sheigra.systems

Contents

Introduction

The Database Manager Library for DataFlex provides a code-first approach to defining and maintaining database structures. Instead of relying on external tools or manual scripts, you describe your database schema entirely in DataFlex code. Each table is represented by an object (typically defined in its own file), with additional objects for columns, indexes and index segments.

When you compile and run your application, the library ensures the database matches your definitions:

- Automatic creation – if the database doesn't exist, it will be created
- Schema alignment – existing databases are compared to your code definitions, with differences automatically updated to match.
- Version flexibility – applications can upgrade from any version at any time, without needing incremental migration scripts.
- Change history – all modifications are logged in a dedicated database table, giving you a complete record of structural changes over time.

By embedding schema management directly into your application code, the Database Manager Library simplifies deployment, reduces risk, and ensures consistency across environments.

Design goals

The Database Manager library was created with several core values in mind. Its purpose is not only to simplify schema management in DataFlex applications, but also to make the process intuitive, consistent, and robust across versions.

Simplicity

Defining and maintaining a database schema should be as natural as building any other part of a DataFlex application. By adopting an object-oriented approach, the library allows developers to drag classes from the Studio's Class Palette and configure them directly in the Properties Panel. This makes table, column, and index definitions more discoverable and reduces the need to remember method calls and parameters.

Flexibility

Applications evolve over time, and database structures must evolve with them. The Database Manager library enables you to upgrade a database from any version to any version without writing migration scripts. All differences between the current database and the definitions in your code are automatically detected and resolved at runtime.

Transparency

While object definitions provide a clear picture of the current schema, they do not show when a change was introduced. To address this, the library includes a dedicated history table that records all structural changes. Each entry includes the affected table or column, the action performed, the date and time, and the application version number. This ensures that every modification is fully traceable.

Flexibility

Applications evolve over time, and database structures must evolve with them. The Database Manager library enables you to upgrade a database from any version to any version without writing migration scripts. All differences between the current database and the definitions in your code are automatically detected and resolved at runtime.

Portability

Database maintenance should not require external tools such as SQL Server Management Studio. With this library, schema updates are embedded in the application itself. A third party only needs to run the application, and the library will take care of creating or upgrading the database automatically.

Reliability

Automating database management reduces human error and enforces consistency across environments. Whether you are setting up a new database, upgrading an existing one, or distributing updates to clients, the process is predictable and repeatable.

Before You Begin

Before using the Database Manager library, it is useful to understand a few key points about its scope and requirements.

Supported Databases

This initial release supports the following backends:

- DataFlex Embedded Database
- Microsoft SQL Server (MSSQL)

Support for additional database platforms will be added in future versions.

DataFlex Version

The library was developed and tested using DataFlex 25.0. There is no technical reason why it should not function with earlier versions of DataFlex, but compatibility has not yet been formally verified.

Prerequisites

To follow the examples and make effective use of the library, you should:

- Have DataFlex Studio installed and be comfortable creating and running applications.
- Be familiar with the basics of relational databases (tables, columns, indexes).
- Be able to connect your application to either the embedded database or MSSQL.

No advanced database administration tools (such as SQL Server Management Studio) are required. All database management tasks are performed through the application itself.

Recommendations

For the smoothest experience, we recommend:

- Ensuring you can compile and run the standard Order Entry example application in your environment.
- Using source control to manage your table definition objects, so that schema changes are versioned alongside your application code.

Getting Started with the Example

Before adding the Database Manager library to your own project, we recommend starting with the bundled Library.src example. This will let you see the library in action without changing anything in your workspace.

Running the Example

- Open Library.src in the Studio.
- Uncomment the Use lines that include the table definitions for the Order Entry sample.
- Compile and run the application.

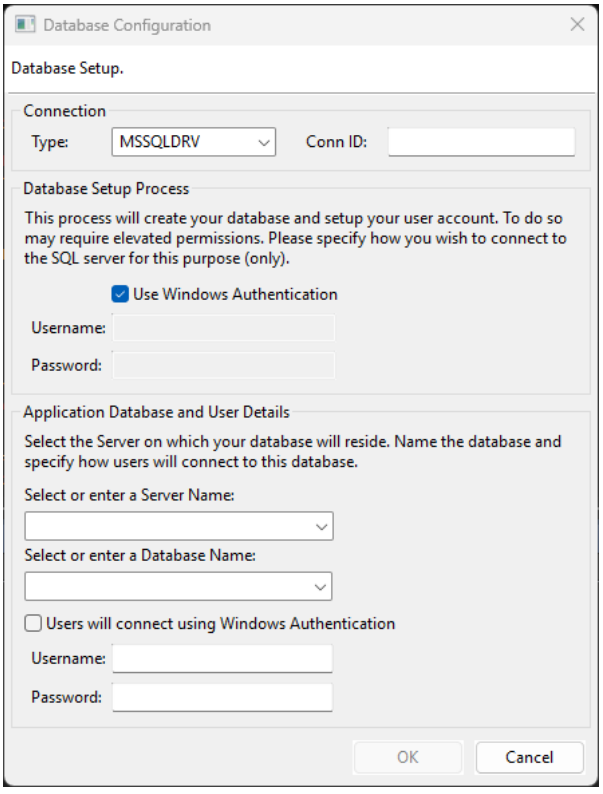
Naming Conventions

When you open the example table definition files, you may notice that object names are not prefixed with the usual "o" Hungarian notation. This is intentional:

- It ensures that object names in the Code Explorer match the corresponding table or column names in the database.
- If you prefer, you can still prefix names with "o".

By default, the object name is used as the table or column name. If you want the database object to use a different name, set the *psName* property of the object. The example code demonstrates this approach. Omitting the 'o' is simply a way of writing less code. This will be discussed later.

The Configuration Dialog



When you run the example for the first time, a Configuration Dialog will appear. This dialog guides you through creating or connecting to the database:

Connection:

- Database type: Choose MSSQLDRV or DATAFLEX.
- Conn ID: Name of managed connection.

Database Setup Process

For the process to create tables it will need permissions that may be above any beyond your normal application permissions. Specify the account that this process should use.

Application Database and User Details

Enter the SQL Server name and the name of a database. If the database does not exist, the process will create it for you.

Specify how your application users will typically connect to this database.

When the process is run, it should:

- Create the SQL database if it didn't already exist.
- Create a SQL login
- Create a Connections.ini file
- Create all required tables
- Create entries in the TableHistory table.

(Note if you are creating a native DataFlex database it will simply create all the tables as required)

Experimenting with Schema Changes

After the initial run, you can explore how the Database Manager responds to changes:

- Open the TableHistory table to see a full record of the actions performed during setup. Each entry includes the date, time, and application version.
- Try reordering column objects in the code to change their position in a table.
- Add a new column or even define a new table.
- Compile and run again.

The Database Manager will detect the differences between your code definitions and the existing database, apply the necessary changes, and log them in TableHistory.

This process shows how schema evolution is handled automatically, without requiring manual SQL scripts or external tools.

How It Fits with Your Application

It is important to be clear about the role of the Database Manager in relation to your own application.

- Standalone project: The Database Manager is not something you compile into your business application. It is a separate project/executable that is compiled and run independently. Its sole responsibility is to ensure that the database schema matches the table and index definitions you maintain in code.
- Why add the library to your workspace?
  You add the library to your existing workspace so that the Database Manager project can work directly with your current environment:
  - It uses your FileList.cfg and DfConnId.ini.
  - It connects to your existing database.
  - It generates and maintains the table definition source files for the same workspace.
- Workflow:
  1. Add the Database Manager library to your workspace.
  2. Create or generate your table object definitions.
  3. Compile the Database Manager project into an executable.
  4. Run this executable to create or update your database.
  5. Once the database is in the correct state, you run your own application as normal – no changes to your application code are required.

In short: your application remains unchanged. The Database Manager simply ensures that the database it depends on is kept in line with the schema you have defined.

Overview of the Main Classes

cDatabaseManager (project-level orchestrator)

Role: The top-level object in the .src file that loads table/view definitions (via Use), connects to the target backend, compares live schema vs. code, applies changes, and logs to TableHistory (if included).

```
Use DfAllEnt.pkg
Use cdmApplication.pkg
Use cdmConnection.pkg
Use cDatabaseManager.pkg

Object oApplication is a cdmApplication

    Object oConnection is a cdmConnection
        Use LoginEncryption.pkg
        Use DatabaseLoginDialog.dg
    End_Object
End_Object

Object oDatabaseManager is a cDatabaseManager
    use TableHistory.pkg

    Use OrderEntry\OrderSystem.pkg
    Use OrderEntry\Customers.pkg
    Use OrderEntry\Vendor.pkg
    Use OrderEntry\SalesPerson.pkg
    Use OrderEntry\Inventory.pkg
    Use OrderEntry\OrderHeader.pkg
    Use OrderEntry\OrderDetail.pkg
End_Object

Start_UI
```

Typical usage:

- Lives in .src as an object with a series of Use statements for tables (and/or SQL views).
- You run/compile once; it creates or upgrades the database and logs actions.

From an administration perspective it makes sense to separate each table definition into its own, separate file. However, this is not a requirement. The only requirement is that each table definition object is a child of the cDatabaseManager object.

Key properties:

peDefaultDriver (e.g. embedded/MSSQL).

By default, each table object has its own peDriver setting set as eDEFAULT which means it takes it value from this peDefaultDriver setting. This saves you having to set it for each table but also allows for some tables to have their own setting. For example, you want some embedded tables even when using a SQL database.

cTableDef (Table Definition)

Role: Defines a real table and how it should exist in the target Database. Child cColumnDef and cTableIndex objects live inside.

Key properties:

- psTableName — logical/table name (defaults to object name if blank).
- psDescription — human-friendly description.
- psOwner — e.g. dbo (MSSQL).
- peDriver — target driver (e.g. embedded/MSSQL). Defaults to use peDefaultDriver setting of cDatabaseManger.
- psConnection — managed connection name. Can be left blank. Typically, only set when table needs a different managed connection to the default used by the application.

- peTableCharacterFormat — e.g. eANSI or eOEM. Default is eANSI.
- psRootName — only set if the table is an alias of another table. Otherwise, will be handled automatically.
- piFileNo — Filelist number (auto-assign supported).
- pbRecnumTable — marks a recnum-style embedded table.
- pbFileCompression – embedded Database only.
- pbJITBinding
- pbSystemTable — Set to True if table is a system (one-record) table.
- pbRequired — whether table must exist in the final schema. If set to FALSE, the table will be deleted from the database.

Key Lifecycle hooks:

- OnCreate — called when creating the table. Code can be added here to populate the table with records.

Usage pattern:

Create a cTableDef object, set high-level table properties, then add cColumnDef and cTableIndex children.

cSqlViewTableDef (SQL View Definition)

Role: Defines a SQL VIEW and adds it to FileList.cfg so DataFlex can treat the view like a table.

Typical flow:

- Define the view as an object (like a table).
- Provide SQL text in OnCreate (the class calls your code to get it).
- The manager will create/refresh the view and register it in FileList.

Key Lifecycle hooks:

- OnCreate ByRef sSQL — called when creating the view. The SQL written to sSQL will be executed to create the SQL view in the target database.

cColumnDef (column definition)

Role: Declares a column inside a cTableDef.

Key properties:

- psName— column name (defaults to object name; set this if you prefer a different physical name).
- peDfDataType – one of DF_ASCII, DF_BCD, DF_TEXT, DF_DATE. Defaults to AUTOASSIGN meaning it will be set automatically according to the value of peNativeDataType.

- peNativeDataType – one of SQL_CHAR, SQL_VARCHAR, SQL_WCHAR, SQL_WVARCHAR, SQL_WVARCHARMAX, SQL_INTEGER, SQL_SMALLINT, SQL_TINYINT, SQL_NUMERIC, SQL_TYPE_TIMESTAMP, SQL_DATETIME, SQL_TYPE_DATE.
- piLength, piPrecision — size/precision for text/decimal types.
- pbIdentityColumn — true for SQL identity columns.
- piRelatedTable, piRelatedColumn — for foreign key mapping.
- piFieldIndex — Index best used for this column (AUTOASSIGN supported).

By default, the object name becomes the column name; set psName if you want a different table name (useful when you prefer "o"-prefixed object names in code explorer but want clean column names or when the preferred column name is not allowed as an object name).

Usage pattern:

Drag column objects from the class palette and into the Table object. Simply use the Studio Properties Panel to apply the required settings.

Key Lifecycle hooks:

- OnCreate – called on the creation of a table. Useful for populating a new column with default values.
- OnChange – called when a column is resized or has its data type or relationships changed.

cIndexDef (index definition)

Role: Declares an index for a table; contains one or more cIndexSegment children.

Key properties:

- piIndex — index number (AUTOASSIGN supported).
- pbPrimaryIndex — mark as primary index (DataFlex primary).
- pbSqlPrimaryKey — mark as SQL PRIMARY KEY.
- piIndexType- e.g. DF_INDEX_TYPE_ONLINE/DF_INDEX_TYPE_BATCH

Usage pattern:

- Create a cTableIndex.
- Add cIndexSegment children for each column/segment.
- Mark primary/PK as appropriate (you can have DF primary and SQL PK aligned).

cIndexSegment (index segment)

Role: One segment (column + options) of an index.

Key properties:

- psColumn — the column name; if blank, the class infers it from the segment object's name.
- pbDescending — descending sort.
- pbIgnoreCase — case-insensitive segment (where supported).

Views vs. Tables — quick note

- Tables (cTableDef) are physical tables; the manager creates/updates columns and indexes and logs each action to TableHistory
- Views (cSqlViewTableDef) are SQL views; the manager creates/refreshes the view and adds a FileList entry so your DataFlex code can use the view as if it were a table. Column Objects are not supported but Index objects are allowed.

Typical Definition Skeletons

```
Object Customer is a cTableDef
    Set psTableName   to "Customer"

    Object Id is a cColumnDef
        Set psName           to "CustomerId"  // optional; defaults to object name (“Id”)
        Set peNativeDataType to SQL_INTEGER
        Set pbIdentityColumn to True          // use SQL identity
    End_Object

    Object Name is a cColumnDef
        Set peNativeDataType to SQL_WVARCHAR
        Set piLength         to 80
    End_Object

    Object ixId is a cTableIndex
        Set pbPrimaryIndex  to True
        Set pbSqlPrimaryKey to True

        Object segId is a cIndexSegment
            Set psColumn to "CustomerId"
        End_Object
    End_Object

    Object ixCustomerName is a cTableIndex

        Object segName is a cIndexSegment
            Set psColumn to "Name"
        End_Object

        Object segId is a cIndexSegment
            Set psColumn to "CustomerId"
        End_Object
    End_Object
End_Object
```

```
Object CustomerView is a cSqlViewTableDef
    Set psTableName   to "vwCustomer"

    Procedure OnCreate String ByRef sSql
        Move @SQL"""
        SELECT "Customer_Number","Name","Address","City","Zip" FROM "Customers"
        """ to sSql
    End_Procedure

    Object oIndex1 is a cTableIndex
        Set pbPrimaryIndex to True

        Object Customer_Number is a cIndexSegment
        End_Object
    End_Object
End_Object
```

Adding the Library to Your Workspace

The Database Manager Library is added to a workspace in the same way as any other DataFlex library. From within the Studio, select Tools and Maintain Libraries before browsing to the Database Manager library folder.

Once the library is added, the Studio will be updated with new components:

The Class Palette will contain a new Database Manager group, including the following classes:

- cColumnDef
- cIndexDef
- cIndexSegment

The Create New… dialog (Studio toolbar) will include one new entry on the Project tab and two new entries on the Other tab:

- Database Manager Project (Project)
- Database Manager Table (Other)
- Database Manager SQL View (Other)

At this point the library is fully available in your workspace, and you are ready to start defining tables and columns.

Logging and History

One of the most powerful features of the Database Manager is the ability to track schema changes over time. Every change the manager makes to your database is written to a dedicated TableHistory table. This allows you to see exactly what changed, when it changed, and under which application version it was applied.

The TableHistory table

The TableHistory table is created automatically (unless you remove it from your build). It contains one row for each action performed by the Database Manager.

Key columns include:

- Id – unique identifier for each logged change.
- Date / Time – when the change occurred.
- Table – the table affected.
- Column – the column affected (if applicable).
- Changed – the action taken (e.g. *Created*, *Modified*, *Deleted*).
- VersionMajor / Minor / Patch / Build – version information of the application that ran the change.

Application versioning

The Database Manager can automatically log the version number of your application. To make use of this:

1. In your Database Manager project, open Project Properties.
2. Enable Include Version Information.
3. Ensure that both your application and its corresponding Database Manager project are built with the same version number.

When the Database Manager runs, it will extract this information and store it alongside each change in the TableHistory table.

This provides a clear audit trail: not only can you see *what* was changed, but also *which version of your software introduced the change*.

Why it matters

- Auditing – you can always check who/what changed your database.
- Debugging – if a deployment goes wrong, TableHistory shows which version made which modifications.
- Compliance – for some clients, being able to demonstrate controlled database changes is a requirement.
- Rollback/Recovery support – while the library doesn't provide rollback directly, TableHistory tells you exactly when and where a structure changed, making it easier to reconstruct or revert if required.

Source Code Generator

Starting with this release, the Database Manager library includes a utility called SourceCodeGenerator.src. This program allows you to bootstrap your project by automatically generating all the required table definition files from an existing database.

How to use

1. Open the Database Manager Library in the Studio.
   - Alternatively, copy SourceCodeGenerator.src into your own workspace and add it as a project.
2. Compile and run the project into an executable.
3. Run the executable from within your application's Programs folder.
   - If you compiled it in the library workspace you will need to move the executable into your own application's Programs folder before running.
   - If you compiled it directly from your own workspace, you can run it in place.

The generator will scan your current FileList.cfg and produce the following:

- DatabaseManager.src – created inside your workspace's AppSrc folder.
- Tables – a new subfolder inside AppSrc containing one .pkg file per table in your database.

Each table object file represents the schema of a corresponding table, ready for use with the Database Manager.

Why use the generator?

This tool is particularly valuable when starting with an existing database:

- No need to hand-code table definitions for dozens or hundreds of tables.
- Gives you a complete starting point to build on.
- Makes it simple to being an established system under Database Manager control.

Once generated, you can edit or extend the objects as needed, just as you would with manually written definitions.

Error Handling

The Database Manager is designed to make schema management safe and predictable, but as with any process that alters databases, errors can occur. To help you track and diagnose these, the library provides multiple levels of reporting.

On-screen feedback

While the process runs, progress is reported directly to the screen. The current error count is always visible. In normal operation this should remain at zero; if the count increases, you can immediately investigate further.

Log file

All errors and key process messages are written to a log file called Database.log, located inside your DataFlex Data folder. This file is appended to on each run, providing a persistent record of issues or warnings encountered during schema updates.

Recommended practice

Because the library makes real changes to your database schema, we strongly recommend that you:

- Always back up your SQL database before running the Database Manager.
- Always back up your DataFlex Data folder (which includes FileList.cfg and other important files).

If an unexpected issue occurs, having these backups in place will allow you to quickly roll back to a stable state.

Note: Backup management is not currently handled by the library itself.

Road Map

The Database Manager library will continue to evolve. Planned future enhancements include:

1. Automatic backups
   The Database Manager will be able to take its own backups of both the SQL database and the DataFlex environment before applying schema changes.
2. More database backends
   Support will be extended beyond the DataFlex Embedded database and Microsoft SQL Server, with additional platforms planned for future releases.