Migration Utility

The migration utility is a command-line tool built to upgrade the schema of an ODS instance to the latest version. It currently supports data migration from Ed-Fi Data Standard v2.0 and Ed-Fi Data Standard v2.2 to Ed-Fi Data Standard v3.2. The utility has out-of-the-box support for migrating an as-shipped ODS to the latest version. With additional customized scripting, the utility can be easily adapted and used to migrate extended ODS instances. ODS shared instances may take advantage of this utility. For year-specific instances, migration may not be a concern as a new ODS is created at the beginning of every school year.

The following table summarizes the supported scenarios for the migration utility:

<table>
<thead>
<tr>
<th>Database Type</th>
<th>Databases</th>
<th>Upgrade/Migration strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Databases</td>
<td>EdFi_Ods,</td>
<td>The current migration utility release supports an in-place update for the following upgrade paths:</td>
</tr>
<tr>
<td></td>
<td>EdFi_Ods_YYYY,</td>
<td>2.4 -&gt; 3.3</td>
</tr>
<tr>
<td></td>
<td>EdFi_Ods_Sandbox</td>
<td>2.5 -&gt; 3.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.0 -&gt; 3.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The migration utility also supports migrating extensions with additional custom scripts.</td>
</tr>
<tr>
<td>Support Databases</td>
<td>EdFi_Admin,</td>
<td>These databases can either be recreated, or existing Entity Framework Migrations can be used to migrate them.</td>
</tr>
<tr>
<td></td>
<td>EdFi_Security,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EdFi_Bulk</td>
<td></td>
</tr>
<tr>
<td>Transient Databases</td>
<td>EdFi_ODS_Empty,</td>
<td>No upgrade supported.</td>
</tr>
<tr>
<td></td>
<td>EdFi_Ods_Minimal_Template</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EdFi_Ods_Populated_Template</td>
<td></td>
</tr>
</tbody>
</table>

This documentation covers the general principles and concepts required to develop for the Migration Utility. When you understand this material, or if you just want to get a sense of how to use the utility, you can put it into action using the version-specific, step-by-step guidance in:

- How To: Upgrade an ODS Database from Version 2.x to Version 3.3

Developer Quick Start

The basic steps are simple:

- Restore a backup copy of the target ODS to your local SQL Server instance. We recommend:
  - Start with the basic Version 2.x: Sample ODS Download: EdFi.Samples.Ods/2.0.0.21. This is a small sample ODS, ideal for initial development and testing.
  - Move on to the larger Version 2.x: Glendale 2.0 ODS Backup (created 2018-06-13). This is a much larger ODS containing sample data, useful for validation and QA.
- Choose one of the two options below to launch the migration utility:

  **Step 1.** Install the Ed-Fi ODS Migration tool:

  ```
c:\>mkdir {YourInstallFolder}
c:\>dotnet tool install EdFi.Ods.Utilities.Migration --tool-path {YourInstallFolder}
  ```

  **Step 2.** Open a console window and change to the directory containing the executable

  ```
  CD {YourInstallFolder}
  ```

  **Step 3.** Launch the upgrade tool from the command line
Step 1. Clone git repository: https://github.com/Ed-Fi-Alliance/Ed-Fi-MigrationUtility

Step 2. Open the Visual Studio solution file, Migration.sln.

Step 3. Set up command line input for debugging in Visual Studio 2017:

- Right click the EdFi.Ods.Utilities.Migration project.
- Select properties.
- Select debug.
- Add command line arguments:

```
Example Command Line Arguments: Grand Bend ODS

```

Step 4. Set the EdFi.Ods.Utilities.Migration project as your startup project.

Step 5. Launch in debug mode (F5).

### Development Overview: The Basics

The table below describes files and folders used by the Migration Utility along with a description and purpose for each resource.

<table>
<thead>
<tr>
<th>Overview Item</th>
<th>Needed by Whom?</th>
<th>Brief Description &amp; Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Script Directory:</strong> Ed-Fi-Standard XML files containing Descriptors for each ODS version. These XML files are imported directly by the scripting in the Utilities\Migration\Scripts directory above.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ```Directory: \Descriptors``` | Maintainers of the upgrade utility only | All database upgrade scripts go here, including:  
- Ed-Fi upgrade scripts.  
- Custom upgrade scripts (user extensions).  
- Compatibility checks.  
- Dynamic / SQL-based validation.  
- Subdirectories contain code for each supported ODS upgrade version.  
- For a detailed description of directory conventions — which may vary based on target ODS version — please consult the version-specific user guide (e.g., How To: Upgrade an ODS Database from Version 2.x to Version 3.3). |
| **Library/Console:** EdFi.Ods. Utilities. Migration (console application created via dotnet publish) | Maintainers of the upgrade utility only | Small, reusable library making use of DbUp to drive the main upgrade.  
- Executes the SQL scripts contained in the Utilities\Migration\Scripts directory above.  
- Takes a configuration object as input, and chooses the appropriate scripts to execute based on ODS version and current conventions. |
| **Test Project:** EdFi.Ods. Utilities. Migration.Tests | Maintainers of the upgrade utility only (does not contain test coverage for extensions) | Contains integration tests that perform a few test upgrades and assert that the output is as expected.  
- Like the console utility, makes direct use of the EdFi.Ods.Utilities.Migration library described above. |

### Troubleshooting

This section outlines general troubleshooting procedures.
Compatibility Errors

- Before the schema is updated, the ODS data is checked for compatibility. If changes are required, the upgrade will stop and exception will be thrown with instructions on how to proceed.
  
  An example error message follows:

  ![Error Message]

- After making the required changes (or writing custom scripts), simply launch the upgrade utility again. The upgrade will proceed where it left off and retry with the last script that failed.

Other Exceptions During Development

- Similar to compatibility error events, the upgrade will halt and an exception message will be generated during development if a problem is encountered.
- After making updates to the script that failed, simply re-launch the update tool. The upgrade will proceed where it left off starting with the last script that failed.
- If you are testing a version that is not yet released, or if you need to re-execute scripts that were renamed or modified during active development: restore your database from backup and start over.
- Similar to other database migration tools, a log of scripts successfully executed will be stored in the default journal table. DbUp's default location is [dbo].[SchemaVersions].
- A log file containing errors/warnings from the most recent run may be found by default in C:\ProgramData\Ed-Fi-ODS-Migration\Migration.log.

Additional Troubleshooting

- See the version-specific migration guides such as How To: Upgrade an ODS Database from Version 2.x to Version 3.3.

Design/Convention Overview

The table below outlines some important conventions in the as-shipped Migration Utility code.

<table>
<thead>
<tr>
<th>What</th>
<th>Why</th>
<th>Optional Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-place upgrade</td>
<td>Extensions</td>
<td>Users who have extension tables (or any other schema with foreign key dependencies on the edfi schema) will be notified, and must explicitly acknowledge it by adding the BypassExtensionValidationCheck option at the command line when upgrading. This will ensure that the installer is aware that custom upgrade scripts may be required.</td>
</tr>
<tr>
<td>Database upgrades are performed in place rather than creating a new database copy</td>
<td>Preserve all unknown data in extension tables</td>
<td>As a secondary concern, this upgrade method was chosen to ease the upgrade process for a Cloud-based ODS (e.g., on Azure).</td>
</tr>
<tr>
<td>Sequence of events that occur during upgrade</td>
<td>Minimize the number of scripts with complex dependencies on other scripts in the same directory upgrade step.</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Specifics differ for each version, but in general the upgrade sequence executes as follows</td>
<td>• Compatibility checks are designed to run before any changes to the edfi schema have been made. This prevents the user from having to deal with a half-upgraded database while making updates.</td>
<td></td>
</tr>
<tr>
<td>1. Validate user input at the command line</td>
<td>• Initial hash codes used for data validation also must be generated before touching the edfi schema to ensure accuracy.</td>
<td></td>
</tr>
<tr>
<td>2. Create tempdata (tables/stored procedures) that will be used for upgrade</td>
<td>• It is also better for performance to do this step while all of our indexes are still present.</td>
<td></td>
</tr>
<tr>
<td>3. Check current ODS data for upgrade compatibility, and display action messages to the user if existing data requires changes</td>
<td>• Dropping of constraints, views, etc is taken care of before making any schema changes to prevent unexpected sql exceptions.</td>
<td></td>
</tr>
<tr>
<td>4. Before modifying the edfi schema: calculate and store hash values for primary key data that expected NOT to change during upgrade</td>
<td>• New descriptors are imported as an initial step before making changes to the core tables. This ensures that all new descriptor data is available in advance for reference during updates.</td>
<td></td>
</tr>
<tr>
<td>5. Drop views, constraints, stored procs</td>
<td>• After creation of descriptors, the sequence of the next steps is designed to ensure that all data sources exist unmodified on the old schema where we expect it to.</td>
<td></td>
</tr>
<tr>
<td>6. Import descriptor data from XML</td>
<td>1. Create tables that are brand new to the schema only (and populate them with existing data)</td>
<td></td>
</tr>
<tr>
<td>7. Create all new tables for this version that did not previously exist</td>
<td>2. Modify existing tables (add/drop columns, etc)</td>
<td></td>
</tr>
<tr>
<td>8. Update data in existing tables</td>
<td>3. Drop old tables no longer needed</td>
<td></td>
</tr>
<tr>
<td>9. Drop old tables</td>
<td>• Foreign keys, constraints, etc are all added back in once the new table structure is fully in place.</td>
<td></td>
</tr>
<tr>
<td>10. Create views /constraints/stored procs for the new version</td>
<td>• Once the edfi schema is fully upgraded and will receive no further changes, we can perform the final data validation check.</td>
<td></td>
</tr>
<tr>
<td>11. Validation check: recalculate the hash codes generated previously, and make sure that all data that is not supposed to change was not mistakenly modified</td>
<td>The v2x to v3 upgrade is a good example case to demonstrate the upgrade steps working together due its larger scale:</td>
<td></td>
</tr>
<tr>
<td>12. Drop all temporary migration data</td>
<td>• For v2x to v3: All foreign keys and other constraints were dropped during this upgrade in order to adopt the new naming conventions.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Also for v2x to v3: ODS types were replaced with new descriptors. This change impacted nearly every table on the existing schema.</td>
<td></td>
</tr>
</tbody>
</table>
### One script per table in each directory, where possible

Scripts are named in the format:

```sql
### TableName [optional_tags].sql
```

This convention does not apply to operations that are performed dynamically.

### Troubleshooting, Timeout prevention

Custom, unknown extensions on the ODS are common. As part of the process of upgrading a highly-customized ODS, an installer is likely to run into a sql exception somewhere in the middle of upgrade (usually caused by a foreign key dependency, schema bound view, etc).

In general, we do not want to attempt to modify an unknown/custom extension on an installer's behalf to try and prevent this from happening. It is important that a installer be aware of each and every change applied to their custom tables. Migration of custom extensions will be handled by the installer.

Considering the above, in the event an exception does occur during upgrade, we want to make the troubleshooting process as easy as possible. If an exception is thrown, an installer should immediately be able to tell:

- Which table was being upgraded when it occurred (from the file name)
- What were the major changes being applied (from the file tags)
- What went wrong (from the exception message)
- Where to find the code that caused it

Many issues may be fixable from the above information alone. If more detail is needed, the installer can view the code in the referenced script file. By separating script changes by table, we make an effort to ensure that there are only a few lines to look through (rather than hundreds)

In addition, each script will be executed in a separate transaction. Operations such as index creation can take a long time on some tables with a large ODS. Splitting the code into separate transactions helps prevent unexpected timeout events.

### The major downside of this approach is the large number of files it can produce. For example, the v2x to v3 upgrade was a case where all existing tables saw modifications. This convention generates a change script for every table in more than one directory.

With updates becoming more frequent in the future, future versions should not be impacted as heavily.

### Most change logic is held in sql scripts (as of V3)

As of v3: Most of the upgrade logic is performed from the SQL scripts, rather than using .NET based upgrade utility to write database changes directly.

As of v3, most upgrade tasks are simple enough where they can be executed straight from SQL (given a few stored procedures to get started).

Given this advantage, effort was made to ensure that each part of the migration tool (console utility, library, integration tests) could be replaced individually as needed.

The current upgrade utility contains a library making use of DbUp to drive the upgrade process. In the future, if/when this tool no longer suits our needs, we should be able to take existing scripting and port it over to an alternative upgrade tool (such as RoundhousE), or even a custom built tool if the need ever arises.

This convention could (and should) change in the future if upgrade requirements become too complex to execute from SQL scripting alone.
<table>
<thead>
<tr>
<th>Two types of data validation/testing options</th>
<th>Prevent data loss</th>
<th>The dynamic validation is performed via reusable stored procedures that are already created and available during upgrade.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Dynamic, SQL-based</td>
<td>The first type of validation, (dynamic, sql based) is executed on data that we know should not ever change during the upgrade.</td>
<td>See scripts in the &quot;Source Validation Check&quot; and &quot;Destination Validation Check&quot; directories for example usages.</td>
</tr>
<tr>
<td>• Ensures data that is expected to remain the same does not change</td>
<td>• The source and destination tables do not need to be the same. This validation type is most commonly used to verify that data was correctly moved to the expected destination table during upgrade.</td>
<td></td>
</tr>
<tr>
<td>• Can run on any ODS in the field even if the data is unknown</td>
<td>• Can be executed on any field ODS to ensure that unknown datasets do not cause unexpected data loss during upgrade.</td>
<td></td>
</tr>
<tr>
<td>• Integration tests</td>
<td>• The data in these tables does not need to be known</td>
<td></td>
</tr>
<tr>
<td>• Runs on a known, given set of inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Used to test logic in areas where changes should occur</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Upgrade Issue Resolution Approach

It is common to encounter scenarios where data cannot be upgraded directly from one major version to another due to schema changes. A common example is that a primary key change causes records allowable by the previous version schema to be considered duplicates on the upgrade version schema and therefore not allowed.

The general approaches included here are a result of collaboration with the community on how to resolve these common situations, and are documented here as a reference for utility developers to apply in their own work.

- **Approach 1**: Make non-breaking, safe changes to data on the user’s behalf. This is the preferred option when practical and safe to do so.
  - Example: Duplicate session names in the v2.x schema that will break the new v3.x primary key will get a term name appended during upgrade to meet schema requirements.
  - Consider logging a warning for the user to review to inform them that a minor change has taken place, and mention the log file in documentation.

- **Approach 2**: Throw a compatibility exception asking the user to intervene with updates. Used when we are unable to safely make assumptions on the user's behalf to perform the update for them.
  - See the troubleshooting section of this document for an example of this.
  - The returned user-facing message should explain what updates are required, and very briefly include the reason why if it is not immediately obvious.
  - Compatibility exceptions should be thrown before proceeding with core upgrade tasks.
  - We want to make sure that we are not asking someone to make updates to a database in an intermediate upgrade state.
  - Each message class has a corresponding error code, mirrored in code by an enumeration.
  - This is done so that specific compatibility scenarios can be covered by integration tests where needed.

- **Approach 3**: Back up deprecated objects that cannot be upgraded into another schema before dropping them. Last resort option that should be mainly restricted to deprecated items that are no longer have a place in the new model.
  - As a general approach, it is preferred to avoid dropping data items without a way to recover them in case of disaster. The user may choose to delete the backup if they desire.
  - Avoid this option for tables that exist in both models but simply cannot be upgraded. Should this (hopefully rare) situation occur, consider the option throwing a compatibility exception instead and ask the user back up/empty the table before proceeding.

In general, the option requiring the least amount of user effort while safely preserving all data has been chosen in order to reduce user burden as much as we are able.