerAgent;
    private readonly IWebHostEnvironment env;

    // Injected thanks to Dependency Injection
    public SyncController(WebServerAgent webServerAgent, IWebHostEnvironment env)
    {
        this.webServerAgent = webServerAgent;
        this.env = env;
    }

    /// <summary>
    /// This POST handler is mandatory to handle all the sync process
    /// </summary>
    /// <returns></returns>
    [HttpPost]
    public Task Post()
        => webServerAgent.HandleRequestAsync(this.HttpContext);

    /// <summary>
    /// This GET handler is optional. It allows you to see the configuration hosted␣
↪on the server
    /// </summary>
    [HttpGet]
    public async Task Get()
    {
        if (env.IsDevelopment())
        {
            await this.HttpContext.WriteHelloAsync(webServerAgent);
        }
        else
        {
            var stringBuilder = new StringBuilder();

            stringBuilder.AppendLine("<!doctype html>");
            stringBuilder.AppendLine("<html>");
            stringBuilder.AppendLine("<title>Web Server properties</title>");
```

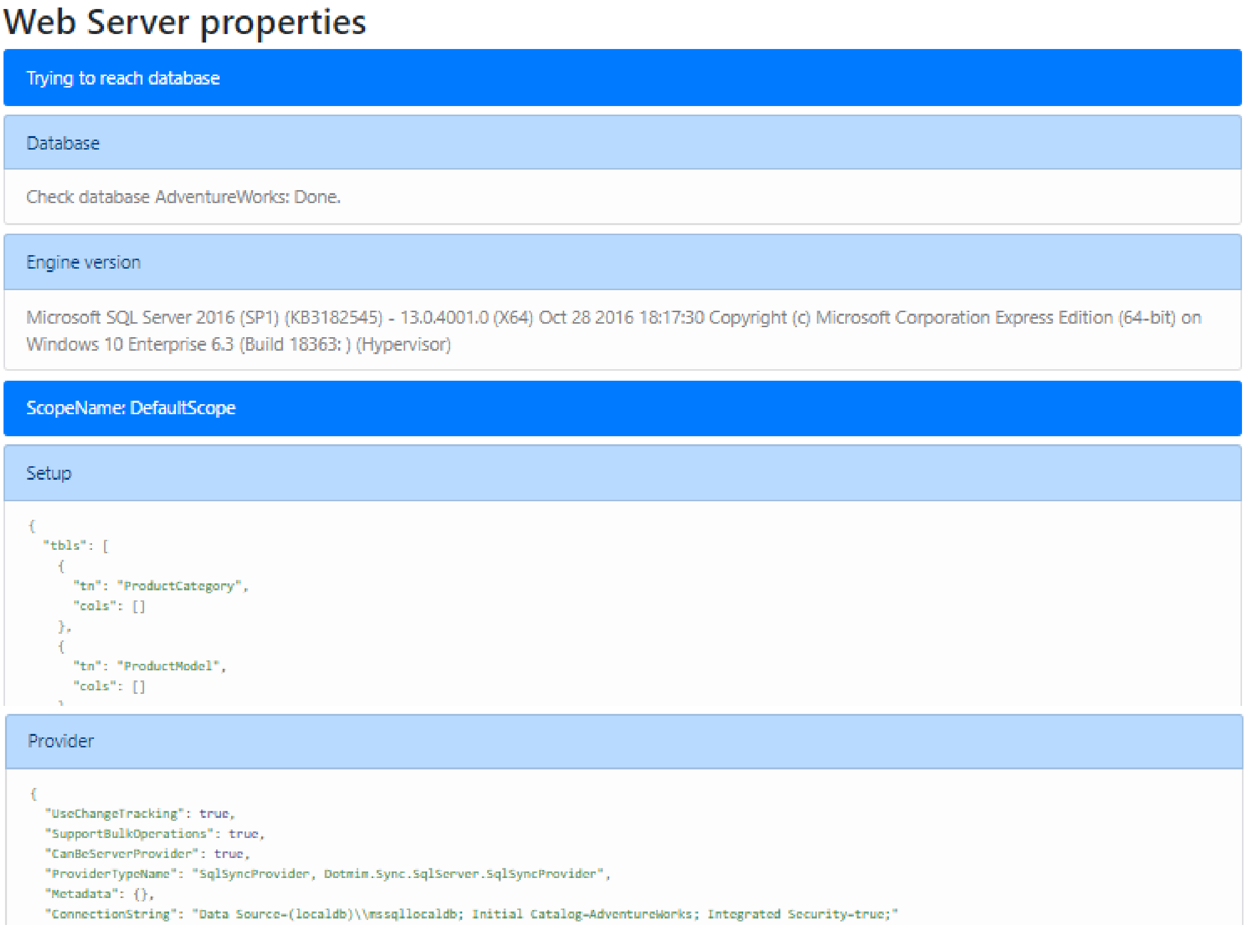
(continues on next page)

(continued from previous page)

```
                stringBuilder.AppendLine("<body>");
                stringBuilder.AppendLine(" PRODUCTION MODE. HIDDEN INFO ");
                stringBuilder.AppendLine("</body>");
                await this.HttpContext.Response.WriteAsync(stringBuilder.ToString());
            }
        }

}
```

Launch your browser and try to reach *sync* web page. (Something like https://localhost:{[}YOUR_PORT{]}/api/sync)

You should have useful information, like a test to reach your server database, your `SyncSetup`, your `SqlSyncProvider`, your `SyncOptions` and your `WebServerOptions` configuration:



Web Server properties

Trying to reach database

Database

Check database AdventureWorks: Done.

Engine version

Microsoft SQL Server 2016 (SP1) (KB3182545) - 13.0.4001.0 (X64) Oct 28 2016 18:17:30 Copyright (c) Microsoft Corporation Express Edition (64-bit) on Windows 10 Enterprise 6.3 (Build 18363:) (Hypervisor)

ScopeName: DefaultScope

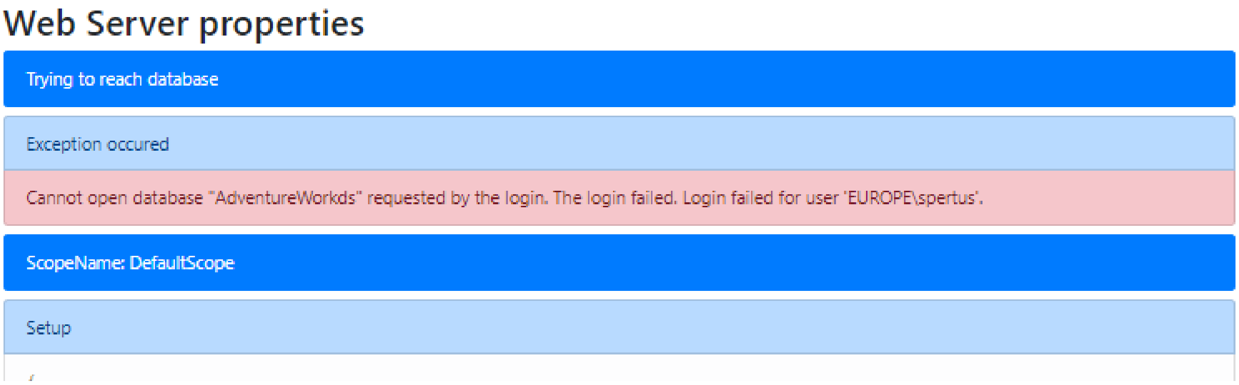
Setup

```
{
  "tbls": [
    {
      "tn": "ProductCategory",
      "cols": []
    },
    {
      "tn": "ProductModel",
      "cols": []
```

Provider

```
{
  "UseChangeTracking": true,
  "SupportBulkOperations": true,
  "CanBeServerProvider": true,
  "ProviderTypeName": "SqlSyncProvider, Dotmim.Sync.SqlServer.SqlSyncProvider",
  "Metadata": {},
  "ConnectionString": "Data Source=(localdb)\\mssqllocaldb; Initial Catalog=AdventureWorks; Integrated Security=true;"
```

If your configuration is not correct, you should have an error message, like this:



Web Server properties

Trying to reach database

Exception occured

Cannot open database "AdventureWorkds" requested by the login. The login failed. Login failed for user 'EUROPE\spertus'.

ScopeName: DefaultScope

Setup

Multi Scopes

If you need to handle multi scopes, here is the implementation with 2 scopes : "prod", "cust".

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));

    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"SqlConnection"];

    var options = new SyncOptions { };

    var tables1 = new string[] {"ProductCategory", "ProductModel", "Product" };
    var tables2 = new string[] {"Address", "Customer", "CustomerAddress"};

    services.AddSyncServer<SqlSyncChangeTrackingProvider>(connectionString,
            "prod", tables1, options);
    services.AddSyncServer<SqlSyncChangeTrackingProvider>(connectionString,
            "cust", tables2, options);

}
```

Once we have correctly configured our sync process, we can create our controller:

- Create a new controller (for example `SyncController`)
- In this newly created controller, inject your `IEnumerable<WebServerAgent>` instance.

```
[ApiController]
[Route("api/[controller]")]
public class SyncController : ControllerBase
{
    private IEnumerable<WebServerAgent> webserverAgents;
    private readonly IWebHostEnvironment env;

    // Injected thanks to Dependency Injection
    public SyncController(IEnumerable<WebServerAgent> webServerAgents,
                          IWebHostEnvironment env)
    {
        this.webServerAgents = webServerAgents;
```

(continues on next page)

(continued from previous page)

```
        this.env = env;
    }

    /// <summary>
    /// This POST handler is mandatory to handle all the sync process
    /// </summary>
    /// <returns></returns>
    [HttpPost]
    public Task Post()
    {
        var scopeName = HttpContext.GetScopeName();

        var webserverAgent = webserverAgents.FirstOrDefault(
                c => c.ScopeName == scopeName);

        await webserverAgent.HandleRequestAsync(HttpContext).ConfigureAwait(false);
    }

    /// <summary>
    /// This GET handler is optional.
    /// It allows you to see the configuration hosted on the server
    /// </summary>
    [HttpGet]
    public async Task Get()
    {
        if (env.IsDevelopment())
        {
            await this.HttpContext.WriteHelloAsync(this.webserverAgents);
        }
        else
        {
            var stringBuilder = new StringBuilder();

            stringBuilder.AppendLine("<!doctype html>");
            stringBuilder.AppendLine("<html>");
            stringBuilder.AppendLine("<title>Web Server properties</title>");
            stringBuilder.AppendLine("<body>");
            stringBuilder.AppendLine(" PRODUCTION MODE. HIDDEN INFO ");
            stringBuilder.AppendLine("</body>");
            await this.HttpContext.Response.WriteAsync(stringBuilder.ToString());
        }
    }

}
```

2.10.3 Client side

The client side is pretty similar to the starter sample, except we will have to use a *proxy orchestrator* instead of a classic *remote orchestrator*:

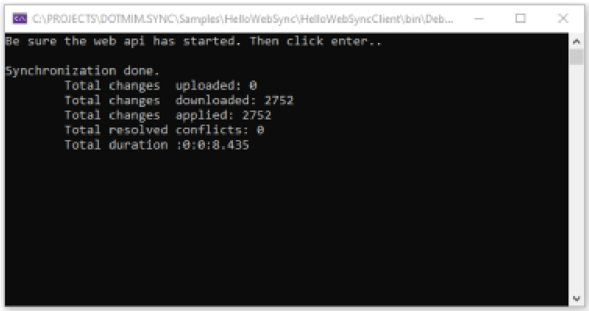
```
var serverOrchestrator = new WebRemoteOrchestrator("https://localhost:44342/api/sync
↪");

// Second provider is using plain old Sql Server provider,
// relying on triggers and tracking tables to create the sync environment
```

(continues on next page)

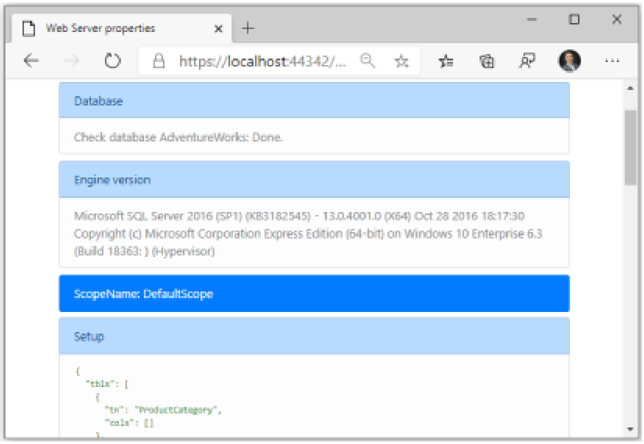
(continued from previous page)

```
var clientProvider = new SqlSyncProvider(clientConnectionString);

// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, serverOrchestrator);

do
{
    // Launch the sync process
    var s1 = await agent.SynchronizeAsync();
    // Write results
    Console.WriteLine(s1);

} while (Console.ReadKey().Key != ConsoleKey.Escape);

Console.WriteLine("End");
```

Now we can launch both application, The Web Api on one side, and the Console application on the other side. Just hit Enter and get the results from your synchronization over http.


C:\PROJECTS\DOTMIM.SYNC\Samples\HelloWebSync\HelloWebSyncClient\bin\Deb...

```
Be sure the web api has started. Then click enter..

Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 2752
        Total changes  applied: 2752
        Total resolved conflicts: 0
        Total duration :0:0:8.435
```



Database

Check database AdventureWorks: Done.

Engine version

Microsoft SQL Server 2016 (SP1) (KB3182545) - 13.0.4001.0 (X64) Oct 28 2016 18:17:30 Copyright (c) Microsoft Corporation Express Edition (64-bit) on Windows 10 Enterprise 6.3 (Build 18363:) (Hypervisor)

ScopeName: DefaultScope

Setup

```
{
  "tbls": [
    {
      "tn": "ProductCategory",
      "cols": []
```

2.11 ASP.NET Core Web Authentication

2.11.1 Overview

The `Dotmim.Sync.Web.Server` package used to expose `DMS` through **ASP.Net Core Web Api** is *just* a wrapper using the web `HttpContext` object to figure out what should be done, internally.

Hint: You will find the auth sample here : Web Authentication Sample

Just as a reminder, the **Web Server** code looks like this:

```
[ApiController]
[Route("api/[controller]")]
public class SyncController : ControllerBase
{
    private WebServerAgent webServerAgent;
    private readonly IWebHostEnvironment env;
```

(continues on next page)

(continued from previous page)

```
    // Injected thanks to Dependency Injection
    public SyncController(WebServerAgent webServerAgent, IWebHostEnvironment env)
    {
        this.webServerAgent = webServerAgent;
        this.env = env;
    }

    /// <summary>
    /// This POST handler is mandatory to handle all the sync process
    /// </summary>
    /// <returns></returns>
    [HttpPost]
    public Task Post()
        => webServerAgent.HandleRequestAsync(this.HttpContext);

    /// <summary>
    /// This GET handler is optional. It allows you to see the configuration hosted on the server
    /// </summary>
    [HttpGet]
    public async Task Get()
    {
        if (env.IsDevelopment())
        {
            await this.HttpContext.WriteHelloAsync(webServerAgent);
        }
        else
        {
            var stringBuilder = new StringBuilder();

            stringBuilder.AppendLine("<!doctype html>");
            stringBuilder.AppendLine("<html>");
            stringBuilder.AppendLine("<title>Web Server properties</title>");
            stringBuilder.AppendLine("<body>");
            stringBuilder.AppendLine(" PRODUCTION MODE. HIDDEN INFO ");
            stringBuilder.AppendLine("</body>");
            await this.HttpContext.Response.WriteAsync(stringBuilder.ToString());
        }
    }

}
```

As you can see, we are completely integrated within the **ASP.Net Core** architecture. So far, protecting our API is just like protecting any kind of ASP.NET Core Api.

If you want to rely on a strong **OAUTH2 / OpenID Connect** provider, please read:

- Microsoft : Mobile application calling a secure Web Api, using Azure AD
- AWS : Securing a Web API using AWS Cognito
- Google : OAUTH2 with Google APIS
- Identity Server : Protecting an API using Identity Server

`DMS` relies on the ASP.NET Core Web Api architecture. So far, you can secure *DMS* like you're securing any kind of exposed Web API:

- Configuring the controller

- Configuring the identity provider protocol
- Calling the controller with an authenticated client, using a bearer token

Note: More information about ASP.Net Core Authentication here : Overview of ASP.NET Core authentication

2.11.2 Server side

We are going to use a **Bearer token** validation on the server side:

- **Unsecure** but easier: Using an hard coded bearer token (Do not use this technic in production)
- **Secured** but relying on an external token provider: Using for example Azure Active Directory Authentication.

Configuration

You need to configure your Web API project to be able to secure any controller.

In your `Startup.cs`, you should add authentication services, with JWT Bearer protection.
It involves using `services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(options =>{})`

Here is a quick sample, **without** relying on any external cloud identity provider (once again, **DON'T** do that in production, it's **INSECURE** and just here for the sake of explanation)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));

    // Adding a default authentication system
    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear(); // => remove default␣
↪claims

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(options =>
            {
                ValidIssuer = "Dotmim.Sync.Bearer",
                ValidAudience = "Dotmim.Sync.Bearer",
                IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
↪"RANDOM_KEY"))
            });

    // [Required]: Get a connection string to your server data source
    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"SqlConnection"];

    // [Required] Tables involved in the sync process:
    var tables = new string[] {"ProductCategory", "ProductModel", "Product",
```

(continues on next page)

(continued from previous page)

```
        "Address", "Customer", "CustomerAddress", "SalesOrderHeader",
↪"SalesOrderDetail" };

    // [Required]: Add a SqlSyncProvider acting as the server hub.
    services.AddSyncServer<SqlSyncProvider>(connectionString, tables);
}
```

As an example, if you’re using **Azure AD** authentication, your code should be more like:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    // [Required]: Handling multiple sessions
    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));

    // Using Azure AD Authentication
    services.AddMicrosoftIdentityWebApiAuthentication(Configuration)
            .EnableTokenAcquisitionToCallDownstreamApi()
            .AddInMemoryTokenCaches();

    // [Required]: Get a connection string to your server data source
    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"SqlConnection"];

    // [Required] Tables involved in the sync process:
    var tables = new string[] {"ProductCategory", "ProductModel", "Product",
        "Address", "Customer", "CustomerAddress", "SalesOrderHeader",
↪"SalesOrderDetail" };

    // [Required]: Add a SqlSyncProvider acting as the server hub.
    services.AddSyncServer<SqlSyncProvider>(connectionString, tables);
}
```

Note: More on Code Configuration Here.

Finally, do not forget to add the **Authentication Middlewares** (and Session Middleware) as well:

```
// This method gets called by the runtime. Use this method to configure the HTTP␣
↪request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();
    app.UseSession();
```

(continues on next page)

(continued from previous page)

```
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Securing the controller

This part is the most easier one. Yo can choose to secure all the controller, using the [Authorize] attribute on the class itself, or you can use either [Authorize] / [AllowAnonymous] on each controller methods:

The simplest controller could be written like this, using the [Authorize] attribute:

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
public class SyncController : ControllerBase
{
    ...
}
```

Maybe you'll need to expose the GET method to see the server configuration. In that particular case, we can use both [Authorize] and [AllowAnonymous]:

```
[ApiController]
[Route("api/[controller]")]
public class SyncController : ControllerBase
{
    private WebServerAgent webServerAgent;

    public SyncController(WebServerAgent webServerAgent)
        => this.webServerAgent = webServerAgent;

    [HttpPost]
    [Authorize]
    public async Task Post() => webServerAgent.HandleRequestAsync(this.HttpContext);

    [HttpGet]
    [AllowAnonymous]
    public Task Get() => this.HttpContext.WriteHelloAsync(webServerAgent);

}
```

And eventually, you can even have more control, using the HttpContext instance, from within your POST handler:

```
[HttpPost]
public async Task Post()
{
    // If you are using the [Authorize] attribute you don't need to check
    // the User.Identity.IsAuthenticated value
    if (!HttpContext.User.Identity.IsAuthenticated)
    {
        this.HttpContext.Response.StatusCode = StatusCodes.Status401Unauthorized;
        return;
```

(continues on next page)

(continued from previous page)

```
    }

    // using scope and even claims, you can have more grain control on your␣
↪authenticated user
    string scope = (User.FindFirst("http://schemas.microsoft.com/identity/claims/scope
↪"))?.Value;
    string user = (User.FindFirst(ClaimTypes.NameIdentifier))?.Value;
    if (scope != "access_as_user")
    {
        this.HttpContext.Response.StatusCode = StatusCodes.Status401Unauthorized;
        return;
    }

    await orchestrator.HandleRequestAsync(this.HttpContext);
}
```

2.11.3 Client side

From you mobile / console / desktop application, you just need to send your **Bearer Token** embedded into your *HttpClient* headers.

The `WebRemoteOrchestrator` object allows you to use your own `HttpClient` instance. So far, create an instance and add your bearer token to the `DefaultRequestHeaders.Authorization` property.

```
// Getting a JWT token
// You should get a Jwt Token from an identity provider like Azure, Google, AWS or␣
↪other.
var token = GenerateJwtToken(...);

HttpClient httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer
↪", token);

// Adding the HttpClient instance to the web client orchestrator
var serverOrchestrator = new WebRemoteOrchestrator(
                "https://localhost:44342/api/sync", client:httpClient);

var clientProvider = new SqlSyncProvider(clientConnectionString);
var agent = new SyncAgent(clientProvider, serverOrchestrator);

var result = await agent.SynchronizeAsync();
```

Xamaring sample

Note: More on mobile token acquisition : Acquire token from mobile application

MSAL allows apps to acquire tokens silently and interactively.
When you call `AcquireTokenSilent()` or `AcquireTokenInteractive()`, MSAL returns an access token for the requested scopes.
The correct pattern is to make a silent request and then fall back to an interactive request.

```
string[] scopes = new string[] {"user.read"};
var app = PublicClientApplicationBuilder.Create(clientId).Build();
var accounts = await app.GetAccountsAsync();

AuthenticationResult result;
try
{
    result = await app.AcquireTokenSilent(scopes, accounts.FirstOrDefault())
                .ExecuteAsync();
}
catch(MsalUiRequiredException)
{
    result = await app.AcquireTokenInteractive(scopes)
                .ExecuteAsync();
}
```

2.12 Converters and Serializers

2.12.1 Overview

You can create your own customer serializer, changing the default JSON serializer to any kind of serializer.

As well, if you have a special type that **DMS** is unable to convert correctly, you can use your own custom converter with your own type conversion.

Note: Using serializers and converters are only useful if you have an **HTTP** architecture.

When using the **HTTP** mode, **DMS** uses two additional components:

- A **serializer**, to transforms a database row into a serialized stream. The default serializer used by **DMS** is **JSON**
- A **converter**, to converts a data type into another one. For example a `byte[]` array to `base64` string. **DMS** is not using any default converter, relying on the serializer default converter.

2.12.2 Custom Serializer

Before seeing how to create a custom serializer, we should explain the serialization mechanism:

Warning: Something really important to notice : Client dictates its own serialization mechanism.

When you run a synchronization, The `WebRemoteOrchestrator` sends a special HTTP header `dotmim-sync-serialization-format`, containing two information:

- First one is specifying the serialization format to use. The server then knows how to deserialize the messages and also uses the same serialization format when sending back messages to the client.
- Second one is specifying if the client needs batch mode or not.

Here is an example of one header sent by the client to the server, during a sync session:

```
dotmim-sync-serialization-format: {
    "f":"json",
    "s":500
}
```

The meaning of this header is:

- Client requests to sends and receives messages serializated in a **Json** format
- Client requests to have multiple files with a overall max length of **500** ko approximatively.

Once the server received this payload, contained in the header, he knows he has to serialize everything in a **JSON** format, and then will generate batch files, with approximatively **500 ko** for each payload.

Note: Batch mode is explained later in the chapter Configuration

MessagePack serializer

Hint: You will find the sample used for this chapter, here : Converter & Serializer

We can now set our own serializer.

To be able to use a new serializer, we should:

- Implement the interfaces `ISerializerFactory` and `ISerializer<T>`
- Reference this serializer on both side (client and server)

```
/// <summary>
/// Represents a factory of generic serializers.
/// This object should be able to get a serializer of each type of T
/// </summary>
public interface ISerializerFactory
{
    string Key { get; }
    ISerializer<T> GetSerializer<T>();
}

/// <summary>
/// Represents a generic serializer for a defined type of T
/// </summary>
public interface ISerializer<T>
{
    Task<T> DeserializeAsync(Stream ms);
    Task<byte[]> SerializeAsync(T obj);
}
```

Here is an example using a new serializer based on **MessagePack**, using the package MessagePack-CSharp

```
public class CustomMessagePackSerializerFactory : ISerializerFactory
{
    public string Key => "mpack";
    public ISerializer<T> GetSerializer<T>() => new CustomMessagePackSerializer<T>();
}

```

(continues on next page)

(continued from previous page)

```
public class CustomMessagePackSerializer<T> : ISerializer<T>
{
    public CustomMessagePackSerializer() =>
        MessagePackSerializer.SetDefaultResolver(ContractlessStandardResolver.
↪Instance);

    public T Deserialize(Stream ms) => MessagePackSerializer.Deserialize<T>(ms);
    public byte[] Serialize(T obj) => MessagePackSerializer.Serialize(obj);
}
```

This class should be added to both the server side and the client side.

On the server side, add the serializer to the web server serializers collection:

```
var connectionString = Configuration.GetSection("ConnectionStrings")["SqlConnection"];
var tables = new string[] {"ProductCategory", "ProductModel", "Product",
"Address", "Customer", "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail" };

// To add a converter, create an instance and add it to the special WebServerOptions
var webServerOptions = new WebServerOptions();
webServerOptions.Serializers.Add(new CustomMessagePackSerializerFactory());

// Don't forget to add this converter when calling the DI AddSyncServer() method !
services.AddSyncServer<SqlSyncChangeTrackingProvider>
            (connectionString, tables, null, webServerOptions);
```

On the client side, add this serializer as the default serializer:

```
// Create a web proxy Orchesrtrator with a custom serializer
var serverProxyOrchestrator = new WebRemoteOrchestrator("https://localhost:44342/api/
↪sync")
{
    SerializerFactory = new CustomMessagePackSerializerFactory()
};

var clientProvider = new SqlSyncProvider(clientConnectionString);
var agent = new SyncAgent(clientProvider, serverOrchestrator);
```

Now the communication between the server side and the client side will be completely made in a **MessagePack** format !

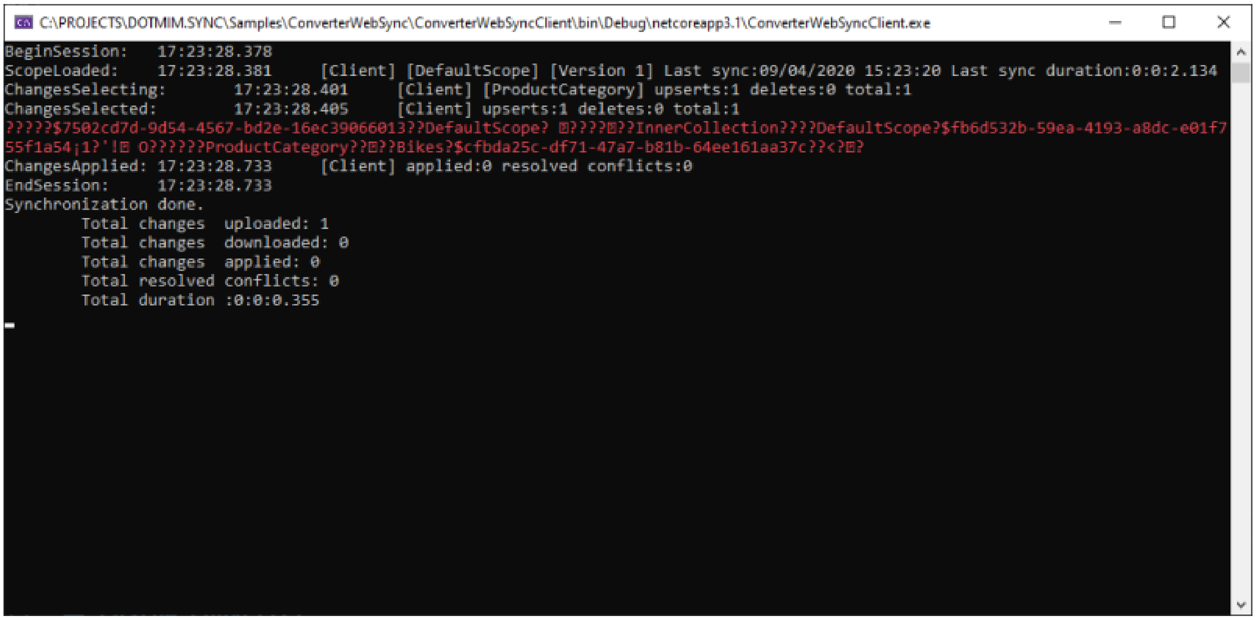
To check if everything is serialized correctly, you can use a web debugging proxy, like Fiddler or you can use an `Interceptor<T>`, available from the `WebRemoteOrchestrator` orchestrator instance:

```
//Spy the changes sent
serverProxyOrchestrator.OnSendingChanges(args =>
{
    using (var ms = new MemoryStream(args.Content))
    {
        using (var reader = new StreamReader(ms))
        {
            var text = reader.ReadToEnd();
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(text);
            Console.ResetColor();
        }
    }
```

(continues on next page)

(continued from previous page)

```
});
```



C:\PROJECTS\DOTMIM.SYNC\Samples\ConverterWebSync\ConverterWebSyncClient\bin\Debug\netcoreapp3.1\ConverterWebSyncClient.exe

```
BeginSession:    17:23:28.378
ScopeLoaded:     17:23:28.381     [Client] [DefaultScope] [Version 1] Last sync:09/04/2020 15:23:20 Last sync duration:0:0:2.134
ChangesSelecting:       17:23:28.401     [Client] [ProductCategory] upserts:1 deletes:0 total:1
ChangesSelected:        17:23:28.405     [Client] upserts:1 deletes:0 total:1
?????$7502cd7d-9d54-4567-bd2e-16ec39066013??DefaultScope? ⊡????⊡??InnerCollection????DefaultScope?$fb6d532b-59ea-4193-a8dc-e01f7
55f1a54¡1?'!⊡ O??????ProductCategory??⊡??Bikes?$cfbda25c-df71-47a7-b81b-64ee161aa37c??<?⊡?
ChangesApplied: 17:23:28.733     [Client] applied:0 resolved conflicts:0
EndSession:     17:23:28.733
Synchronization done.
        Total changes  uploaded: 1
        Total changes  downloaded: 0
        Total changes  applied: 0
        Total resolved conflicts: 0
        Total duration :0:0:0.355
```

2.12.3 Custom converter

DMS relies on the serializer's converter to convert each value from each row.

But you can create and use your own converter, that will be called on each row, before and after the serialization process.

Like the `ISerializerFactory`, you can create your own `IConverter`:

- This converter should be available both on the client and the server.
- The server should registers all converters used by any client
- The client registers its own converter.

```
public interface IConverter
{

    /// <summary>
    /// get the unique key for this converter
    /// </summary>
    string Key { get; }

    /// <summary>
    /// Convert a row before being serialized
    /// </summary>
    void BeforeSerialize(SyncRow row);

    /// <summary>
    /// Convert a row afeter being deserialized
    /// </summary>
    void AfterDeserialized(SyncRow row);
}
```

Example of a simple *IConverter*:

```
public class CustomConverter : IConverter
{
    public string Key => "cuscom";

    public void BeforeSerialize(SyncRow row)
    {
        // Each row belongs to a Table with its own Schema
        // Easy to filter if needed
        if (row.Table.TableName != "Product")
            return;

        // Encode a specific column, named "ThumbNailPhoto"
        if (row["ThumbNailPhoto"] != null)
            row["ThumbNailPhoto"] = Convert.ToBase64String((byte[])row["ThumbNailPhoto
↪"]);

        // Convert all DateTime columns to ticks
        foreach (var col in row.Table.Columns.Where(c => c.GetDataType() ==␣
↪typeof(DateTime)))
        {
            if (row[col.ColumnName] != null)
                row[col.ColumnName] = ((DateTime)row[col.ColumnName]).Ticks;
        }
    }

    public void AfterDeserialized(SyncRow row)
    {
        // Only convert for table Product
        if (row.Table.TableName != "Product")
            return;

        // Decode photo
        row["ThumbNailPhoto"] = Convert.FromBase64String((string)row["ThumbNailPhoto
↪"]);

        // Convert all DateTime back from ticks
        foreach (var col in row.Table.Columns.Where(c => c.GetDataType() ==␣
↪typeof(DateTime)))
        {
            if (row[col.ColumnName] != null)
                row[col.ColumnName] = new DateTime(Convert.ToInt64(row[col.
↪ColumnName]));
        }
    }
}
```

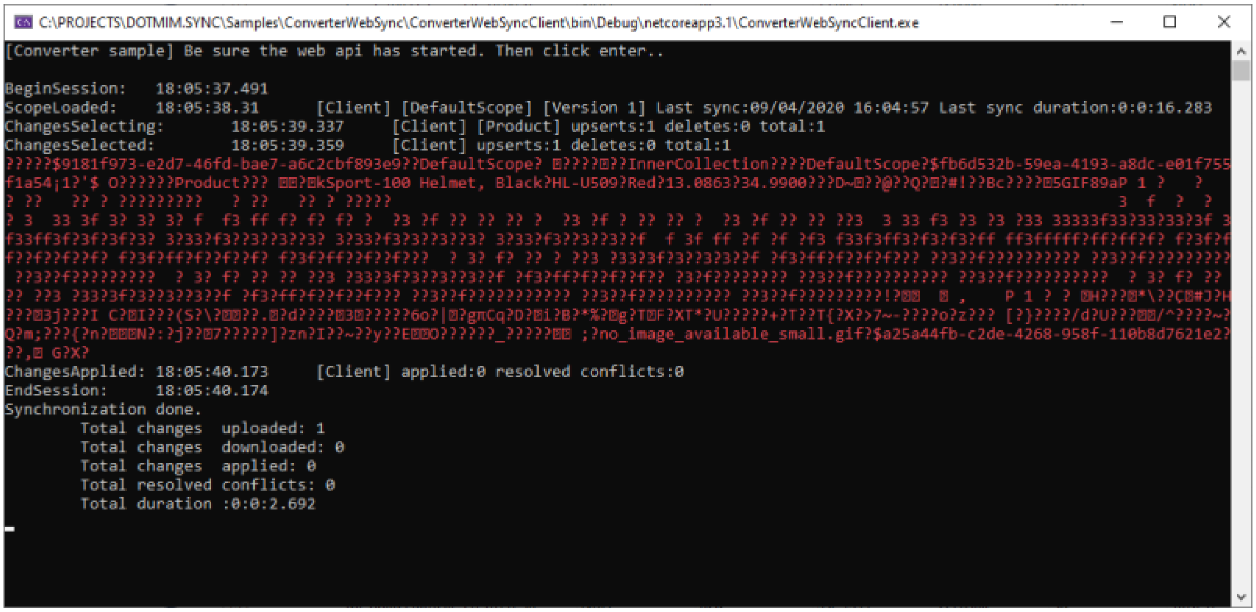
On client side, register this converter from your `WebRemoteOrchestrator`:

```
// Create the web proxy client provider with specific options
var proxyClientProvider = new WebRemoteOrchestrator
{
    SerializerFactory = new CustomMessagePackSerializerFactory(),
    Converter = new CustomConverter()
};
```

On server side, add this converter to the list of available converters:

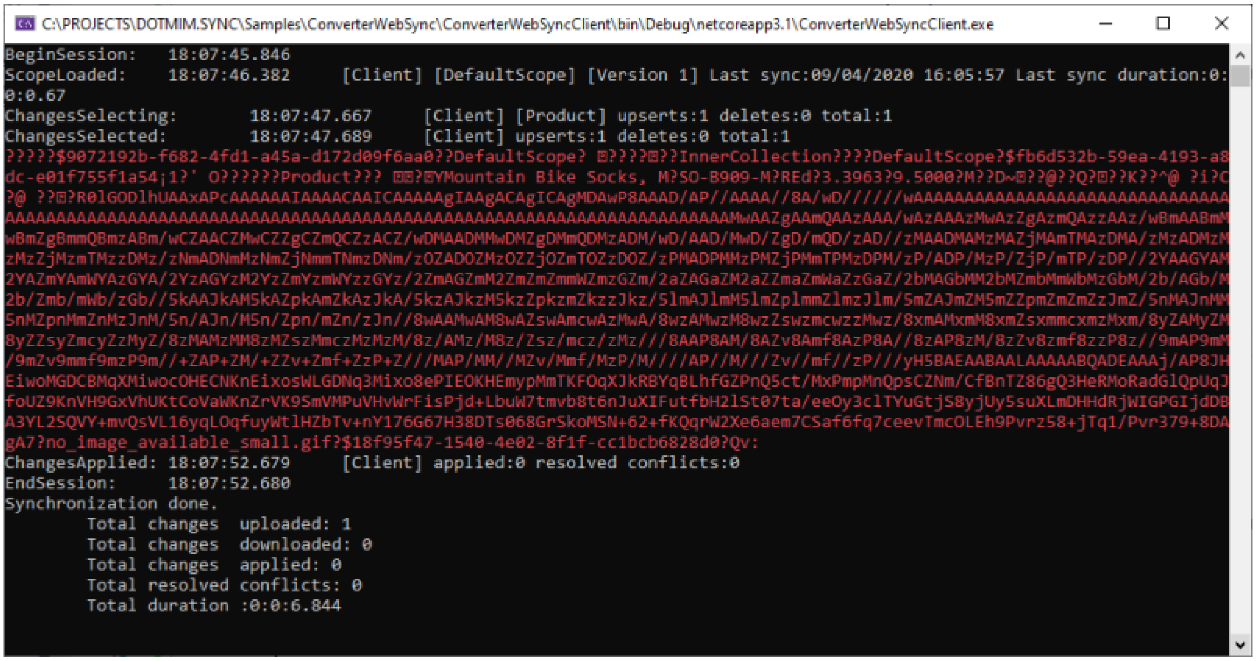
```
var webServerOptions = new WebServerOptions
{
...
};
webServerOptions.Serializers.Add(new CustomMessagePackSerializerFactory());
webServerOptions.Converters.Add(new CustomConverter());
```

Without Converter:


C:\PROJECTS\DOTMIM.SYNC\Samples\ConverterWebSync\ConverterWebSyncClient\bin\Debug\netcoreapp3.1\ConverterWebSyncClient.exe

```
[Converter sample] Be sure the web api has started. Then click enter..

BeginSession:   18:05:37.491
ScopeLoaded:    18:05:38.31      [Client] [DefaultScope] [Version 1] Last sync:09/04/2020 16:04:57 Last sync duration:0:0:16.283
ChangesSelecting:       18:05:39.337     [Client] [Product] upserts:1 deletes:0 total:1
ChangesSelected:        18:05:39.359     [Client] upserts:1 deletes:0 total:1
?????$9181f973-e2d7-46fd-bae7-a6c2cbf893e9??DefaultScope? □????□??InnerCollection????DefaultScope?$fb6d532b-59ea-4193-a8dc-e01f755
f1a54¡1?'$ O??????Product??? □□?□kSport-100 Helmet, Black?HL-U509?Red?13.0863?34.9900???D~□??@??Q?□?#!??Bc????□5GIF89aP 1 ?   ?
? ??   ?? ? ?????????   ? ??   ?? ? ?????                                                                          3  f  ?  ?
? 3  33 3f 3? 3? 3? f  f3 ff f? f? f? ?  ?3 ?f ?? ?? ?? ?  ?3 ?f ? ?? ?? ?  ?3 ?f ?? ?? ??3  3 33 f3 ?3 ?3 ?33 33333f33?33?33?3f 3
f33ff3f?3f?3f?3? 3?33?f3??3??3??3? 3?33?f3?3??3??3? 3?33?f3??3??3??f  f 3f ff ?f ?f ?f3 f33f3ff3?f3?f3?ff ff3fffff?ff?ff?f? f?3f?f
f??f??f??f? f?3f?ff?f??f??f? f?3f?ff??f??f???  ? 3? f? ?? ? ??3 ?33?3f?3??3?3??f ?f3?ff?f??f?f??? ??3??f?????????? ??3??f?????????
 ??3??f????????  ? 3? f? ?? ?? ??3 ?33?3f?3??3??3??f ?f3?ff?f??f??f?? ??3?f????????? ??3??f?????????? ??3??f??????????  ? 3? f? ??
?? ??3 ?33?3f?3??3??3??f ?f3?ff?f??f??f??? ??3??f??????????? ??3??f?????????? ??3??f?????????!?□□  □ ,    P 1 ? ? □H???□*\??Ç□#J?H
???□3j???I C?□I???(S?\?□□??.□?d????□3□?????6o?|□?gπCq?D?□i?B?*%?□g?T□F?XT*?U?????+?T??T{?X?>7~-????o?z??? [?}????/d?U???□□/^????~?
Q?m;???{?n?□□□N?:?j??□7?????]?zn?I??~??y??E□□O??????_?????□□ ;?no_image_available_small.gif?$a25a44fb-c2de-4268-958f-110b8d7621e2?
??,□ G?X?
ChangesApplied: 18:05:40.173     [Client] applied:0 resolved conflicts:0
EndSession:     18:05:40.174
Synchronization done.
        Total changes  uploaded: 1
        Total changes  downloaded: 0
        Total changes  applied: 0
        Total resolved conflicts: 0
        Total duration :0:0:2.692

_
```

With Converter:


C:\PROJECTS\DOTMIM.SYNC\Samples\ConverterWebSync\ConverterWebSyncClient\bin\Debug\netcoreapp3.1\ConverterWebSyncClient.exe

```
BeginSession:   18:07:45.846
ScopeLoaded:    18:07:46.382     [Client] [DefaultScope] [Version 1] Last sync:09/04/2020 16:05:57 Last sync duration:0:
0:0.67
ChangesSelecting:       18:07:47.667     [Client] [Product] upserts:1 deletes:0 total:1
ChangesSelected:        18:07:47.689     [Client] upserts:1 deletes:0 total:1
?????$9072192b-f682-4fd1-a45a-d172d09f6aa0??DefaultScope? □????□??InnerCollection????DefaultScope?$fb6d532b-59ea-4193-a8
dc-e01f755f1a54¡1?' O??????Product??? □□?□YMountain Bike Socks, M?SO-B909-M?REd?3.3963?9.5000?M??D~□??@??Q?□??K??^@ ?i?C
?@ ??□?R0lGODlhUAAxAPcAAAAAAIAAAACAAICAAAAAgIAAgACAgICAgMDAwP8AAAD/AP//AAAA//8A/wD//////wAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMwAAZgAAmQAAzAAA/wAzAAAzMwAzZgAzmQAzzAAz/wBmAABmM
wBmZgBmmQBmzABm/wCZAACZMwCZZgCZmQCZzACZ/wDMAADMMwDMZgDMmQDMzADM/wD/AAD/MwD/ZgD/mQD/zAD//zMAADMAMzMAZjMAmTMAzDMA/zMzADMzM
zMzZjMzmTMzzDMz/zNmADNmMzNmZjNmmTNmzDNm/zOZADOZMzOZZjOZmTOZzDOZ/zPMADPMMzPMZjPMmTPMzDPM/zP/ADP/MzP/ZjP/mTP/zDP//2YAAGYAM
2YAZmYAmWYAzGYA/2YzAGYzM2YzZmYzmWYzzGYz/2ZmAGZmM2ZmZmZmmWZmzGZm/2aZAGaZM2aZZmaZmWaZzGaZ/2bMAGbMM2bMZmbMmWbMzGbM/2b/AGb/M
2b/Zmb/mWb/zGb//5kAAJkAM5kAZpkAmZkAzJkA/5kzAJkzM5kzZpkzmZkzzJkz/5lmAJlmM5lmZplmmZlmzJlm/5mZAJmZM5mZZpmZmZmZzJmZ/5nMAJnMM
5nMZpnMmZnMzJnM/5n/AJn/M5n/Zpn/mZn/zJn//8wAAMwAM8wAZswAmcwAzMwA/8wzAMwzM8wzZswzmcwzzMwz/8xmAMxmM8xmZsxmmcxmzMxm/8yZAMyZM
8yZZsyZmcyZzMyZ/8zMAMzMM8zMZszMmczMzMzM/8z/AMz/M8z/Zsz/mcz/zMz///8AAP8AM/8AZv8Amf8AzP8A//8zAP8zM/8zZv8zmf8zzP8z//9mAP9mM
/9mZv9mmf9mzP9m//+ZAP+ZM/+ZZv+Zmf+ZzP+Z///MAP/MM//MZv/Mmf/MzP/M////AP//M///Zv//mf//zP///yH5BAEAABAALAAAAABQADEAAAj/AP8JH
EiwoMGDCBMqXMiwocOHECNKnEixosWLGDNq3Mixo8ePIEOKHEmypMmTKFOqXJkRBYqBLhfGZPnQ5ct/MxPmpMnQpsCZNm/CfBnTZ86gQ3HeRMoRadGlQpUqJ
foUZ9KnVH9GxVhUKtCoVaWKnZrVK9SmVMPuVHvWrFisPjd+LbuW7tmvb8t6nJuXIFutfbH2lSt07ta/eeOy3clTYuGtjS8yjUy5suXLmDHHdRjWIGPGIjdDB
A3YL2SQVY+mvQsVL16yqLOqfuyWtlHZbTv+nY176G67H38DTs068GrSkoMSN+62+fKQqrW2Xe6aem7CSaf6fq7ceevTmcOLEh9Pvrz58+jTq1/Pvr379+8DA
gA7?no_image_available_small.gif?$18f95f47-1540-4e02-8f1f-cc1bcb6828d0?Qv:
ChangesApplied: 18:07:52.679     [Client] applied:0 resolved conflicts:0
EndSession:     18:07:52.680
Synchronization done.
        Total changes  uploaded: 1
        Total changes  downloaded: 0
        Total changes  applied: 0
        Total resolved conflicts: 0
        Total duration :0:0:6.844
```

2.13 Increasing timeout

If you're not working on **TCP** but more likely on **HTTP** using a web api to expose your sync process, you will probably have to face some issues with timeout.

Note: Before increasing timeout, be sure you have already setup a snapshot for all your new clients.

By default, `Timeout` is fixed to 2 minutes.

To increase the overall timeout, you will have to work on both side:

- Your web server api project.
- Your client application.

2.13.1 Server side

There is no way to increase the `Timeout` period on your web api using code, with **.Net Core**.

The only solution is to provide a `web.config`, that you add manually to your project.

Note: More information here : increase-the-timeout-of-asp-net-core-application

Here is a `web.config` example where `requestTimeout` is fixed to **20** minutes:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*"
           modules="AspNetCoreModule" resourceType="Unspecified"/>
    </handlers>
    <aspNetCore requestTimeout="00:20:00"  processPath="%LAUNCHER_PATH%"
                arguments="%LAUNCHER_ARGS%" stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="false"/>
  </system.webServer>
</configuration>
```

2.13.2 Client side

On the client side, the web orchestrator `WebRemoteOrchestrator` instance uses its own `HttpClient` instance unless you specify your own `HttpClient` instance.

So far, to increase the timeout, you can either:

- Provide your own `HttpClient` instance with the `Timeout` property correctly set:

```
var handler = new HttpClientHandler { AutomaticDecompression = DecompressionMethods.
↪GZip };
var client = new HttpClient(handler) { Timeout = TimeSpan.FromMinutes(20) };
var clientProvider = new WebRemoteOrchestrator("http://my.syncapi.com:88/Sync", null,␣
↪null, client);
```

- Increase the existing `HttpClient` instance, created by `WebRemoteOrchestrator`:

```
var clientProvider = new WebRemoteOrchestrator("http://my.syncapi.com:88/Sync");
clientProvider.HttpClient.Timeout = TimeSpan.FromMinutes(20);
```

2.14 Snapshot

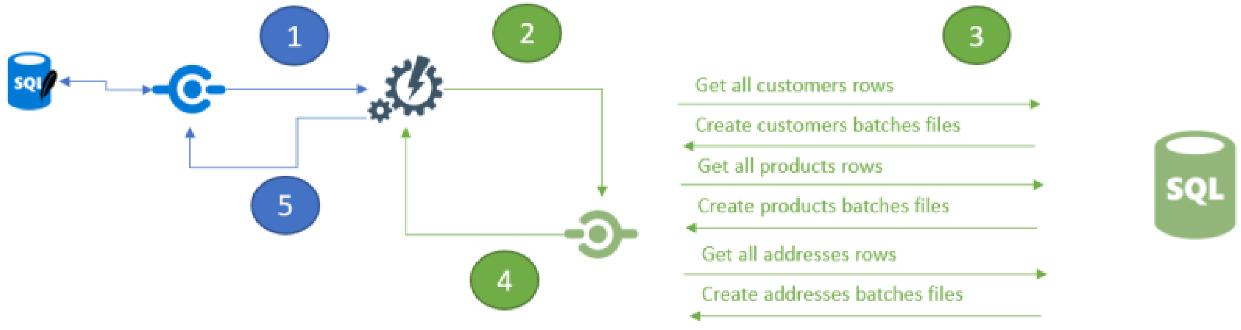
2.14.1 Overview

Sometimes, client initialization can be a problem due to the time needed for the server to generate the first batches.

The idea is to reduce this time for initialization of new clients.

Wihtout snapshot, we could have some troubles due to the heavy work from the server side, when initializing a new client:

1) A new **Client** launches a synchronization. Its local database is empty and should be synced with all rows from all tables part of the sync configuration.
2) **Server** orchestrator gets the request, initializes the medata stores, sends back the schema if needed, and then launches the sync process internally.
3) **Server** prepares batches files, based on all tables involved in the sync (using `_Initialize` stored procedures).
4) **Server** streams back the files to the **client** orchestrator.
5) **Client** orchestrator applies the rows to the local database using the client provider.


1
2
3
5
4
SQL
Get all customers rows
Create customers batches files
Get all products rows
Create products batches files
Get all addresses rows
Create addresses batches files
SQL

Depending on numbers of tables, and rows, the step **3** could take a lot of times.
During this generation time the client has to wait, with no response from server, until the batches are ready.

Warning: In a **TCP** mode, it will work since the client will wait until a response from the server. But in an **HTTP** mode you can eventually have a **timeout** exception raised. . .

Hint: In **HTTP** mode, you can increase the timeout duration , but it's not ideal. . .

The **snapshot** feature comes in here to resolve this issue.

The idea is quite simple : Creating a snapshot of the server database on time **TS**, available for all new clients.

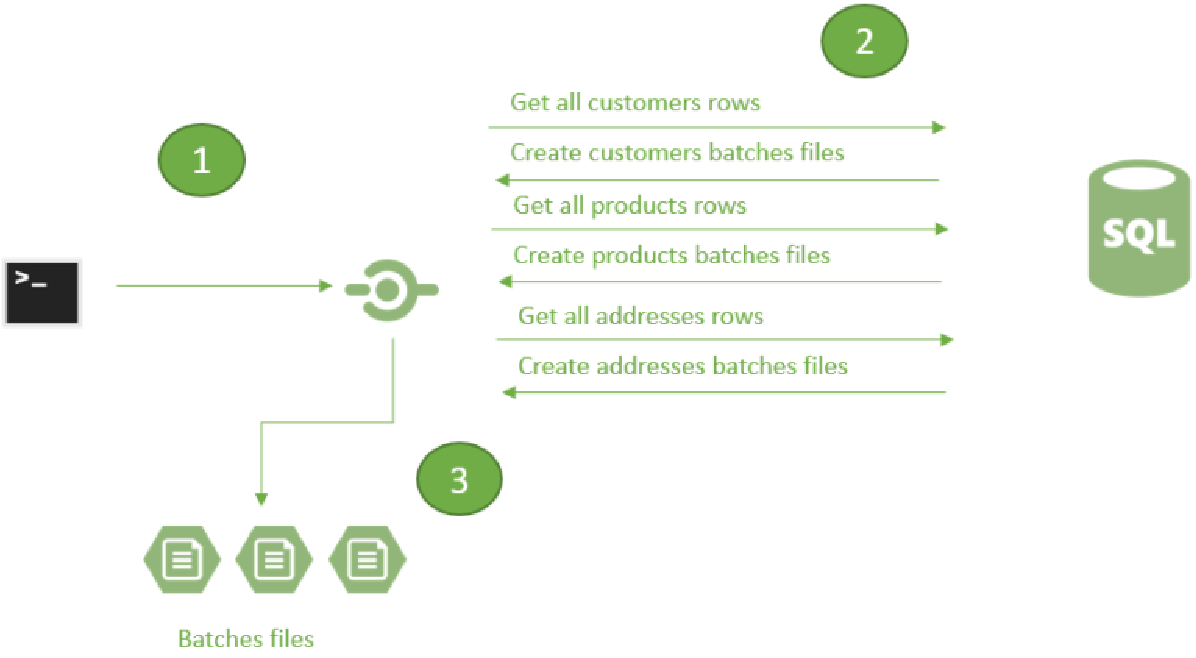
A common scenario would be:

- Create a snapshot every 2 weeks on the server side, to get the most relevant and up to date data.
- Every new client will download and apply this snapshot on initialization.
- This new client will then synchronize all new datas in between the snapshot (so TS) and T.

Here is the steps to create a server snapshot and the configuration from both server and client side:

2.14.2 Server side

Create a new method, that will generate a *snapshot* at a current time *T* with all rows / tables, available for all new clients:


2
Get all customers rows
Create customers batches files
Get all products rows
Create products batches files
Get all addresses rows
Create addresses batches files
1
SQL
3
Batches files

Note: Creates a new project, a console application for example, to create a snapshot.

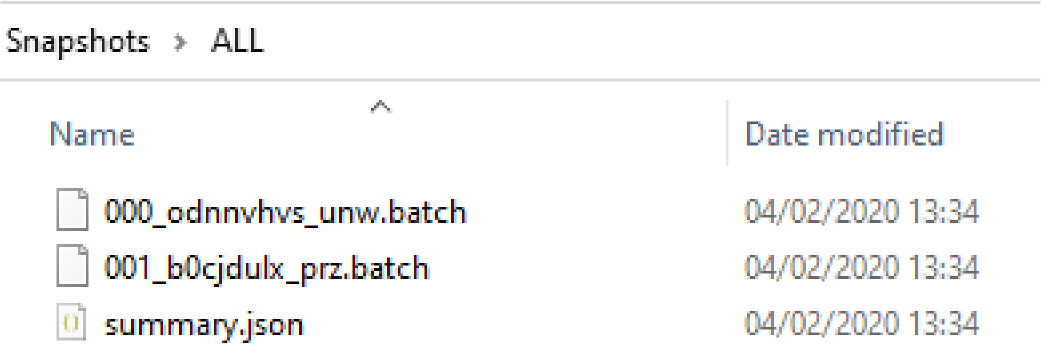
```
var serverProvider = new SqlSyncProvider("Data Source= ...");

// new setup with all tables involved
var setup = new SyncSetup(allTables);

// snapshot directory
var snapshotDirectoryName = "snapshots";
var snapshotDirctory = Path.Combine(Environment.CurrentDirectory,␣
↪snapshotDirctoryName);

var options = new SyncOptions
{
    SnapshotsDirectory = snapshotDirctory,
    BatchSize = 3000
};

// Create a remote orchestrator
var remoteOrchestrator = new RemoteOrchestrator(serverProvider, options);
```

(continues on next page)

(continued from previous page)

```
// Create a snapshot
await remoteOrchestrator.CreateSnapshotAsync(setup);
```

Once created, the folder looks like this:



Snapshots > ALL

Name	Date modified
000_odnnvhvs_unw.batch	04/02/2020 13:34
001_b0cjdulx_prz.batch	04/02/2020 13:34
summary.json	04/02/2020 13:34

- Some **.batch* files containing all the rows, for all the sync tables.
- A *summary.json* contains all the mandatory information regarding this snapshot

```
{
    "dirname": "ALL",
    "dir": "C:\\Users\\spertus.EUROPE\\Snapshots",
    "ts": 2001,
    "parts": [
        {
            "file": "000_fnwkoou5_tdj.batch",
            "index": 0,
            "last": false,
            "tables": [
                {
                    "n": "ProductCategory"
                },
                {
                    "n": "ProductModel"
                },
                {
                    "n": "Product"
                }
            ]
        },
        {
            "file": "001_02zy0swq_nce.batch",
            "index": 1,
            "last": true,
            "tables": [
                {
                    "n": "Product"
                },
                {
                    "n": "Address"
                },
                {
                    "n": "Customer"
```

(continues on next page)

(continued from previous page)

```
                },
                {
                    "n": "CustomerAddress"
                },
                {
                    "n": "SalesOrderHeader"
                },
                {
                    "n": "SalesOrderDetail"
                }
            ]
        }
    ]
}
```

We have here, the server `timestamp` when the snapshot was generated, all the files, ordered, with table contained in each file.

Filtered clients

For filtered client, the snapshot will be a little bit different, since it will not contains all the data. More, each filtered client will have its own snapshot, based on its filter parameters values !

To generate a filtered snapshot, just add the `SyncParameters` values to the new `SyncContext` instance argument:

```
// Setup with a filter on CustomerId, on table Customer
var setup = new SyncSetup(allTables);
setup.Filters.Add("Customer", "CustomerId");

// Create a filtered snapshot
SyncParameters parameters = new()
   {
      new("CustomerId", "1001"),
   };

await Server.RemoteOrchestrator.CreateSnapshotAsync(setup, parameters);
```

Activate the snapshot option for all new clients

To activate this snapshot, the server should know where each snapshot is located.

The *SyncOptions* has a new property called *SnapshotsDirectory*:

```
// Options used for client and server when used in a direct TCP mode:
var options = new SyncOptions {
    SnapshotsDirectory = Path.Combine(
                    Environment.GetFolderPath(Environment.SpecialFolder.UserProfile),
                    "Snapshots")
    };
```

HTTP mode with ASP.Net Core Web API

The `ASP.NET Core` web api looks like this, now:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));


    // Get a connection string for your server data source
    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"DefaultConnection"];

    // Set the web server Options
    var options = new SyncOptions()
    {
        SnapshotsDirectory = Path.Combine(
                Environment.GetFolderPath(Environment.SpecialFolder.UserProfile),
                "Snapshots")
    };


    // Create the setup used for your sync process
    var tables = new string[] {"ProductCategory",
                    "ProductDescription", "ProductModel",
                    "Product", "ProductModelProductDescription",
                    "Address", "Customer", "CustomerAddress",
                    "SalesOrderHeader", "SalesOrderDetail" };

    var setup = new SyncSetup(tables);

    // add a SqlSyncProvider acting as the server hub
    services.AddSyncServer<SqlSyncProvider>(connectionString, setup, options);
}

// This method gets called by the runtime. Use this method to configure the HTTP␣
↪request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
        app.UseDeveloperExceptionPage();

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseSession();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

2.14.3 Client side

On the client side, you don't have anything to do, just a normal new sync processus:

```
var s = await agent.SynchronizeAsync(progress);
```

Here is an output of new client coming with a new client database :

```
BeginSession     14:00:22.651
ScopeLoading     14:00:22.790    Id:b3d33500-ee06-427a-bccc-7518a9dfec93 LastSync:␣
↪LastSyncDuration:0
TableSchemaApplied       14:00:26.95     TableName: ProductCategory Provision:All
TableSchemaApplied       14:00:26.234    TableName: ProductModel Provision:All
TableSchemaApplied       14:00:26.415    TableName: Product Provision:All
TableSchemaApplied       14:00:26.466    TableName: Address Provision:All
TableSchemaApplied       14:00:26.578    TableName: Customer Provision:All
TableSchemaApplied       14:00:26.629    TableName: CustomerAddress Provision:All
TableSchemaApplied       14:00:26.777    TableName: SalesOrderHeader Provision:All
TableSchemaApplied       14:00:26.830    TableName: SalesOrderDetail Provision:All
SchemaApplied    14:00:26.831    Tables count:8 Provision:All
TableChangesApplied      14:00:28.101    ProductCategory State:Modified Applied:41␣
↪Failed:0
TableChangesApplied      14:00:28.252    ProductModel State:Modified Applied:128␣
↪Failed:0
TableChangesApplied      14:00:28.449    Product State:Modified Applied:201 Failed:0
TableChangesApplied      14:00:28.535    Product State:Modified Applied:295 Failed:0
TableChangesApplied      14:00:28.686    Address State:Modified Applied:450 Failed:0
TableChangesApplied      14:00:28.874    Customer State:Modified Applied:847 Failed:0
TableChangesApplied      14:00:29.28     CustomerAddress State:Modified Applied:417␣
↪Failed:0
TableChangesApplied      14:00:29.165    SalesOrderHeader State:Modified Applied:32␣
↪Failed:0
TableChangesApplied      14:00:29.383    SalesOrderDetail State:Modified Applied:542␣
↪Failed:0
DatabaseChangesApplied   14:00:29.385    Changes applied on database Client: Applied:␣
↪2752 Failed: 0
ScopeSaved       14:00:29.455    Id:b3d33500-ee06-427a-bccc-7518a9dfec93 LastSync:04/
↪02/2020 13:00:29 LastSyncDuration:68091840
EndSession       14:00:29.457
BeginSession     14:00:29.460
ScopeLoading     14:00:29.466    Id:b3d33500-ee06-427a-bccc-7518a9dfec93 LastSync:04/
↪02/2020 13:00:29 LastSyncDuration:68091840
TableChangesSelected     14:00:29.481    ProductCategory Upserts:0 Deletes:0␣
↪TotalChanges:0
TableChangesSelected     14:00:29.491    ProductModel Upserts:0 Deletes:0␣
↪TotalChanges:0
TableChangesSelected     14:00:29.504    Product Upserts:0 Deletes:0 TotalChanges:0
TableChangesSelected     14:00:29.514    Address Upserts:0 Deletes:0 TotalChanges:0
TableChangesSelected     14:00:29.524    Customer Upserts:0 Deletes:0 TotalChanges:0
TableChangesSelected     14:00:29.535    CustomerAddress Upserts:0 Deletes:0␣
↪TotalChanges:0
TableChangesSelected     14:00:29.544    SalesOrderHeader Upserts:0 Deletes:0␣
↪TotalChanges:0
TableChangesSelected     14:00:29.553    SalesOrderDetail Upserts:0 Deletes:0␣
↪TotalChanges:0
TableChangesApplied      14:00:29.722    ProductCategory State:Modified Applied:1␣
↪Failed:0
DatabaseChangesApplied   14:00:29.732    Changes applied on database Client: Applied:␣
↪1 Failed: 0
ScopeSaved       14:00:29.772    Id:b3d33500-ee06-427a-bccc-7518a9dfec93 LastSync:04/
↪02/2020 13:00:29 LastSyncDuration:71205855
EndSession       14:00:29.773
Synchronization done.
        Total changes downloaded: 2753
```

(continues on next page)

(continued from previous page)

```
        Total changes uploaded: 0
        Total conflicts: 0
        Total duration :0:0:7.120
```

As you can see, we have basically **2** Sync in a row.

- First one get the **schema**, and apply all the **batches** from the snapshot
- Second one get all the rows added / deleted / modified from the snapshot `TimeStamp T-1` and the last server `TimeStamp T` (in our sample just one `ProductCategory`)

2.15 Setup & Options

You can configure your synchronization model with some parameters, available through the `SyncSetup` and `SyncOptions` objects :

What's the differences between `SyncSetup` and `SyncOptions` ?

- `SyncSetup` contains all the parameters related to your schema, and shared between the server and all the clients.
  - In **Http mode**, the `SyncSetup` parameters are set by the **Server** and will be send to all **Clients**.
- `SyncOptions` contains all the parameters **not shared** between the server and all the clients.

2.15.1 SyncSetup

If we look at the `SyncSetup` object, we mainly have properties about your synced tables schema:

```
public class SyncSetup
{
    /// <summary>
    /// Gets or Sets the scope name
    /// </summary>
    public string ScopeName { get; set; }

    /// <summary>
    /// Gets or Sets all the synced tables
    /// </summary>
    public SetupTables Tables { get; set; }

    /// <summary>
    /// Specify all filters for each table
    /// </summary>
    public SetupFilters Filters { get; set; }

    /// <summary>
    /// Specify a prefix for naming stored procedure. Default is empty string
    /// </summary>
    public string StoredProceduresPrefix { get; set; }

    /// <summary>
    /// Specify a suffix for naming stored procedures. Default is empty string
    /// </summary>
    public string StoredProceduresSuffix { get; set; }
```

(continues on next page)

(continued from previous page)

```
    /// <summary>
    /// Specify a prefix for naming stored procedure. Default is empty string
    /// </summary>
    public string TriggersPrefix { get; set; }

    /// <summary>
    /// Specify a suffix for naming stored procedures. Default is empty string
    /// </summary>
    public string TriggersSuffix { get; set; }

    /// <summary>
    /// Specify a prefix for naming tracking tables. Default is empty string
    /// </summary>
    public string TrackingTablesPrefix { get; set; }

    /// <summary>
    /// Specify a suffix for naming tracking tables.
    /// </summary>
    public string TrackingTablesSuffix { get; set; }

}
```

The `SynchronizeAsync()` method creates a `SyncSetup` instance automatically when called.

For instance, these two instructions are equivalent:

```
var tables = new string[] {"ProductCategory", "ProductModel", "Product",
    "Address", "Customer", "CustomerAddress", "SalesOrderHeader", "SalesOrderDetail" }
↪;

var agent = new SyncAgent(clientProvider, serverProvider);

var r = await agent.SynchronizeAsync(tables);
```

```
var setup = new SyncSetup("ProductCategory", "ProductModel", "Product",
        "Address", "Customer", "CustomerAddress", "SalesOrderHeader",
↪"SalesOrderDetail");

var agent = new SyncAgent(clientProvider, serverProvider);

var r = await agent.SynchronizeAsync(setup);
```

The main advantage of using `SyncSetup` is you can personalize what you want from your database:

Schema

Note: The schema feature is only avaialable for SQL Server

One great feature in **SQL Server** is the schema option.

You can configure your sync tables with schema if you target the `SqlSyncProvider`.

You have two way to configure schemas:

- Directly during the tables declaration, as string.

```
var tables = new string[] { "SalesLT.ProductCategory", "SalesLT.ProductModel",
↪"SalesLT.Product",
                                "Address", "Customer", "CustomerAddress"};
```

- On each table, from the `SyncSetup` setup instance.

```
var setup = new SyncSetup ("ProductCategory", "ProductModel", "Product",
                            "Address", "Customer", "CustomerAddress");

setup.Tables["ProductCategory"].SchemaName = "SalesLt";
setup.Tables["ProductModel"].SchemaName = "SalesLt";
setup.Tables["Product"].SchemaName = "SalesLt";
```

Warning: Schemas are not replicated if you target `SqliteSyncProvider` or `MySqlSyncProvider` as client providers.

Filtering Columns

Once your `SyncSetup` instance is created (with your tables list), you can specify the columns you want to sync:

```
var setup = new SyncSetup("ProductCategory", "ProductModel", "Product",
        "Address", "Customer", "CustomerAddress", "SalesOrderHeader",
↪"SalesOrderDetail" );

// Filter columns
setup.Tables["Customer"].Columns.AddRange(new string[] {
    "CustomerID", "EmployeeID", "NameStyle", "FirstName", "LastName" });

setup.Tables["Address"].Columns.AddRange(new string[] {
    "AddressID", "AddressLine1", "City", "PostalCode" });
```

For instance, table `Customer` and `Address` won't sync all their columns, but only those specified.

Filtering Rows

From your `SyncSetup` instance, you can also specify a `SetupFilter` on each table, allowing you to filter rows.

```
setup.Filters.Add("Customer", "CustomerID");
setup.Filters.Add("CustomerAddress", "CustomerID");
setup.Filters.Add("SalesOrderHeader", "CustomerID", "SalesLT");
```

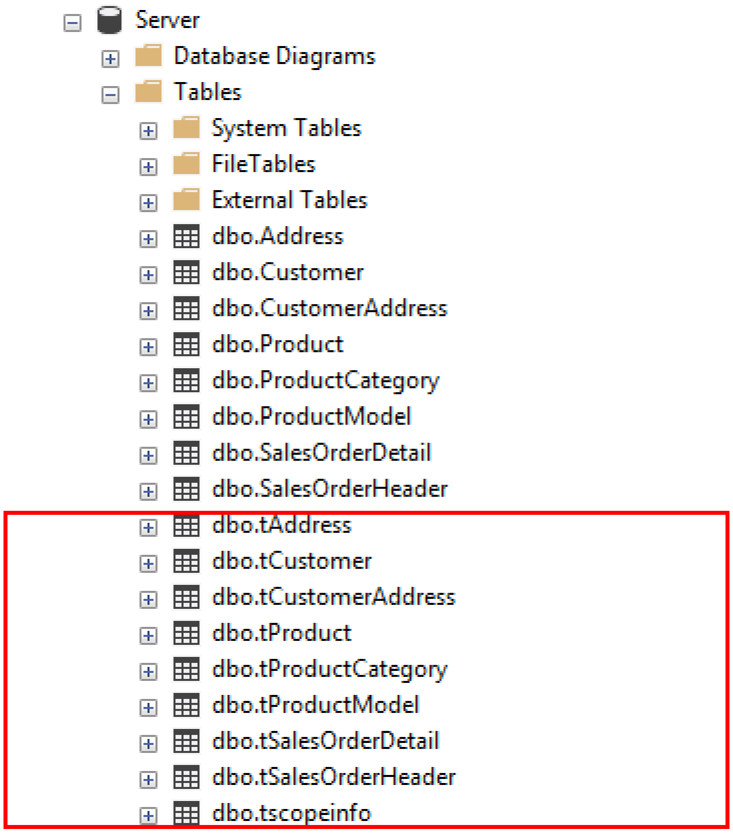
Tables `Customer`, `CustomerAddress` and `SalesLT.SalesOrderHeader` will filter their rows, based on the `CustomerID` column value.

Note: Filtering rows is a quite complex thing. A full chapter is dedicated to rows filtering: Filters

Database configuration

You can personalize how are created the **tracking tables**, **triggers** and **stored procedures** tables in your database:

```
var setup = new SyncSetup(tables)
{
    StoredProceduresPrefix = "s",
    StoredProceduresSuffix = "",
    TrackingTablesPrefix = "t",
    TrackingTablesSuffix = "",
    TriggersPrefix = "",
    TriggersSuffix = "t"
};
```



```
Server
    Database Diagrams
    Tables
        System Tables
        FileTables
        External Tables
        dbo.Address
        dbo.Customer
        dbo.CustomerAddress
        dbo.Product
        dbo.ProductCategory
        dbo.ProductModel
        dbo.SalesOrderDetail
        dbo.SalesOrderHeader
        dbo.tAddress
        dbo.tCustomer
        dbo.tCustomerAddress
        dbo.tProduct
        dbo.tProductCategory
        dbo.tProductModel
        dbo.tSalesOrderDetail
        dbo.tSalesOrderHeader
        dbo.tscopeinfo
```

HTTP mode

In a more realistic scenario, you will probably have a web proxy in front of your **Server** database.
You must provide your configuration values on the server side, not on the client side, since the server side will always override the values from the client.

As we saw in the Web chapter, we are using the **ASP.NET Dependency injection** system to create our **Server** remote provider.
It’s the best place to setup your sync configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
```

(continues on next page)

(continued from previous page)

```
    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));

    // Get a connection string for your server data source
    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"DefaultConnection"];


    // Create the setup used for your sync process
    var tables = new string[] {"ProductCategory",
                    "ProductDescription", "ProductModel",
                    "Product", "ProductModelProductDescription",
                    "Address", "Customer", "CustomerAddress",
                    "SalesOrderHeader", "SalesOrderDetail" };

    var setup = new SyncSetup(tables)
    {
        StoredProceduresPrefix = "s",
        StoredProceduresSuffix = "",
        TrackingTablesPrefix = "t",
        TrackingTablesSuffix = "",
        TriggersPrefix = "",
        TriggersSuffix = "t"
    };

    // add a SqlSyncProvider acting as the server hub
    services.AddSyncServer<SqlSyncProvider>(connectionString, setup);
}

// This method gets called by the runtime. Use this method to configure the HTTP␣
↪request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
        app.UseDeveloperExceptionPage();

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseSession();
    app.UseEndpoints(endpoints => endpoints.MapControllers());
}
```

Warning: The prefix and suffix properties, are not shared betweeen server and client.

2.15.2 SyncOptions

On the other side, `SyncOptions` can be customized on server and on client, with their own different values. For instance, we can have a different value for the the `BatchDirectory` (representing the tmp directory when batch is enabled) on server and on client.

```
/// <summary>
/// This class determines all the options you can set on Client & Server,
/// that could potentially be different
/// </summary>
public class SyncOptions
{
    /// <summary>
    /// Gets or Sets the directory used for batch mode.
    /// Default value is [User Temp Path]/[DotmimSync]
    /// </summary>
    public string BatchDirectory { get; set; }

    /// <summary>
    /// Gets or Sets the directory where snapshots are stored.
    /// This value could be overwritten by server is used in an http mode
    /// </summary>
    public string SnapshotsDirectory { get; set; }

    /// <summary>
    /// Gets or Sets the size used (approximatively in kb, depending on the␣
↪serializer)
    /// for each batch file, in batch mode.
    /// Default is 0 (no batch mode)
    /// </summary>
    public int BatchSize { get; set; }

    /// <summary>
    /// Gets or Sets the log level for sync operations. Default value is false.
    /// </summary>
    public bool UseVerboseErrors { get; set; }

    /// <summary>
    /// Gets or Sets if we should use the bulk operations. Default is true.
    /// If provider does not support bulk operations, this option is overrided to␣
↪false.
    /// </summary>
    public bool UseBulkOperations { get; set; } = true;

    /// <summary>
    /// Gets or Sets if we should clean tracking table metadatas.
    /// </summary>
    public bool CleanMetadatas { get; set; } = true;

    /// <summary>
    /// Gets or Sets if we should cleaning tmp dir files after sync.
    /// </summary>
    public bool CleanFolder { get; set; } = true;

    /// <summary>
    /// Gets or Sets if we should disable constraints before making apply changes
    /// Default value is true
    /// </summary>
    public bool DisableConstraintsOnApplyChanges { get; set; } = true;

    /// <summary>
    /// Gets or Sets the scope_info table name. Default is scope_info
    /// On the server side, server scope table is prefixed with _server
```

(continues on next page)

(continued from previous page)

```
    /// and history table with _history
    /// </summary>
    public string ScopeInfoTableName { get; set; }

    /// <summary>
    /// Gets or Sets the default conflict resolution policy. This value could␣
↪potentially
    /// be ovewritten and replaced by the server
    /// </summary>
    public ConflictResolutionPolicy ConflictResolutionPolicy { get; set; }

    /// <summary>
    /// Gets or Sets the default logger used for logging purpose
    /// </summary>
    public ILogger Logger { get; set; }
}
```

Note: If nothing is supplied when creating a new `SyncAgent` instance, a default `SyncOptions` is created with default values.

`SyncOptions` has some useful methods, you can rely on:

```
/// <summary>
/// Get the default Batch directory full path ([User Temp Path]/[DotmimSync])
/// </summary>
public static string GetDefaultUserBatchDiretory()

/// <summary>
/// Get the default user tmp folder
/// </summary>
public static string GetDefaultUserTempPath()

/// <summary>
/// Get the default sync tmp folder name (usually 'DotmimSync')
/// </summary>
public static string GetDefaultUserBatchDirectoryName()
```

Batch mode

Batch mode is an important options if you have to deal with *over sized* sync changes.

If you have a lot of changes to download from your server (or changes to upload from your client), maybe you don't want to download / upload one big change object, stored in memory.

Even more, when you're in a web environment, you don't want to make a web request with everything inside of it, which could be way too heavy !

The `BatchSize` property from the `SyncOptions` object allows you to define the maximum size of any payload:

```
var clientOptions = new SyncOptions { BatchSize = 500 };
```

Warning:

Be careful, the batch size value **is** a kb maximum size.
But The maximum size depends on compression, converters and so on. . .
Test and adjust the `BatchSize` value regarding your result and expectation.

Example

Hint: You will find the complete sample here : Batch size sample

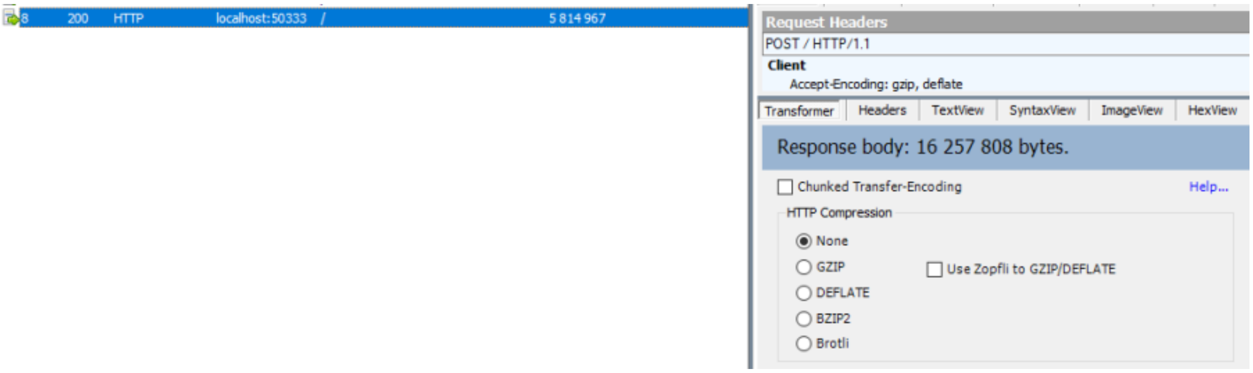
As an example, we make an insert of **100000** product category items in the server database, before making our sync:

```
Insert into ProductCategory (Name)
Select SUBSTRING(CONVERT(varchar(255), NEWID()), 0, 7)
Go 100000
```

By default, here is a sync process, where we download everything from the server, without any `BatchSize` option:

```
var agent = new SyncAgent(clientProvider, proxyClientProvider);
await agent.SynchronizeAsync(setup);
```

Here is the fiddler trace:



8 200 HTTP localhost:50333 / 5 814 967

Request Headers
POST / HTTP/1.1
Client
Accept-Encoding: gzip, deflate

Response body: 16 257 808 bytes.

☐ Chunked Transfer-Encoding

HTTP Compression
● None
○ GZIP
○ DEFLATE
○ BZIP2
○ Brotli
☐ Use Zopfli to GZIP/DEFLATE

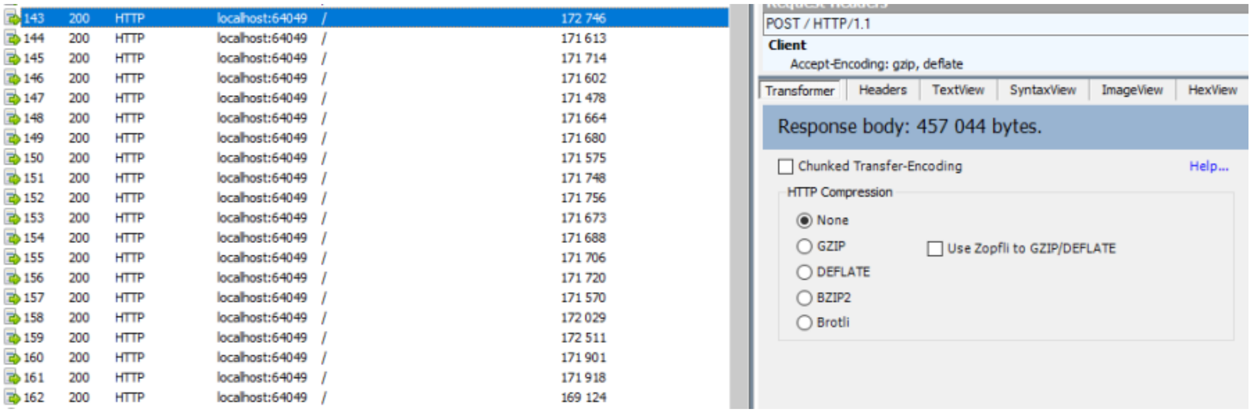
As you can see, the fiddler trace shows a http response around **16 Mb** (approximatively **6 Mb** compressed).
It could be even more, depending on the size of the selected changes from the server.

Here is the same sync, with the batch mode enabled:

```
// ----------------------------------
// Client side
// ----------------------------------
var clientOptions = new SyncOptions { BatchSize = 500 };

var agent = new SyncAgent(clientProvider, proxyClientProvider, clientOptions);
var progress = new SynchronousProgress<ProgressArgs>(pa =>
Console.WriteLine(String.Format("{0} -{1}\t {2}",
            pa.Context.SessionId, pa.Context.SyncStage, pa.Message));
var s = await agent.SynchronizeAsync(progress);
Console.WriteLine(s);
```

Hint: The client side dictates the batch size. The server is always adapting its payload, regarding the client ask.

Here is the fiddler trace:



143	200	HTTP	localhost:64049	/	172 746
144	200	HTTP	localhost:64049	/	171 613
145	200	HTTP	localhost:64049	/	171 714
146	200	HTTP	localhost:64049	/	171 602
147	200	HTTP	localhost:64049	/	171 478
148	200	HTTP	localhost:64049	/	171 664
149	200	HTTP	localhost:64049	/	171 680
150	200	HTTP	localhost:64049	/	171 575
151	200	HTTP	localhost:64049	/	171 748
152	200	HTTP	localhost:64049	/	171 756
153	200	HTTP	localhost:64049	/	171 673
154	200	HTTP	localhost:64049	/	171 688
155	200	HTTP	localhost:64049	/	171 706
156	200	HTTP	localhost:64049	/	171 720
157	200	HTTP	localhost:64049	/	171 570
158	200	HTTP	localhost:64049	/	172 029
159	200	HTTP	localhost:64049	/	172 511
160	200	HTTP	localhost:64049	/	171 901
161	200	HTTP	localhost:64049	/	171 918
162	200	HTTP	localhost:64049	/	169 124

POST / HTTP/1.1

Client

Accept-Encoding: gzip, deflate

Response body: 457 044 bytes.

□ Chunked Transfer-Encoding

HTTP Compression

● None
○ GZIP
○ DEFLATE
○ BZIP2
○ Brotli

□ Use Zopfli to GZIP/DEFLATE

And the progress of the sync process:

```
974f8be9-332d-4d6d-b881-7784b63b4bb7 - BeginSession      10:53:38.762    Session␣
↪Id:974f8be9-332d-4d6d-b881-7784b63b4bb7
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ScopeLoaded       10:53:39.385    [Client]␣
↪[DefaultScope] [Version ] Last sync: Last sync duration:0:0:0.0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - Provisioned       10:53:42.224    [Client]␣
↪tables count:8 provision:Table, TrackingTable, StoredProcedures, Triggers
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesSelected   10:53:42.243    [Client]␣
↪upserts:0 deletes:0 total:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:55.133    [Client]␣
↪[ProductCategory] Modified applied:5171 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:55.702    [Client]␣
↪[ProductCategory] Modified applied:10343 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:56.297    [Client]␣
↪[ProductCategory] Modified applied:15515 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:56.891    [Client]␣
↪[ProductCategory] Modified applied:20687 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:57.620    [Client]␣
↪[ProductCategory] Modified applied:25859 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:58.280    [Client]␣
↪[ProductCategory] Modified applied:31031 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:58.971    [Client]␣
↪[ProductCategory] Modified applied:36203 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:53:59.682    [Client]␣
↪[ProductCategory] Modified applied:41375 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:00.420    [Client]␣
↪[ProductCategory] Modified applied:46547 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:01.169    [Client]␣
↪[ProductCategory] Modified applied:51719 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:01.940    [Client]␣
↪[ProductCategory] Modified applied:56891 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:02.657    [Client]␣
↪[ProductCategory] Modified applied:62063 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:03.432    [Client]␣
↪[ProductCategory] Modified applied:67235 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:04.192    [Client]␣
↪[ProductCategory] Modified applied:72407 resolved conflicts:0
```

(continues on next page)

(continued from previous page)

```
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:05.82     [Client]␣
↪[ProductCategory] Modified applied:77579 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:05.930    [Client]␣
↪[ProductCategory] Modified applied:82751 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:06.787    [Client]␣
↪[ProductCategory] Modified applied:87923 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:07.672    [Client]␣
↪[ProductCategory] Modified applied:93095 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:08.553    [Client]␣
↪[ProductCategory] Modified applied:98267 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:08.972    [Client]␣
↪[ProductCategory] Modified applied:100041 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.113    [Client]␣
↪[ProductModel] Modified applied:128 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.183    [Client]␣
↪[Product] Modified applied:198 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.208    [Client]␣
↪[Product] Modified applied:295 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.255    [Client]␣
↪[Address] Modified applied:450 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.329    [Client]␣
↪[Customer] Modified applied:847 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.375    [Client]␣
↪[CustomerAddress] Modified applied:417 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.414    [Client]␣
↪[SalesOrderHeader] Modified applied:32 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplying   10:54:09.476    [Client]␣
↪[SalesOrderDetail] Modified applied:542 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - ChangesApplied    10:54:09.636    [Client]␣
↪applied:102752 resolved conflicts:0
974f8be9-332d-4d6d-b881-7784b63b4bb7 - EndSession        10:54:09.638    Session␣
↪Id:974f8be9-332d-4d6d-b881-7784b63b4bb7
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 102752
        Total changes  applied: 102752
        Total resolved conflicts: 0
        Total duration :0:0:30.886
```

As you can see, most of the product category items come from different batch requests.

UseBulkOperations

This option is only available when using `SQL Server` providers.

It allows you to use bulk operations from within **SQL Server** using **Table Value Parameters** as input to the stored procedures.

When using `UseBulkOperations`, each table will have new stored procedures and one table value parameter:

- Stored procedure `CustomerAddress_bulkdelete`
- Stored procedure `CustomerAddress_bulkupdate`
- Table value parameter `Customer_BulkType`

Using this option will increase your performances, so do not hesitate to use it !

CleanMetadatas

The CleanMetadatas option allows you to clean the _tracking tables from your client databases.

Once enabled, the client database will delete all metadatas from the tracking tables, after every successful sync.

Be careful, the delete method will:

- Work only if client download *something* from server. If there is no changes downloaded and applied on the client, DeleteMetadasAsync is not called
- Work only on **T-2** metadatas. To be more secure, the **T-1** values stays in the tracking tables.

You can also manually delete metadatas from both server or client, using the method DeleteMetadatasAsync, available from both LocalOrchestrator and RemoteOrchestrator:

```
var clientProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(clientDbName));
var localOrchestrator = new LocalOrchestrator(clientProvider);
await localOrchestrator.DeleteMetadatasAsync();
```

Note: If you're using SqlSyncChangeTrackingProvider, the metadatas cleansing is automatically handled by the change tracking feature.

DisableConstraintsOnApplyChanges

The DisableConstraintsOnApplyChanges will disable all constraint on your database, before the sync process is launched, and will be enabled after. Use it if you're not sure of the table orders.

ScopeInfoTableName

This option allows you to customize the scope info table name. Default is *scope_info*.

On the server side, server scope table is prefixed with **_server** and history table with **_history**

ConflictResolutionPolicy

Define the default conflict resolution policy. See more here : Conflict

2.16 Provision, Deprovision & Migration

2.16.1 Overview

Since your sync architecture will evolve over the time, you may need to update the sync generated code as well.
Regarding the **DMS** architecture, we have two situations, the first one is automatically handled by **DMS** and the other one, not.
Fortunately, **DMS** provides some useful methods for all these scenario.

2.16.2 Provision / Deprovision

The `ProvisionAsync` and `DeprovisionAsync` methods are used internally by **DMS**

For instance, during the first sync, **DMS** will provision everything, on the server side and on the client side.

When you launch for the first time a sync process, **DMS** will:

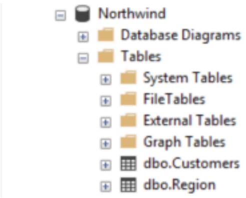
- **[Server Side]**: Get the database schema from the server database.
- **[Server Side]**: Create **Stored procedures**, **triggers** and **tracking tables**.
- **[Client Side]**: Fetch the server schema.
- **[Client Side]**: Create **tables** on the client database, if needed.
- **[Client Side]**: Create **Stored procedures**, **triggers** and **tracking tables**

Note: If you're using the `SqlSyncChangeTrackingProvider`, **DMS** will skip the creation of triggers and tracking tables, relying on the *Change Tracking* feature from SQL Server.

Basically, all these steps are managed by the `RemoteOrchestrator` on the server side, and by the `LocalOrchestrator` on the client side.
All the methods used to provision and deprovision tables are available from both the `LocalOrchestrator` and `RemoteOrchestrator` instances.

```
public async Task<SyncSet> ProvisionAsync(SyncProvision provision)
public async Task<SyncSet> ProvisionAsync(SyncSet schema, SyncProvision provision)

public async Task DeprovisionAsync(SyncProvision provision)
public virtual async Task DeprovisionAsync(SyncSet schema, SyncProvision provision)
```

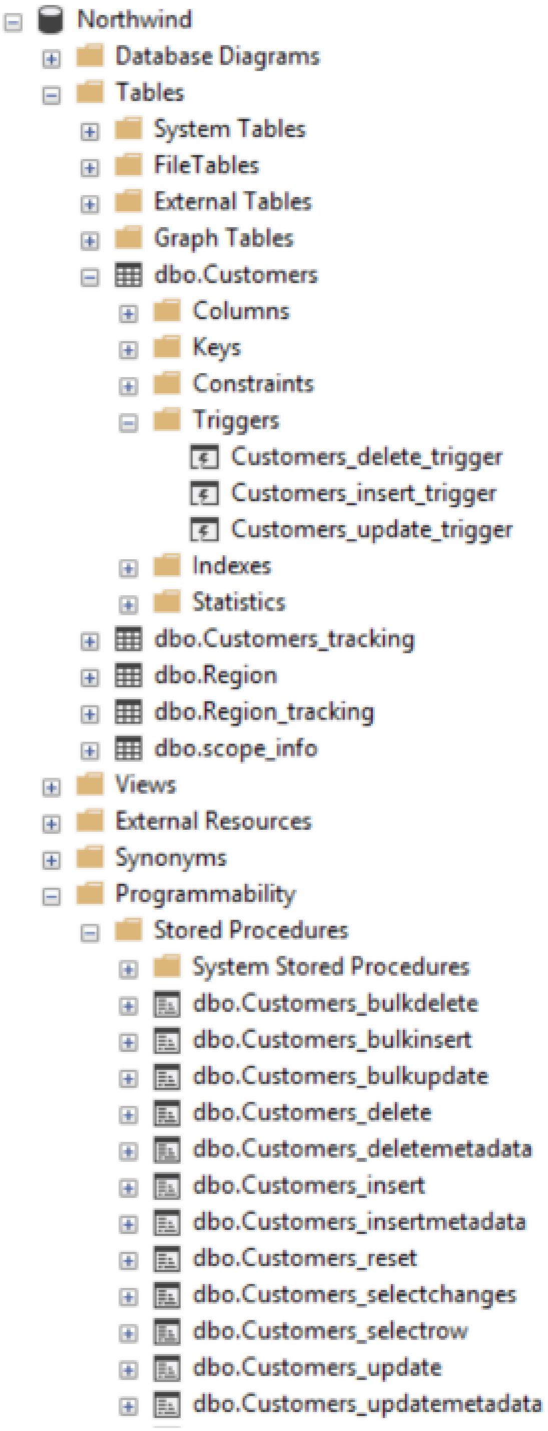
Let's start with a basic example, where you have a simple database containing two tables *Customers* and *Region*:



And here the most straightforward code to be able to sync a client db :

```
SqlSyncProvider serverProvider = new SqlSyncProvider(GetDatabaseConnectionString(
↪"Northwind"));
SqlSyncProvider clientProvider = new SqlSyncProvider(GetDatabaseConnectionString("NW1
↪"));
var setup = new SyncSetup("Customers", "Region");
var agent = new SyncAgent(clientProvider, serverProvider);
var result = await agent.SynchronizeAsync(setup);

Console.WriteLine(result);
```

Once your sync process is finished, you will have a full configured database :



- Northwind
  - Database Diagrams
  - Tables
    - System Tables
    - FileTables
    - External Tables
    - Graph Tables
    - dbo.Customers
      - Columns
      - Keys
      - Constraints
      - Triggers
        - Customers_delete_trigger
        - Customers_insert_trigger
        - Customers_update_trigger
      - Indexes
      - Statistics
    - dbo.Customers_tracking
    - dbo.Region
    - dbo.Region_tracking
    - dbo.scope_info
  - Views
  - External Resources
  - Synonyms
  - Programmability
    - Stored Procedures
      - System Stored Procedures
      - dbo.Customers_bulkdelete
      - dbo.Customers_bulkinsert
      - dbo.Customers_bulkupdate
      - dbo.Customers_delete
      - dbo.Customers_deletemetadata
      - dbo.Customers_insert
      - dbo.Customers_insertmetadata
      - dbo.Customers_reset
      - dbo.Customers_selectchanges
      - dbo.Customers_selectrow
      - dbo.Customers_update
      - dbo.Customers_updatemetadata

DMS has provisioned:

- One tracking table per table from your setup.
- Three triggers on each table.
- Several stored procedures for each table.

Provision

In some circumstances, you may want to provision manually your database, on the server using a remote orchestrator, or on the client side using a local orchestrator.

- If you have a really big database, the provision step could be really long, so it could be better to provision the server side before any sync process happens.
- If you have to modify your schema, you will have to **deprovision**, **edit** your schema and finally **provision** again your database.

That's why **DMS** exposes several methods to let you control how, and when, you want to provision and deprovision your database.

Each orchestrator has two main methods, basically:

```
ProvisionAsync(SyncSet schema, SyncProvision provision)
DeprovisionAsync(SyncSet schema, SyncProvision provision)
```

The `SyncProvision` enum parameter lets you decide which kind of objects (tables, stored proc, triggers or tracking tables) you will provision on your target database.

```
[Flags]
public enum SyncProvision
{
    NotSet = 0,
    Table = 1,
    TrackingTable = 2,
    StoredProcedures = 4,
    Triggers = 8,
    ScopeInfo = 16,
    ScopeInfoClient = 32
}
```

Note: `SyncProvision.NotSet` is the default value, and will provision everything, depending on the orchestrator used to provision.

The remote (server side) provisioning is quite simple, since the schema is already there.

But the local (client side) provisioning could a little bit more tricky since we may miss tables.
In that particular case, we will rely on the schema returned by the remote orchestrator.

Hint: You will find this complete sample here : Provision & Deprovision sample

Provisioning from server side, using a remote orchestrator:

```
var serverProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(serverDbName));

// Create standard Setup and Options
var setup = new SyncSetup("Address", "Customer", "CustomerAddress");

```

(continues on next page)

(continued from previous page)

```
// -----------------------------------------------------------------
// Server side
// -----------------------------------------------------------------

// This method is useful if you want to provision by yourself the server database
// You will need to :
// - Create a remote orchestrator with the correct setup to create
// - Provision everything

// Create a server orchestrator used to Deprovision and Provision only table Address
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

// Provision everything needed (sp, triggers, tracking tables)
// Internally provision will fectch the schema a will return it to the caller.
var sScopeInfo = await remoteOrchestrator.ProvisionAsync(setup);
```

Provision on the client side is quite similar, despite the fact we will rely on the server schema to create any missing table.

```
// Create 2 Sql Sync providers
var serverProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(serverDbName));
var clientProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(clientDbName));

// Create standard Setup and Options
var setup = new SyncSetup("Address", "Customer", "CustomerAddress");

// -----------------------------------------------------------------
// Client side
// -----------------------------------------------------------------

// This method is useful if you want to provision by yourself the client database
// You will need to :
// - Create a local orchestrator with the correct setup to provision
// - Get the schema from the server side using a RemoteOrchestrator or a␣
↪WebRemoteOrchestrator
// - Provision everything locally

// Create a local orchestrator used to provision everything locally
var localOrchestrator = new LocalOrchestrator(clientProvider);

// Because we need the schema from remote side, create a remote orchestrator
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

// Getting the scope info from server side
var serverScope = await remoteOrchestrator.GetScopeInfoAsync();

// At this point, if you need the schema and you are not able to create a␣
↪RemoteOrchestrator,
// You can create a WebRemoteOrchestrator and get the schema as well
// var proxyClientProvider = new WebRemoteOrchestrator("https://localhost:44369/api/
↪Sync");
// var serverScope = proxyClientProvider.GetServerScopeInfoAsync();

// Provision everything needed (sp, triggers, tracking tables, AND TABLES)
await localOrchestrator.ProvisionAsync(serverScope);
```

Deprovision

Like provisioning, deprovisioning uses basically the same kind of algorithm.

Deprovisioning from server side, using a remote orchestrator:

```
// Create server provider
var serverProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(serverDbName));

// Create a server orchestrator used to Deprovision everything on the server side
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

// Deprovision everything
var p = SyncProvision.ServerScope | SyncProvision.ServerHistoryScope |
        SyncProvision.StoredProcedures | SyncProvision.TrackingTable |
        SyncProvision.Triggers;

// Deprovision everything
await remoteOrchestrator.DeprovisionAsync(p);
```

Deprovisioning from client side, using a local orchestrator:

```
// Create client provider
var clientProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(clientDbName));

// Create a local orchestrator used to Deprovision everything
var localOrchestrator = new LocalOrchestrator(clientProvider);

var p = SyncProvision.ClientScope |
        SyncProvision.StoredProcedures | SyncProvision.TrackingTable |
        SyncProvision.Triggers;

// Deprovision everything
await localOrchestrator.DeprovisionAsync(p);
```

Drop All

The `DropAllAsync()` method is almost the same as `DeprovisionAsync()`. The main difference is that `DropAllAsync()` will make a `DeprovisionAsync()` on all scopes.

Basically `DroppAllAsync()` removes barely everything, and will let your database in a state without anything related to **DMS**

> **Warning:** Be extremely cautious with DropAllAsync as you will lost all the tracking rows information of your database.

```
// Create client provider
var clientProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(clientDbName));

// Create a local orchestrator used to Deprovision everything
var localOrchestrator = new LocalOrchestrator(clientProvider);

```

(continues on next page)

(continued from previous page)

```
// Drop everything
await localOrchestrator.DropAllAsync();
```

2.16.3 Migrating a database schema

During any dev cycle, you will probably have to make some evolutions on your server database.
Adding or deleting columns will break the sync process.
Manually, without the `ProvisionAsync()` and `DeprovisionAsync()` methods, you will have to edit all the stored procedures, triggers and so on to be able to recreate a full sync processus.

Before going further, you need to decide:

- When your server is upgraded with a new schema, is any client should still be able to sync their old version, without the new columns / tables added (or removed) ?
- When your server is upgraded with a new schema, is any client should upgrade their local database before being able to eventually make a new sync on the new schema ?

The first scenario is the easiest one to handle, since we will not allow any client to make a sync if they don't have a schema up to date.

The second scenario is a little bit more complex, but we can handle it by using different scopes.

Before going further, here is the starting point, using a quite simple Synchronization:

Basically, we can imagine having a sync process already in place:

```
// Create the server Sync provider
var serverProvider = new SqlSyncProvider(serverConnectionString);

// Create 2 clients. First will migrate, 2nd will stay without new column
var client1Provider = new SqlSyncProvider(clientConnectionString);
var databaseName = $"{Path.GetRandomFileName().Replace(".", "").ToLowerInvariant()}.db
↪";
var client2Provider = new SqliteSyncProvider(databaseName);

// Create standard Setup
var setup = new SyncSetup("Address", "Customer", "CustomerAddress");

// Creating agents that will handle all the process
var agent1 = new SyncAgent(client1Provider, serverProvider);
var agent2 = new SyncAgent(client2Provider, serverProvider);

// Using the Progress pattern to handle progession during the synchronization
var progress = new SynchronousProgress<ProgressArgs>(
    args => Console.WriteLine($"{args.ProgressPercentage:p}:\t{args.Message}"));

// First sync to have a starting point
// To make a full example, we are going to use differente scope name (v0, v1)
// v0 is the initial database
// v1 will contains the new column in the Address table
var s1 = await agent1.SynchronizeAsync("v0", setup, progress);
Console.WriteLine("Initial Sync on Sql Server Client 1");
Console.WriteLine(s1);
```

(continues on next page)

(continued from previous page)

```
var s2 = await agent2.SynchronizeAsync("v0", setup, progress);
Console.WriteLine("Initial Sync on Sqlite Client 2");
Console.WriteLine(s2);
```

In place migration

The in place migration will use the same algorithm as the provisioning and deprovisioning.
Eventually, we will just foce the `ProvisionAsync` call on both side.

The easiest method, **BUT** the downside is that once your server is upgraded, no clients will be able to sync unless they upgrade on their side as well.

Server Side

Now, we are adding a new column on the server side, in the **Address** table, and we are adding a new table **Product**.

Hint: Here, using a tool like EF Migrations could be really useful.

```
// -----------------------------------------------------------------
// Migrating a table by adding a new column on the server side
// -----------------------------------------------------------------

// Adding a new column called CreatedDate to Address table, on the server, and on the␣
↪client.
await AddNewColumnToAddressAsync(serverProvider.CreateConnection());
// Adding a new table Product
await AddNewTableProductAsync(serverProvider.CreateConnection());
```

Then, upgrading the server scope:

```
// -----------------------------------------------------------------
// Server side
// -----------------------------------------------------------------

// Create standard Setup
var setup = new SyncSetup("Product", "Address", "Customer", "CustomerAddress");

// Create a server orchestrator used to Deprovision and Provision only table Address
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

// Calling the ProvisionAsync method with overwrite to true, will force a refresh
await remoteOrchestrator.ProvisionAsync("v0", setup, overwrite:true,␣
↪progress:progress);
Console.WriteLine("Server migration with new column CreatedDate and new table Product␣
↪done.");
```

Client Side

On the client side, as on the server, you're still responsible of migrating your schema. Once it's done, the code is almost the same, a part from that you need to get the `ServerScopeInfo` from the server to be able to call `ProvisionAsync`:

```
// -----------------------------------------------------------------
// Client side
// -----------------------------------------------------------------

// Provision client with the new the v0 scope, again
// Getting the scope from server and apply it locally
var serverScope = await agent1.RemoteOrchestrator.GetServerScopeInfoAsync("v0",␣
↪progress: progress);

// provision
var clientScope = await agent1.LocalOrchestrator.ProvisionAsync(serverScope,
                                                                overwrite:true,␣
↪progress:progress);

Console.WriteLine("Sql Server client1 Provision done.");
```

You can use an interceptor as well, that can automate this step (if you are not able to update your client application after a database schema update).

Basically, the interceptor `OnConflictingSetup` is called every time a setup from the server is different from the one on the client.

You can choose then to update your database accordingly:

```
// -----------------------------------------------------------------
// Client side
// -----------------------------------------------------------------

agent.LocalOrchestrator.OnConflictingSetup(async args =>
{
    if (args.ServerScopeInfo != null)
    {
        args.ClientScopeInfo = await localOrchestrator.ProvisionAsync(args.
↪ServerScopeInfo, overwrite: true);

        // this action will let the sync continue
        args.Action = ConflictingSetupAction.Continue;

        return;
    }
    // if we raise this step, just and the sync without raising an error
    args.Action = ConflictingSetupAction.Abort;

    // The Rollback Action will raise an error
    // args.Action = ConflictingSetupAction.Rollback;
    }
});
```

Multi scopes migration

We are going to handle, with a little example, how we could add a new column on an already existing sync architecture:

Hint: You will find this complete sample here : Migration sample

The main constraint we have is to ensure that a client "not yet upgraded" will still continue to work, and a client "upgraded" will have a full sync available.

So far, we are going to use a multi scopes architecture:

- scope name "**v0**" will have the initial version of the schema.
- scope name "**v1**" will have the new version of the schema.

In our sample, we are going to migrate 2 clients.

- First client (running on **Sql Server**) will receive the upgrade on the finally
- Second client (running on **SQLite**) will stay on first version and then will eventually upgrade later.

Here is the macro processus:

- Client 1 & 2 are synced with the server, using the scope "**v0**".
- Server will upgrade its schema by adding a new column (with null values allowed !).
- Server will create a new SyncSetup on a new scope "**v1**" to have this new column handled.
- Client 1 will upgrade on its own its schema, adding the new column as well.
- Client 1 will request the new scope "**v2**" and will sync successfully the new column.
- Client 2 remains on scope "**v0**" and is still able to sync (but not the new column values).
- Eventually Client 2 upgrade its schema.
- Client 2 gets the new scope "**v1**".
- Client 2 makes a `Reinitialize` sync to get all rows with the correct values.

Warning: Remember : You are in charge of migrating the schema of your server and your clients !

Server side

Now, we are adding a new column on the server side, in the **Address** table:

Hint: Here, using a tool like EF Migrations could be really useful.

```
// -----------------------------------------------------------------
// Migrating a table by adding a new column on the server side
// -----------------------------------------------------------------

// Adding a new column called CreatedDate to Address table, on the server, and on the␣
↪client.
await AddNewColumnToAddressAsync(serverProvider.CreateConnection());
```

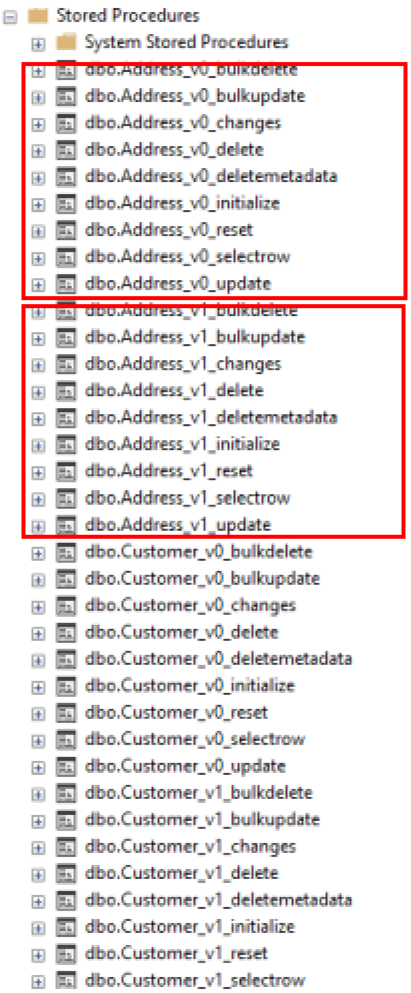
Then, create a new scope "**v1**":

```
// -----------------------------------------------------------------
// Server side
// -----------------------------------------------------------------

// Creating a new setup with the same tables
// We are going to provision a new scope (v1)
// Since this scope is not existing yet, it will force DMS to refresh the schema and
// get the new column
var setupAddress = new SyncSetup("Address", "Customer", "CustomerAddress");

// Create a server orchestrator used to Deprovision and Provision only table Address
var remoteOrchestrator = new RemoteOrchestrator(serverProvider);

// Provision everything again for this new scope v1,
// This provision method will fetch the address schema from the database,
// since the new scope name is not existing yet
// so it will contains all the columns, including the new Address column added
await remoteOrchestrator.ProvisionAsync("v1", setupAddress, progress:progress);
Console.WriteLine("Server migration with new column CreatedDate done.");
```

At this point, server database has two scopes:

- **v0** : first scope with Address table without the new column
- **v1** : second scope with Address table with the new column CreatedDate

If we look the database stored procedures, we see clearly the differences:



Stored Procedures
System Stored Procedures
dbo.Address_v0_bulkdelete
dbo.Address_v0_bulkupdate
dbo.Address_v0_changes
dbo.Address_v0_delete
dbo.Address_v0_deletemetadata
dbo.Address_v0_initialize
dbo.Address_v0_reset
dbo.Address_v0_selectrow
dbo.Address_v0_update
dbo.Address_v1_bulkdelete
dbo.Address_v1_bulkupdate
dbo.Address_v1_changes
dbo.Address_v1_delete
dbo.Address_v1_deletemetadata
dbo.Address_v1_initialize
dbo.Address_v1_reset
dbo.Address_v1_selectrow
dbo.Address_v1_update
dbo.Customer_v0_bulkdelete
dbo.Customer_v0_bulkupdate
dbo.Customer_v0_changes
dbo.Customer_v0_delete
dbo.Customer_v0_deletemetadata
dbo.Customer_v0_initialize
dbo.Customer_v0_reset
dbo.Customer_v0_selectrow
dbo.Customer_v0_update
dbo.Customer_v1_bulkdelete
dbo.Customer_v1_bulkupdate
dbo.Customer_v1_changes
dbo.Customer_v1_delete
dbo.Customer_v1_deletemetadata
dbo.Customer_v1_initialize
dbo.Customer_v1_reset
dbo.Customer_v1_selectrow

And if we are opening the stored procedures we see that "**v1**" is handling the new column.

As an example, let's add a new row on the server side, with this new column:

```
// Now add a row on the server (with the new column)
var addressId = await Helper.InsertOneAddressWithNewColumnAsync(
    new SqlConnection(serverConnectionString));
Console.WriteLine($"New address row added with pk {addressId}");
```

Now that the server is migrated, we are going to handle the clients:

- Client 1 migrates then sync on "**v1**".
- Client 2 stays on "**v0**" and sync.

Client 1

```
// -----------------------------------------------------------------
// Client side
// -----------------------------------------------------------------


// adding the column to the client
```

(continues on next page)

(continued from previous page)

```
await Helper.AddNewColumnToAddressAsync(new SqlConnection(clientConnectionString));
Console.WriteLine("Sql Server client1 migration with new column CreatedDate done.");

// Provision client with the new the V1 scope
// Getting the scope from server and apply it locally
var sScopeInfo = await agent1.RemoteOrchestrator.GetScopeInfoAsync("v1");

// provision this new scope
var v1cScopeInfo = await agent1.LocalOrchestrator.ProvisionAsync(sScopeInfo);
Console.WriteLine("Sql Server client1 Provision done.");
```

Now a tricky part to understand:

You have created a new scope locally, but this scope “**v1**” is **NEW** and

- **DMS** considers that all the tables in this scope are empty.
- If we launch the sync on this new scope, **DMS** will download all the rows from the server.

In this scenario we want to sync the **v1** scope on the same basis timeframe as the **v0** scope.

That’s why we are going to “**shadow copy**” the required properties from the “**v0**” scope to the “**v1**” scope using the `ShadowScope()` method:

```
// TRICKY PART
/*
    The scope v1 is new.
    If we sync now, since v1 is new, we are going to sync all the rows from start
    What we want is to sync from the last point we sync the old v0 scope
    That's why we are shadowing the metadata info from v0 into v1
*/
var v1cScopeInfoClient= await agent1.LocalOrchestrator.GetScopeInfoClientAsync("v1");
var v0cScopeInfoClient = await agent1.LocalOrchestrator.GetScopeInfoClientAsync("v0");
v1cScopeInfoClient.ShadowScope(v0cScopeInfoClient);
await agent1.LocalOrchestrator.SaveScopeInfoClientAsync(v1cScopeInfoClient);
```

Now our client 1 is upgraded, has the new scope and can eventually launch a new sync on this new scope:

```
// Now test a new sync, on this new scope v1
var s4 = await agent1.SynchronizeAsync("v1", progress: progress);
Console.WriteLine($"Sql Server client1 migrated, doing a sync on second scope v1:");
Console.WriteLine(s4);

// If we get the client row from the client database, it should contains the value
var client1row = await Helper.GetLastAddressRowAsync(
    new SqlConnection(clientConnectionString), addressId);
```

Optionally, we can remove the old scope on the Client 1, that we don’t need anymore:

- On Sql Server, we just need to remove the stored procedures for **v0**
- We can (but optional) remove the “**v0**” scope information from the scope info table

```
// On this new client, migrated, we no longer need the v0 scope
// we can deprovision it
await agent1.LocalOrchestrator.DeprovisionAsync("v0", SyncProvision.StoredProcedures);

await agent1.LocalOrchestrator.DeleteClientScopeInfoAsync(v0clientScope);
```

(continues on next page)

(continued from previous page)

```
Console.WriteLine($"Deprovision of old scope v0 done on Sql Server client1");
```

Client 2

Client 2 is still syncing on the "**v0**" scope and will eventually migrate to "**v1**".

Hint: Notice we are using a ReinitializeWithUpload argument at the end to get all rows from server

```
// -----------------------------------------------------------------
// SQLite Client will stay on old schema (without the new CreatedDate column)
// -----------------------------------------------------------------

// First of all, we are still able to sync the local database without having to␣
↪migrate the client
// allows old clients that do not have the new column, to continue sync normally
// these old clients will continue to sync on the v0 scope

var s3 = await agent2.SynchronizeAsync("v0", setup, progress: progress);
Console.WriteLine($"Sqlite not migrated, doing a sync on first scope v0:");
Console.WriteLine(s3);

// If we get the row from the client, we have the new row inserted on server,
// but without the new column
var client2row = await Helper.GetLastAddressRowAsync(client2Provider.
↪CreateConnection(), addressId);
Console.WriteLine(client2row);

// -----------------------------------------------------------------
// SQLite Client will eventually migrate to v1
// -----------------------------------------------------------------

// It's time to migrate the sqlite client
// Adding the column to the SQLite client
await Helper.AddNewColumnToAddressAsync(client2Provider.CreateConnection());
Console.WriteLine($"Column eventually added to Sqlite client2");

// Provision SQLite client with the new the V1 scope
var v1cScopeInfo2= await agent2.LocalOrchestrator.ProvisionAsync(sScopeInfo);
Console.WriteLine($"Provision v1 done on SQLite client2");

// ShadowScope old scope info client to new scope info client
var v1cScopeInfoClient2 = await agent2.LocalOrchestrator.GetScopeInfoClientAsync("v1
↪");
var v0cScopeInfoClient2 = await agent2.LocalOrchestrator.GetScopeInfoClientAsync("v0
↪");
v1cScopeInfoClient2.ShadowScope(v0cScopeInfoClient2);
await agent2.LocalOrchestrator.SaveScopeInfoClientAsync(v1cScopeInfoClient2);

// let's try to sync firstly
// Now test a new sync, on this new scope v1
// Obviously, we don't have anything from the server
var s5 = await agent2.SynchronizeAsync("v1", progress: progress);
```

(continues on next page)

(continued from previous page)

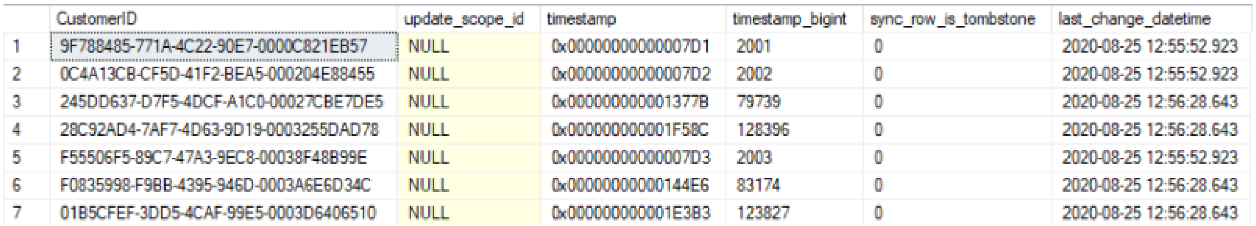
```
Console.WriteLine(s5);

// If we get the row from client, we have the new column, but value remains null
// since this row was synced before client migration
client2row = await Helper.GetLastAddressRowAsync(client2Provider.CreateConnection(),␣
↪addressId);
Console.WriteLine(client2row);

// What we can do here, is just make a sync with Renit
var s6 = await agent2.SynchronizeAsync("v1", SyncType.Reinitialize, progress:␣
↪progress);
Console.WriteLine($"Making a full Reinitialize sync on SQLite client2");
Console.WriteLine(s6);

// And now the row is correct
// If we get the row from client, we have the new column, but value remains null
// since this row was synced before client migration
client2row = await Helper.GetLastAddressRowAsync(client2Provider.CreateConnection(),␣
↪addressId);
Console.WriteLine(client2row);
```

2.17 Metadatas

All tracking tables maintains the state of each row. Especially for deleted rows.

For example, here is the content of the `[Customer_tracking]` after a successful sync:

```
SELECT * FROM [Customer_tracking]
```



	CustomerID	update_scope_id	timestamp	timestamp_bigint	sync_row_is_tombstone	last_change_datetime
1	9F788485-771A-4C22-90E7-0000C821EB57	NULL	0x00000000000007D1	2001	0	2020-08-25 12:55:52.923
2	0C4A13CB-CF5D-41F2-BEA5-000204E88455	NULL	0x00000000000007D2	2002	0	2020-08-25 12:55:52.923
3	245DD637-D7F5-4DCF-A1C0-00027CBE7DE5	NULL	0x000000000001377B	79739	0	2020-08-25 12:56:28.643
4	28C92AD4-7AF7-4D63-9D19-0003255DAD78	NULL	0x000000000001F58C	128396	0	2020-08-25 12:56:28.643
5	F55506F5-89C7-47A3-9EC8-00038F48B99E	NULL	0x00000000000007D3	2003	0	2020-08-25 12:55:52.923
6	F0835998-F9BB-4395-946D-0003A6E6D34C	NULL	0x00000000000144E6	83174	0	2020-08-25 12:56:28.643
7	01B5CFEF-3DD5-4CAF-99E5-0003D6406510	NULL	0x000000000001E3B3	123827	0	2020-08-25 12:56:28.643

So, over time, we can have an increase of these tracking tables, with a lot of rows that are not useful anymore.
These **metadatas rows** are present on the server side of course, and also on the **client side**.

Note: If you are using the `SqlSyncChangeTrackingProvider` provider, you do not have to maintains and manage the metadatas, since it's handled by the **SQL Server** engine.

2.17.1 Client side

On the client side, once the client has made a synchronization with success, we can easily purge the metadata rows from all the local tracking tables.

The `CleanMetadatas` option (boolean `true` / `false` available through the `SyncOptions` object) allows you to clean automatically the `_tracking` tables metadata rows from your client databases.

If enabled, the client database will basically delete all the metadata rows from the tracking tables, after every successful sync.

Note: The metadata rows purge mechanism will work only:

- If the client has downloaded *something* from the server. If there is no changes downloaded and applied on the client, `DeleteMetadasAsync()` is not called
- On **T-2** metadata rows. To be more secure, the **T-1** values stays in the tracking tables.

So far, the client side is easy to maintain, since it's by default, automatic… magic…

2.17.2 Server side

There is no automatic mechanism on the server side. Mainly because **DMS** does not know *when* it should clean the metadata rows on the server.

Note: Indeed we can launch the metadata rows cleanup routine after *every* client synchronization, but it will lead to an non-necessary overhead and will extend the time needed for each sync

---

Basically, the most important is to keep the metadata rows as long as one client needs them to retrieve the deleted / updated rows.
Once all clients have made a sync and are up to date at time **T**, we can theoricaly supposing that the metadata rows from **0** to **T-1** are not needed anymore.

The easiest way to achieve that, on the server side, is to create a schedule task and call the `DeleteMetadatasAsync` method (from a console application, service windows, whatever…) with this kind of code:

```
var rmOrchestrator = new RemoteOrchestrator(serverProvider);
await rmOrchestrator.DeleteMetadatasAsync();
```

DMS will delete the metadata rows in the safest way to ensure no client become *out-dated*.

How does it work

What happens under the hood ?

DMS will try to get the *min* timestamp available from the *scope_info_history* table to ensure that no clients becomes *out-dated*.

Basically, if you have this kind of *scope_info_history* table :

```
SELECT [sync_scope_id] ,[sync_scope_name] ,[scope_last_sync_timestamp], [scope_last_sync]
FROM [AdventureWorks].[dbo].[scope_info_history]
```

Server database:

sync_scope_id	sync_scope_name	scope_last_sync_timestamp	scope_last_sync
9E9722CD-…	DefaultScope	2090	2020-04-01
AB4122AE-…	DefaultScope	2100	2020-04-10
DB6EEC7E-…	DefaultScope	**2000**	2020-03-20
E9CBB51D-…	DefaultScope	2020	2020-03-21
CC8A9184-…	DefaultScope	2030	2020-03-22
D789288E-…	DefaultScope	2040	2020-03-23
95425970-…	DefaultScope	2050	2020-03-24
5B6ACCC0-…	DefaultScope	2060	2020-03-25

The `Min(scope_last_sync_timestamp)` will be **2000** and then **DMS** will internally call `remoteOrchestrator.DeleteMetadatasAsync(2000);`

Going further

Now imagine we have one client that did a first sync, and then **never did a sync again for 3 years** … This situation will lead to this kind of rows in the *scope_info_history* table:

```
SELECT [sync_scope_id] ,[sync_scope_name] ,[scope_last_sync_timestamp], [scope_last_
↪sync]
FROM [AdventureWorks].[dbo].[scope_info_history]
```

Server database:

sync_scope_id	sync_scope_name	scope_last_sync_timestamp	scope_last_sync
9E9722CD-…	DefaultScope	**100**	**2017-04-01**
AB4122AE-…	DefaultScope	2100	2020-04-10
DB6EEC7E-…	DefaultScope	2000	2020-03-20
E9CBB51D-…	DefaultScope	2020	2020-03-21
CC8A9184-…	DefaultScope	2030	2020-03-22
D789288E-…	DefaultScope	2040	2020-03-23
95425970-…	DefaultScope	2050	2020-03-24
5B6ACCC0-…	DefaultScope	2060	2020-03-25

Once again, if you call the `remoteOrchestrator.DeleteMetadatasAsync()` from your schedule task, internally **DMS** will delete all rows where timestamp is inferior to **100** (and so far, all metadata rows existing before year 2017)

It's not really interesting to keep **all** the metadata rows from **2017** to **2020**, just because of **One** client who never did a sync since 2017…

Eventually we can assume this client has removed the app or changed his mobile device or whatever. We can argue that this client can be considered as *out-dated* and will have to **reinitialize** everything if he tries to sync again.

Then how to create a scheduled taks with that will workaround this situation ?

Well, can make this assumption:

- We will run the `DeleteMetadatasAsync()` every month (or weeks, choose the best interval for you)
- Each run will take the `Min(scope_last_sync_timestamp)` from the *scope_info_history* table for all client that have, at least, sync during the last **30** days.

The code became:

```
// get all scope info clients
var sScopeInfoClients = await remoteOrchestrator.GetAllScopeInfoClientsAsync();

// select only clients that have synced at least 30 days earlier
var oneMonthMaxScopeInfoClients = sScopeInfoClients.Where(
    sic => sic.LastSync.HasValue && sic.LastSync.Value >= DateTime.Now.AddDays(-30));

// Get the min timestamp
var minTimestamp = oneMonthMaxScopeInfoClients.Min(h => h.LastSyncTimestamp);

// Call the delete metadatas with this timestamp
await remoteOrchestrator.DeleteMetadatasAsync(minTimestamp.Value);
```

Grab this code, create a *routine* to execute every month, and your server database won't growth too much because of the tracking tables metadata rows.

2.18 Conflicts

2.18.1 Overview

Conflicts occurs when a client update / delete / insert a record that is updated / deleted or inserted on the server as well, *before any sync happened*.

As an example, we can imagine a conflict occuring during an update on a column called ‘‘Street‘:

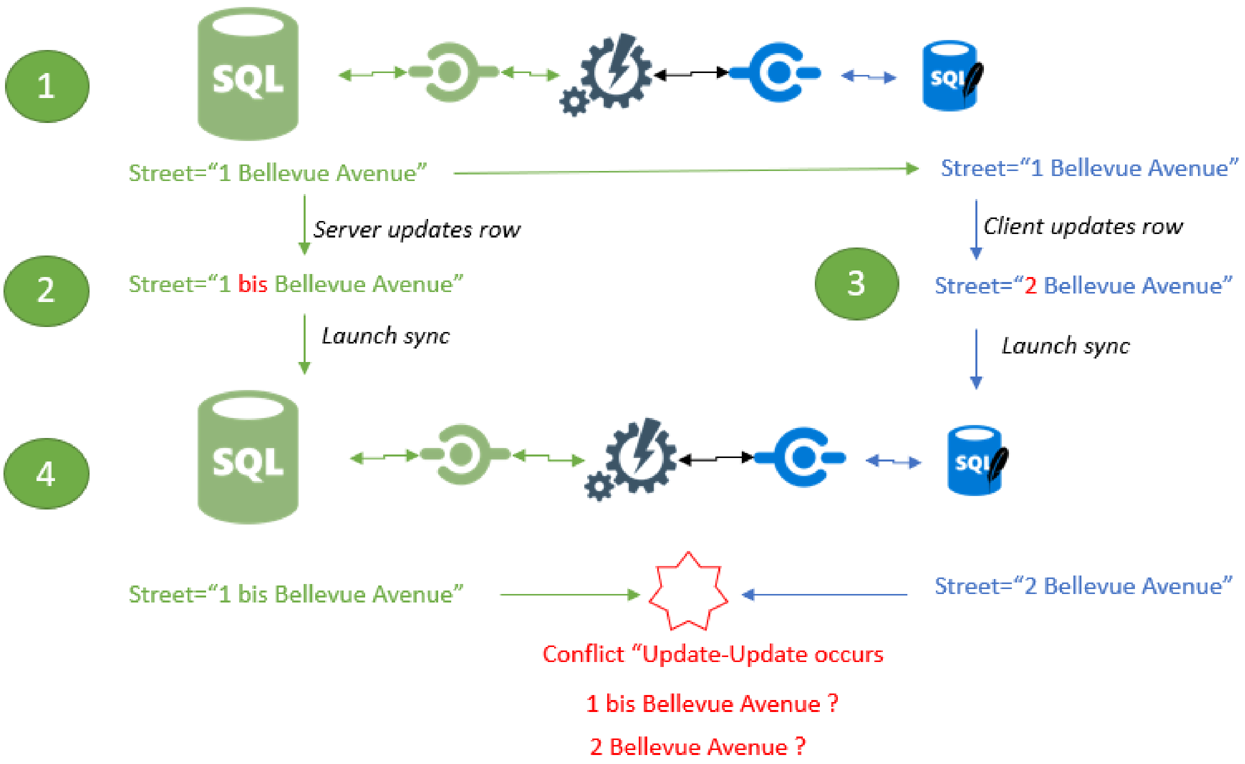
1) As a starting point, both server and client has a value of `Street=1 Bellevue Avenue` after an initial sync (where no conflicts occured).

2) Server is updating the row with a value of "*1 bis Bellevue Avenue*".

3) Client is updating as well the same row value with "*2 Bellevue Avenue*".

4) Sync is launched, and a conflict is raised **on the server side**.

Here is the diagram of the situation:


1
SQL
Street="1 Bellevue Avenue"
Street="1 Bellevue Avenue"
Server updates row
Client updates row
2
Street="1 bis Bellevue Avenue"
3
Street="2 Bellevue Avenue"
Launch sync
Launch sync
4
SQL
Street="1 bis Bellevue Avenue"
Street="2 Bellevue Avenue"
Conflict "Update-Update occurs
1 bis Bellevue Avenue ?
2 Bellevue Avenue ?

By default, conflicts are resolved automaticaly using the configuration policy property `ConflictResolutionPolicy` set in the `SyncOptions` object :

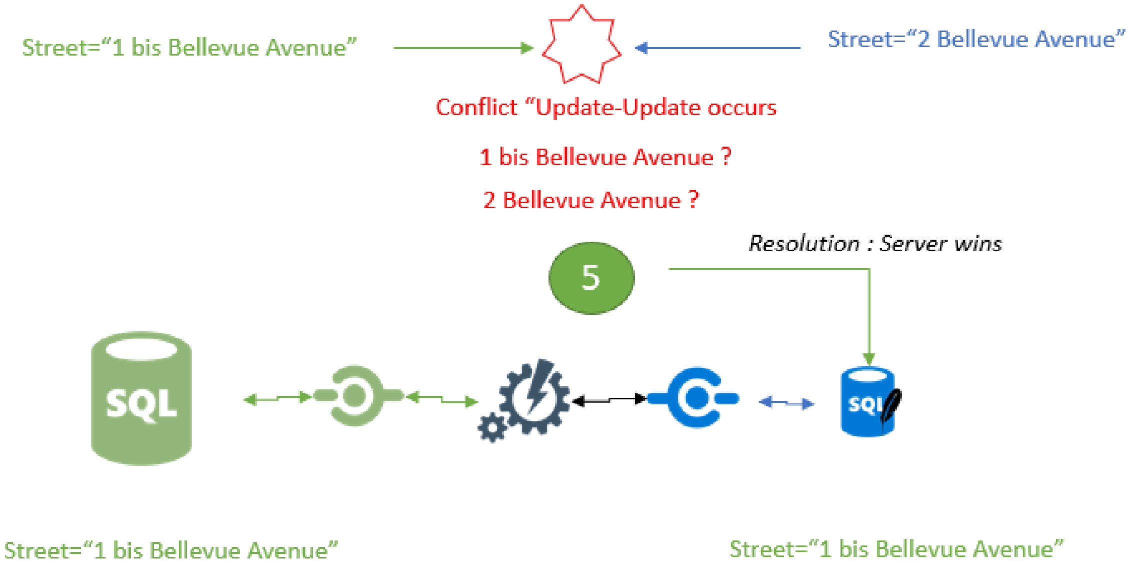
You can choose:

- `ConflictResolutionPolicy.ServerWins` : The server is the winner of any conflict. this behavior is the default behavior.
- `ConflictResolutionPolicy.ClientWins` : The client is the winner of any conflict.

Hint: Default value is `ServerWins`.

```
var options = new SyncOptions { ConflictResolutionPolicy = ConflictResolutionPolicy.
↪ServerWins };
```

Here is the same diagram with the final step, where resolution is set to `ServerWins` (default value, by the way)


Street="1 bis Bellevue Avenue"
Street="2 Bellevue Avenue"
Conflict "Update-Update occurs
1 bis Bellevue Avenue ?
2 Bellevue Avenue ?
Resolution : Server wins
5
SQL
SQL
Street="1 bis Bellevue Avenue"
Street="1 bis Bellevue Avenue"

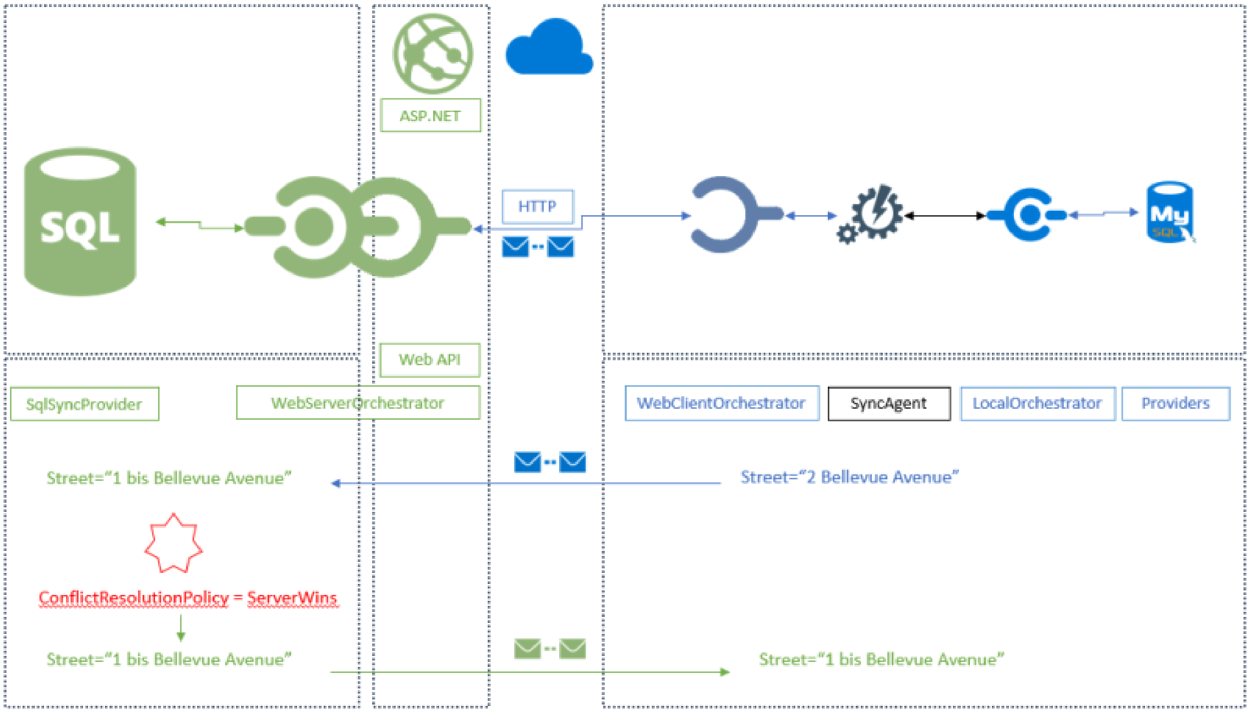
2.18.2 Resolution

Warning: A conflict is always resolved on the server side.

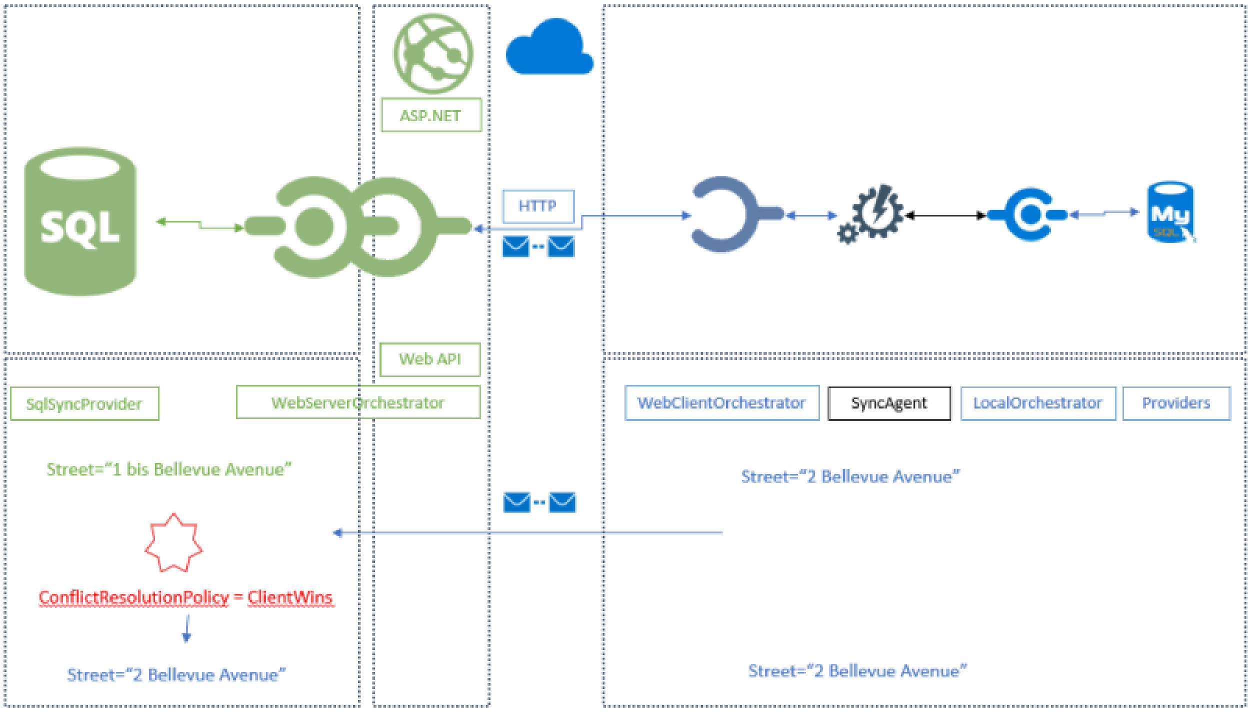
Depending on your policy resolution, the workflow could be:

- A conflict is generated on the client and the server side.
- The client is launching a sync processus.
- The server tries to apply the row and a conflict is generated.
- The server resolves the conflict on the server side.
- If the server wins, the resolved server row is sent to the client and is *force-applied* on the client database.
- If the client wins, the server will *force-apply* the client row on the server. Nothing happen on the client, since the row is correct.

Here is the workflow, when the conflict resolution is set to `ServerWins` in an **HTTP** mode:


SQL
ASP.NET
HTTP
Web API
My SQL
SqlSyncProvider
WebServerOrchestrator
WebClientOrchestrator
SyncAgent
LocalOrchestrator
Providers
Street="1 bis Bellevue Avenue"
Street="2 Bellevue Avenue"
ConflictResolutionPolicy = ServerWins
Street="1 bis Bellevue Avenue"
Street="1 bis Bellevue Avenue"

Here is the same workflow, when the conflict resolution is now set to `ClientWins`:


SQL
ASP.NET
HTTP
Web API
My SQL
SqlSyncProvider
WebServerOrchestrator
WebClientOrchestrator
SyncAgent
LocalOrchestrator
Providers
Street="1 bis Bellevue Avenue"
Street="2 Bellevue Avenue"
ConflictResolutionPolicy = ClientWins
Street="2 Bellevue Avenue"
Street="2 Bellevue Avenue"

2.18.3 Handling conflicts manually

If you decide to manually resolve a conflict, the `ConflictResolutionPolicy` option will be ignored.

To be able to resolve a conflict, you just have to *Intercept* the `ApplyChangedFailed` method and choose the correct version.

```
agent.OnApplyChangesFailed(args =>
{
// do stuff and choose correct resolution policy
});
```

The ApplyChangeFailedEventArgs argument contains all the required properties to be able to resolve your conflict.

You will determinate the correct version through the *Action* property of type ConflictResolution:

```
public enum ConflictResolution
{
    /// <summary>
    /// Indicates that the change on the server is the conflict winner
    /// </summary>
    ServerWins,

    /// <summary>
    /// Indicates that the change sent by the client is the conflict winner
    /// </summary>
    ClientWins,

    /// <summary>
    /// Indicates that you will manage the conflict by filling the final row and sent␣
↪it to
    /// both client and server
    /// </summary>
    MergeRow,

    /// <summary>
    /// Indicates that you want to rollback the whole sync process
    /// </summary>
    Rollback
}
```

- ConflictResolution.ClientWins : The client row will be applied on server, even if there is a conflict, so the client row wins.
- ConflictResolution.ServerWins : The client row won't be applied on the server, so the server row wins.
- ConflictResolution.MergeRow : It's up to you to choose the correct row to send on both server and client. the FinalRow instance will be used instead of Server or Client row.

You are able to compare the row in conflict through the Conflict property of type SyncConflict:

- Conflict.LocalRow : Contains the conflict row from the client side. This row is readonly.
- Conflict.RemoteRow : Contains the conflict row from the server side. This row is readonly.
- Conflict.Type : Gets the ConflictType enumeration. For example ConflictType.RemoteUpdateLocalUpdate represents a conflict row beetween an updated row on the server and the same row updated on the client as well.

You can use the current connection during this event to be able to perform actions on the server side through the DbConnection and DbTransaction properties.
If you decide to rollback the transaction, all the sync process will be rollback.

Eventually, the `FinalRow` property is used when you specify an Action to `ConflictAction.MergeRow`. You decide what will contains the row applied on both server and client side. Be careful, the `FinalRow` property is null until you specify the `Action` property to `ConflictAction.MergeRow` !

TCP mode

Manually resolving a conflict based on a column value:

```
agent.OnApplyChangesFailed(e =>
{
    if (e.Conflict.RemoteRow.Table.TableName == "Region")
    {
        e.Action = (int)e.Conflict.RemoteRow["Id"] == 1 ?
                ConflictResolution.ClientWins :
                ConflictResolution.ServerWins;
    }
}
```

Manually resolving a conflict based on the conflict type :

```
agent.OnApplyChangesFailed(args =>
{
    switch (args.Conflict.Type)
    {
        //
        case ConflictType.RemoteExistsLocalExists:
        case ConflictType.RemoteExistsLocalIsDeleted:
        case ConflictType.RemoteIsDeletedLocalExists:
        case ConflictType.RemoteIsDeletedLocalIsDeleted:
        case ConflictType.RemoteCleanedupDeleteLocalUpdate:
        case ConflictType.RemoteExistsLocalNotExists:
        case ConflictType.RemoteIsDeletedLocalNotExists:
        default:
            break;
    }
});
```

Resolving a conflict by specifying a merged row :

```
agent.OnApplyChangesFailed(e =>
{
    if (e.Conflict.RemoteRow.Table.TableName == "Region")
    {
        e.Action = ConflictResolution.MergeRow;
        e.FinalRow["RegionDescription"] = "Eastern alone !";
    }
}
```

Note: Be careful, the `e.FinalRow` is null until you set the `Action` property to `ConflictAction.MergeRow` !

HTTP Mode

We saw that conflicts are resolved on the server side, if you are in an **HTTP** mode, involving a server web side, it is there that you need to intercept failed applied changes:

```
[Route("api/[controller]")]
[ApiController]
public class SyncController : ControllerBase
{
    private WebServerAgent webServerAgent;

    // Injected thanks to Dependency Injection
    public SyncController(WebServerAgent webServerAgent)
        => this.webServerAgent = webServerAgent;

    public async Task Post()
    {
        webServerAgent.RemoteOrchestrator.OnApplyChangesFailed(e =>
        {
            if (e.Conflict.RemoteRow.SchemaTable.TableName == "Region")
            {
                e.Resolution = ConflictResolution.MergeRow;
                e.FinalRow["RegionDescription"] = "Eastern alone !";
            }
            else
            {
                e.Resolution = ConflictResolution.ServerWins;
            }
        });

        // handle request
        await webServerAgent.HandleRequestAsync(this.HttpContext);

    }

    /// <summary>
    /// This Get handler is optional.
    /// It allows you to see the configuration hosted on the server
    /// The configuration is shown only if Environmenent == Development
    /// </summary>
    [HttpGet]
    public Task Get() => this.HttpContext.WriteHelloAsync(webServerAgent);
}
```

2.18.4 Handling conflicts from the client side

As we said, all the conflicts are resolved from the server side.
But, using a **Two sync trick**, you are able to resolve the conflict from the client side.

Basically the process is occuring in this order: - The first sync will raise the conflict and will be resolved on the server. - The first sync will send back the resolved conflict to the client, containing the server row and the client row - From the client side, you will now be able to ask the client to choose the correct version - The second sync will then send back the *new* version of the row to the server.

Warning: To be able to use this technic, the ConflictResolutionPolicy MUST be set to ConflictResolutionPolicy.ServerWins

Here is a full example using this special trick:

```
var agent = new SyncAgent(clientProvider, serverProvider, options, setup);

var localOrchestrator = agent.LocalOrchestrator;
var remoteOrchestrator = agent.RemoteOrchestrator;

// Conflict resolution MUST BE set to ServerWins
options.ConflictResolutionPolicy = ConflictResolutionPolicy.ServerWins;

// From client : Remote is server, Local is client
// From here, we are going to let the client decides
// who is the winner of the conflict :
localOrchestrator.OnApplyChangesFailed(acf =>
{
    // Check conflict is correctly set
    var localRow = acf.Conflict.LocalRow;
    var remoteRow = acf.Conflict.RemoteRow;

    // From that point, you can easily letting the client decides
    // who is the winner
    // Show a UI with the local / remote row and
    // letting him decides what is the good row version
    // for testing purpose; will just going to set name to some fancy BLA BLA value

    // SHOW UI
    // OH.... CLIENT DECIDED TO SET NAME TO "BLA BLA BLA"

    // BE AS FAST AS POSSIBLE IN YOUR DESICION,
    // SINCE WE HAVE AN OPENED CONNECTION / TRANSACTION RUNNING

    remoteRow["Name"] = clientNameDecidedOnClientMachine;

    // Mandatory to override the winner registered in the tracking table
    // Use with caution !
    // To be sure the row will be marked as updated locally,
    // the scope id should be set to null
    acf.SenderScopeId = null;
});

// First sync, we allow server to resolve the conflict and send back the result to client
var s = await agent.SynchronizeAsync();

Assert.Equal(1, s.TotalChangesDownloaded);
Assert.Equal(1, s.TotalChangesUploaded);
Assert.Equal(1, s.TotalResolvedConflicts);

// From this point the Server row Name is STILL "SRV...."
// And the Client row NAME is "BLA BLA BLA..."
// Make a new sync to send "BLA BLA BLA..." to Server

s = await agent.SynchronizeAsync();
```

(continues on next page)

(continued from previous page)

```
Assert.Equal(0, s.TotalChangesDownloaded);
Assert.Equal(1, s.TotalChangesUploaded);
Assert.Equal(0, s.TotalResolvedConflicts);
```

2.19 Errors

2.19.1 Overview

Errors can happens during a synchronization process.

By default, if an error occurs, the synchronization process is stopped and the transaction is rolled back.

Note: The Error Resolution can be different on each side.

That’s being said, you can configure the synchronization process to continue even if an error occurs, and adapt the resolution process.

- The default error resolution can be defined from the `SyncOptions` instance on each side.
- You can use the `OnApplyChangesConflictOccured` interceptor to handle errors on a per-row basis.

```
// OPTION 1: Default error resolution using SyncOptions:
var syncOptions = new SyncOptions { ErrorResolutionPolicy = ErrorResolution.
↪RetryOnNextSync };

var agent = new SyncAgent(clientProvider, serverProvider, options);

// OPTION 2: Per-row error resolution using OnApplyChangesConflictOccured event:
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW: {args.ErrorRow}");
    args.Resolution = ErrorResolution.RetryOnNextSync;
});
```

Here is the description of the `ErrorResolution` enum, used to define the default error resolution:

```
/// <summary>
/// Determines what kind of action should be taken when an error is raised from the␣
↪datasource
/// during an insert / update or delete command
/// </summary>
public enum ErrorResolution
{
    /// <summary>
    /// Ignore the error and continue to sync. Error will be stored
    /// locally in a separate batch info file
    /// <para>
    /// Row is stored locally with a state of <see cref="SyncRowState.
↪ApplyDeletedFailed"/>
    /// or <see cref="SyncRowState.ApplyModifiedFailed"/> depending on the row state.
    /// </para>
```

(continues on next page)

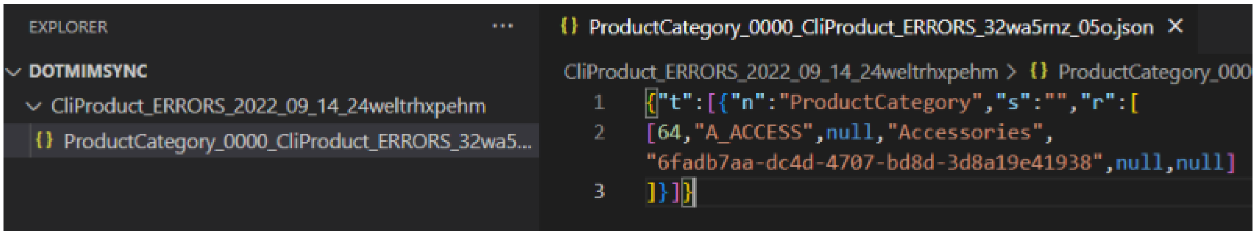
(continued from previous page)

```
    /// </summary>
    ContinueOnError,

    /// <summary>
    /// Will try one more time once after all the others rows in the table.
    /// <para>
    /// If the error is raised again, an exception is thrown and transaction is rolled back
    /// </para>
    /// </summary>
    RetryOneMoreTimeAndThrowOnError,

    /// <summary>
    /// Will try one more time once after all the others rows in the table.
    /// <para>
    /// If the error is raised again, Sync continues normally and error will be stored locally in a
    /// separate batch info file with a state of <see cref="SyncRowState.ApplyDeletedFailed"/>
    /// or <see cref="SyncRowState.ApplyModifiedFailed"/> depending on the row state.
    /// </para>
    /// </summary>
    RetryOneMoreTimeAndContinueOnError,

    /// <summary>
    /// Row is stored locally and will be applied again on next sync. Sync continues normally and
    /// row is stored locally with a state of <see cref="SyncRowState.RetryDeletedOnNextSync"/>
    /// or <see cref="SyncRowState.RetryModifiedOnNextSync"/> depending on the row state.
    /// </summary>
    RetryOnNextSync,

    /// <summary>
    /// Considers the row as applied.
    /// </summary>
    Resolved,

    /// <summary>
    /// Throw the error. Default value.
    /// </summary>
    Throw
}
```

Each time a row is in error (due to various reasons like **Unique constraint** or **ForeignKeys constraint** or any other failure that could happens), the row in error will be logged locally in a `BatchInfo` directory.

Usually the batch info file has "*ERROR*" in its directory name.

Note: The batch info file is a json file, and can be opened with any text editor.


EXPLORER
DOTMIMSYNC
CliProduct_ERRORS_2022_09_14_24weltrhxpehm
ProductCategory_0000_CliProduct_ERRORS_32wa5...
ProductCategory_0000_CliProduct_ERRORS_32wa5rnz_05o.json
CliProduct_ERRORS_2022_09_14_24weltrhxpehm > ProductCategory_000

```
{"t":[{"n":"ProductCategory","s":"","r":[
[64,"A_ACCESS",null,"Accessories",
"6fadb7aa-dc4d-4707-bd8d-3d8a19e41938",null,null]
]}]}
```

You can read all the lines from the error folders, using the `LoadBatchInfosAsync` method from your `LocalOrchestrator` or `RemoteOrchestrator` instance.

```
var batchInfos = await agent.LocalOrchestrator.LoadBatchInfosAsync();

foreach (var batchInfo in batchInfos)
{
    Console.WriteLine($"BatchInfo: {batchInfo.DirectoryName}");

    // Load all tables from the batch info files
    var syncTables = agent.LocalOrchestrator.LoadTablesFromBatchInfoAsync(batchInfo);

    await foreach (var syncTable in syncTables)
    {
        Console.WriteLine(syncTable.TableName);
        foreach (var syncRow in syncTable.Rows)
        {
            Console.WriteLine($"Row: {syncRow}");
        }
    }
}
```

2.19.2 Resolution

To emphazise, we are going to see the error resolution in action. We are creating a foreign key error on the *ProductCategory* table:

```
CREATE TABLE [ProductCategory](
    [ProductCategoryID] [nvarchar](50) NOT NULL,
    [ParentProductCategoryId] [nvarchar](50) NULL,
    [Name] [nvarchar](50) NOT NULL,
    [rowguid] [uniqueidentifier] NULL,
    [ModifiedDate] [datetime] NULL,
    [Attribute With Space] [nvarchar](max) NULL,
CONSTRAINT [PK_ProductCategory] PRIMARY KEY CLUSTERED ([ProductCategoryID] ASC))

GO
ALTER TABLE [ProductCategory] WITH CHECK ADD CONSTRAINT [FK_ParentProductCategoryId]
FOREIGN KEY([ParentProductCategoryId]) REFERENCES [ProductCategory]␣
↪([ProductCategoryID])

GO
BEGIN TRAN
    ALTER TABLE [ProductCategory] NOCHECK CONSTRAINT ALL
    INSERT [ProductCategory] ([ProductCategoryID], [ParentProductCategoryId], [Name])
    VALUES (N'A', 'B', N'A Sub category')

    INSERT [ProductCategory] ([ProductCategoryID], [ParentProductCategoryId], [Name])
```

(continues on next page)

(continued from previous page)

```
    VALUES (N'B', NULL, N'B Category');
    ALTER TABLE [ProductCategory] CHECK CONSTRAINT ALL
COMMIT TRAN;
```

As you can see the **A** *ProductCategory* belongs to **B** *ProductCategory* . But because **A** is before **B**, on a Select statement, the **A** row will come firstly, before **B**.

We have generated a **ForeignKey constraint failure**.

Note: For the explanation purpose, we are going to use the `OnApplyChangesConflictOccured` interceptor to see the error message on each row.

You can do the same using the `SyncOptions` `ErrorResolutionPolicy` property.

Here is the description of the `ErrorResolution` enum, used to define the error resolution, for this particular foreign key constraint failure:

ErrorResolution.Throw

The default ErrorResolution is `ErrorResolution.Throw`. This means that the sync will stop on the first error, and will throw an exception.

In the following example, we are going to see the error message on each row.

No need to specify `args.Resolution = ErrorResolution.Throw;` as it is the default value. We do it here to emphazise the error resolution behavior:

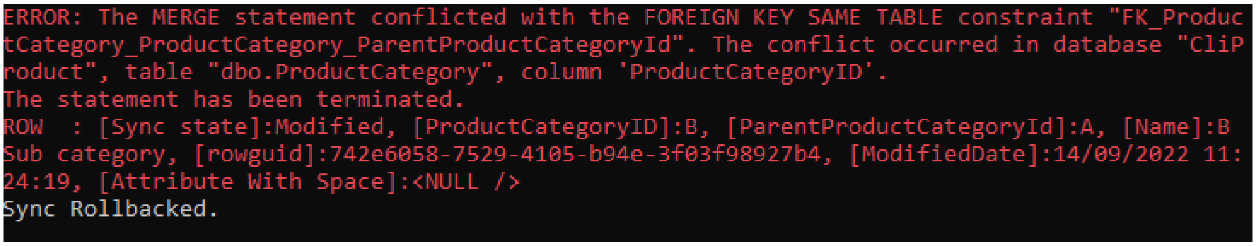
```
// ErrorResolution.Throw is the default resolution. No need to explicitly set it.
// It's done here for the demo explanation.
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW  : {args.ErrorRow}");
    Console.ResetColor();

    args.Resolution = ErrorResolution.Throw;
});

var serverProvider = new SqlSyncProvider(serverConnectionString)
{ UseBulkOperations = false };

var clientProvider = new SqlSyncProvider(clientConnectionString)
{ UseBulkOperations = false };

var setup = new SyncSetup("ProductCategory");
var agent = new SyncAgent(clientProvider, serverProvider);

do
{
    try
    {
        Console.ResetColor();
        result = await agent.SynchronizeAsync(setup);
```

(continues on next page)

(continued from previous page)

```
        Console.WriteLine(result);
        Console.WriteLine("Sync Ended. Press a key to start again, or Escapte to end
↪");
    }
    catch (Exception e)
    {
        Console.ResetColor();
        Console.WriteLine("Sync rolled back.");
    }
} while (Console.ReadKey().Key != ConsoleKey.Escape);
```

The error is raised for the second line, as it's the one who triggers the foreign key constraint failure:



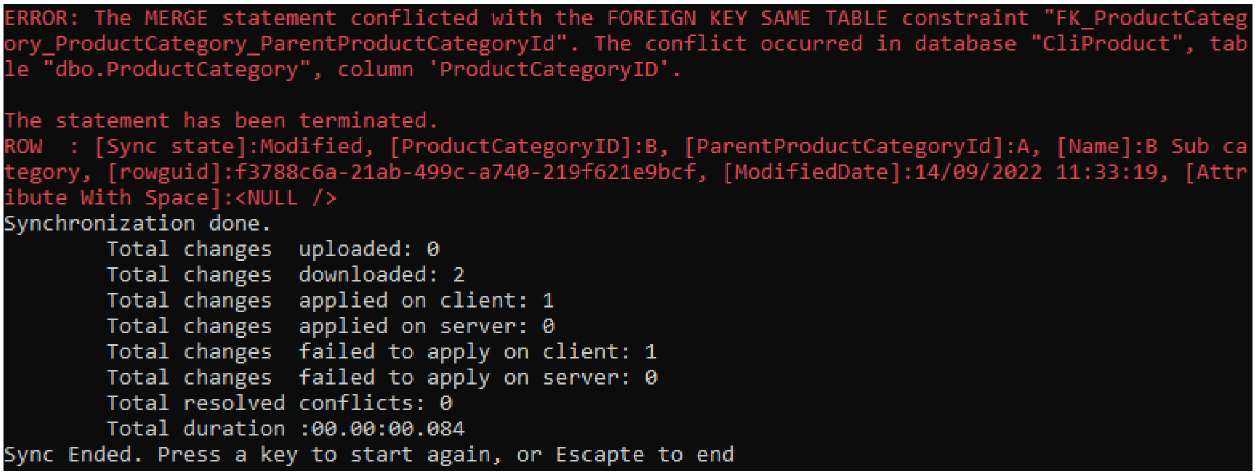
```
ERROR: The MERGE statement conflicted with the FOREIGN KEY SAME TABLE constraint "FK_Produc
tCategory_ProductCategory_ParentProductCategoryId". The conflict occurred in database "CliP
roduct", table "dbo.ProductCategory", column 'ProductCategoryID'.
The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:B, [ParentProductCategoryId]:A, [Name]:B
Sub category, [rowguid]:742e6058-7529-4105-b94e-3f03f98927b4, [ModifiedDate]:14/09/2022 11:
24:19, [Attribute With Space]:<NULL />
Sync Rollbacked.
```

Be careful, we do not have any files in the `BatchInfo` directory, as the sync has been rolled back.

ErrorResolution.ContinueOnError

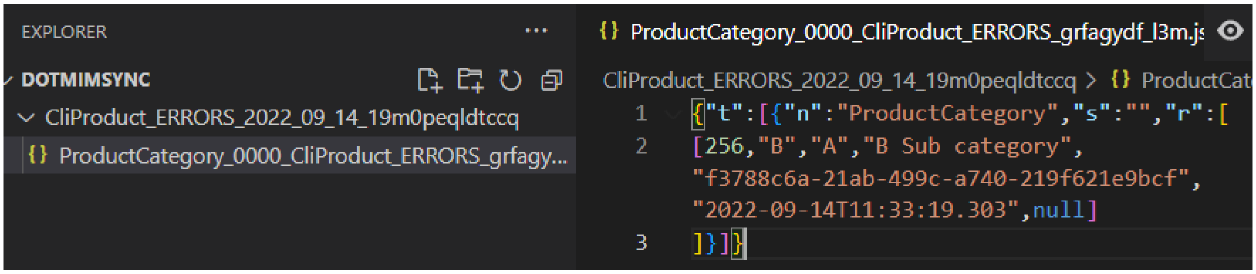
The `ErrorResolution.ContinueOnError` will continue the sync, and will not roll back the transaction. Error is logged in the error's batch info directory:

```
// ErrorResolution.Throw is the default resolution. No need to explicitly set it.
// It's done here for the demo explanation.
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW  : {args.ErrorRow}");
    Console.ResetColor();

    args.Resolution = ErrorResolution.ContinueOnError;
});
```



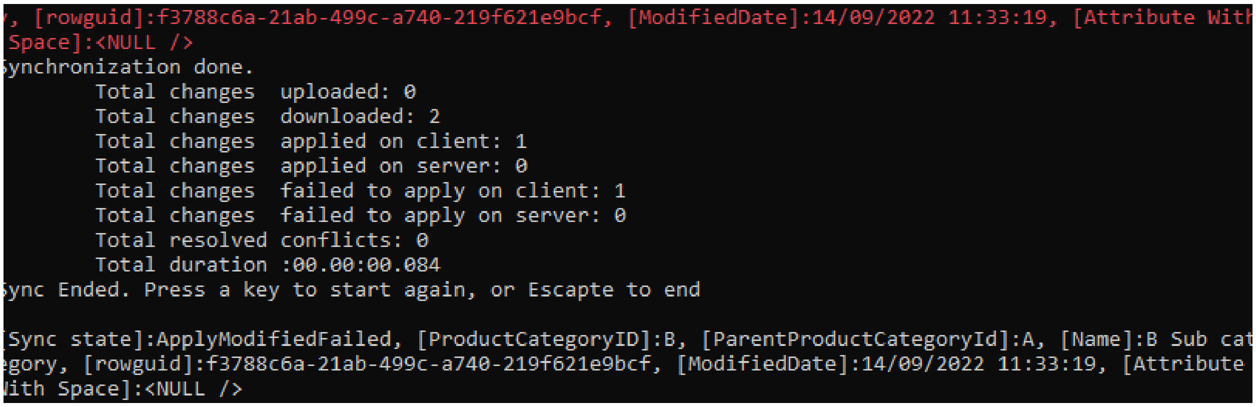
```
ERROR: The MERGE statement conflicted with the FOREIGN KEY SAME TABLE constraint "FK_ProductCateg
ory_ProductCategory_ParentProductCategoryId". The conflict occurred in database "CliProduct", tab
le "dbo.ProductCategory", column 'ProductCategoryID'.

The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:B, [ParentProductCategoryId]:A, [Name]:B Sub ca
tegory, [rowguid]:f3788c6a-21ab-499c-a740-219f621e9bcf, [ModifiedDate]:14/09/2022 11:33:19, [Attr
ibute With Space]:<NULL />
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 2
        Total changes  applied on client: 1
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 1
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.084
Sync Ended. Press a key to start again, or Escapte to end
```

The error's batch info directory contains the error file:


EXPLORER
DOTMIMSYNC
CliProduct_ERRORS_2022_09_14_19m0peqldtccq
{} ProductCategory_0000_CliProduct_ERRORS_grfagy...
{} ProductCategory_0000_CliProduct_ERRORS_grfagydf_l3m.js
CliProduct_ERRORS_2022_09_14_19m0peqldtccq > {} ProductCat

```
{"t":[{"n":"ProductCategory","s":"","r":[
[256,"B","A","B Sub category",
"f3788c6a-21ab-499c-a740-219f621e9bcf",
"2022-09-14T11:33:19.303",null]
]}]}
```

And you can get it using the `LoadBatchInfosAsync` method:



```
, [rowguid]:f3788c6a-21ab-499c-a740-219f621e9bcf, [ModifiedDate]:14/09/2022 11:33:19, [Attribute With
 Space]:<NULL />
ynchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 2
        Total changes  applied on client: 1
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 1
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.084
ync Ended. Press a key to start again, or Escapte to end

[Sync state]:ApplyModifiedFailed, [ProductCategoryID]:B, [ParentProductCategoryId]:A, [Name]:B Sub cat
egory, [rowguid]:f3788c6a-21ab-499c-a740-219f621e9bcf, [ModifiedDate]:14/09/2022 11:33:19, [Attribute
ith Space]:<NULL />
```

ErrorResolution.RetryOneMoreTimeAndThrowOnError

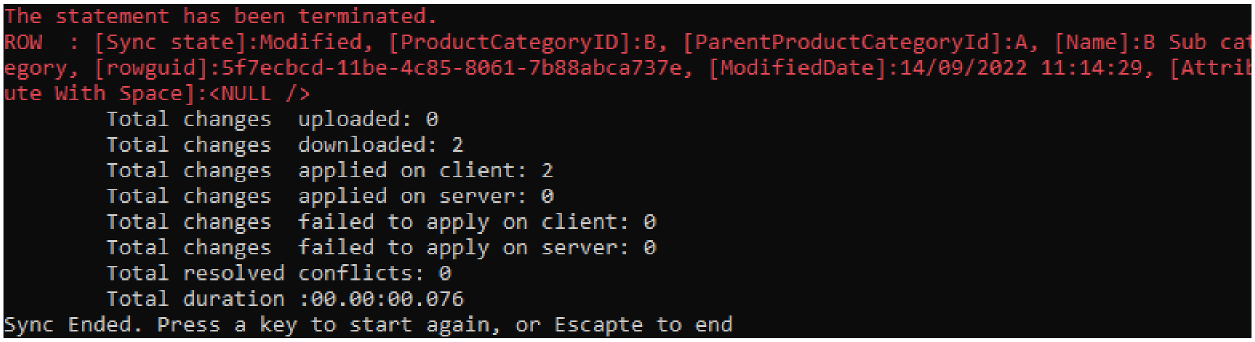
The `ErrorResolution.RetryOneMoreTimeAndThrowOnError` will retry the row one more time, and if it fails again, will throw an exception.

```
// ErrorResolution.Throw is the default resolution. No need to explicitly set it.
// It's done here for the demo explanation.
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW  : {args.ErrorRow}");
```

(continues on next page)

(continued from previous page)

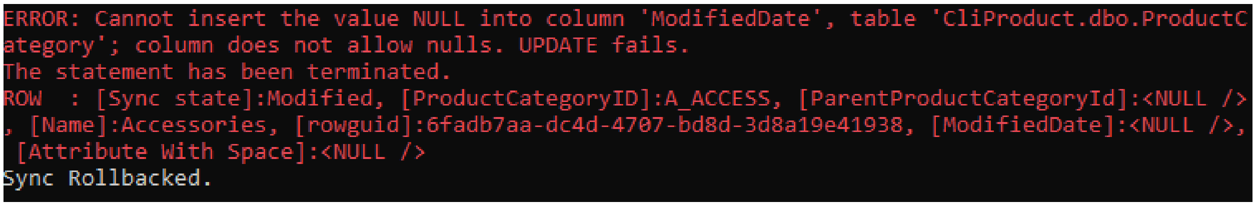
```
    Console.ResetColor();

    args.Resolution = ErrorResolution.RetryOneMoreTimeAndThrowOnError;
});
```



```
The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:B, [ParentProductCategoryId]:A, [Name]:B Sub cat
egory, [rowguid]:5f7ecbcd-11be-4c85-8061-7b88abca737e, [ModifiedDate]:14/09/2022 11:14:29, [Attrib
ute With Space]:<NULL />
        Total changes  uploaded: 0
        Total changes  downloaded: 2
        Total changes  applied on client: 2
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 0
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.076
Sync Ended. Press a key to start again, or Escapte to end
```

As you can see, we have an error raised **BUT** because we tried **AGAIN** to applied the failed row one more time, the sync has been successfully applied.

As a demo purpose, we are going to generate a new error (A **Not Null Constraint** error), on the *ProductCategory* table, to see the difference:



```
ERROR: Cannot insert the value NULL into column 'ModifiedDate', table 'CliProduct.dbo.ProductC
ategory'; column does not allow nulls. UPDATE fails.
The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:A_ACCESS, [ParentProductCategoryId]:<NULL />
, [Name]:Accessories, [rowguid]:6fadb7aa-dc4d-4707-bd8d-3d8a19e41938, [ModifiedDate]:<NULL />,
 [Attribute With Space]:<NULL />
Sync Rollbacked.
```

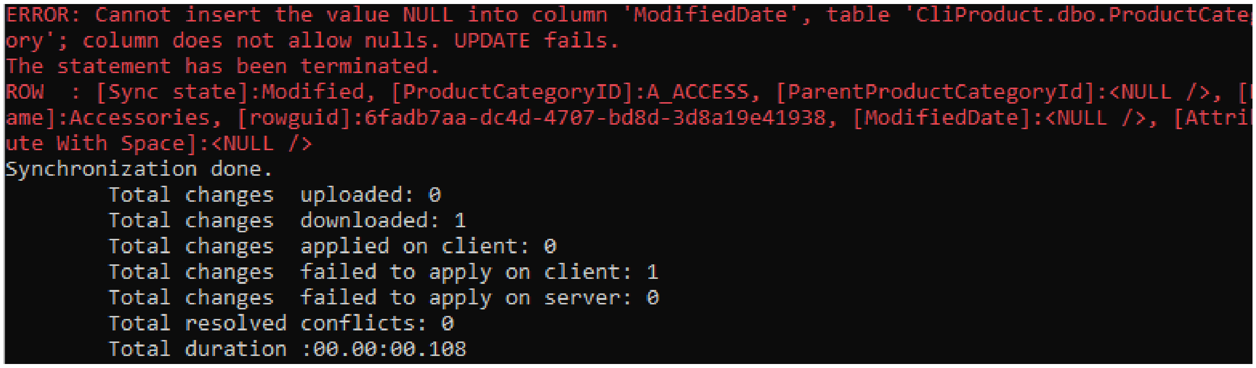
Ok, this time, the error can't be resolved, even if we tried to apply the row twice. So the sync has been rolled back, and the error has been raised.

ErrorResolution.RetryOneMoreTimeAndContinueOnError

The `ErrorResolution.RetryOneMoreTimeAndContinueOnError` will retry the row one more time, and if it fails again, will ignore the error and continue the sync.

Note: We are going to use the same **Not Null Constraint** error on the *ProductCategory* table, as we did in the previous example.

```
// ErrorResolution.Throw is the default resolution. No need to explicitly set it.
// It's done here for the demo explanation.
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW  : {args.ErrorRow}");
    Console.ResetColor();

    args.Resolution = ErrorResolution.RetryOneMoreTimeAndContinueOnError;
});
```



```
ERROR: Cannot insert the value NULL into column 'ModifiedDate', table 'CliProduct.dbo.ProductCate
ory'; column does not allow nulls. UPDATE fails.
The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:A_ACCESS, [ParentProductCategoryId]:<NULL />, [
ame]:Accessories, [rowguid]:6fadb7aa-dc4d-4707-bd8d-3d8a19e41938, [ModifiedDate]:<NULL />, [Attri
ute With Space]:<NULL />
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 1
        Total changes  applied on client: 0
        Total changes  failed to apply on client: 1
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.108
```

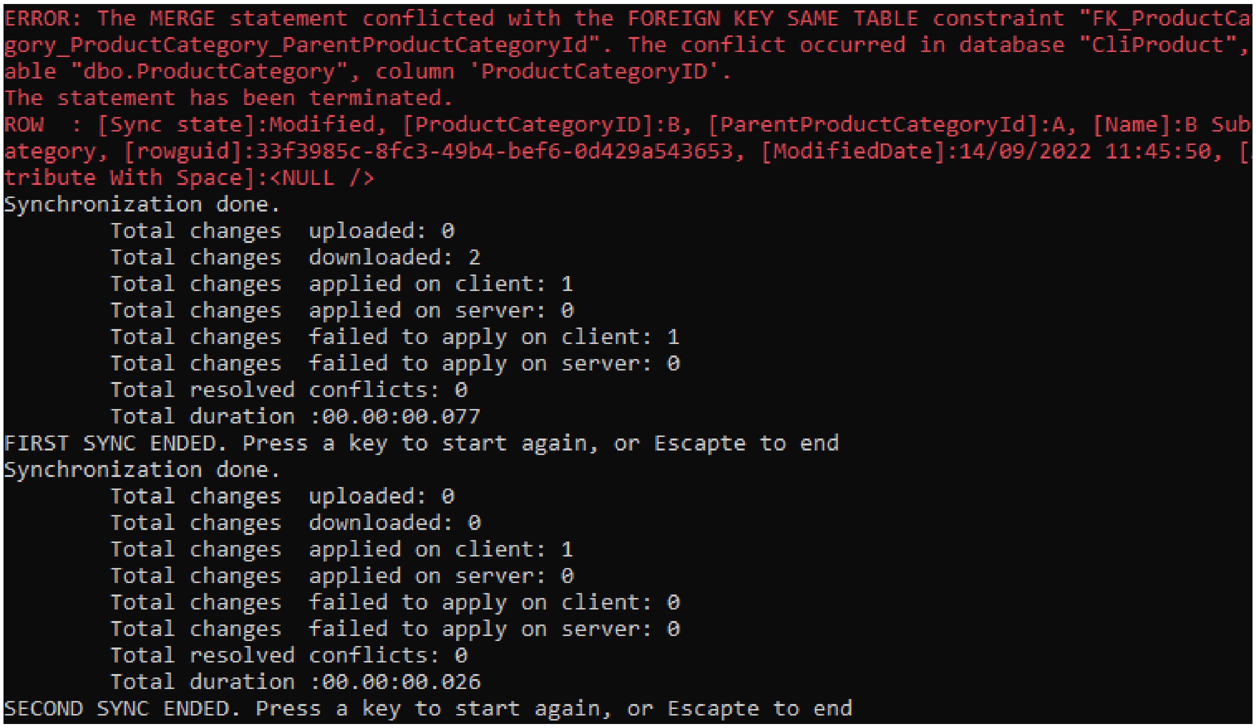
We tried to apply the row twice, and then the error has been ignored, and logged in the batch info directory.

ErrorResolution.RetryOnNextSync

Last option, the `ErrorResolution.RetryOnNextSync` will retry to apply the row on the next sync, and if it fails again, will try again and again, until the server (or client) resolves the issue.

As demo purpose we are chaining multiple call to `SynchronizeAsync` to see the error being retried on the next sync.

```
// ErrorResolution.Throw is the default resolution. No need to explicitly set it.
// It's done here for the demo explanation.
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW   : {args.ErrorRow}");
    Console.ResetColor();

    args.Resolution = ErrorResolution.RetryOnNextSync;
});
```



```
ERROR: The MERGE statement conflicted with the FOREIGN KEY SAME TABLE constraint "FK_ProductCa
gory_ProductCategory_ParentProductCategoryId". The conflict occurred in database "CliProduct",
able "dbo.ProductCategory", column 'ProductCategoryID'.
The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:B, [ParentProductCategoryId]:A, [Name]:B Sub
ategory, [rowguid]:33f3985c-8fc3-49b4-bef6-0d429a543653, [ModifiedDate]:14/09/2022 11:45:50, [
tribute With Space]:<NULL />
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 2
        Total changes  applied on client: 1
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 1
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.077
FIRST SYNC ENDED. Press a key to start again, or Escapte to end
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 0
        Total changes  applied on client: 1
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 0
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.026
SECOND SYNC ENDED. Press a key to start again, or Escapte to end
```

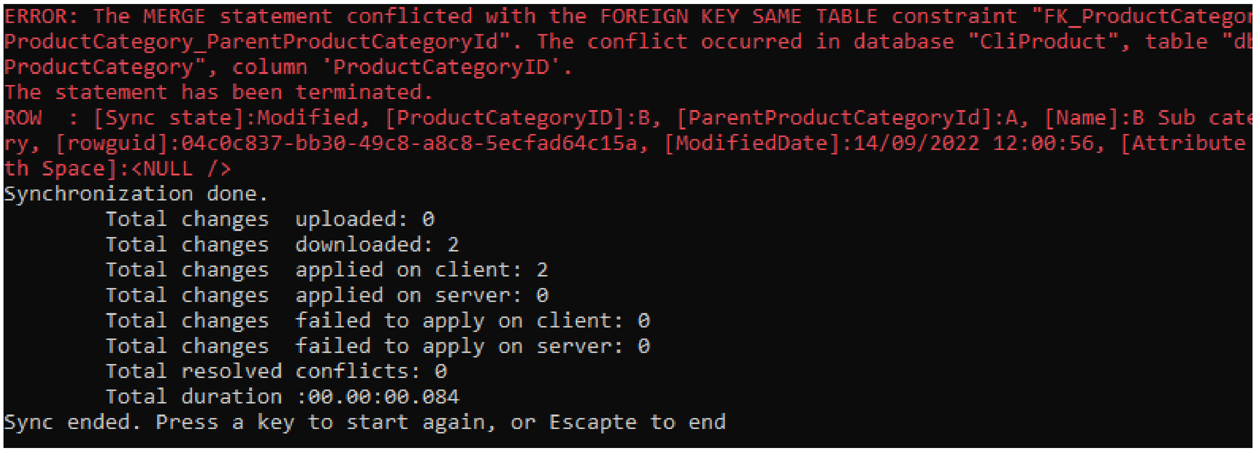
As you can see, the error has been logged in the batch info directory during the first sync, and has been automatically retried on the second sync.

Note: IF you look carefully the result on the second sync, you will see that we did not download any rows, but the error has been resolved.

ErrorResolution.Resolved

Last option, the `ErrorResolution.Resolved` will mark the error as resolved, and will not retry to apply the row on the next sync.

```
// ErrorResolution.Throw is the default resolution. No need to explicitly set it.
// It's done here for the demo explanation.
agent.LocalOrchestrator.OnApplyChangesErrorOccured(args =>
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"ERROR: {args.Exception.Message}");
    Console.WriteLine($"ROW  : {args.ErrorRow}");
    Console.ResetColor();

    args.Resolution = ErrorResolution.Resolved;
});
```



```
ERROR: The MERGE statement conflicted with the FOREIGN KEY SAME TABLE constraint "FK_ProductCategor
ProductCategory_ParentProductCategoryId". The conflict occurred in database "CliProduct", table "db
ProductCategory", column 'ProductCategoryID'.
The statement has been terminated.
ROW  : [Sync state]:Modified, [ProductCategoryID]:B, [ParentProductCategoryId]:A, [Name]:B Sub cate
ry, [rowguid]:04c0c837-bb30-49c8-a8c8-5ecfad64c15a, [ModifiedDate]:14/09/2022 12:00:56, [Attribute
th Space]:<NULL />
Synchronization done.
        Total changes  uploaded: 0
        Total changes  downloaded: 2
        Total changes  applied on client: 2
        Total changes  applied on server: 0
        Total changes  failed to apply on client: 0
        Total changes  failed to apply on server: 0
        Total resolved conflicts: 0
        Total duration :00.00:00.084
Sync ended. Press a key to start again, or Escapte to end
```

Warning: Be careful, if you use this resolution, you will have to manually resolve the error on the server (or client).

DMS will consider the row as successfully applied, and will not retry to apply it on the next sync. As you can see, we have a Total Changes Applied On Client equals to 2, but the server has only 1 row applied.

2.20 Filters

You can apply a filter on any table, even if the filtered column belongs to another table.

For instance, you can apply a filter on the **Customer** table, even if the filter is set on the **Address** table on the **City** column.

In a nutshell, adding a filter for a specific table requires:

- Creating a `SetupFilter` instance for this table (you can not have more than one `SetupFilter` per table)
- Creating a *[parameter]* with a type and optionally a default value.
- Creating a *[where]* condition to map the *[parameter]* and a column from your table.
- If your filtered table is not the base table, you will have to specify one or more *[joins]* methods to reach the base filtered table.

2.20.1 Simple Filter

Note: You will find a complete sample here : Simple Filter sample

You have a straightforward method to add a filter, derivated from your `SyncSetup` instance:

```
setup.Filters.Add("Customer", "CustomerID");
```

Basically, this method will add a filter on the `Customer` table, based on the `CustomerID` column.

Internally, this method will:

- Creates a `SetupFilter` instance for the table `Customer`.

- Creates a *Parameter* called `CustomerID` that will have the same type as the `CustomerID` column from the `Customer` table.
- Creates a *Where* condition where the `CustomerID` *parameter* will be compared to the `CustomerID` column from the `Customer` table.

Since you are creating a filter based on a table and a column existing in your `SyncSetup`, you don't have to specify type, joins and where clauses.

Here is another way to create this simple filter:

```
var filter = new SetupFilter("Customer");
// Add a column as parameter. This column will be automaticaly added in the tracking table
filter.AddParameter("CustomerID", "Customer");
// add the side where expression, mapping the parameter to the column
filter.AddWhere("CustomerID", "Customer", "CustomerID");
// add this filter to setup
setup.Filters.Add(filter);
```

This code is a little bit more verbose, but is a little bit more flexible in some circumstances

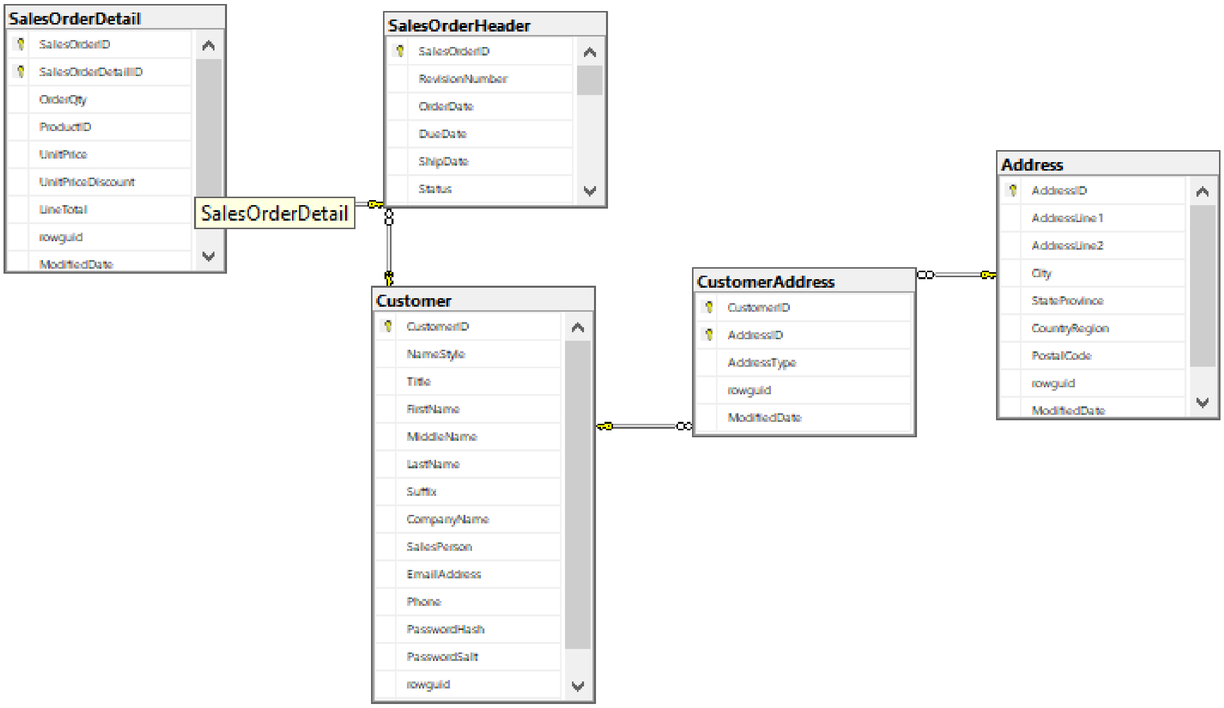
2.20.2 Complex Filter

Note: You will find a complete sample here : Complex Filter sample

Usually, you have more than one filter, especially if you have foreign keys in between. So far, you will have to manage the links between all your filtered tables.

To illustrate how it works, here is a straightforward scenario:

1) We Want only **Customers** from a specific **City** and a specific **Postal code**.
2) Each customer has **Addresses** and **Sales Orders** which should be filtered as well.



We will have to filter on each level:

- Level zero: **Address**
- Level one: **CustomerAddress**
- Level two: **Customer**, **SalesOrderHeader**
- Level four: **SalesOrderDetail**

The main difference with the *easy way* method, is that we will details all the methods on the `SetupFilter` to create a fully customized filter.

The `SetupFilter` class

The `SetupFilter` class will allows you to personalize your filter on a defined table (`Customer` in this example):

```
var customerFilter = new SetupFilter("Customer");
```

> **Warning:** Be careful, you can have only **one** `SetupFilter` instance per table. Obviously, this instance will allow you to define multiple parameters / criterias!

The `.AddParameter()` method

Allows you to add a new parameter to the `_changes` stored procedure.

This method can be called with two kind of arguments:

- Your parameter is a **custom** parameter. You have to define its name and its `DbType`. Optionally, you can define if it can be null and its default value (SQL Server only)

- Your parameter is a **mapped** column. Easier, you just have to define its name and the mapped column. This way, `Dotmim.Sync` will determine the parameter properties, based on the schema

For instance, the parameters declaration for the table `Customer` looks like:

```
customerFilter.AddParameter("City", "Address", true);
customerFilter.AddParameter("postal", DbType.String, true, null, 20);
```

- `City` parameter is defined from the `Address.City` column.
- `postal` parameter is a **custom** defined parameter.
  - *Indeed we have a ‘‘PostalCode‘‘ column in the ‘‘Address‘‘ table, that could be used here. But we will use a custom parameter instead, for the example*

At the end, the generation code should looks like:

```
ALTER PROCEDURE [dbo].[sCustomerAddress_Citypostal__changes]
    @sync_min_timestamp bigint,
    @sync_scope_id uniqueidentifier,
    @City varchar(MAX) NULL,
    @postal nvarchar(20) NULL
```

Where `@City` is a mapped parameter and `@postal` is a custom parameter.

The `.AddJoin()` method

If your filter is applied on a column in the actual table, you don't need to add any `join` statement.

But, in our example, the `Customer` table is two levels below the `Address` table (where we have the filtered columns `City` and `PostalCode`)

So far, we can add some join statement here, going from `Customer` to `CustomerAddress` then to `Address`:

```
customerFilter.AddJoin(Join.Left, "CustomerAddress")
  .On("CustomerAddress", "CustomerId", "Customer", "CustomerId");

customerFilter.AddJoin(Join.Left, "Address")
  .On("CustomerAddress", "AddressId", "Address", "AddressId");
```

The generated statement now looks like:

```
FROM [Customer] [base]
RIGHT JOIN [tCustomer] [side]ON [base].[CustomerID] = [side].[CustomerID]
LEFT JOIN [CustomerAddress] ON [CustomerAddress].[CustomerId] = [base].[CustomerId]
LEFT JOIN [Address] ON [CustomerAddress].[AddressId] = [Address].[AddressId]
```

As you can see **DMS** will take care of quoted table / column names and aliases in the stored procedure.

Just focus on the name of your table.

The `.AddWhere()` method

Now, and for each parameter, you will have to define the where condition.

Each parameter will be compare to an existing column in an existing table.

For instance:

- The `City` parameter should be compared to the `City` column in the `Address` table.

- The `postal` parameter should be compared to the `PostalCode` column in the `Address` table:

```
// Mapping City parameter to Address.City column
addressFilter.AddWhere("City", "Address", "City");
// Mapping the custom "postal" parameter to Address.PostalCode
addressFilter.AddWhere("PostalCode", "Address", "postal");
```

The generated sql statement looks like this:

```
WHERE (
(
 (
   ([Address].[City] = @City OR @City IS NULL) AND ([Address].[PostalCode] = @postal␣
↪OR @postal IS NULL)
  )
 OR [side].[sync_row_is_tombstone] = 1
)
```

The `.AddCustomWhere()` method

If you need more, this method will allow you to add your own `where` condition.

Be careful, this method takes a `string` as argument, which will not be parsed, but instead, just added at the end of the stored procedure statement.

> **Warning:** If you are using the AddCustomWhere method, you **NEED** to handle deleted rows.

Using the `AddCustomWhere` method allows you to do *whatever you want* with the `Where` clause in the select changes.

For instance, here is the code that is generated using a `AddCustomWhere` clause:

```
var filter = new SetupFilter("SalesOrderDetail");
filter.AddParameter("OrderQty", System.Data.DbType.Int16);
filter.AddCustomWhere("{{{OrderQty}}} = @OrderQty");
```

```
SELECT DISTINCT ..............
WHERE (
(
    [OrderQty] = @OrderQty
)
AND
    [side].[timestamp] > @sync_min_timestamp
    AND ([side].[update_scope_id] <> @sync_scope_id OR [side].[update_scope_id] IS␣
↪NULL)
)
```

Note: The {{{ and }}} characters are used to escape the column `OrderQty`, and will be replaced by the escaper character of the database engine.

- For **SQL Server** and **SQLite** it will be `[` and `]`
- For **MySQL** and **MariaDB** it will be `
- For **Postgres**, it will be `"`

The problem here is pretty simple.

1) When you are deleting a row, the tracking table marks the row as deleted (`sync_row_is_tombstone = 1`)
2) Your row is not existing anymore in the `SalesOrderDetail` table.
3) If you are not handling this situation, this deleted row will never been selected for sync, because of your `where` custom clause . . .

Fortunately for us, we have a pretty simple workaround: Add a **custom condition** to also **retrieve deleted rows** in your custom where clause.

How to get deleted rows in your Where clause ?

Basically, all the deleted rows are stored in the tracking table.

- This tracking table is *aliased* and should be called in your clause with the alias `side`.
- Each row marked as deleted has a **bit** flag called `sync_row_is_tombstone` set to **1**.

You don't have to care about any timeline, since it's done automatically in the rest of the generated **SQL** statement.

That being said, you have eventually to add `OR side.sync_row_is_tombstone = 1` to your `AddCustomWhere` clause.

Here is the good `AddCustomWhere` method where deleted rows are handled correctly:

```
var filter = new SetupFilter("SalesOrderDetail");
filter.AddParameter("OrderQty", System.Data.DbType.Int16);
filter.AddCustomWhere("{{{OrderQty}}} = @OrderQty OR {{{side}}}.{{{sync_row_is_
↪tombstone}}} = 1");
setup.Filters.Add(filter);
```

2.20.3 Complete Sample

Here is the full sample, where we have defined the filters (`City` and `postal` code) on each filtered tables: `Customer`, `CustomerAddress`, `Address`, `SalesOrderHeader` and `SalesOrderDetail`

You will find the source code in the last commit, project `Dotmim.Sync.SampleConsole.csproj`, file `program.cs`, method `SynchronizeAsync()`:

```
var setup = new SyncSetup(new string[] {"ProductCategory",
    "ProductModel", "Product",
    "Address", "Customer", "CustomerAddress",
    "SalesOrderHeader", "SalesOrderDetail" });

// ----------------------------------------------------
// Horizontal Filter: On rows. Removing rows from source
// ----------------------------------------------------
// Over all filter : "we Want only customer from specific city and specific postal␣
↪code"
// First level table : Address
// Second level tables : CustomerAddress
// Third level tables : Customer, SalesOrderHeader
// Fourth level tables : SalesOrderDetail
```

(continues on next page)

(continued from previous page)

```
// Create a filter on table Address on City Washington
// Optional : Sub filter on PostalCode, for testing purpose
var addressFilter = new SetupFilter("Address");

// For each filter, you have to provider all the input parameters
// A parameter could be a parameter mapped to an existing colum :
// That way you don't have to specify any type, length and so on ...
// We can specify if a null value can be passed as parameter value :
// That way ALL addresses will be fetched
// A default value can be passed as well, but works only on SQL Server (MySql is a␣
↪damn ... thing)
addressFilter.AddParameter("City", "Address", true);

// Or a parameter could be a random parameter bound to anything.
// In that case, you have to specify everything
// (This parameter COULD BE bound to a column, like City,
//  but for the example, we go for a custom parameter)
addressFilter.AddParameter("postal", DbType.String, true, null, 20);

// Then you map each parameter on wich table / column the "where" clause should be␣
↪applied
addressFilter.AddWhere("City", "Address", "City");
addressFilter.AddWhere("PostalCode", "Address", "postal");
setup.Filters.Add(addressFilter);

var addressCustomerFilter = new SetupFilter("CustomerAddress");
addressCustomerFilter.AddParameter("City", "Address", true);
addressCustomerFilter.AddParameter("postal", DbType.String, true, null, 20);

// You can join table to go from your table up (or down) to your filter table
addressCustomerFilter.AddJoin(Join.Left, "Address")
    .On("CustomerAddress", "AddressId", "Address", "AddressId");

// And then add your where clauses
addressCustomerFilter.AddWhere("City", "Address", "City");
addressCustomerFilter.AddWhere("PostalCode", "Address", "postal");
setup.Filters.Add(addressCustomerFilter);

var customerFilter = new SetupFilter("Customer");
customerFilter.AddParameter("City", "Address", true);
customerFilter.AddParameter("postal", DbType.String, true, null, 20);
customerFilter.AddJoin(Join.Left, "CustomerAddress")
    .On("CustomerAddress", "CustomerId", "Customer", "CustomerId");
customerFilter.AddJoin(Join.Left, "Address")
    .On("CustomerAddress", "AddressId", "Address", "AddressId");
customerFilter.AddWhere("City", "Address", "City");
customerFilter.AddWhere("PostalCode", "Address", "postal");
setup.Filters.Add(customerFilter);

var orderHeaderFilter = new SetupFilter("SalesOrderHeader");
orderHeaderFilter.AddParameter("City", "Address", true);
orderHeaderFilter.AddParameter("postal", DbType.String, true, null, 20);
orderHeaderFilter.AddJoin(Join.Left, "CustomerAddress")
    .On("CustomerAddress", "CustomerId", "SalesOrderHeader", "CustomerId");
orderHeaderFilter.AddJoin(Join.Left, "Address")
    .On("CustomerAddress", "AddressId", "Address", "AddressId");
orderHeaderFilter.AddWhere("City", "Address", "City");
```

(continues on next page)

(continued from previous page)

```
orderHeaderFilter.AddWhere("PostalCode", "Address", "postal");
setup.Filters.Add(orderHeaderFilter);

var orderDetailsFilter = new SetupFilter("SalesOrderDetail");
orderDetailsFilter.AddParameter("City", "Address", true);
orderDetailsFilter.AddParameter("postal", DbType.String, true, null, 20);
orderDetailsFilter.AddJoin(Join.Left, "SalesOrderHeader")
    .On("SalesOrderHeader", "SalesOrderID", "SalesOrderDetail", "SalesOrderID");
orderDetailsFilter.AddJoin(Join.Left, "CustomerAddress")
    .On("CustomerAddress", "CustomerId", "SalesOrderHeader", "CustomerId");
orderDetailsFilter.AddJoin(Join.Left, "Address")
    .On("CustomerAddress", "AddressId", "Address", "AddressId");
orderDetailsFilter.AddWhere("City", "Address", "City");
orderDetailsFilter.AddWhere("PostalCode", "Address", "postal");
setup.Filters.Add(orderDetailsFilter);

// ----------------------------------------------------
```

And you `SyncAgent` now looks like:

```
// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, serverProvider);

// Adding 2 parameters
// Because I've specified that "postal" could be null,
// I can set the value to DBNull.Value (and the get all postal code in Toronto city)
var parameters = new SyncParameters
{
    { "City", new Guid("Toronto") },
    { "postal", DBNull.Value }
};

// [Optional]: Get some progress event during the sync process
var progress = new SynchronousProgress<ProgressArgs>(
    pa => Console.WriteLine('$'"{pa.PogressPercentageString}\t {pa.Message}"));

var s1 = await agent.SynchronizeAsync(setup, parameters, progress);
```

2.20.4 Http mode

Note: You will find a complete sample here : Complex Web Filter sample

If you're using the http mode, you will notice some differences between the **client side** and the **server side**:

- The **server side** will declare the filters.
- The **client side** will declare the paramaters.

Server side

You have to declare your `SetupFilters` from within your `ConfigureServices()` method.

Pretty similar from the last example, excepting you do not add any `SyncParameter` value at the end:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDistributedMemoryCache();
    services.AddSession(options => options.IdleTimeout = TimeSpan.FromMinutes(30));

    // Get a connection string for your server data source
    var connectionString = Configuration.GetSection("ConnectionStrings")[
↪"DefaultConnection"];

    // Set the web server Options
    var options = new SyncOptions
    {
        BatchDirectory = Path.Combine(SyncOptions.GetDefaultUserBatchDiretory(),
↪"server")
    };


    // Create the setup used for your sync process

    var setup = new SyncSetup("ProductCategory", "ProductModel", "Product",
                              "Address", "Customer", "CustomerAddress",
                              "SalesOrderHeader", "SalesOrderDetail")
    {
        StoredProceduresPrefix = "s",
        StoredProceduresSuffix = "",
        TrackingTablesPrefix = "s",
        TrackingTablesSuffix = ""
    };

    // Create a filter on table Address on City Washington
    // Optional : Sub filter on PostalCode, for testing purpose
    var addressFilter = new SetupFilter("Address");
    addressFilter.AddParameter("City", "Address", true);
    addressFilter.AddParameter("postal", DbType.String, true, null, 20);
    addressFilter.AddWhere("City", "Address", "City");
    addressFilter.AddWhere("PostalCode", "Address", "postal");
    setup.Filters.Add(addressFilter);

    var addressCustomerFilter = new SetupFilter("CustomerAddress");
    addressCustomerFilter.AddParameter("City", "Address", true);
    addressCustomerFilter.AddParameter("postal", DbType.String, true, null, 20);
    addressCustomerFilter.AddJoin(Join.Left, "Address")
        .On("CustomerAddress", "AddressId", "Address", "AddressId");
    addressCustomerFilter.AddWhere("City", "Address", "City");
    addressCustomerFilter.AddWhere("PostalCode", "Address", "postal");
    setup.Filters.Add(addressCustomerFilter);

    var customerFilter = new SetupFilter("Customer");
    customerFilter.AddParameter("City", "Address", true);
    customerFilter.AddParameter("postal", DbType.String, true, null, 20);
    customerFilter.AddJoin(Join.Left, "CustomerAddress")
        .On("CustomerAddress", "CustomerId", "Customer", "CustomerId");
    customerFilter.AddJoin(Join.Left, "Address")
        .On("CustomerAddress", "AddressId", "Address", "AddressId");
    customerFilter.AddWhere("City", "Address", "City");
```

(continues on next page)

(continued from previous page)

```
    customerFilter.AddWhere("PostalCode", "Address", "postal");
    setup.Filters.Add(customerFilter);

    var orderHeaderFilter = new SetupFilter("SalesOrderHeader");
    orderHeaderFilter.AddParameter("City", "Address", true);
    orderHeaderFilter.AddParameter("postal", DbType.String, true, null, 20);
    orderHeaderFilter.AddJoin(Join.Left, "CustomerAddress")
        .On("CustomerAddress", "CustomerId", "SalesOrderHeader", "CustomerId");
    orderHeaderFilter.AddJoin(Join.Left, "Address")
        .On("CustomerAddress", "AddressId", "Address", "AddressId");
    orderHeaderFilter.AddWhere("City", "Address", "City");
    orderHeaderFilter.AddWhere("PostalCode", "Address", "postal");
    setup.Filters.Add(orderHeaderFilter);

    var orderDetailsFilter = new SetupFilter("SalesOrderDetail");
    orderDetailsFilter.AddParameter("City", "Address", true);
    orderDetailsFilter.AddParameter("postal", DbType.String, true, null, 20);
    orderDetailsFilter.AddJoin(Join.Left, "SalesOrderHeader")
        .On("SalesOrderHeader", "SalesOrderID", "SalesOrderDetail", "SalesOrderID");
    orderDetailsFilter.AddJoin(Join.Left, "CustomerAddress")
        .On("CustomerAddress", "CustomerId", "SalesOrderHeader", "CustomerId");
    orderDetailsFilter.AddJoin(Join.Left, "Address")
        .On("CustomerAddress", "AddressId", "Address", "AddressId");
    orderDetailsFilter.AddWhere("City", "Address", "City");
    orderDetailsFilter.AddWhere("PostalCode", "Address", "postal");
    setup.Filters.Add(orderDetailsFilter);

    // add a SqlSyncProvider acting as the server hub
    services.AddSyncServer<SqlSyncProvider>(connectionString, setup, options);
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
        app.UseDeveloperExceptionPage();

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseSession();
    app.UseEndpoints(endpoints => endpoints.MapControllers());
}
```

Client side

The client side shoud be familiar to you:

```
// Defining the local provider
var clientProvider = new SqlSyncProvider(DbHelper.
↪GetDatabaseConnectionString(clientDbName));

// Replacing a classic remote orchestrator
// with a web proxy orchestrator that point on the web api
var proxyClientProvider = new WebRemoteOrchestrator("http://localhost:52288/api/Sync
↪");
```

(continues on next page)

(continued from previous page)

```
// Set the web server Options
var options = new SyncOptions
{
    BatchDirectory = Path.Combine(SyncOptions.GetDefaultUserBatchDiretory(), "client")
};

// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, proxyClientProvider, options);

// [Optional]: Get some progress event during the sync process
var progress = new SynchronousProgress<ProgressArgs>(
   pa => Console.WriteLine($"{pa.ProgressPercentage:p}\t {pa.Message}"));

// Adding 2 parameters
// Because I've specified that "postal" could be null,
// I can set the value to DBNull.Value (and the get all postal code in Toronto city)
var parameters = new SyncParameters
{
    { "City", new Guid("Toronto") },
    { "postal", DBNull.Value }
};

var s1 = await agent.SynchronizeAsync(parameters, progress);
```

2.21 Sqlite Encryption

2.21.1 Overview

- **SQLite** doesn't support encrypting database files by default.
- Instead, we need to use a modified version of SQLite like SEE , SQLCipher , SQLiteCrypt , or wxSQLite3 .
- This article demonstrates using an unsupported, open-source build of **SQLCipher**, but the information also applies to other solutions since they generally follow the same pattern.

Hint: You will find more information about Sqlite Encryption with **Microsoft.Data.Sqlite** Here .

Hint: You will find the sqlite encryption sample here : Sqlite Encryption Sample

2.21.2 Tweak the nuget packages

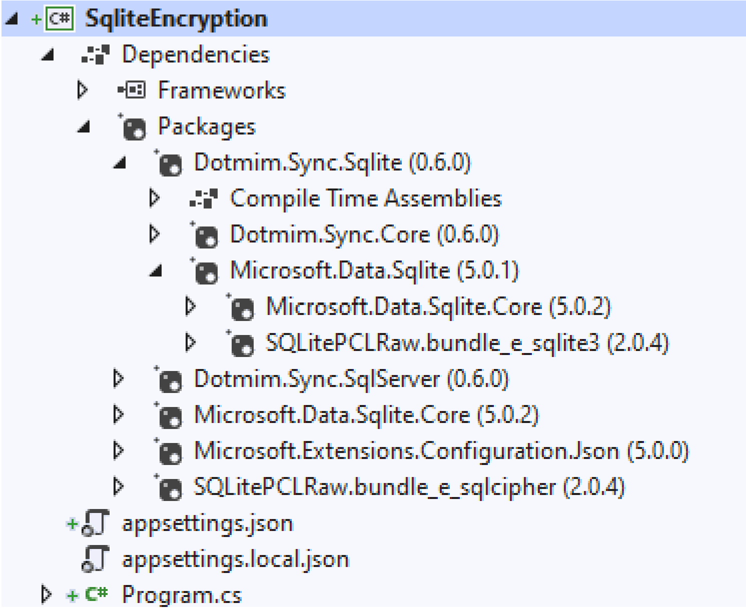
Basically, installing the packages needed to use Sqlite encryption is pretty simple. Just override packages:

```
dotnet add package Microsoft.Data.Sqlite.Core
dotnet add package SQLitePCLRaw.bundle_e_sqlcipher
```

Your project file should be something like this:

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>netcoreapp3.1</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Dotmim.Sync.Sqlite" Version="0.6.0" />
        <PackageReference Include="Microsoft.Data.Sqlite.Core" Version="5.0.2" />
        <PackageReference Include="SQLitePCLRaw.bundle_e_sqlcipher" Version="2.0.4" />
    </ItemGroup>
</Project>
```

Here is a screenshot of Visual Studio, after installing the packages:



- SqliteEncryption
  - Dependencies
    - Frameworks
    - Packages
      - Dotmim.Sync.Sqlite (0.6.0)
        - Compile Time Assemblies
        - Dotmim.Sync.Core (0.6.0)
        - Microsoft.Data.Sqlite (5.0.1)
          - Microsoft.Data.Sqlite.Core (5.0.2)
          - SQLitePCLRaw.bundle_e_sqlite3 (2.0.4)
      - Dotmim.Sync.SqlServer (0.6.0)
      - Microsoft.Data.Sqlite.Core (5.0.2)
      - Microsoft.Extensions.Configuration.Json (5.0.0)
      - SQLitePCLRaw.bundle_e_sqlcipher (2.0.4)
  - appsettings.json
  - appsettings.local.json
  - Program.cs

- As you can see, the `Dotmim.Sync.Sqlite` is referencing the `Microsoft.Data.Sqlite` package that is referencing `Microsoft.Data.Sqlite.Core` and `SQLitePCLRaw.bundle_e_sqlite3`.
- Because we made references at the root level of `Microsoft.Data.Sqlite.Core` and `SQLitePCLRaw.bundle_e_sqlcipher`, these two packages will be used in place of the `Microsoft.Data.Sqlite`'s packages.

2.21.3 Code

The code is prett much the same code, just ensure you're filling a **Password** in your **Sqlite** connection string:

```
// connection string should be something like "Data Source=AdventureWorks.db;
↪Password=..."
var sqliteConnectionString = configuration.GetConnectionString("SqliteConnection");
var clientProvider = new SqliteSyncProvider(sqliteConnectionString);

// You can use a SqliteConnectionStringBuilder() as well, like this:
//var builder = new SqliteConnectionStringBuilder();
//builder.DataSource = "AdventureWorks.db";
//builder.Password = "...";
```

2.22 Tables & Rows already existing

How to handle existing **clients** databases, with **existing** rows…

2.22.1 Default behavior

Before going further let's see the default behavior of `DMS`, regarding this particular scenario where you have existing rows in your client databases:

Basically, `DMS` will not take care of any existing client rows. On the first sync, these rows will stay on the client and will not be uploaded to the server (On the other part, of course the server rows will be downloaded to the client)

(Obviously, after this first sync, if you are updating locally any of these existing rows, they will be handled on the next sync)

The reason behind this behavior is to fit the scenario where you want to use a client database with some pre-existing rows (for example a server backup downloaded to the client ?) and where you don't wan't to upload them to the server (because they are already existing on the server)

Now, we can have a second scenario where you actually want to upload these pre-existing rows.

For this scenario, you have a special method, available on the `LocalOrchestrator` only, called `UpdateUntrackedRowsAsync` that will mark all non tracked rows for the next sync.

2.22.2 UpdateUntrackedRowsAsync

Note: You will find a complete sample here : Already Existing rows

Assuming you have a client database with some pre-existing rows and before going further, be sure that your server and client table has the same schema.

The workflow to handle these lines is:

- **Make a first sync, to be sure we have all the required metadata locally (tracking tables, triggers, stored proc …)**
  - During this first sync, you will download the server rows as well.
- Call the `UpdateUntrackedRowsAsync` method to mark all non tracked client rows.
- Make a second sync to upload these rows to server.

Here is a small sample, following this workflow:

```
// Tables involved in the sync process:
var setup = new SyncSetup("ServiceTickets");

// Creating an agent that will handle all the process
var agent = new SyncAgent(clientProvider, serverProvider);

// Launch the sync process
// This first sync will create all the sync architecture
// and will get the server rows
var s1 = await agent.SynchronizeAsync(setup);

// This first sync did not upload the client rows.
```

(continues on next page)

(continued from previous page)

```
// We only have rows from server that have been downloaded
// The important step here, done by 1st Sync,
// is to have setup everything locally (triggers / tracking tables ...)
Console.WriteLine(s1);

// Now we can "mark" original clients rows as "to be uploaded"
await agent.LocalOrchestrator.UpdateUntrackedRowsAsync();

// Then we can make a new synchronize to upload these rows to server
// Launch the sync process
var s2 = await agent.SynchronizeAsync();
Console.WriteLine(s2);
```

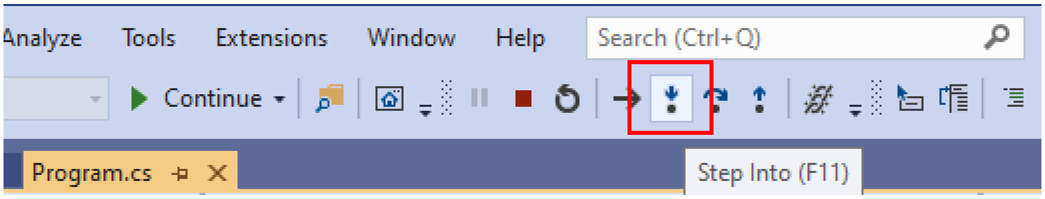
2.23 Debugging DMS

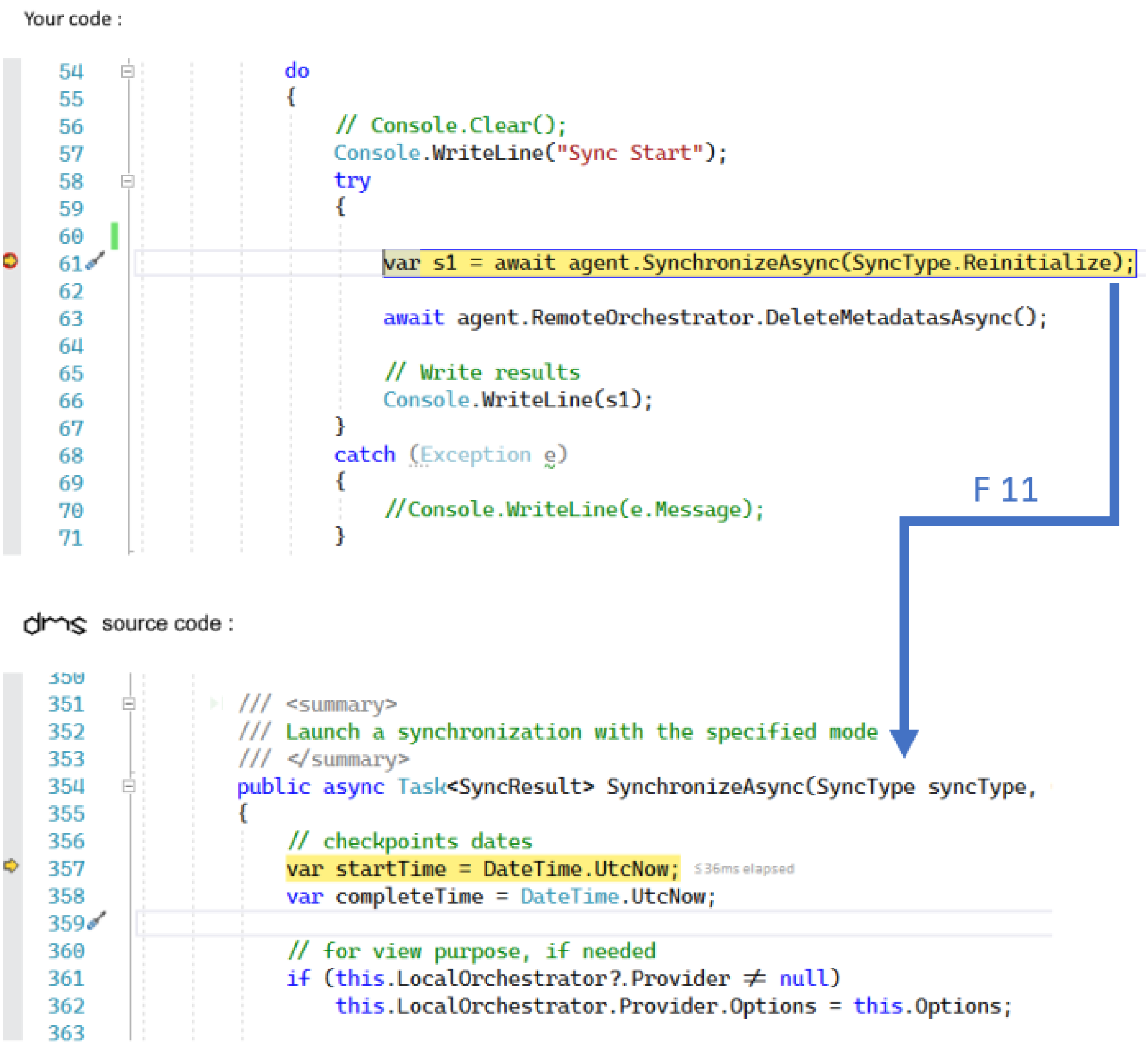
Thanks to **Symbol files** and **Source Link**, we're able to debug the **DMS** framework from within our application, without having to donwload the source code locally.

To be able to **Step Into** the code, we should configure **Visual Studio** to:

- Download the symbol files from nuget if available.
- Enable the source link to redirect the pdb information to the source code hosted on Github.

Once we've correctly configured our Visual Studio environment, we are able to **Step Into** the **DMS** code, during a debugging session (or press **F11**):





Your code :

```
do
{
    // Console.Clear();
    Console.WriteLine("Sync Start");
    try
    {

        var s1 = await agent.SynchronizeAsync(SyncType.Reinitialize);

        await agent.RemoteOrchestrator.DeleteMetadatasAsync();

        // Write results
        Console.WriteLine(s1);
    }
    catch (Exception e)
    {
        //Console.WriteLine(e.Message);
    }
```

dms source code :

```
/// <summary>
/// Launch a synchronization with the specified mode
/// </summary>
public async Task<SyncResult> SynchronizeAsync(SyncType syncType,
{
    // checkpoints dates
    var startTime = DateTime.UtcNow;
    var completeTime = DateTime.UtcNow;

    // for view purpose, if needed
    if (this.LocalOrchestrator?.Provider ≠ null)
        this.LocalOrchestrator.Provider.Options = this.Options;
```

As you can see in the previous screenshot, we are actually *step into* the `SynchronizeAsync` method directly from your code.

Behinds the scene, the **.pdb file** retrieves the correct filename and position and the **Source link** download the correct file from the DMS Github repository .

Let's see in details how to configure your Visual Studio environment:

2.23.1 Symbols packages

Symbol files (*.pdb) are produced by the .NET compiler alongside assemblies. **Symbol files** map execution locations to the original source code so you can step through source code as it is running using a debugger.

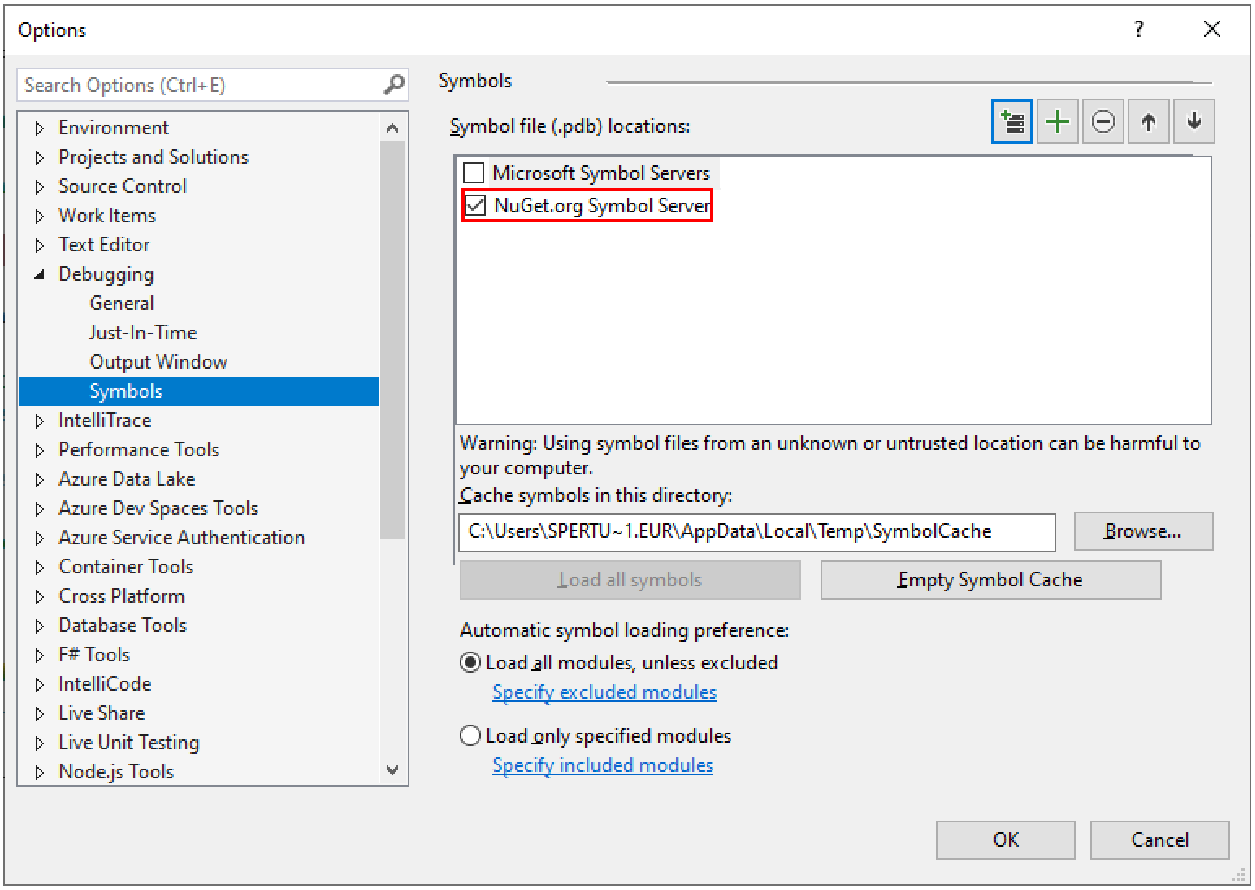
DMS publishes the symbols packages, containing the .pdb files, for each release to the nuget symbols server.

To be able to get the symbols, we should check we have **Nuget** as a symbol server availabe from our Visual Studio options:

Go to **Tools** > **Options** > **Debugging** > **Symbols**:

- Verify we have **Nuget.Org Symbol** Servers checked.
- Uncheck **Microsoft Symbol Servers**, unless we want also to debug the .NET Core assemblies from within our application.

Hint: If you don't have the Nuget.Org Symbol option, you can add this url directly : https://symbols.nuget.org/download/symbols



Symbol file (.pdb) locations:

□ Microsoft Symbol Servers

☑ NuGet.org Symbol Server

Warning: Using symbol files from an unknown or untrusted location can be harmful to your computer.

Cache symbols in this directory:

C:\Users\SPERTU~1.EUR\AppData\Local\Temp\SymbolCache

Automatic symbol loading preference:

● Load all modules, unless excluded

Specify excluded modules

○ Load only specified modules

Specify included modules

Now we are able to map the execution to the original source code location, but we still miss… the source code itself ! That's why need also the **Source link** options.

2.23.2 Source link

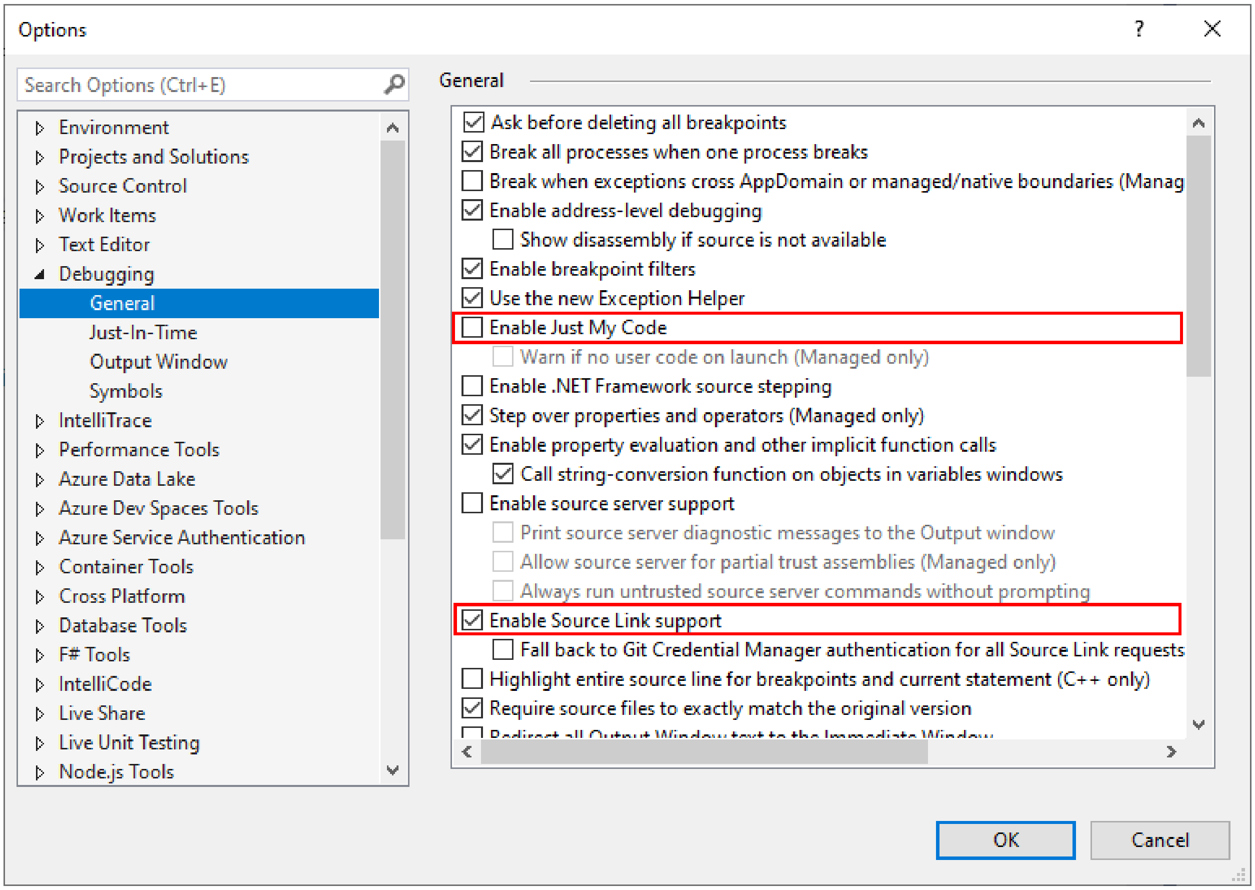
Source Link is a technology that enables source code debugging of .NET assemblies from NuGet by developers.
Source Link executes when creating the NuGet package and embeds source control metadata inside assemblies and the package.
Developers who download the package and have **Source Link** enabled in Visual Studio can step into its source code.
Source Link provides source control metadata to create a great debugging experience.

Note: More information on source link technology: SourceLink

To be able to use the **Source link** technolgy, we should verify the option is checked from within our Visual Studio options:

Go to **Tools** > **Options** > **Debugging** > **General**:

- Uncheck **Enable Just My Code**
- Check **Enable Source Link support**



General

☑ Ask before deleting all breakpoints
☑ Break all processes when one process breaks
□ Break when exceptions cross AppDomain or managed/native boundaries (Manag
☑ Enable address-level debugging
  □ Show disassembly if source is not available
☑ Enable breakpoint filters
☑ Use the new Exception Helper
□ Enable Just My Code
  □ Warn if no user code on launch (Managed only)
□ Enable .NET Framework source stepping
☑ Step over properties and operators (Managed only)
☑ Enable property evaluation and other implicit function calls
  ☑ Call string-conversion function on objects in variables windows
□ Enable source server support
  □ Print source server diagnostic messages to the Output window
  □ Allow source server for partial trust assemblies (Managed only)
  □ Always run untrusted source server commands without prompting
☑ Enable Source Link support
  □ Fall back to Git Credential Manager authentication for all Source Link requests
□ Highlight entire source line for breakpoints and current statement (C++ only)
☑ Require source files to exactly match the original version
□ Redirect all Output Window text to the Immediate Window

We can now debug our code, and **Step Into** the DMS code as well.

If you need more information, you can check this documentation: Using Pdb and Source code