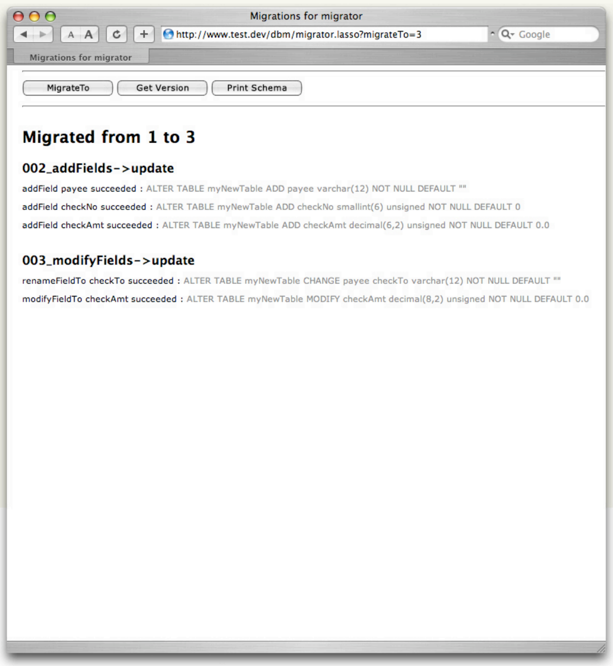
L-Migrator beta 1

Schema Migrations Utility for Lasso Professional Server 8.1/8.5

Rev 1 • March 25, 2007 • Greg Willits

http://www.l-unit.org/downloads/l-migrator/l-migrator.zip



```
Migrated from 1 to 3

002_addFields->update
addField payee succeeded : ALTER TABLE myNewTable ADD payee varchar(12) NOT NULL DEFAULT ""
addField checkNo succeeded : ALTER TABLE myNewTable ADD checkNo smallint(6) unsigned NOT NULL DEFAULT 0
addField checkAmt succeeded : ALTER TABLE myNewTable ADD checkAmt decimal(6,2) unsigned NOT NULL DEFAULT 0.0

003_modifyFields->update
renameFieldTo checkTo succeeded : ALTER TABLE myNewTable CHANGE payee checkTo varchar(12) NOT NULL DEFAULT ""
modifyFieldTo checkAmt succeeded : ALTER TABLE myNewTable MODIFY checkAmt decimal(8,2) unsigned NOT NULL DEFAULT 0.0
```

Losing track of changes to your data schema can be problematic. If you're maintaining instances for test and production, or have several developers to keep synchronized, well, that's just plain asking for trouble.

Migrations is a methodology for keeping track of each change in a database, helping multiple instances of databases stay up to date with schema changes, and placing schema management under version control.

The L-Migrator utility and framework provides Lasso developers with a toolset to integrate migrations schema management in Lasso projects.

Why Use Migrations?

Your code is under version control, so why shouldn't your database be?

If you're using version control, you have the ability to save small incremental stages of development. As you introduce changes to your code you may find yourself stuck with a change that seems to have broken things unexpectedly. Whether you need to roll back to regroup, or just to compare previous code to current code, version control proves to be a very useful capability. Additionally, version control can be used to keep multiple developers coordinated.

Migrations is method for adding these kinds of capabilities to your database schema. As you evolve an application, it's data schema usually evolves as well. Quite often that evolution is performed by making changes directly to the database via command line. Let's say you've made a couple dozen changes to the schema while developing the next update to your app. When it is time to apply those same changes to a test site or to the production database, are you going to remember ever change you made?

Migrations is way to maintain a functional history of schema changes which can be applied to multiple instances of databases. Once you've created a migration for your development copy, you can run that migration against the test and production database and know you'll end up with exact copies.

What Are Migrations?

Quite simply, a migration is a simple code file which makes the schema changes to your database programmatically. By scripting these updates rather than applying them directly through a database utility, the changes are captured for reuse and for a self documenting history. Each change script is kept in a file a with a sequential number. Each number represents the current version number of the database.

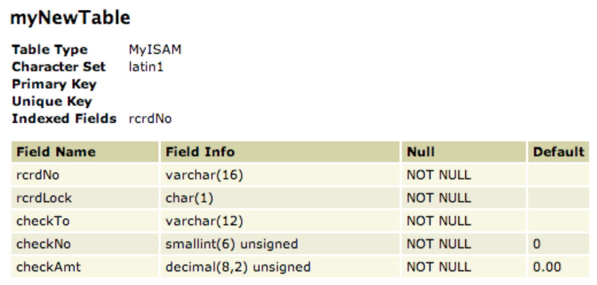
A utility like L-Migrator organizes these scripts, provides the tools to execute them, and most importantly provides a framework in which to author these scripts to make them easy to write and also make them largely database agnostic so writing scripts is almost identical regardless of the database engine.

Working with L-Migrator

The L-Migrator application organizes, executes, and displays the results of the developer's migrations,

The L-Migrator application runs in a web browser on the developer's local machine or on a test server on the LAN. (It can run on a remote server, but the app isn't security hardened, and it's not recommended to run it across the internet).

The application has 3 simple commands: migrateTo, getVersion, and printSchema. MigrateTo will update or rollback the database to the version specified. GetVersion will simply retrieve the current version and display it. PrintSchema will grab the schema for each table in the database and print an HTML table suitable for printouts.



myNewTable

Table Type	MyISAM
Character Set	latin1
Primary Key	
Unique Key	
Indexed Fields	rcrdNo

Field Name	Field Info	Null	Default
rcrdNo	varchar(16)	NOT NULL	
rcrdLock	char(1)	NOT NULL	
checkTo	varchar(12)	NOT NULL	
checkNo	smallint(6) unsigned	NOT NULL	0
checkAmt	decimal(8,2) unsigned	NOT NULL	0.00

In this Beta release, the MigrateTo command is executed by modifying the version value in the URL where `migrator.lasso?migrateTo=4` would modify the database to version 4.

Installing L-Migrator

In Lasso SiteAdmin, use the File Extensions setup panel to allow Lasso to process .ctyp, .dsp, and .lgc files. Additioally allow Lasso file tags to work with .cnfg file extensions.

Copy the /dbm/ folder (it stands for database migrator) into your web application's root folder.

Open the migrator.cnfg file and edit the three configuration lines. L-Migrator is database agnostic. It can work with any SQL database, but currently only the MySQL is completed. (If you can contribute a little time to create an adaptor for other database please contact me). So, the adaptor setting should be `mysql`. If you go with a default install setup, the path value should stay as `/dbm/migrations/`. The thing you need to set is the database name value. Right now, L-Migrator will only work with one database at a time.

Inside the migrator.cnfg file you'll find a CREATE and an INSERT statement for a table named _schemaInfo. All instances of your database (dev, test, production, etc) need to have that table included. This is where L-Migrator stores the current version of the database. Eventually, additional info might be stored here as well.

Once you have these steps completed, you should be able to load the L-Migrator page at your root domain /dbm/migrator.lasso.

There are a few demo scripts included in the installation package. If you execute the command `migrator.lasso?migrateTo=1`, then a sample table named myNewTable will be added to your database. You can then play with all of L-Migrator's commands and set `migrateTo` from 1 through 5 to see how it works.

Writing Migration Scripts

Start by looking at the sample scripts in the /dbm/migrations/ folder. You'll notice that each script begins with a sequential number of 001_, 002_, 003_, etc. Each script must be numbered like this. This is what allows L-Migrator know which script will upgrade the database to or roll it back to that version number. The rest of the file name is inconsequential and can be anything you need to help you know what that script does. The file must end with a .ctyp extension.

Each script is a simple ctype definition that inherits the base type of `migrationCommands`. Each script has member tags `->update` and `->rollback`. Inside the `->update` tag you write the steps to migrate the schema to the next version you need. The `->rollback` tag is used to restore the database to the state prior to the `->update` tag's steps. Whatever you do in `update`, you need to write a step to counteract it in `rollback`. The example scripts should help make this process clear.

Below are a basic set uof update and rollback scripts.

```
define_tag:'update';

  self->(addField:
    -table  = 'myNewTable',
    -name   = 'payee',
    -type   = 'string',
    -size   = '12');

  self->(modifyField:
    -table  = 'myNewTable',
    -name       = 'checkAmt',
    -type       = 'decimal',
    -precision = '8',
    -scale      = '2',
    -unsigned   = true);

/define_tag;

define_tag:'rollback';

  self->(removeField:
    -table  = 'myNewTable',
    -name   = 'payee');

  self->(modifyField:
    -table  = 'myNewTable',
    -name       = 'checkAmt',
    -type       = 'decimal',
    -precision = '6',
    -scale      = '2',
    -unsigned   = true);

/define_tag;
```

In the update tag we added a field and we modified a field. In the rollback tag you can see that reversing the addField is a simple removeField command. For the modified field, what was it before we modified it? You'll have to know what that is, and use a modifyField command to set it back to what it was.

You'll notice that each command requires a `table` parameter. That gets tedious having to repeat it, so if each command in the tag applies to the same table, you can eliminate the redundant parameters and use the following line one time before using any commands:

```
    (self->'tblName')='myNewTable';
```

The next sections covers the available commands and their options.

Schema Modification Commands

L-Migrator is written to mostly be database agnostic. Most script details you'll write will not be database engine specific. However, there will be times when you'll need to issue database-specific commands, and that can be done. the following commands are what this version of L-Migrator currents supports.

addField

```
-required = 'name'
-optional = 'table'
-optional = 'type'
-optional = 'size'
-optional = 'null'
-optional = 'default'
-optional = 'unsigned'
-optional = 'precision'
-optional = 'scale'
-optional = 'after'
```

- name is the field name
- table is the table name
- type is the column data type. See the data type reference below. If not specified, it will be set to string.
- size is the column width. It is required for all fields except the decimal type of fields which use precision and scale instead.
- null is set to True or False to indicate NULL or NOT NULL settings. If not specified, it is set to NOT NULL.
- default is used to set an explicit default value. If not specified a standard value will be used.
- unsigned is set to True or False for numeric data types only.
- precision sets the column width for numeric fields with fractional values. Precision is the 8 out of the 8,2 setting value.
- scale sets the number of fractional value columns. Scale is the 2 out of the 8,2 setting value.
- after is the fieldname of the field that the new field should be inserted after. If not specified the new field will be added based on the bahavior of the database engine (usually the end of the table)

```
  self->(addField:
    -table  = 'myNewTable',
    -name   = 'payee',
    -type   = 'string',
    -size   = '12',
    -null   = false,
    -after  = 'checkNumber');

  self->(addField:
    -table     = 'myNewTable',
    -name      = 'checkAmt',
    -type      = 'decimal',
    -precision = '8',
    -scale     = '2',
    -unsigned  = true);
```

renameField

Uses the same options as addField except that instead of name, we have

```
-required = 'fromName'
-required = 'toName'
```

With MySQL (and perhaps others?), a rename field requires that all definition elements be repeated, so in effect you have to declare all the same things you would if you were using addField.

```
self->(renameField:
    -table     = 'myNewTable',
    -fromName  = 'checkAmt',
    -toName    = 'checkAmount',
    -type      = 'decimal',
    -precision = '8',
    -scale     = '2',
    -unsigned  = true);
```

removeField

```
-required = 'name'
-optional = 'table'

self->(removeField:
    -table  = 'myNewTable',
    -name   = 'payee');
```

modifyField

Uses the same options as addField (except `after` is not used). With MySQL (and perhaps others?), a modify field requires that all definition elements be specified, so in effect you have to declare all the same things you would if you were using addField.

```
self->(modifyField:
    -table     = 'myNewTable',
    -name      = 'checkAmt',
    -type      = 'decimal',
    -precision = '6',
    -scale     = '2',
    -unsigned  = true);
```

addTable

In MySQL, the create table statement must include the definition of at least one field. So, the options for the addTable command are the same as as addField except `after` is not used.

```
self->(addTable:
    -table  = 'secondNewTable',
    -name   = 'rcrdNo',
    -type   = 'string',
    -size   = '16');
```

renameTable

```
-required = 'fromName'
-required = 'toName'

self->(renameTable:
    -fromName   = 'usrs',
    -toName     = 'users');
```

removeTable

```
-required = 'name'

self->(removeTable:
    -name   = 'usrs');
```

addIndex

```
-optional = 'table'
-required = 'name'
-optional = 'type'
-required = 'field'
-optional = 'length'
```

For MySQL the only required elements are the index name and the field to be indexed. The `type` option accepts the string values of UNIQUE, FULLTEXT, or SPATIAL.

```
self->(addIndex:
  -table    = 'myNewTable',
  -name     = 'rcrdNo',
  -field    = 'rcrdNo');
```

removeIndex

```
-optional = 'table'
-required = 'name'

self->(removeIndex:
  -table    = 'myNewTable',
  -name     = 'rcrdNo');
```

execute

The `execute` command will run any SQL statement provided as a single string. This allows migrations to include features not yet abstracted, or that are too cumbersome to abstract.

If you only ever work with one database engine, you might prefer to do all your migrations with execute commands. There's nothing wrong with that. The abstracted commands are beneficial for when you work with multiple engines so you don't have to use engine-specific syntax.

Data Type Mappings for MySQL

The values on the left are valid values for the `type` parameter in fields commands like addField. the right column is how the map to MySQL column types. (As new adaptors are created, the options on the left will stay constant). If a value is used that is not in the left column, then that value will be passed as is in the query. This allows the migrations to pass values that the migrator doesn't yet abstract.

`string`	`varchar`
`fixedString`	`char`
`radioBtns`	`enum`
`checkBoxes`	`set`
`text`	`text`
`tinytext`	`tinytext`
`smalltext`	`smalltext`
`mediumtext`	`mediumtext`
`largetext`	`longtext`
`blob`	`blob`
`tinyblob`	`tinyblob`
`smallblob`	`smallblob`
`mediumblob`	`mediumblob`
`largeblob`	`longblob`
`binary`	`blob`
`int`	`int`
`integer`	`int`
`tinyInt`	`tinyint`
`smallInt`	`smallint`
`mediumInt`	`mediumint`
`largeInt`	`bigint`
`decimal`	`decimal`
`float`	`float`
`double`	`double`
`date`	`date`
`time`	`time`
`year`	`year`
`datetime`	`datetime`
`timestamp`	`timestamp`

Inside the L-Migrator Application

This release is a beta product. There's more to do to make the GUI pretty, and also to allow an installation to work with multiple databases conveniently.

The bulk of L-Unit's innards can be found in the /_resources/ folder. To read sequentially through the code, start with the file /migrator.lasso which immediately includes /_resources/migrator.lgc which is the main application controller. It's written as a standard Lasso include, but the majority of the application logic is object-oriented.

The migrator.lasso is a bit of hack at this point as it is the rough idea of what will become little fancier UI. Nevertheless it is functional.

Most of this code will likely be reorganized for the final version. This version is pretty much the straight first draft "scratchpad" code.

Application Source Code

- `migrationEngine` — the main controller which manages the update and rollback processes and a few other general tasks
- `migrationCommands` — the base cass for migration scripts. This class holds the abstracted fields and table modification commands.
- `migrationAdaptor` — provides the translation of the commands into database specfic queries.
- `migrationSchema` — extracts schema details for printouts
- `migratorStartup` — some basic procedural code for kick-starting the application by loading the config data and the class libraries
- `schemaView` — the html template for the schema table printouts.