Migration Tool Developer's Guide

Overview

This document describes an implementation details of Migration Tool and how to extend its functionality. It is recommended to read Migration Tool User Guide before for better understanding of Migration Tool in general.

Repositories

Migration tool repository https://github.com/magento/data-migration-tool

System requirements

The same as for Magneto 2

Internal structure

Directory Structure

Next diagram represents directory structure of Migration Tool

```
├ bin
│ └ migrate                  -- entry point
├ etc                        -- configurations of tool
│ ├ ce-1.x.x                 -- folder contains version specific set of config
files
│ │ ├ map.xml.dist
│ │ ├ config.xml.dist
│ │ ├ settings.xml.dist
│ │ └ ...
│ ├ config.xsd
│ ├ map.xsd
│ ├ magento_path.php
│ └ ...
├ src
│ └ Migration
│   ├ App
│   │ └ Shell.php              -- shell application
│   ├ Handler                  -- contains handlers for specific cases of
processing data
│   │ ├ Manager.php
│   │ ├ HandlerInterface.php
│   │ └ AbstractHandler.php
│   ├ Resource                 -- contains adapter for connection to data storage
and classes to work with structured data
│   │ ├ Adapter
│   │ ├ Document
│   │ ├ Record
│   │ ├ Structure
│   │ ├ Source.php
│   │ └ Destination.php
│   ├ Logger                    -- classes for processing log information
│   │ ├ ConsoleHandler.php
│   │ ├ FileHandler.php
│   │ ├ Logger.php
│   │ ├ Manager.php
│   │ ├ MessageFormatter.php
│   │ └ MessageProcessor.php
│   ├ Step
│   │ ├ EAV
│   │ ├ Map
│   │ └ UrlRewrite
│   ├ Config.php
│   └ Migration.php    -- application
├ tests
│ ├ unit
│ │ └ phpunit.xml.dist
│ ├ static
│ │ └ phpunit.xml.dist
│ └ integration
│   └ phpunit.xml.dist
├ composer.json
└ README.md
```

Entry Point

Script that runs migration process located
magento-root/vendor/magento/migration-tool/bin/migrate

Configuration

The Schema for configuration file **config.xsd** is placed under **etc/** directory. Default configuration files **config.xml.dist** created for each version of Magento 1.x. It placed in separate directories under **etc/**.

Default configuration file can be replaced by custom one via CLI(see **--config <value>** parameter).

Configuration file has next structure:

```
<config xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xs:noNamespaceSchemaLocation="config.xsd">
    <steps mode="settings">
        <step title="Settings step">
            <integrity>Migration\Step\Settings</integrity>
            <data>Migration\Step\Settings</data>
        </step>
    </steps>
    <steps mode="data">
        <step title="Map step">
            <integrity>Migration\Step\Map\Integrity</integrity>
            <data>Migration\Step\Map\Data</data>
            <volume>Migration\Step\Map\Volume</volume>
        </step>
        ...
    </steps>
    <steps mode="delta">
        <step title="Map step">
            <delta>Migration\Step\Map\Delta</delta>
            <volume>Migration\Step\Map\Volume</volume>
        </step>
        ...
    </steps>
    <source>
        <database host="localhost" name="magento1" user="root" password=""/>
    </source>
    <destination>
        <database host="localhost" name="magento2" user="root" password=""/>
    </destination>
    <options>
        <map_file>map-file.xml</map_file>
        <settings_map_file>settings-map-file.xml</settings_map_file>
        <bulk_size>100</bulk_size>
        <custom_option>custom_option_value</custom_option>
        <source_prefix />
        <dest_prefix />
        ...
    </options>
</config>
```

- **steps** - describes all steps are processed during migration.

- **source** - configuration for data source. Available source types: **database.**

- **destination** - configuration for data destination. Available destination types: **database.**

- **options** - list of parameters. Contains both mandatory(**map_file**, **settings_map_file**, **bulk_size**) and optional (**custom_option**, **resource_adapter_class_name**, **prefix_source**, **prefix_dest**, **log_file**) parameters.

Use **prefix** option when documents have a prefix. In that case you're not need to set prefix part for documents in a map file.
It can be set to source and to destination documents. Use the "source_prefix" and "dest_prefix" configuration options accordingly

Configuration data is accessible via \Migration\Config class.

Step internals

Migration Process consists of steps.
Step is - unit that provides functionality required for migration some separated data. Step can consist of one or more stages e.g. **integrity check**, **data**, **volume check**, **delta**.

By default there are several steps (Map, EAV, URL Rewrites ...). But developer can add his own.
Steps related classes are placed into **src/Migration/Step** directory.
To be executed Step class should be defined in config.xml file.

```
<config xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xs:noNamespaceSchemaLocation="config.xsd">
    <steps mode="mode_name">
        <step title="Step Name">
            <integrity>Migration\Step\StepName\Inegrity</integrity>  <!--
integrity check stage of the step -->
            <data>Migration\Step\StepName\Data</data>
            <volume>Migration\Step\StepName\Volume</volume>
        </step>
        ...
    </steps>
    ...
</config>
```

Every stage class must implement StageInterface.

```
class StageClass implements StageInterface
{
  /**
   * Perform the stage
   *
   * @return bool
   */
  public function perform()
  {
  }
}
```

If **data** stage supports rollback it should implement RollbackInterface interface.

Visualization of step running is provided by Symfony's ProgressBar component (see http://symfony.com/doc/current/components/console/helpers/progressbar.html). It

accessible inside step as **LogLevelProcessor**
Main methods for use are:

```
$this->progress->start();
$this->progress->advance();
$this->progress->finish();
```

Stages

Integrity check

Each step has to check that the structure of data source (Magento 1 by default) and the structure of data destination (Magento 2) are compatible. If not an error will be shown with entities that are not compatible.

Data Transfer

In case integrity check is passed, transferring data is run. In case when some error appears then rollback is run to revert to previous state of Magneto 2. If a step class implements RollbackInterface then "rollback" method will be executed in case an error.

Volume check

After data has been migrated Volume Check provides additional check that all data was transferred correctly.

Delta delivery

Delta functionality is responsible for delivering the rest of data that was added after main migration.

Running Modes

The tool should be run in three different modes in particular order:

1. settings - migration of system settings
2. data - main migration of data
3. delta - migration the rest of data that was added after main migration

Each mode has its own list of steps to be executed. See config.xml

Settings migration mode

Settings migration mode of this tool used to transfer following entities:

1. Websites, stores, store views.
2. Store configuration (mainly Stores->Configuration in M2 or System->Configuration in M1)

All store configuration keeps its data in core_config_data table in DB. settings.xml file contains rules for this table that are applied during migration process. This file describes

settings that should be ignored, renamed or should change their values. settings.xml file has following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xs:noNamespaceSchemaLocation="settings.xsd">
    <key>
        <ignore>
            <path>path/to/ignore*</path>
        </ignore>
        <rename>
            <path>path/to/rename</path>
            <to>new/path/renamed</to>
        </rename>
    <key>
    <value>
        <transform>
            <path>some/key/to/change</path>
            <handler class="Some\Handler\Class"/>
        </transform>
    </value>
</settings>
```

Nodes, with paths indicated under <ignore> option will be ignored and will not be migrated. Wildcards can be used in this node. All other settings not listed in ignore node, will be migrated. If path of some setting has been changed in M2, it should be added to //key/rename node, where old path should be added to //key/rename/path node and new setting path should be added to //key/rename/to node.

Value of the setting can be transformed during migration using **handler**. The //value/transform nodes consist path to setting and transformation handler class name. Handlers are implements Migration\Handler\HandlerInterface.

Data migration mode

In this mode most of the data will be migrated. Before data migration the integrity check stages runs for each step. If integrity check passed the Migration Tool installs deltalog tables (with prefix m2_cl_*) and corresponding triggers to Magento 1.x database. And runs data migration stage of steps. When migration completed without errors the volume check checks data consistency. Next the most valuable migration steps are described. It is Map Step, URL Rewrite Step, EAV Step

Map Step

Map step is responsible for transfer most of data from Magento 1 to Magento 2. This step reads instructions from map.xml file (located in etc dir). The file describes differences between data structures of source (Magento 1) and destination (Magento 2). In case Magento 1 contains tables or fields belonged to some extension that does not exist in Magento 2 then these entities can be placed here to ignore them by Map Step. Otherwise it will show an error message.
Map file has next format:

```
<?xml version="1.0" encoding="UTF-8"?>
<map xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xs:noNamespaceSchemaLocation="map.xsd">
    <source>
        <document_rules>
```

```
            <ignore>
                <document>some_document2</document>
            </ignore>
            <rename>
                <document>some_document</document>
                <to>some_dest_document</to>
            </rename>
            <log_changes>
                <document key="primary_key">some_dest_document</document>
            </log_changes>
        </document_rules>

        <field_rules>
            <move>
                <field>some_document1.field1</field>
                <to>some_document1.field2</to>
            </move>
            <ignore>
                <field>some_document3.field8</field>
            </ignore>
            <transform>
                <field>some_document1.field1</field>
                <handler class="\Migration\Handler\Convert">
                    <param name="map"
value="[value1:value2;value3:value4;value5:value6;]" />
                </handler>
            </transform>
        </field_rules>
    </source>
    <destination>
        <document_rules>
            <ignore>
                <document>some_document8</document>
            </ignore>
        </document_rules>

        <field_rules>
            <transform>
                <field>some_document5.field3</field>
                <handler class="\Migration\Handler\SetValue">
                    <param name="value" value="10" />
                </handler>
            </transform>
        </field_rules>
    </destination>
</map>
```

Areas:

source - contains rules of source database.

destination - contains rules of destination database.

Options:

ignore - document or field marked with this option will be ignored and not transferred to destination document.

rename - describes name relations between documents with the different name. In a case when destination document name is not the same with the source document - you can use rename option to set source document name similar to destination table name.

move - set rule to move specified field from source document to destination document. NOTE: destination document name should be the same with the source document name. If source and destination document names are different - you need to use **rename** option for document that contains moved field.

transform - is a option that allows user to migrate fields according to behavior described in handlers.

handler - describes transformation behavior for fields. To call the handler you need to specify a handler class name in a <handler> tag. Use <param> tag with the parameter name and value data to pass it to handler.

Source available operations:

Document	Field
ignore	ignore
rename	move
	transform

Destination available operations:

Document	Field
ignore	ignore

Wildcards
To ignore documents with similar parts (e.g. document_name_1, document_name_2 e.t.c), you can use wildcard functionality. Just put * symbol instead of repeating part (e.g. document_name_*) and this mask will cover all source or destination documents that meet this mask.

URL Rewrite Step

This step is quite complex because of there are many different algorithms were developed in Magento 1 which are not compatible with Magento 2. For different versions of Magento 1 can be different algorithms. Thus under Step/UrlRewrite folder there are classes that developed for some of particular version of Magento and Migration\Step\UrlRewrite\Version191to2000 is one of them. It can transfer URL Rewrites data from Magento 1.9.1 to Magento 2.

EAV Step

This step transfers all attributes (e.g. product, customer, RMA) from Magento 1 to Magento 2. It uses map-eav.xml file that contains rules alike in map.xml file for specific cases of to processing data.

Some of the tables that are processed in the step:

- eav_attribute
- eav_attribute_group
- eav_attribute_set
- eav_entity_attribute
- catalog_eav_attribute

- customer_eav_attribute
- eav_entity_type
- ...

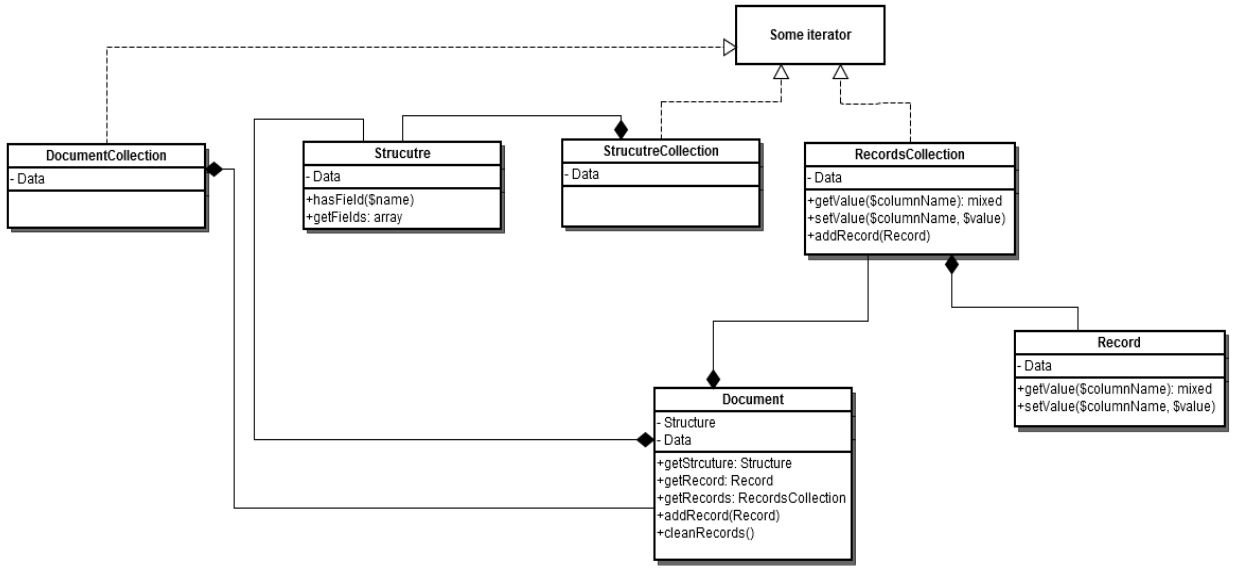
Delta migration mode

After main migration some data could have been added to DB of Magento 1 e.g. by customers on store-front. To track this data, database triggers are setup for tables in the beginning of main migration. If some extension has its own tables that need to be tracked for changing its data then a developer should

1. add these tables into deltalog.xml file
2. create its own delta class which extends Migration\App\Step\AbstractDelta
3. add name of this class to config.xml into delta mode section

Data sources

To reach to the data sources of Magento 1 and Magento 2 and operate with its data (select, update, insert, delete) there are many classes in Resource folder. Migration\Resource\Source and Migration\Resource\Destination are main classes. All migration steps use it to operate with data. This data contains in classes like Migration\Resource\Document, Migration\Resource\Record, Migration\Resource\Structure etc. Here is a class diagram of these classes



You need to upgrade your Gliffy Plugin License. Your license entitles you to 500 users but you currently have a Confluence license for 2000 users. Please upgrade your license promptly.

Logging

In order to implement output of migration process and control all possible levels PSR logger, which is used in Magento, is applied. \Migration\Logger\Logger class was implemented to provide logging functionality. To use the logger you should inject it via constructor dependency injection

```
class SomeClass
{
    ...
    protected $logger;

    public function __construct(\Migration\Logger\Logger $logger)
    {
        $this->logger = $logger;
    }
    ...
}
```

After that you can use this class for logging some events:

```
$this->logger->info("Some information message");
$this->logger->debug("Some debug message");
$this->logger->error("Message about error operation");
```

There is a possibility to customize where log information should be written. You can do that by adding handler to logger using pushHandler() method of the logger. Each handler should implement \Monolog\Handler\HandlerInterface interface. As for now there are two handlers:

- ConsoleHandler: writes messages to console
- FileHandler: writes messages to log file that has been set in "log_file" config option

Also it is possible to implement any additional handler. There exists the set of handlers in Magento framework. Example of adding handlers to logger:

```
// $this->consoleHandler is the object of Migration\Logger\ConsoleHandler class
// $this->logger is the object of Migration\Logger\Logger class
$this->logger->pushHandler($this->consoleHandler);
```

To set additional data for logger (e.g. current mode, table name e.t.c) you can use logger processors. There is one existing processor (MessageProcessor). It's created to add "extra" data for logging messages and will be called each time when log method executed. MessageProcessor have protected $extra var, which contain empty values for 'mode', 'stage', 'step' and 'table'. Extra data can be passed to processor as a second parameter (context) for log method. Currently additional data sets to processor in AbstractStep->runStage (pass current mode, stage and step to processor) method and data classes where used logger->debug method (pass migrating table name). Example of adding processors to logger:

```
// $this->processoris the object of Migration\Logger\messageProcessor class
// $this->logger is the object of Migration\Logger\Logger class
$this->logger->pushProcessor([$this->processor, 'setExtra']);
// As a second array value you need to pass method that should be executed when
processor called
```

There is a possibility to set the level of verbosity. As for now there are 3 level: ERROR(write only errors to the log), INFO(only important information is written to the log, default value), DEBUG(everything is written). Verbosity log level can be set for each handler separately by calling setLevel() method. If you want to set verbosity level via command line parameter, you should change 'verbose' option at application launch as follows:

```
magento2$ php -f vendor/magento/migration-tool/migration.php -- --verbose ERROR
```

```
magento2$ php -f vendor/magento/migration-tool/migration.php -- --verbose INFO
magento2$ php -f vendor/magento/migration-tool/migration.php -- --verbose DEBUG
```

There is a possibility to format log messages via monolog formatter. To make formatter functionality work it's need be set to specified log handler using setFormatter() method. Currently we have one formatter class (MessageFormatter) that set certain format (depends on verbosity level) during message handling (via format() method executed from handler).

As for now manipulation with logger, adding handler(s), processor(s) to it and processing verbose mode is performed in process() method of Migration\Logger\Manager class. Mentioned method is called during application start.

Extension Points

Custom Resource Type of Source

By default migration tool works with MySQL DB of Magento 1 as source of data to transfer it to Magento 2. But source data type can be changed to CSV as example. There is **resource_adapter_class_name** option in **config.xml** that can hold custom class name to resource adapter which can be implemented to work with CSV as example or any other data type.

Map Step configuration

In most cases modification of map will be enough.

Custom Handler

For cases where data in a field should be transformed with more complex algorithm that already present out of the box a Custom Handler can be applied to this field. To apply custom handler to the field add a **transform** node to the **field_rules** (see Map Step)

Custom Steps

Migration tool provides possibility to add custom steps to migration procedure (see Step internals).

Automatic tests

There are 3 types of tests in migration tool: static, unit and integration tests. They all are located in tests/ directory of the tool and they are located in folders, which is the same as the type of the test (e.g. unit tests are located in tests/unit folder). To launch the test you should have phpunit installed. In such case you should change current folder to the folder of test and launch phpunit. See the example below.

```
[10:32 AM]-[vagrant@debian-70rc1-x64-vbox4210]-
[/var/www/magento2/vendor/magento/migration-tool]-[git master]
$ cd tests/unit

[10:33 AM]-[vagrant@debian-70rc1-x64-vbox4210]-
[/var/www/magento2/vendor/magento/migration-tool/tests/unit]-[git master]
$ phpunit
PHPUnit 4.1.0 by Sebastian Bergmann.
....
```