L-Migrator, Migrations in General, Definition and Scope, Usage within Lasso, Best Practices and Shortcuts.

By Brian Loomis - <www.virtualrelations.us>
PageBlocks: <www.pageblocks.org>

1. Introduction

1.0 Scope

L-Migrator is one of the componenets of PageBlocks, a framework developed by Greg Willits. It is the standard migrations package for Lasso and is similar to many other migrations for other programming languages.

Migrations for database agnostic schema generation are part of the working toward the Agile Manifesto. http://agilemanifesto.org/principles.html

1.1 Who Is This Aimed At?

L-Migrator is for people who are working in a collaborative environment, people who work on multiple stages of development tiers, or anyone that is interested in putting their databases on version control.

This is intended for audiences that are more experienced with programming concepts, advanced strategies, and other methods for standardizing workflow and automating deployment.

This is not intended to replace or augment the L-migrator documentation, its just the conference paper, incidentally there may be content that is similar to the L-Migrator docs.

1.2 What are Migrations?

Dictionary.com defines `migrate` as follows.

mi·grate –verb (used without object), -grat·ed, -grat·ing.

1. to go from one country, region, or place to another.

2. to pass periodically from one region or climate to another, as certain birds, fishes, and animals: The birds migrate southward in the winter.

3. to shift, as from one system, mode of operation, or enterprise to another.

4. Physiology. (of a cell, tissue, etc.) to move from one region of the body to another, as in embryonic development.

5. Chemistry.

   a. (of ions) to move toward an electrode during electrolysis.

   b. (of atoms within a molecule) to change position.

6. (at British universities) to change or transfer from one college to another.

So in this sense, a database migration would be definition number 3, a shift from one system, mode of operation, or enterprise to another. When we think of a database migration we can look at it from several of these perspectives, and in the case of using migrations to automate deployment, we can also see that definition 1 is accurate also, because we can use migrations to push out database schema from one location to another. Well discuss that later in the paper.

In short migrations allow you to make Lasso code that will update and increment or decrement your database schema to keep it inline with the current application requirements, and automate installation or deployment – for example, rolling out multiple Lasso Site instances under a load balancer.

1.3 Why do we want to use migrations?

There are a number of reason s we would want to do this.

Migrations are Database Agnostic – you can write your schema in L-Migrator and deploy it against, SQL-Lite, MySQL, or PosgreSQL.

No need to remember what the syntax was for that SQL syntax.

Migrations allow for different versions of a database. You can easily add the schema to the database and roll up or roll down your database version. It beats writing alter statements.

Dovetails with iterative development like Agile or Scrum.

Working on a team, when you develop against a local database, once you checkout your code from the repository, all you have to do is run the migrations to bring your copy of the database up to speed.

Allows Adaptive Planning

1.4 How does this factor into the workflow?

Underpinning the Agile development manifesto is the methods which increase effectiveness in self organizing teams. People with different skills and needs need to be able to communicate clearly and effectively.

Evolutionary design techniques recognize that people need to learn things by trying things out. Take a moment if you will and imagine back to when you were first starting to learn Lasso techniques. Remember how it was so important to get code snppets and samples before you were able to code efficiently? For people that are stil learning you recognize the importance of the sites like Lasso Forge, Tagswap, and LassoBin as well as the Lasso Language Guide and more.

Most develops work by coding on a local machine, either a workstation or a laptop and pushing those changes to a server, either over a FPT connection, over ssh or using a deployment methodology using a version control system like git or subversion.

In a tradtional development environment the programmer would work on their own machine developing and modifying the database schema from the command line or a utility such as Navicat or CocoaMySQL. When new code is deployed that would involve new database table s and schema changes to the master server, the devloper would have to perform a schema dump, and then load that SQL schema change on the server or by making the changes to the schema on both systems simultaneously.

In larger organizations with a DBA (Database Administrator), changes must be coordinated with the DBA and this takes time and communiation. Migrations allow developers to submit their changes, and have the DBA review and either commit or modify their changes.

Wouldn't it be great to be able to automate that?

1.5 What is a database migration?

A data base migration is a text file that connects to a database adaptor file and produces database change schema (CREATE TABLE, ALTER TABLE, ADD COLUMN etc.) from the text file. This allows the same text file to create schema by connecting to multiple database types through the adapter to migrate database changes from a laptop ot workstation, using SQLLite, or MySQL to a production system using PostgreSQL, SQL Server, Oracle or MySQL.

2.0 Structure of L-Migrator

The structure of L-Migrator is contained in a zip file which when uncompressed leaves you with a folder called /dbm/ . This is placed at the root of the site. Let's examine the contents:

`/dbm/resources/`

This folder contains the various resources for the package to work:

`/dbm/resources/migrator.lasso`

This is the initial file that is loaded that triggers the migrations and is the main display page for the package.

`/dbm/resources/migrator.cnfg`

This is the config file. It included a directive to create the initial table that tracks the migrations. In future versions we may include this initial config as a migration of its own. Note that it is not encased in LassoScript tags or square brackets.

```
dbName    = l_migrator
migAdaptor  = mysql
migPath   = /dbm/migrations/
messageOn = yes
#---------------------------------------------------------------------
# Before you can use L-Migrator, add this table to your database
#
# CREATE TABLE `_schemaInfo` (
#    `id` int(4) unsigned NOT NULL default '0',
#    `version` smallint(5) unsigned NOT NULL default '0'
# ) ENGINE=MyISAM DEFAULT CHARSET=utf8
#
# INSERT INTO `_schemaInfo` (`id`,`version`) VALUES ('1','0');
#---------------------------------------------------------------------
```

dbname is obviously the name of the database that need to be migrated and will be connected to. MigAdapter is the name of the database type that is to used. If you are using GIT or subversion or using mirror in an FTP applicaiton you would want to be excluding .cnfg files from those processes, In later versions we may put this variable into some type of environment file so that it is dependent on the production value of the environment, development , test or production, etc. Migration path should bemaintained unless you have changed the way the package is structured. MessageOn is a new variable we have introduced to show the message array at the bottom of the page after the migrations have run.

`/dbm/resources/style.css`

This created the styles that are used in the display of the migrations. Self explanatory.

`/dbm/resources/*.gif`

Graphics for triggering migrations.

`/dbm/resources/schemaView.dsp`

This draws the tables that show the schema on the page when you view the schema.

`/dbm/resources/migratorStartup.lgc`

The migrator startup file contains the logic necessary to prime the pump for the migrations, the configuration file is parsed and

```
<?lassoscript
auth;

var:'configFile' = 'migrator.cnfg';
//--------------------------------------

var:
   'dbName'    = string,
   'migAdaptor'  = string,
   'migPath'     = string,
   'configData'  = null,
   'thisLine'    = string,
   'varPair'   = pair,
   'varName'   = string,
   'message'   = array,
   'varValue'    = string;

if: (file_exists:$configFile);
   $message->Insert:('Config File Exists');
   $configData = string:(file_read: $configFile);
   $configData->(removeleading:bom_utf8);

// split lines

   $configData->trim;
   $configData = (string_replace: $configData, -find='\r\n', -replace='\r');
   $configData = (string_replace: $configData, -find='\n', -replace='\r');
   $configData = ($configData->(split:'\r'));

// remove comments

   if: $configData && ($configData->type == 'array');
      loop: -from=$configData->size, -to=1, -by=(-1);
         $thisLine = $configData->(get:loop_count);
         if: ($thisLine == '')
            || ((string_findregexp:
               $thisLine,
               -find='\\S+')->size == 0)
            || ($thisLine >> 'output_none')
            || ($thisLine->(beginswith:'#'))
```

```
        || ($thisLine->(beginswith:'//'));
          $configData->(remove:loop_count);
      /if;
    /loop;
  /if;
//make vars

  iterate: $configData, $varPair;
    $varName  = ($varPair->(split:'='))->get:1;
    $varValue = ($varPair->(split:'='))->get:2;
    $varName->trim;
    $varValue->trim;
    (var:$varName) = $varValue;
  /iterate;
/if;
//--------------------------------------
library:'/dbm/_resources/migrationEngine.ctyp';
$message->Insert:('End of Migrations Engine');
library:'/dbm/_resources/migrationCommands.ctyp';
$message->Insert:('End of Migration Commands');
library:'/dbm/_resources/migrationAdaptor_MySQL.ctyp';
$message->Insert:('End of Migration Adapters');
library:'/dbm/_resources/migrationSchema.ctyp';
$message->Insert:('End of Schema Reading');
//--------------------------------------
var:
  'migrator'       = (migrationEngine:
                -path     = $migPath,
                -adaptor  = $migAdaptor,
                -db       = $dbName),
  'runResult'      = string,
  'tableSchemaData'  = null,
  'tableSchemas'   = map,
  'thisTable'      = string;
$migrator->getCurrentVersion;
$migrator->findMigrationLimits;
?>
```

The next component, the migration adapter is the portion of the system that actually creates the syntax to communicate withthe database. As of now only the MySQL adapter is completed, additional help is needed to create the adaptor files to connect to the other datbases, such as Oracle, SQL Server, SQL Lite and PostGreSQL.

`/dbm/resources/migrationAdaptor_mysql.ctyp`

This file is a custom type that is extremely long, it does not need to be added into here , however it's important that we examine one of the tags that's incuded in thte type, the `convertFieldType` tag

```
define_tag:'convertFieldType',
  -required = 'fieldType', -type = 'string';

//since my projects and most of the Lasso community's projects
//are on MySQL, we'll let MySQl column types prevail.
//so, other than showing an example of how string is converted to varchar
//this converter pretty much returns the same data type to feed it
//doing this for all adaptors pretty much eliminates the db agnostic
//intention of an adaptor, but the architecture is here to support it

  select: #fieldType;
    case:'string';     return:'varchar';
    case:'fixedString';  return:'char';
    case:'radioBtns';    return:'enum';
    case:'checkBoxes';   return:'set';
```

```
    case:'text';          return:'text';
    case:'tinytext';   return:'tinytext';
    case:'smalltext';     return:'smalltext';
    case:'mediumtext';    return:'mediumtext';
    case:'largetext';     return:'longtext';

    case:'blob';          return:'blob';
    case:'tinyblob';   return:'tinyblob';
    case:'smallblob';     return:'smallblob';
    case:'mediumblob';    return:'mediumblob';
    case:'largeblob';     return:'longblob';
    case:'binary';     return:'blob';

    case:'int';           return:'int';
    case:'integer';       return:'int';
    case:'tinyInt';       return:'tinyint';
    case:'smallInt';   return:'smallint';
    case:'mediumInt';     return:'mediumint';
    case:'largeInt';   return:'bigint';
    case:'decimal';       return:'decimal';
    case:'float';         return:'float';
    case:'double';     return:'double';

    case:'date';          return:'date';
    case:'time';          return:'time';
    case:'year';          return:'year';
    case:'datetime';   return:'datetime';
    case:'timestamp';     return:'timestamp';

    case;                 return:#fieldType;
  /select;

/define_tag;
```

Of equal importance is the `setDefaultFor` ctag, it sets the default values for each field that the specific database needs .

```
define_tag:'setDefaultFor',
  -required = 'fieldType', -type = 'string';

  select: #fieldType;
    case:'string';     return:' "" ';
    case:'fixedString';  return:' "" ';
    case:'radioBtns';     return:' "" ';
    case:'checkBoxes';    return:' "" ';

    case:'text';          return:' "" ';
    case:'tinytext';   return:' "" ';
    case:'smalltext';     return:' "" ';
    case:'mediumtext';    return:' "" ';
    case:'largetext';     return:' "" ';

    case:'blob';          return:' "" ';
    case:'tinyblob';   return:' "" ';
    case:'smallblob';     return:' "" ';
    case:'mediumblob';    return:' "" ';
    case:'largeblob';     return:' "" ';
    case:'binary';     return:' "" ';

    case:'int';           return: 0;
    case:'integer';       return: 0;
    case:'tinyInt';       return: 0;
    case:'smallInt';   return: 0;
    case:'mediumInt';     return: 0;
    case:'largeInt';   return: 0;
```

```
    case:'decimal';       return: 0.0;
    case:'float';         return: 0;
    case:'double';     return: 0;

    case:'date';          return:'0000-00-00';
    case:'time';          return:'00:00:00';
    case:'year';          return:'0000';
    case:'datetime';   return:'0000-00-00 00:00:00';
    case:'timestamp';     return:'0000-00-00 00:00:00';

    case;                 return:' "" ';
  /select;
/define_tag;
```

One thing that is important to note here is the way the L-Migrator commands are mapped to the Database specific commands for their respective syntax. The other commands in this file are are ctags that allow the type to generate and output the syntax necesary to execute the commands.

Lets examine the other files in the dbm/resources as well.

`dbm/resources/migrationCommands.ctyp`

This file is the string and glue that binds the migrations to the database adapter. It's rather long to look at the whole thing but as with the migrations:. Here's the `addTable` ctag from the `migrationCommands.ctyp`

```
define_tag:'addTable',
  -required = 'name',     -type = string;

  local:
    'thisQueryStatement'  = string,
    'thisQueryError'   = false;

  #thisQueryStatement = (self->'queryAdaptor')->(addTable: params);

  self->(execute: #thisQueryStatement,
       -operation = 'addTable',
       -itemName  = #name);

/define_tag;
```

Not real impressive in itself but lets look at the execute tag that is called from within the `addTable` tag. See how the `addTable` tag calls the `queryAdapter` which gets the job done. This is the power of Lasso's custom types and the way the reflexive properties work. Now that the queryAdapter has returned the params for addtable the query is executed by the execute tag.

```
define_tag:'execute',
  -required = 'thisQueryStatement', -type = string,
  -optional = 'table',         -type = string,
  -optional = 'operation',       -type = string,
  -optional = 'itemName',        -type = string;


  !(params >> '-table')
    ? (local:'table') = (self->'tblName');


  !(params >> '-operation')
    ? (local:'operation') = 'custom query';

  !(params >> '-itemName')
    ? (local:'itemName') = '';

  local:'executionError' = string;
```

```
  (self->'queryStatements')->(insert:#thisQueryStatement);

  inline:
    -database   = (self->'dbName'),
    -table    = #table,
    -sql      = #thisQueryStatement;

    #executionError = error_currentError;
           $message->Insert:('SQL SChema');
           $message->Insert:('  ' action_statement);
           $message->Insert:('  ' error_currenterror);
  /inline;

  (#executionError == 'No Error')
    ? (self->'queryErrors')->(insert: #operation + ' ' + #itemName + ' succeeded')
    | (self->'queryErrors')->(insert: #operation + ' ' + #itemName + ' failed due to
#executionError);

/define_tag;
```

We've added some hooks here that help us with out debugging in the `migrator.lasso` file. See how our variable `$message` uses the insert statement to add to the array of messages from the migration? We can toggle this with our variable in the `migrator.cnfg` file.

```
           $message->Insert:('SQL Schema');
           $message->Insert:('  ' action_statement);
           $message->Insert:('  ' error_currenterror);
```

The migrationEngine.ctype file is the heart of the system and handles all the transactions to and from the migrations, it writes the changes to the _schmaInfo table and handles the errors produced in the migrations.

`/dbm/resources/migrationEngine.ctyp`

Lets look at a couple key comonents of the `migrationEngine.ctyp`

Update Schema

```
define_tag:'updateSchema';
  local:
    'migrationClassName'    = string,
    'migrationVersion'      = integer,
    'migrationObj'        = null,
    'migrationErrors'       = array,
    'lastMigrationSucceeded'  = true;


  iterate: (self->'migrationClassNames'), #migrationClassName;

    #migrationVersion = integer:((#migrationClassName->(split:'_'))->get:1);

    if: #lastMigrationSucceeded
      && (#migrationVersion > (self->'currentVersion'))
      && (#migrationVersion <= (self->'migrateToVersion'));

      library:((self->'classPath') + #migrationClassName + '.ctyp');

      #migrationObj = (\#migrationClassName)->asType;
      #migrationObj->(init:
        -adaptor  = (self->'adaptorName'),
        -db       = (self->'dbName'));


      #migrationObj->update;
      #migrationErrors = #migrationObj->getErrors;
```

```
      (self->'allQueryErrors')->insert: (#migrationClassName + '->update') = (#migrationObj-
>'queryErrors');
      (self->'allQueryStatements')->insert: (#migrationClassName + '->update') = (#migrationObj-
>'queryStatements');


      #lastMigrationSucceeded = self->(analyzeResultOf: #migrationErrors);
      $message->Insert:('Last Migration Succeeded ' #lastMigrationSucceeded);
      if: #lastMigrationSucceeded;
        inline:
          -database   = (self->'dbName'),
          -table    = '_schemaInfo',
          -keyField = 'id',
          -keyValue = '1',
          'version' = #migrationVersion,
          -update;
          $message->Insert:('Update Schema');
          $message->Insert:('  ' action_statement);
          $message->Insert:('  ' error_currenterror);
        /inline;
      /if;
    /if;
  /iterate;
/define_tag;
```

and the rollback portion of this.

```
define_tag:'rollbackSchema';

  local:
    'migrationClassName'    = string,
    'migrationVersion'      = integer,
    'migrationObj'        = null,
    'migrationErrors'       = array,
    'lastMigrationSucceeded'  = true;

  iterate: (self->'migrationClassNames'), #migrationClassName;

    #migrationVersion = integer:((#migrationClassName->(split:'_'))->get:1);

    if: #lastMigrationSucceeded
      && (#migrationVersion <= (self->'currentVersion'))
      && (#migrationVersion > (self->'migrateToVersion'));

      library:((self->'classPath') + #migrationClassName + '.ctyp');

      #migrationObj = (\#migrationClassName)->asType;
      #migrationObj->(init:
        -adaptor  = (self->'adaptorName'),
        -db       = (self->'dbName'));


      #migrationObj->rollback;
      #migrationErrors = #migrationObj->getErrors;


      (self->'allQueryErrors')->insert: (#migrationClassName + '->rollback') = (#migrationObj-
>'queryErrors');
      (self->'allQueryStatements')->insert: (#migrationClassName + '->rollback') = (#migrationObj-
>'queryStatements');
```

```
        #lastMigrationSucceeded = self->(analyzeResultOf: #migrationErrors);

        if: #lastMigrationSucceeded;
          inline:
            -database   = (self->'dbName'),
            -table    = '_schemaInfo',
            -keyField = 'id',
            -keyValue = '1',
            'version' = #migrationVersion-1,
            -update;
            $message->Insert:('RollBack Schema');
            $message->Insert:('  ' action_statement);
            $message->Insert:('  ' error_currenterror);
          /inline;
        /if;
      /if;
  /iterate;
/define_tag;
```

You'll notice that in these two tags also we have added the hooks for the messaging tags.

3.0 Writing Migrations

Writing migrations is as simple as creating a glossary item in BBEdit or snippets in Coda, if anyone is interested in creating syntax snippets. This way you can create migration steps with simple keyboard shortcuts.

Migrations are stored in the `/dbm/migrations/` directory.

Migrations follow a specific naming constraint. Note the examples here:

```
001_Begin.ctyp
002_createTable.ctyp
003_createTable.ctyp
004_createTable.ctyp
005_createTableMonkey.ctyp
006_createMore.ctyp
```

When writing the migrations the naming should be the numeric value of the migration, and the name after that can be anything BUT that name must match the name of the custom type inside that migrations file.

Lets look at some simple migrations:

```
<?LassoScript
  define_type:'001_Begin', 'migrationCommands';

//-------------------------------------------------------------------------------   define_tag:'update';

  /define_tag;


//-------------------------------------------------------------------------------
  define_tag:'rollback';

  /define_tag;
/define_type;
?>
```

Notice in the migrations how the enclosing type always matches the filename. This is essential! Also note that the two tags in the migrations file will always be update and rollback.

Here's the syntax for a simple migration:

```
<?lassoscript
define_type:'002_createTable', 'migrationCommands';

//-------------------------------------------------------------------------------
```

```
  define_tag:'update';

  // define tblName once for multiple operations,
  // or with -table in each operation
  self->(addTable:
    -table = 'concerts',
    -name  = 'ID',
    -type  = 'int',
    -size  = '11',
    -autoinc=true,
    -pkey=true);
    self->'tblName' = 'concerts';
    self->(addField:
      -name  = 'rcrdLock',
      -type  = 'fixedString',
      -size  = '1');

    self->(addIndex:
      -name    = 'rcrdNo',
      -field   = 'ID');

  /define_tag;

//-----------------------------------------------------------------------------
  define_tag:'rollback';

    self->(removeTable: -name='concerts');

  /define_tag;
/define_type;
?>
```

When using the addtable command it requires the option to add the first field, we use an autoincrement with a primary key. This is a new feature that was not available in the original release of PageBlocks or L-Migrator. Please see the L-Migrator docs to explain exaclt how the migrations are written.

4.0 Executing Migrations.