2011-09-29

Migrate

Adempiere Migration Tool

Tool for Upgrading, Transferring, or Converting Databases

User Manual

Stefan Christians

Migrate User Manual

Adempiere Migration Tool
by Stefan Christians

2011-09-29

This program is part of Adempiere ERP Bazaar
http://www.adempiere.org

Contributors:
Stefan Christians

Sponsors:
K.K. Alice

Tool for Upgrading, Transferring, or Converting Databases

While tools such as migration scripts for upgrading or DDLUtils for converting databases are suitable for Adempiere's application developers to maintain the seed database, they are a bit challenging for the average user to maintain their live database.

Migrate provides a graphical user interface for upgrading databases.

It can also be used for converting between database vendors (like Oracle and postgreSQL) or applications (like Compiere and Adempiere).

Table of Contents

List of Figures

List of Tables

1

Introduction

What is Data Migration?

Welcome to MIGRATE, ADEMPIERE's universal migration tool for upgrading, transferring, and converting databases.

“Migrating” means moving from one place to another. Specifically for databases, “migrating data” can have either of the following meanings:

a. Transferring

   The process of transferring data between storage types or computer systems. Like copying data from hard disk to floppies, or from one server to another. This is commonly referred to as Copying, Transferring, Moving, or Replicating.

b. Converting

   The process of converting data from one format or system to another. For example, if your company changes its database system from a proprietary vendor to an open source alternative, the data needs to be manipulated to fit into the new database's format. This is commonly referred to as Converting or Translating.

c. Upgrading

   The process of upgrading a database's structure to enable new or different functionality. Newer software versions may have introduced new functionality or bug fixes which require a different database structure than was available in previous versions. In such cases, your database needs to be adjusted to the new structure so that it can be correctly utilized by the new software version. This is commonly referred to as version migration or upgrading.

MIGRATE can do all three types of migration, therefore we call it a Universal Migration Tool.

You can use MIGRATE for following tasks:

- converting your database from ORACLE to POSTGRESQL
- converting your database for use by COMPIERE to use by ADEMPIERE
- upgrading your database for use by a different ADEMPIERE version

History

Before ADEMPIERE forked from the COMPIERE project, version migration was available to COMPIERE users for a fee. The user had to load the newest reference database, which was distributed with the COMPIERE software package, and then start a closed-source proprietary migration program, which would check the license validity and download SQL scripts from COMPIERE's web server to correctly upgrade the live database by copying the reference database's structure. This was done through a graphical user interface which was straight-forward and worked very well, but it had one disadvantage (apart from the obvious cost factor and being closed-source): it was not very flexible.

This is inherent in the nature of scripts – they run a number of commands in sequence to get from origin A to target B. It is not possible to get to a target C or D. For COMPIERE it meant that it was only possible to upgrade from older versions to the newest version, not to a version in between or downgrade to a lower version. You had to load the newest reference database to work with the newest scripts. As a consequence, you were forced to do a full upgrade every time, introducing many bugs and trial features, which was not ideal for business environments.

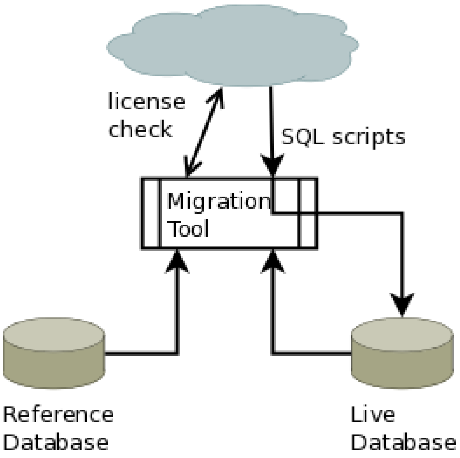


Figure 1.1. COMPIERE's proprietary upgrade service

Since COMPIERE's version migration was proprietary, it was not included in the code base from which ADEMPIERE forked out, and a new solution had to be found quickly to be able to do any version migration at all. Karsten Thiemann programmed a nice little tool called DBDIFFERENCE, which would generate SQL-scripts based on the structural differences between the reference and target database. The SQL-scripts would then be manually applied to upgrade the target database.

As the user is actively involved in SQL-script generation and can also review and edit the scripts before they are applied to the target database, there is of course much more flexibility and control than was possible with COMPIERE's solution. But for the casual database user the task was daunting, and real world implementations with numerous extensions and customizations messing up DBDIFFERENCE's logic required heavy interventions which were not always feasible. DBDIFFERENCE also relied mainly on the reference database's design, without giving much thought to the contents of the Application Dictionary, a storage of meta-data and rules defining the data's use by ADEMPIERE where also most customizations are defined.

Once your data reflected the structure required by an ADEMPIERE release, things got easier because you could use scripts pre-generated by the ADEMPIERE team (if you took good care of your customizations), but getting to that point was a major task.

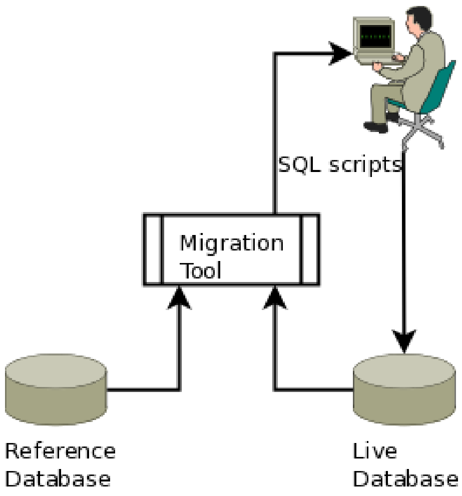


Figure 1.2. ADEMPIERE's script-based solution required massive user intervention

To make things worse, with ADEMPIERE you had the choice of using POSTGRESQL, a free and open-source database system. So if you previously used a proprietary database system, you had to do a conversion migration to translate your data to POSTGRESQL. Another set of tools (DDLUTILS) was used for this purpose, also requiring heavy user intervention.

So although the migration tools introduced by ADEMPIERE were very flexible and in many cases proven to be workable, they lacked the ease of use old hands were accustomed to from COMPIERE's version migration tool. Being very suitable for ADEMPIERE's application developers to maintain the seed database, they are a bit challenging for the average ADEMPIERE user.

MIGRATE solves these disadvantages by providing a graphical user interface which makes it easy to use for the uninitiated, and giving up on the script concept entirely by using algorithms instead. Also MIGRATE uses a reference database against which the live database is checked, but the algorithms also make heavy use of the meta-data available in ADEMPIERE's Application Dictionary and thus are also aware of any customizations and extensions. Any changes to the live database are made directly, no scripts are generated or need to be applied.

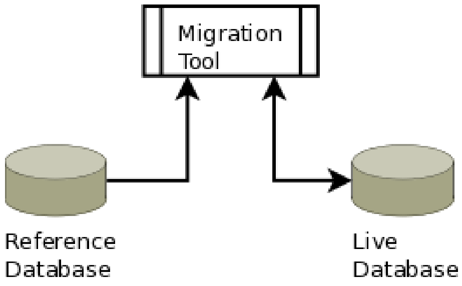


Figure 1.3. MIGRATE simplifies automated migration using algorithms instead of scripts

Functionality

Transfer Mode

MIGRATE reads the structure and data from a source database and writes it into a target database. In its most simple form, this corresponds to what we previously described as Transfer Migration.

You can therefore use migrate to transfer or copy a database from server A to server B, though it is not recommendable. This kind of migration is very straight-forward and does not require any overhead logic, and the tools provided by your database vendor (**exp** and **imp** for ORACLE, **pg_dump** and **pg_restore** for POSTGRESQL) are much more suitable and extremely efficient. MIGRATE is much too slow and bulky for this task.

However, MIGRATE comes in handy if the source and target are for different database vendors, for example if you want to transfer your data from ORACLE to POSTGRESQL. This is what we previously described as Conversion Migration.

In this case, MIGRATE reads content from the source database, translates it to a format understood by the target database, and then writes it to the target.

Note that although MIGRATE attempts to correctly translate content to the target's format, this is not always possible. Converting data types and indexes is relatively safe, converting views is a bit more difficult, and translating functions and procedural languages, such as from PL/SQL to PL/PGSQL, is virtually impossible if you do not program a full-fledged command interpreter. Consequently, the user will be given warning messages to check on views that have been translated, but the translation of functions is currently not implemented at all.

In both above cases, data is read from the source and a new target is created, or an existing target is overwritten, to contain the source's data. The only difference is whether or not the source and target vendors are different. In MIGRATE, this kind of migration is called “Transfer Mode”.

Upgrade Mode

Things get more interesting if the target does not get overwritten, but if source data is merged into existing target data: The table structure etc. of your live data in the target table is modified to reflect the structure provided as reference from the source table. Data records missing in the target will be added from the source. Views and Functions defined in the target will be replaced by those defined in the source. So if a new ADEMPIERE version required new tables or views or functions, that functionality would be copied to your live data from the source database. We therefore call this kind of migration “Upgrade Mode”, and the source is the reference database and the target is your live database.

This version migration is what will most often be used.

Note that version migration only refers to ADEMPIERE versions, not versions of the database engine. Your database vendor will provide you tools to upgrade the database version, if necessary. Normally this is also achieved very efficiently by exporting (or dumping) data, installing the new database version, and then importing (or restoring) from the dump file.

Putting it all Together

Say you are currently running COMPIERE on an ORACLE database, and you want to change over to ADEMPIERE on a POSTGRESQL database. You would do this migration in two steps (each step will take approximately 3-5 hours, depending on the size of your live database):

First you would transfer your data from ORACLE to POSTGRESQL. MIGRATE will take care that all data types are correctly translated and move the data. All tables, indexes, sequences, foreign keys etc. will be applied in the target database. An attempt will be made to translate views. Functions will be commented out (so that you can review the original code) and replaced with compilable stubs.

This translation is intended as a one-way step. If you try to translate back and forth between database vendors, you will eventually end up with gibberish.

As second step, you would load the reference database and run a version migration. Now the views and functions will be replaced by those defined in the reference database. So only your custom views need to be checked and custom functions need to be translated manually.

When done, you are ready to use ADEMPIERE running on POSTGRESQL. From now on, you will only require version migrations each time you upgrade ADEMPIERE, and they will run significantly faster.

Process Description

MIGRATE performs the following steps to run a migration:

Connect to Databases

MIGRATE uses JDBC to connect to the source and target databases.

If conducting a transfer migration, any existing data in the target database is erased.

Load Meta-Data

As a first step, some tests are made to detect and correct buggy behavior by some JDBC drivers.

Meta-data on the database's structure (tables, indexes, views, functions, sequences, foreign keys, etc.) is loaded.

The Application Dictionary is accessed to gather information on customizations, system clients, and languages used.

Structural Migration

To get rid of overhead, MIGRATE first of all removes all kinds of database objects which are not tables from the target database:

- check constraints
- unique constraints
- foreign keys
- views
- operators
- triggers
- functions

- primary keys[1]
- indexes[1]

With the database reduced to this state, MIGRATE can pretty much do whatever it wants without running into constraint issues or being slowed down for integrity checks.

Then temporary tables are truncated to reduce the amount of data that needs to be migrated and thus increase performance:

- Data from temporary tables (T_…) is removed
- Records from Import tables (I_…) which have already been imported are removed
- Records from the TEST table (Test) are removed
- Processes and Errors are removed (AD_PInstance, AD_Find, AD_Error)
- Changes which are not customizations are removed (AD_ChangeLog)
- Sessions older than a week are removed (AD_Session)
- Notes which have been processed are removed (AD_Note)
- Log entries older than a week are removed (…Log)

The GardenWorld demonstration client is dropped, and all system records which are not referenced by real clients are purged.

Any sequences defined in the target are synchronized with the reference database, and sequences which are not yet defined are added.

Finally, the main structural migration task of synchronizing the target's table structure starts:

- Non-customized tables are dropped from the target if they do not exist in the reference database
- Tables existing only in the reference database are added to the target
- Tables existing in both the target and the reference database are synchronized:
  - Target tables are renamed to have the same name as their counterparts in the reference database[2]
  - Non-customized columns are dropped from the target if they do not exist in the reference database
  - Columns existing only in the reference database are added to the target
  - Columns existing in both the target and the reference database are synchronized so that the target column has the same properties as the column in the reference database:
    - column name
    - data type and size
    - default values
    - nullable constraint

After table synchronization, any non-customized sequences are dropped from the target if they do not exist in the reference database.

Database objects are recreated – all objects existing in the reference database are created in the target, and those target objects which are customizations are re-created:

---

[1] For performance reasons, primary keys and indexes are actually dropped at a later stage, and also temporary indexes are created and later dropped again during the migration process. These performance enhancements do not affect the functionality of the actual migration process and are omitted in this description for simplicity's sake.

[2] This feature is not implemented yet.

- functions
- triggers
- operators
- views
- indexes[3]
- primary keys[3]

Data Migration

Data records are transferred from the reference database to the target:

- if the record does not yet exist in the target, it is added.
- if the record already exists in the target, the target record is updated to contain the same data in all columns as the reference database.

New parent tables are populated[4] (only for upgrade migrations). If new tables are added to the target which use previously existing independent tables as child tables, records must be added to the parent table to reflect already existing data in the child tables.

Parent links are preserved (only for upgrade migrations). If a target table did not contain a column which is used as part of a foreign key constraint in the reference database, that column will have been added with a default value which does not reference any parent record. The correct parent must be found and the default value replaced with a link to the parent record.

Orphaned data is removed (only for upgrade migrations). Records who's parent records have been purged during migration are orphans which are no longer required and must be deleted.

Check constraints are enforced (only for upgrade migrations). Records containing values which would violate a check constraint are modified to comply with the constraint.

Cleanup

Cleanup operations are performed only for upgrade migrations:

Customizations are re-applied. Users may modify windows and processes in ADEMPIERE, but those modifications would be overwritten and reset by the migration process. Modifications which should be preserved can be marked as customization in the change log, and they will be re-applied.

Sequence counters are checked to ensure that the next number is larger than any number already used in the database. Missing sequence counters are added (Sequence counters defined in the application dictionary as well as native database sequence counters).

Missing translations are added. If translation records are required but do not exist yet, they are added with the original text from the main record.

Terminology is synchronized:

---

[3] For performance reasons, indexes and primary keys are actually recreated at a later stage after data migration.

[4] This feature is not implemented yet.

- New elements are created in the application dictionary for any columns or parameters which have no base element defined yet.
- unused elements are deleted
- consistent terminology is deployed throughout the application dictionary

Trees are re-organized so that customized nodes are inserted back into their original locations.

Security settings are verified and role access records updated or added.

Version information stored in the application dictionary is updated.

Enforce Constraints

Constraints are recreated – all constraints existing in the reference database are created in the target, and those target constraints which are customizations are re-created:

- foreign keys
- check constraints
- unique constraints

Close Database Connections

The source connection is closed and, if appropriate, the reference database is dropped.

Any remaining changes are committed to the target and the target connection is closed. If requested, the live database will be optimized.

2

Marking Customizations

Customizations are preserved through migrations. Entities which are not recognized as customizations will be dropped or overwritten from the reference database.

Migrate recognizes four different levels of customization:

`CUSTOMPREFIXED`
: An entity is named with a special prefix which identifies it as a customization. Prefixes are stored in the Application Dictionary.

`CUSTOMMARKED`
: An entity is marked as customization in the Application Dictionary.

`CUSTOMIMPLIED`
: An entity itself is not customized, but it contains customized components.

`CUSTOMNONE`
: An entity is not customized.

The only way to determine the customization level is by consulting the Application Dictionary, which means you must have informed the Application Dictionary about your customizations before you start Migrate.

Registering Custom Entity Types

You can register four-letter entity types to identify your customizations. These four letters can also be used as prefix to name database objects which are not maintained by the application dictionary.

For example, if you decide to identify your customizations by entity type `QRST`, then you can create a custom index and name it `QRST_MyIndexName`. Because `QRST` is registered as custom entity type in the Application Dictionary, Migrate understands that `QRST_MyIndexName` is a custom index and will preserve it.[1]

It is good practice to also name those objects which are maintained by the Application Dictionary using your custom prefix, like `QRST_MyTableName` and `QRST_MyColumnName`. This makes the customizations also easily recognizable by human database administrators.

---

[1] Exception: If the same four letters are also registered as entity type in the reference database, they will not be considered as customization markers. The reasoning behind this is that if you use a customized reference database, those customizations contained in the reference database should also be maintained and controlled by the reference database and not protected by Migrate.

Of course you can also use different entity types for different topics, like `QRS1` for security related customizations, `QRS2` for accounting related customizations, etc.

To register your custom entity type, log in as **`System`** and open the window `Application Dictionary` → `Entity Type`.

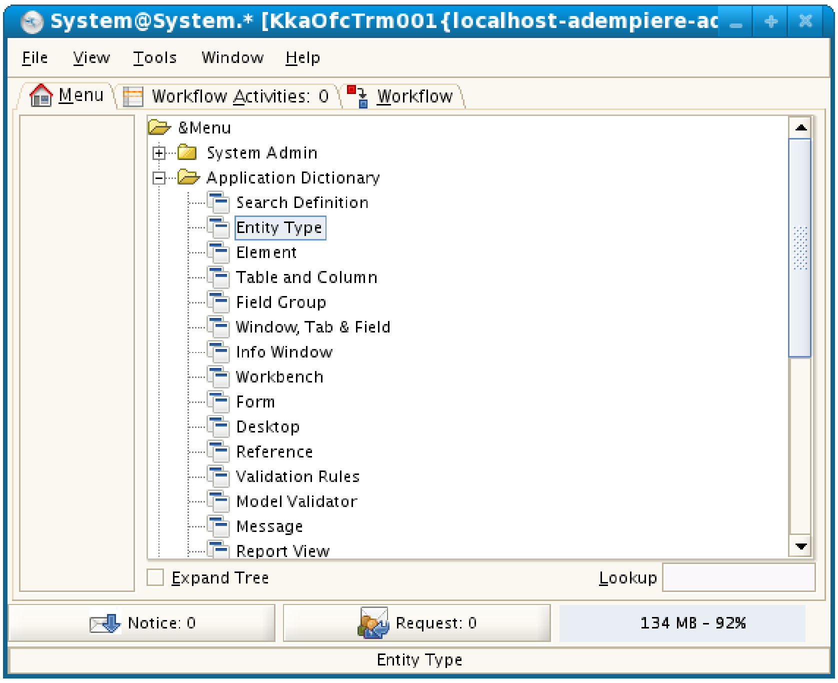
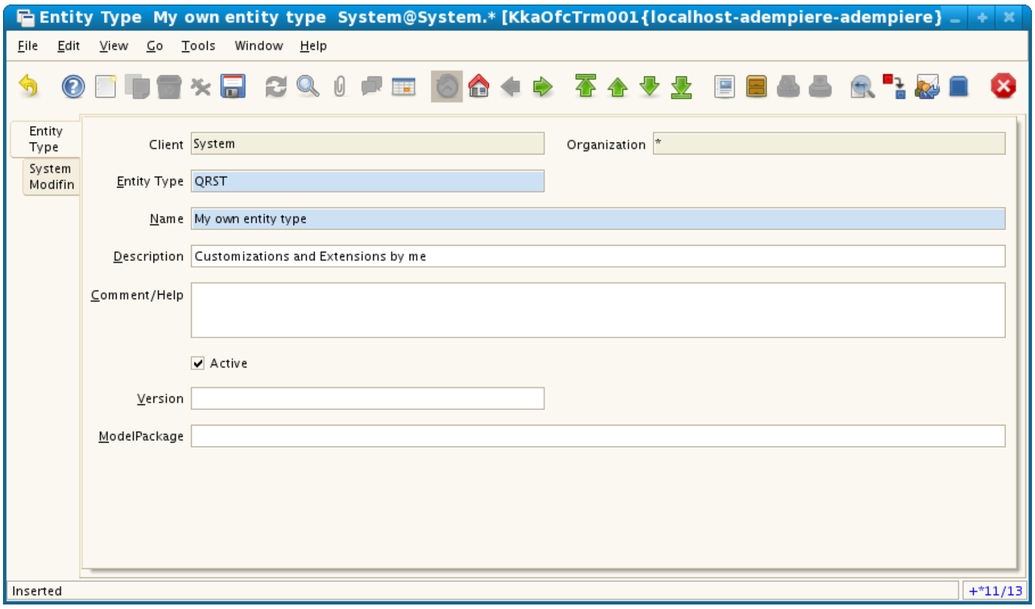


Figure 2.1. Select `Entity Type` from the `Application Dictionary` menu

Create a new record, enter four letters as your new entity type, and give it a short name and a description.


Entity Type My own entity type System@System.* [KkaOfcTrm001{localhost-adempiere-adempiere}
File
Edit
View
Go
Tools
Window
Help
Entity Type
System Modifin
Inserted
+*11/13

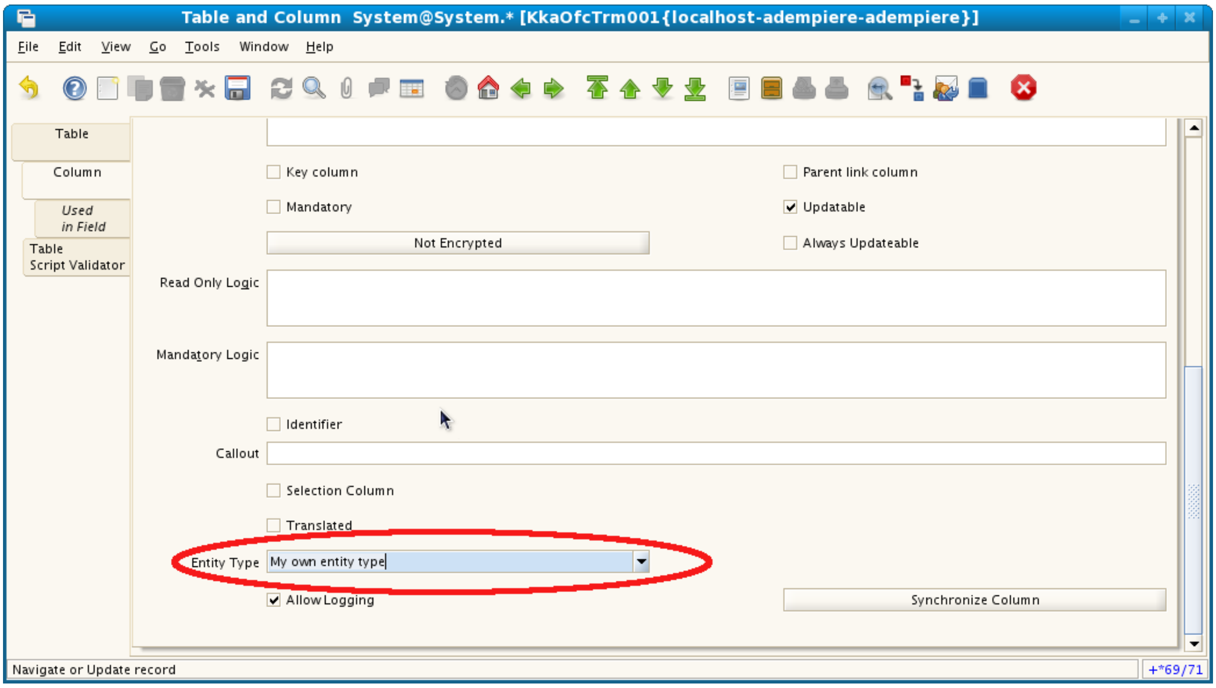
Client: System
Organization: *
Entity Type: QRST
Name: My own entity type
Description: Customizations and Extensions by me
Comment/Help:
☑ Active
Version:
ModelPackage:

Figure 2.2. Register your custom entity type in the Application Dictionary

Mark Customizations in the Application Dictionary

You can now use your new entity type to mark your customizations in the Application Dictionary.

For example, if you add a new column to a table, you can define it as being of your new entity type:



☐ Key column
☐ Parent link column
☐ Mandatory
☑ Updatable
☐ Always Updateable
Read Only Logic
Mandatory Logic
☐ Identifier
Callout
☐ Selection Column
☐ Translated
Entity Type: My own entity type
☑ Allow Logging

Navigate or Update record
+*69/71

Figure 2.3. Select your custom entity type for newly created objects

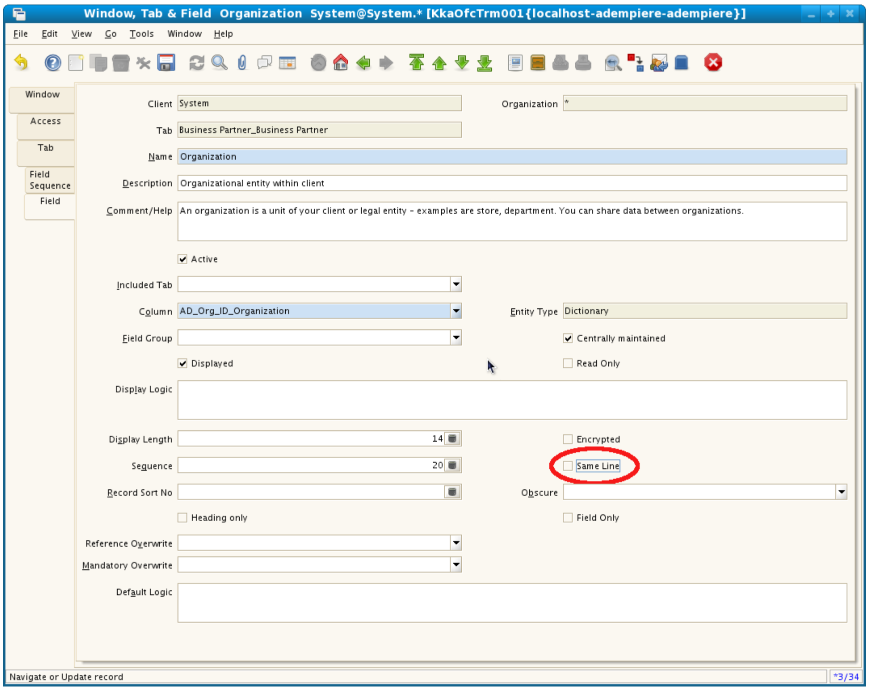
Apart from your own entity types, you can of course also mark your customizations with one of the predefined types `User maintained`, `Applications`, `Other Customizations`, `Extensions`, or `Other Extensions`.

Do not use `Adempiere` or `Dictionary`, which mark your changes as system-maintained and they will be dropped during the next version migration.

Mark Customizations in the Change Log

In some cases it is not possible to identify your changes with a custom entity type.

For example, if you wanted to change the Business Partner window so that the organization field is not displayed next to the client field but below it in the next row. Logged in as **`System`**, you would make the changes in the window `Application Dictionary` → `Window, Tab & Field`. Navigate to the `Organization` field, and deselect `Same Line` so that the field gets displayed in the next row.

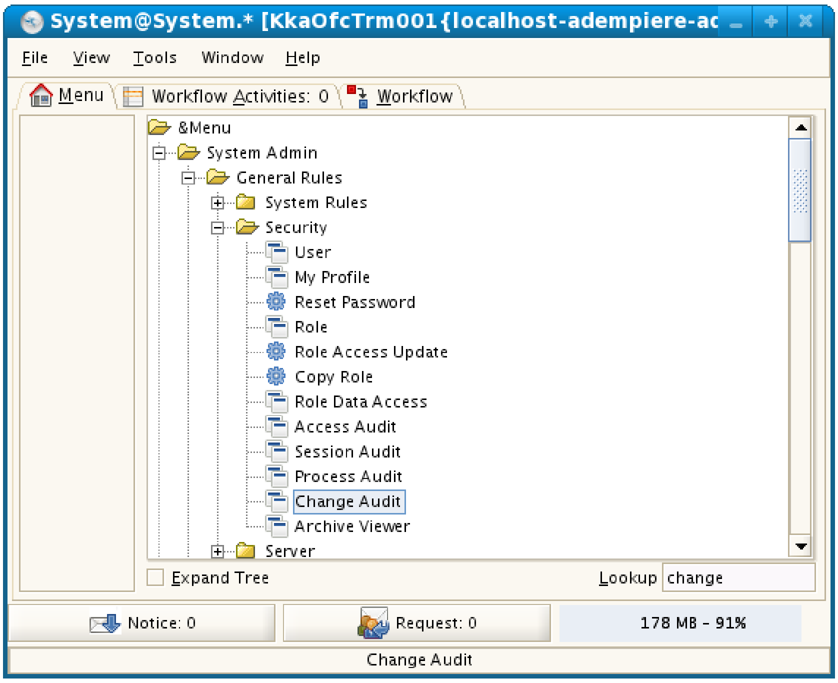

Window, Tab & Field Organization System@System.* [KkaOfcTrm001{localhost-adempiere-adempiere}]
File
Edit
View
Go
Tools
Window
Help
Window
Access
Tab
Field Sequence
Field

Client: System
Organization: *
Tab: Business Partner_Business Partner
Name: Organization
Description: Organizational entity within client
Comment/Help: An organization is a unit of your client or legal entity - examples are store, department. You can share data between organizations.
☑ Active
Included Tab:
Column: AD_Org_ID_Organization
Entity Type: Dictionary
Field Group:
☑ Centrally maintained
☑ Displayed
☐ Read Only
Display Logic:
Display Length: 14
☐ Encrypted
Sequence: 20
☐ Same Line
Record Sort No:
Obscure:
☐ Heading only
☐ Field Only
Reference Overwrite:
Mandatory Overwrite:
Default Logic:
Navigate or Update record
*3/34

Figure 2.4. Tweaking window appearance

But as you can see, the entity type for this field is already `Dictionary`, and you can not apply your custom entity type.

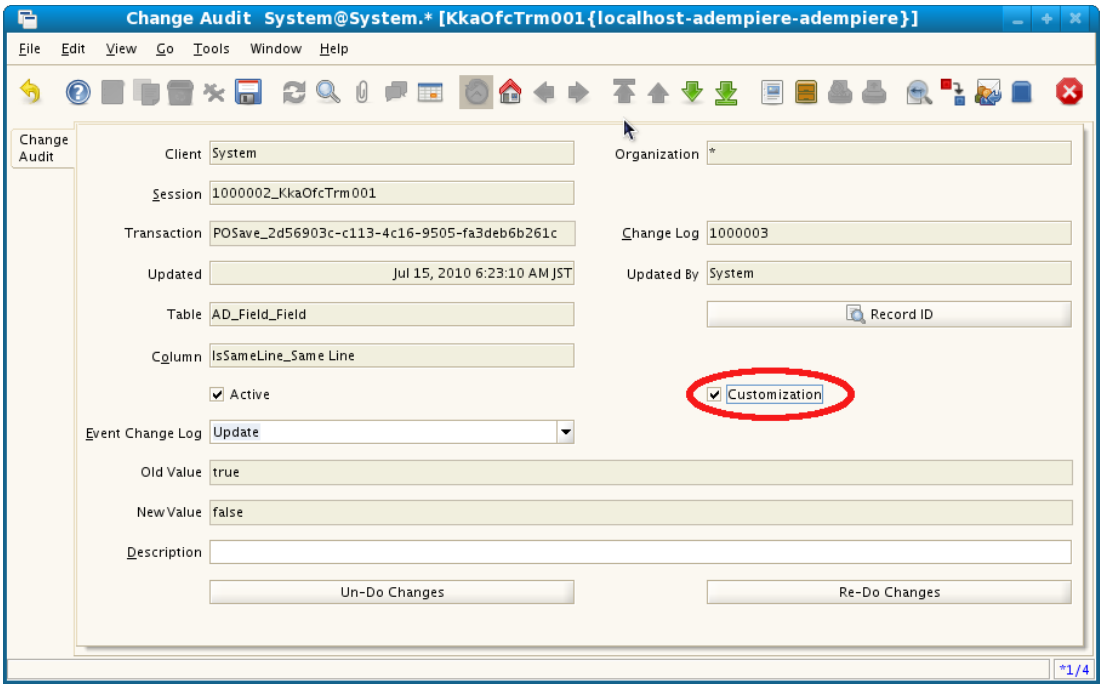
To still protect your change from being undone during the next version migration, you can mark it as customization in the change log. For security reasons, ADEMPIERE keeps a log of changes done to the system. The log can be accessed from the window `System Admin` → `General Rules` → `Security` → `Change Audit`.



☐ Expand Tree
Lookup: change
Change Audit

Figure 2.5. Select `Change Audit` from the `Security` menu

Find the change you want to keep permanently and mark it is customization:



Client System
Organization *
Session 1000002_KkaOfcTrm001
Transaction POSave_2d56903c-c113-4c16-9505-fa3deb6b261c
Change Log 1000003
Updated Jul 15, 2010 6:23:10 AM JST
Updated By System
Table AD_Field_Field
Column IsSameLine_Same Line
☑ Active
☑ Customization
Event Change Log Update
Old Value true
New Value false
Description

Figure 2.6. Marking changes as customization in the Change Log

MIGRATE will preserve changes marked as customization in such way.

3

Migrating a Database

Preperation

Disconnect all Users

The target database should be up and running.

No users should be logged in. Make sure all users are disconnected from the target and source database.

That includes the ADEMPIERE server itself: Shut down the application server.

Create a Backup

You *must* have a backup of your live data before starting the migration process.

Remember the disclaimer at the beginning of this document: This program is distributed without warranty of fitness for a particular purpose. It may migrate your data, or it may completely mess up your database.

The easiest way to quickly create a backup is with **./RUN_DBExport.sh** (or **RUN_DBExport.bat**) in the `utils` directory.

That script will create a file `ExpDat.dmp` in the `data` directory, which can be easily restored using **./RUN_DBRestore.sh** (or **RUN_DBRestore.bat**), if necessary.

Install new ADEMPIERE version

If you want to do an upgrade migration, download the ADEMPIERE version you want to upgrade to and install it.

Then execute **./RUN_setup.sh** (or **RUN_setup.bat**) in `$ADEMPIERE_HOME` to configure ADEMPIERE. The settings saved are also used by MIGRATE.

Import Reference Database

If you want to do an upgrade migration, install the reference database:

Execute **./RUN_ImportReference.sh** (or **RUN_ImportReference.bat**) in the `utils` directory.

If you want to do a transfer migration, make sure the source database is up and running.

Verify Preconditions

Make sure that

- no users are logged in
- the ADEMPIERE application server is shut down
- you have a backup
- the reference database is imported (for upgrade migrations)
- the source or reference database is up and running
- the target database is up and running

Running the Migration Tool

Once all preparations have been done and verified, you can start MIGRATE by executing **./RUN_Migrate.sh** (or **RUN_Migrate.bat**) from the `utils` directory.

This will start the migration tool and display the interactive graphical user interface.[1]

When MIGRATE is started, it will read environment variables for setting parameters and options. Since the **RUN_Migrate** script loads ADEMPIERE's environment before calling MIGRATE, it effectively means that ADEMPIERE's settings will also be used by MIGRATE. Any settings not defined by environment variables will be supplemented with sensible values.

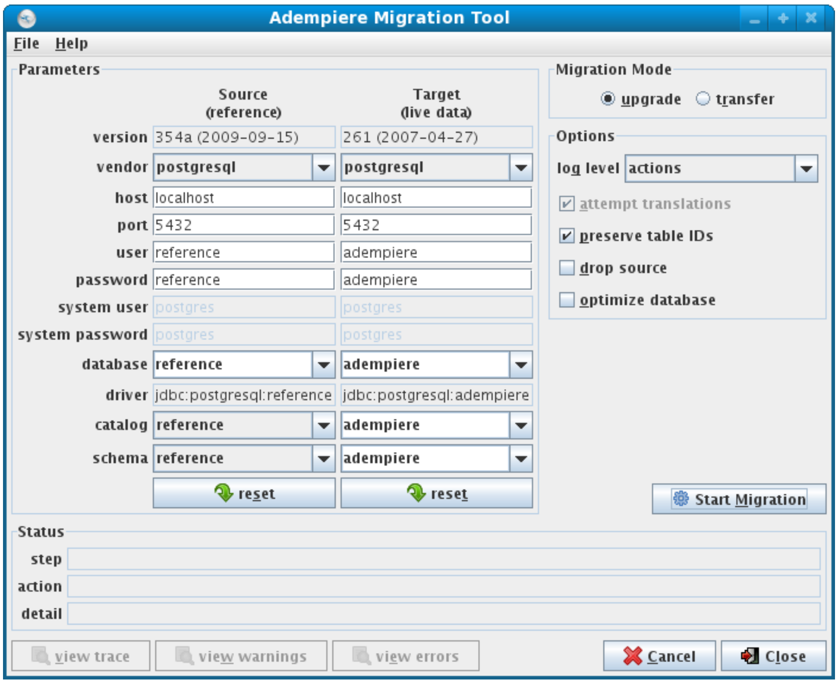
If `$ADEMPIERE_HOME` is defined, MIGRATE looks for a configuration file called `migration.config` in the `$ADEMPIERE_HOME/utils` directory, otherwise it will look for the configuration file in the current directory. If the file exists, configuration settings will be read from that configuration file, and any settings loaded from the environment will be overwritten. Once a migration was run, MIGRATE saves its settings to that configuration file, so next time it is started, your last parameters and options will be used again.

Any command line arguments passed to MIGRATE will override the settings loaded from the configuration file or from the environment so that command line arguments always take precedence.

---

[1] To run in text mode and/or suppress console output, the keywords *`text`* or *`silent`* can be given to the **RUN_Migrate** script as command line arguments.

The User Interface



	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

Migration Mode: ● upgrade ○ transfer

log level: actions

☑ attempt translations

☑ preserve table IDs

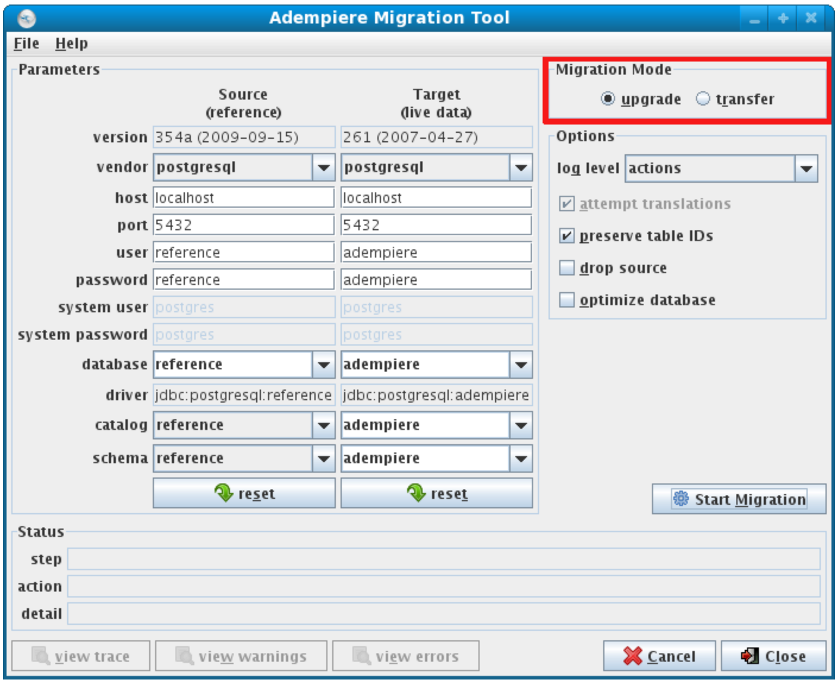
☐ drop source

☐ optimize database

Figure 3.1. MIGRATE's interactive Graphical User Interface

Once the user interface is displayed, you need to select the migration mode, select some options to be used by the migration process, and set the database connection parameters.

Migration Mode



	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

Migration Mode: ● upgrade ○ transfer

log level: actions

☑ attempt translations

☑ preserve table IDs

☐ drop source

☐ optimize database

Figure 3.2. Migration Mode Settings

Select the mode in which to run the migration process.

Two different modes of migration can be performed:

`upgrade`
: Upgrade target to newest version as found in source.

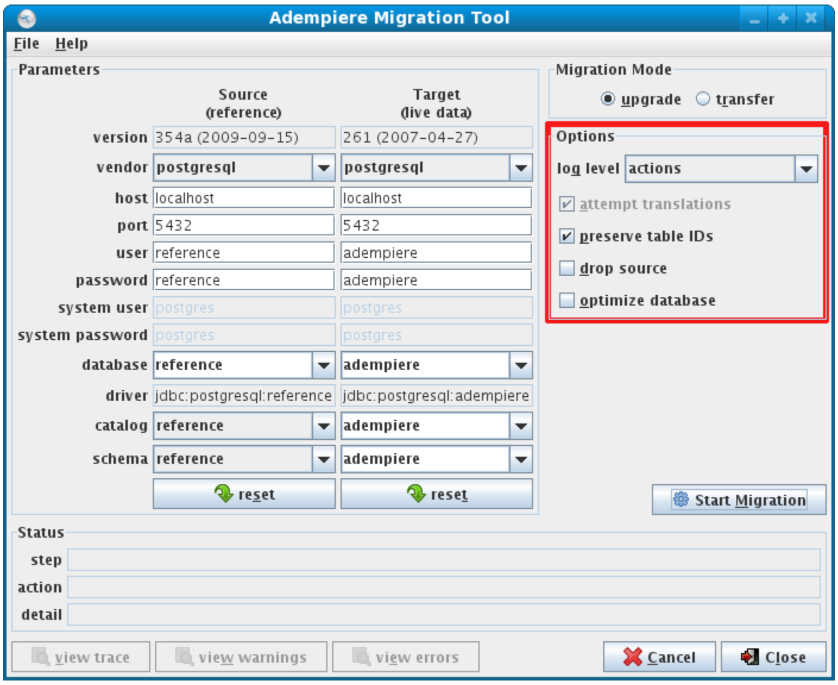
  This mode can also be used to convert from other applications to Adempiere.

`transfer`
: Copy source to target.

  This mode can also be used to convert from other databases to PostgreSQL.

The default is to run an upgrade migration, but if different vendors are used as source and target database (see Parameters below), only a transfer migration can be performed.

Options



	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

Migration Mode: ● upgrade ○ transfer

Options

log level: actions

☑ attempt translations

☑ preserve table IDs

☐ drop source

☐ optimize database

Figure 3.3. Options

Several options can be set to control migration behavior. Which options are available depends on the migration mode.

`log level`
: Migrate creates three log files containing results of the migration process:

  - `migration_`*`timestamp`*`.error.log`

    contains any errors encountered during migration which must be fixed.

  - `migration_`*`timestamp`*`.warning.log` contains hints for the database administrator of what has to be checked or might need to be done manually after migration has finished.

- `migration_`*`timestamp`*`.trace.log`

  contains the output messages of what steps and actions MIGRATE has performed.

The log level option sets the threshold for messages to be recorded in the trace log. Messages with a lower priority will not be logged.

Available log levels in order of descending priority are:

- `no logging`
- `errors only`
- `post-migration tasks` (warnings)
- `migration steps`
- `actions`
- `details`
- `SQL update queries`
- `SQL read queries`
- `everything`

The default log level is `actions`.

Note that levels of `details` or lower can create huge trace files. Be sure to have enough disk space available.

`attempt translations`

This option is only available in transfer mode.

When converting from one database to another, views and functions need to be translated.

If selected, MIGRATE will attempt to translate views and functions, otherwise they will be replaced with a compilable stub.

(Note that currently only translation of views is implemented).

The default is `yes`.

`preserve table IDs`

This option is only available in upgrade mode.

When running an upgrade, all system information is dropped. Table IDs therefore restart with the highest used sequence number available after migration. It may be beneficial, however, to remember higher ID numbers used before migration to ensure consistency over different versions.

If selected, table ID numbers are preserved through migration, otherwise MIGRATE restarts counting after migration

The default is `yes`.

`drop source`

This option is only available in upgrade mode.

When done with upgrading, the source database is no longer required and may be dropped to clear space. However, the database administrator may wish not to drop it for reference purposes.

If selected, the source is dropped after a successful upgrade, otherwise it is kept remaining in the database after migration.

(Note that the source will only be dropped if no errors occurred during migration).

The default is `no`.

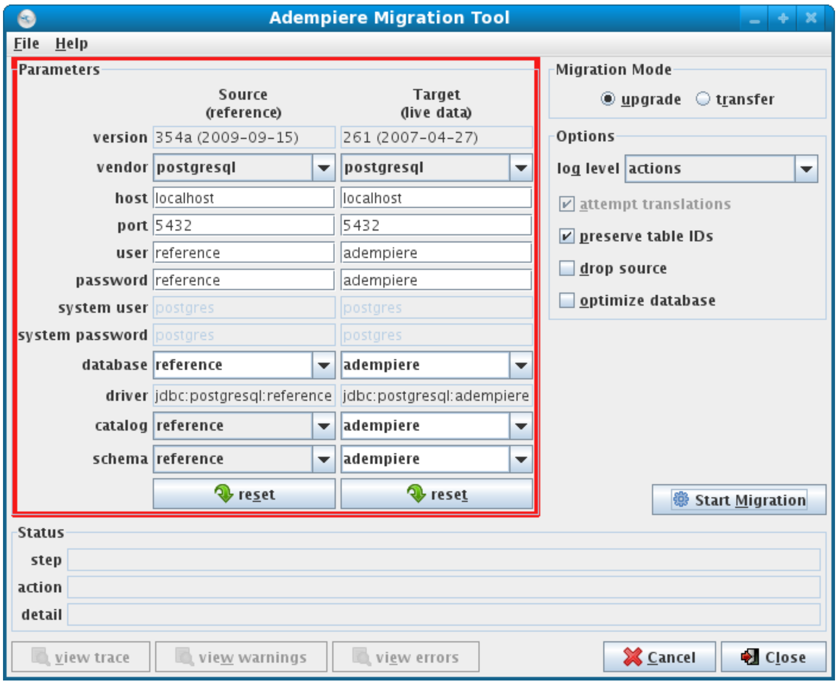
`optimize database`

After migration, the database can be automatically optimized. Most databases nowadays have scheduled processes which regularly run optimization tasks, so it may not be necessary to explicitly run them here. Examples for optimization tasks are space allocation or gathering of statistics, but what is actually performed depends on which kind of database is running.

If selected, the target database is optimized after migration, otherwise it is left to the database's automatic scheduler.

The default is `no`.

Parameters



	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

Migration Mode: ● upgrade ○ transfer

log level: actions

☑ attempt translations

☑ preserve table IDs

☐ drop source

☐ optimize database

Figure 3.4. Connection Parameters

Parameters are used to define the connections to the source and target databases.

In upgrade mode, the source is the reference against which the target's structure is updated, and live data in the target remains intact.

In transfer mode, the source is copied to the target, and all live data in the target is overwritten.

Two identical sets of parameters must be defined, one for the source connection and one for the target connection.

`version`

This field is read-only and displays the ADEMPIERE version number found in the database.

If no version number is displayed, it means that either no connection to the database could be established, or the database contains no ADEMPIERE version information (which means it is not an ADEMPIERE database).

`vendor`

The vendor (or product) of the database. Supported vendors currently are:

- ORACLE
- POSTGRESQL

The default is `postgresql`.

`host`

The name or IP-address of the server on which the database is running.

The default is `localhost`.

`port`

The port on which the database is listening.

Common port numbers are `5432` for POSTGRESQL or `1521` for ORACLE.

The default is `5432`.

`user`

The normal database user as which to log in.

The default is `reference` for source and `adempiere` for target.

`password`

The normal database user's password.

The default is `adempiere` for both source and target.

`system user`

Some databases require a system user for certain operations[2]. This is the name of the system user as which to log in.

The default is `postgres`.

---

[2] The `system user` and `system password` fields are not used if the selected database does not require log in by a system user for migration.

`system password`

The system user's password[2].

The default is `postgres`.

`database`

The name of the database to use.

The default is `reference` for source and `adempiere` for target.

`driver`

This field is read-only and displays the URL which will be used by MIGRATE to connect to the database. The driver and format used depend on the database vendor.

`catalog`

The catalog to use.

The usage and meaning of catalogs varies according to database vendor. If none is given, MIGRATE will try to find a sensible catalog.

`schema`

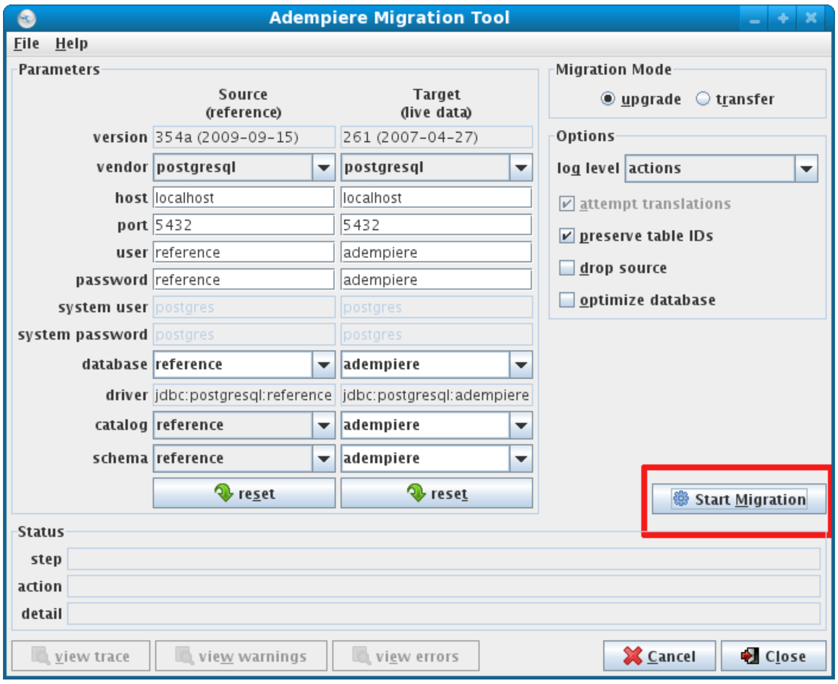
The schema to use.

The usage and meaning of schemas varies according to database vendor. If none is given, MIGRATE will try to find a sensible schema.

`reset`

Pressing this button resets the parameters to their original settings.

Command Buttons



Parameters

	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

Migration Mode

● upgrade ○ transfer

Options

log level actions

☑ attempt translations

☑ preserve table IDs

☐ drop source

☐ optimize database

Status

step

action

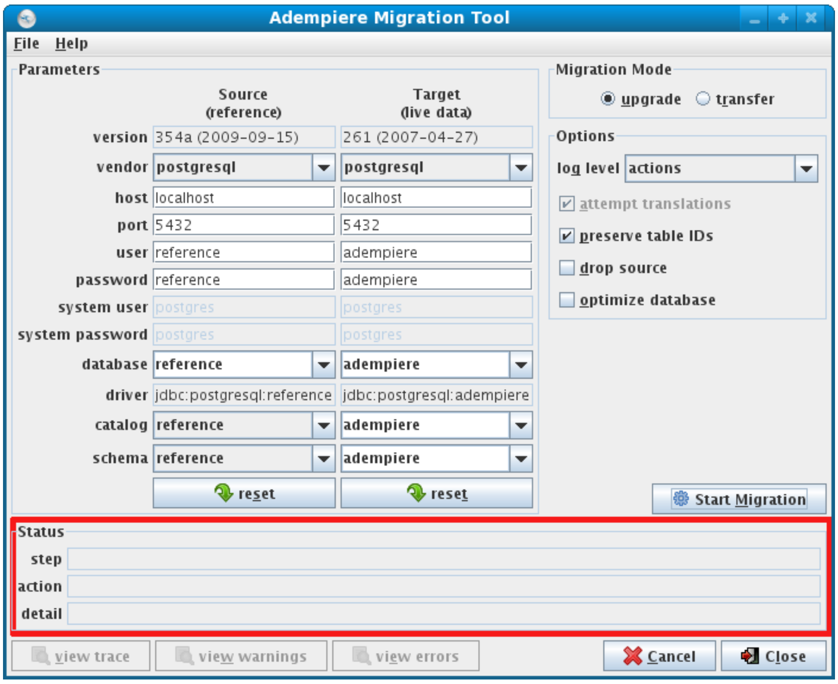
detail

Figure 3.5. Command Buttons

`Start Migration`
: Start the migration process.

Pressing this button runs sanity checks and starts the migration process. Once the target database has been modified, the process must not be interrupted.

Status



	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

Migration Mode: ● upgrade ○ transfer

log level: actions

☑ attempt translations

☑ preserve table IDs

☐ drop source

☐ optimize database

step

action

detail

Figure 3.6. Status Display

The current status of the running migration process is displayed, indicating what action is being performed in which migration step.

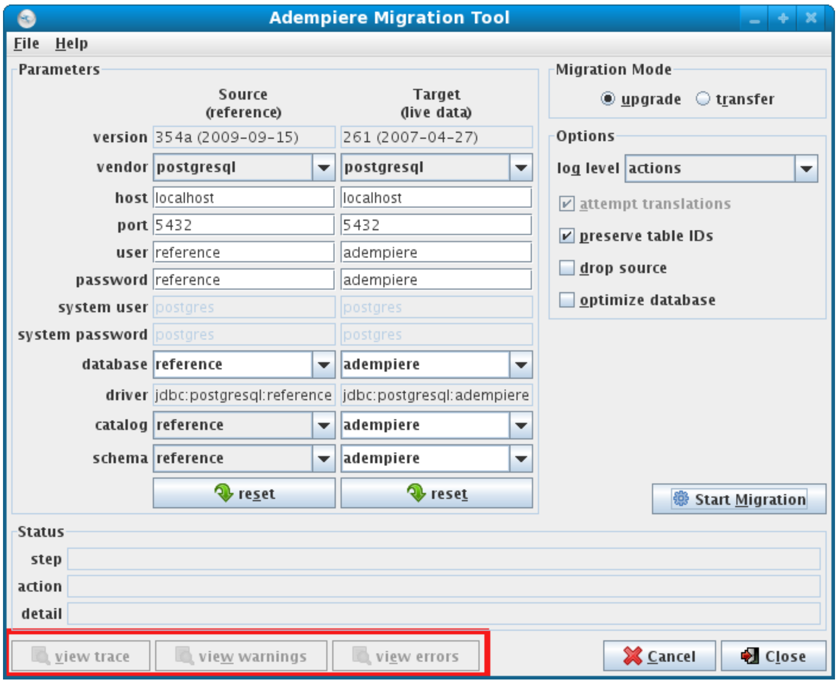
`step`
: This field displays the current migration step being performed, which can be one of:

- `CONNECT TO DATABASES`
- `LOAD METADATA`
- `SYNCHRONIZE TARGET FROM SOURCE`
- `CLOSE DATABASE CONNECTIONS`
- `DONE`

`action`
: This field displays which action or operation is currently being performed within above migration step.

`detail`
: This field displays details of the current action being performed, for example which record is presently being updated.

View Buttons



	Source (reference)	Target (live data)
version	354a (2009-09-15)	261 (2007-04-27)
vendor	postgresql	postgresql
host	localhost	localhost
port	5432	5432
user	reference	adempiere
password	reference	adempiere
system user	postgres	postgres
system password	postgres	postgres
database	reference	adempiere
driver	jdbc:postgresql:reference	jdbc:postgresql:adempiere
catalog	reference	adempiere
schema	reference	adempiere

● upgrade ○ transfer

log level actions

☑ attempt translations
☑ preserve table IDs
☐ drop source
☐ optimize database

Figure 3.7. View Buttons

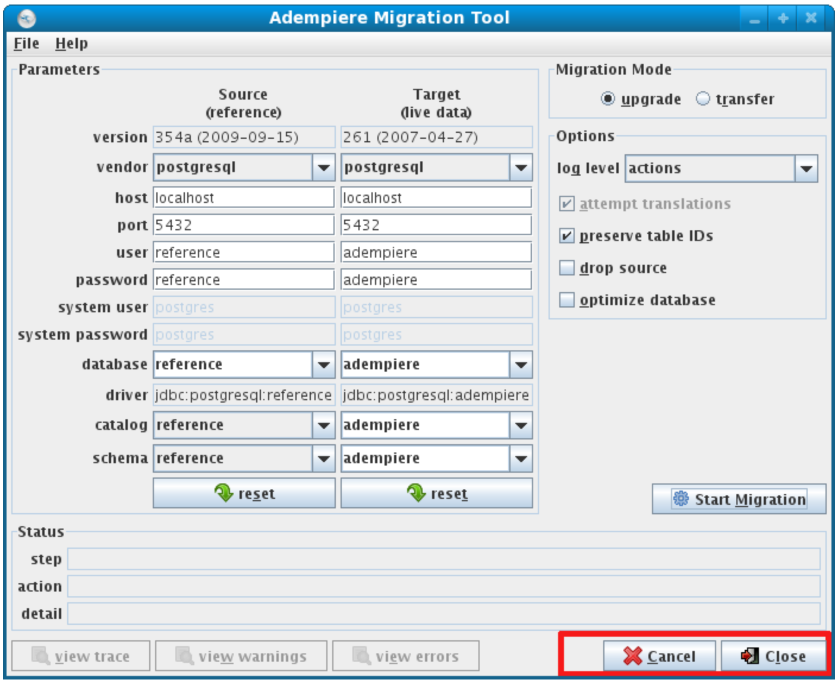
Press one of these buttons to view the different log files.

`view trace`
: View a snapshot of the last 500 lines of the trace log. The trace log contains all output messages as defined with the log level.

`view warnings`
: View a snapshot of the last 500 lines of the warning log. The warning log contains tasks to be performed manually by the database administrator after migration, such as making sure that views and functions were translated correctly.

`view errors`
: View a snapshot of the last 500 lines of the error log. The error log contains all errors which occurred during migration and need to be fixed.

Close Buttons



version 354a (2009-09-15) 261 (2007-04-27)
vendor postgresql postgresql
host localhost localhost
port 5432 5432
user reference adempiere
password reference adempiere
system user postgres postgres
system password postgres postgres
database reference adempiere
driver jdbc:postgresql:reference jdbc:postgresql:adempiere
catalog reference adempiere
schema reference adempiere

● upgrade ○ transfer

log level actions
☑ attempt translations
☑ preserve table IDs
☐ drop source
☐ optimize database

step
action
detail

Figure 3.8. Close Buttons

`Cancel`
: Stop the migration process and close the program without saving any settings.

`Close`
: Stop the migration process and save settings and parameters before closing the program.

Starting from the Command Line

Of course MIGRATE does not have to be started with the **RUN_Migrate** script but can also be started directly from the command line. This allows MIGRATE to be called from other scripts for automating migration, if required.

The command to start MIGRATE from the command line is:

```
java [java Options] -cp classpath [migrate Options] com.kkalice.adempiere.migrate.Migrate
```

JAVA Options
: These are the options used by the Java Runtime Engine.

  Sufficiently high memory settings should be used so that MIGRATE does not run out of memory.

  Recommended are: `-Xms64M -Xmx512M`

  If the database contains large objects, higher settings may be necessary.

Classpath
: The classpath should contain the file `migrate.jar` as well as the JDBC database drivers for the databases to be used, for example:

```
$ADEMPIERE_HOME/lib/migrate.jar:$ADEMPIERE_HOME/lib/postgresql.jar:$ADEMPIERE_HOME/lib/oracle.jar
```

or:

```
migrate.jar:/usr/share/java/postgresql-jdbc.jar:/opt/oracle/jdbc/lib/ojdbc14.jar
```

Of course only the JDBC drivers for the database vendors you will actually be connecting to need to be supplied.

Migrate Options

Options passed to Migrate must be prefixed with **`-D`** so that java knows it must pass the options on to the application as system properties.

It is highly recommended that all options and parameters are explicitly set on the command line to avoid unpleasant surprises when values you were expecting as default are unexpectedly overridden by environment variables or the configuration file.

GUI Mode / Text Mode / Silent Mode

Two options are only available when starting Migrate from the command line:

`-DisText`

Migrate will run in Text mode, the GUI will not be started. All parameters and options must be provided by environment variables, the configuration file, or command line arguments.

`-DisSilent`

All console output will be suppressed. This implies *`-DisText`*.

If none of these arguments are passed, Migrate will run interactively with a Graphical User Interface.

Migration Mode

Upgrade mode or transfer mode is selected by the `isUpgrade` property:

`-DisUpgrade=Y`

run migration in upgrade mode.

`-DisUpgrade=N`

run migration in transfer mode.

Options

`-DmaxLogLevel=<log level>`

Use following Java log levels to correspond to the thresholds which can be selected from the GUI:

```
OFF     = no logging
SEVERE  = errors only
WARNING = post-migration tasks
INFO    = migration steps
CONFIG  = actions
FINE    = details
```

```
FINER   = SQL update queries
FINEST  = SQL read queries
ALL     = everything
```

-DattemptTranslation=Y, N
whether to translate views and functions

-DpreserveTableID=Y, N
whether to preserve table IDs

-DdropSource=Y, N
whether to drop the source database after successful migration

-DoptimizeDatabase=Y, N
whether to optimize the target database

Parameters
Source connection parameters:

```
-DsourceDB_vendor=<database vendor>
-DsourceDB_host=<host>
-DsourceDB_port=<port>
-DsourceDB_name=<database name>
-DsourceDB_catalog=<catalog>
-DsourceDB_schema=<schema>
-DsourceDB_user=<normal user>
-DsourceDB_passwd=<normal password>
-DsourceDB_systemUser=<system user>
-DsourceDB_systemPasswd=<system password>
```

And target connection parameters:

```
-DtargetDB_vendor=<database vendor>
-DtargetDB_host=<host>
-DtargetDB_port=<port>
-DtargetDB_name=<database name>
-DtargetDB_catalog=<catalog>
-DtargetDB_schema=<schema>
-DtargetDB_user=<normal user>
-DtargetDB_passwd=<normal password>
-DtargetDB_systemUser=<system user>
-DtargetDB_systemPasswd=<system password>
```

To pass an empty string, either omit the string after the equal sign or write only the parameter name without any equal sign:

```
-DsourceDB_catalog=
```

or just

```
-DsourceDB_catalog
```

Example:

The following command runs a transfer migration from an Oracle to a postgreSQL database, assuming that `migrate.jar` is in the current directory. Everything should be typed on one line:

```
java -Xms64M -Xmx512M -cp migrate.jar:/usr/share/java/postgresql-jdbc.jar:/opt/oracle/jdbc/lib/ojdbc14
.jar -DisText -DisUpgrade=N -DmaxLogLevel=CONFIG -DattemptTranslation=Y -DoptimizeDatabase=N
 -DsourceDB_vendor=oracle -DsourceDB_host=localhost -DsourceDB_port=1521 -DsourceDB_name=erp
-DsourceDB_schema=compiere -DsourceDB_user=compiere -DsourceDB_passwd=compiere -DsourceDB_
systemUser=system -DsourceDB_systemPasswd=manager -DtargetDB_vendor=postgresql -DtargetDB_ho
st=localhost -DtargetDB_port=5432 -DtargetDB_name=adempiere -DtargetDB_schema=adempiere -Dtarg
etDB_user=adempiere -DtargetDB_passwd=adempiere com.kkalice.adempiere.migrate.Migrate
```

Post-Migration Tasks

Migrate already runs sanity checks and clean-up procedures after migration, so it is not necessary to start any post-migration scripts such as **RUN_PostMigration.sh** (or **RUN_PostMigration.bat**).

However, the database administrator should check the log files to verify whether any manual intervention is required after migration has completed, particularly the warning log and the error log.

For a transfer migration, warnings and errors issued for non-customized objects or system records can usually be ignored, as they will be replaced during the subsequent version migration anyway. Only problems with customized objects or live data of real clients need to be addressed by the database administrator.

Warnings

The warning log contains tasks to be performed manually by the database administrator after migration.

Table 3.1. Warning Messages

Warning	Mode	Cause	Solution
`Preserving node … in tree …`	upgrade	System nodes would normally be purged from trees, but are preserved if they are recognized as a customization (for example, custom entries in the system-wide menu).	Review this list to verify whether all customized system nodes are really needed in the new version.
`Not dropping customized table …`	upgrade	A table not existing in the reference database would normally be dropped, but it is kept alive if recognized as a customized table.	Review this list to verify whether all customized tables are really needed in the new version.
`Must re-write customized trigger function …`	transfer	If data is migrated from a database in which triggers can contain inline code to a database in which triggers themselves can not contain code but only point to functions, the inline code has to be converted to a callable function. At the time of conversion, the number of arguments to the function is unknown, and since also translation of functions is not implemented	Translate the function called by the trigger into the target database's syntax.

Warning	Mode	Cause	Solution
		yet, the trigger is basically rendered useless.	
`Must verify customized object …`	transfer	MIGRATE attempts to translate objects, but the result is not guaranteed to be correct.	Review that the object is translated correctly and works the way it is intended to.
`Must re-write object … [`*`error message`*`]`	transfer	Sometimes translation of an object fails. MIGRATE then just replaces the object's code with a compilable stub and indicates the last error as hint why translation failed.	Manually translate the object into the target database's syntax.
`Modified … rows in … to comply with check constraint …`	upgrade	A table contained values which would violate the check constraint rule. Those values have been modified to comply with the constraint.	Review the table to make sure that the modifications do not disrupt any business logic.
`Could not find correct parent for … from … in … to …`	upgrade	If a new column is added to a table and that column is part of a foreign key, MIGRATE attempts to find the correct parents for records already existing in the child table. This warning is issued if the correct parents could not be found.	If no error is reported when the foreign key is created, this warning can be ignored. Otherwise the child records must be linked to the correct parents manually. (If you know what hint can be used to deduce the correct parent, file a bug report).

Errors

The error log contains all errors which occurred during migration and need to be fixed. If an error was raised by the database driver, the original error message is added as a hint.

Table 3.2. Error Messages

Error	Cause	Solution
`Could not find driver … [`*`error message`*`]`	The required JDBC driver could not be found.	Make sure the JDBC driver is in the classpath.
`Could not connect to database … [`*`error message`*`]`	A connection to the database could not be established.	Make sure host name, port, database name, user name, and user password are correct. Make sure the server is reachable over the network. Make sure access configuration allows connections from your IP address.
`Could not commit changes in … [`*`error message`*`]`	Consult the database vendor's manual about the cause of the error.	Eliminate the cause of the error.
`Could not roll back changes in … [`*`error message`*`]`		
`Could not close … [`*`error message`*`]`		
`Could not determine product vendor for … [`*`error message`*`]`	The database vendor could not be determined or is unsupported.	Explicitly set the database vendor.
`Could not determine catalog for … [`*`error message`*`]`	No meaningful catalog could be determined.	Explicitly set the catalog to use.

Error	Cause	Solution
`Could not determine schema for …` [*error message*]	No meaningful schema could be determined.	Explicitly set the schema to use.
`Could not drop schema …` [*error message*]	The target schema could not be dropped.	Make sure the user has sufficient privileges to drop a schema.
`Could not test character set in …` [*error message*]	Migrate temporarily creates a table with some string fields to check how the JDBC driver reports character sizes. An error occurred while trying to create this table.	Make sure no table with the name `kkax_migr_chartest` previously exists in the database.
`Target table … does not exist`	Tables which were expected to exist for terminology checking could not be found.	Terminology checking will only be successful on databases with an ADempiere-style Application Dictionary.
`Source table … does not exist`		
`Target translation table … does not exist`		
`Join table … does not exist`		
`Extra table … does not exist`		
`Could not set savepoint …` [*error message*]	Consult the database vendor's manual about the cause of the error.	Eliminate the cause of the error.
`Could not get savepoint name` [*error message*]		
`Could not rollback to savepoint …` [*error message*]		
`Could not release savepoint …` [*error message*]		
`Could not prepare statement …` [*error message*]		
`Could not reset prepared statement …` [*error message*]		
`Could not close prepared statement …` [*error message*]		
`Could not count parameters for prepared statement …` [*error message*]		
`Could not set parameter … of prepared statement …` [*error message*]		
`Could not create statement` [*error message*]		
`Could not close statement` [*error message*]		
`Could not execute prepared statement query …` [*error message*]		
`Could not execute sql query …` [*error message*]		
`Could not close resultset …` [*error message*]		

Error	Cause	Solution
Could not move cursor in result set … [*error message*]		
Could not read column … from result set … [*error message*]		
Could not check last column value from result set … [*error message*]		
Could not execute prepared statement command … [*error message*]		
Could not execute sql command … [*error message*]		
unknown data type …	No unambiguous data type ID exists for the data type	File a bug report.
unknown data type or extra logic required for data type ID …	The unambiguous data type ID could not be converted to a vendor-specific data type	File a bug report.
Instantiation Exception for class … [*error message*]	A Java interface could not be instantiated.	File a bug report.
Illegal Access Exception for class … [*error message*]		
Could not find interface … [*error message*]		
A database can not be migrated to itself (source and target must be different)	Source and target connection parameters must point to different databases.	Make sure source and target connection parameters are correct.
Source and target need to be same database vendor for upgrades	Upgrades can only be run if source and target are the same database vendor.	Choose the correct reference database or run a transfer migration.

Start the Application Server

Now that your database has been successfully migrated, all errors have been fixed, and all warnings have been taken care of, the application server may be started again.

Users are welcome to log in.

4

Compiling and Extending

Compiling Migrate

Normally there should be no need to compile Migrate, as it will be installed together with Adempiere.

However, there may be situations when you separately want to compile Migrate, either to modify the code to suit your personal needs, or to fix bugs or extend the code and hopefully contribute your enhancements to the Adempiere project.

Requirements

Migrate requires the Java Development Kit version 1.6 (JDK 6)[1] and therefore also at least version 3.5.3a of Adempiere.

Downloading and Compiling the Source Code

1. Download the Adempiere source.

   ```
   hg clone http://adempiere.hg.sourceforge.net/hgroot/adempiere/adempiere#development .
   ```

2. You can either compile the complete Adempiere project or only the Migrate sub-project.

   - To compile the complete Adempiere project, change to directory `utils_dev`.

     ```
     cd utils_dev
     ```

   - To compile only the Migrate sub-project, change to directory `migrate`.

     ```
     cd migrate
     ```

3. Then execute **RUN_build.sh** (or **RUN_build.bat**).

   ```
   ./RUN_build.sh
   ```

---

[1] There is actually only one reason for this limitation: Some JDBC drivers do not return a readable SQL statement but only an object reference when the toString() method is called on a prepared statement, rendering it useless for logging purposes. Therefore Migrate uses a wrapper around the PreparedStatement class, which overrides the toString() method and returns a human readable string to be used for logging. All other methods are caught to extract variable information which is used to generate the string, and then passed on to the original PreparedStatement class. In Java 1.6, some new methods were added to PreparedStatement, which also accept or return classes new to Java 1.6. Since Java does not allow conditional compiling, a choice had to be made whether to be compatible with version 1.5 or version 1.6. Naturally, a choice was made for the newer version.

4. The resulting JAR file (`migrate.jar`) will be created in the `migrate` project directory and also copied to the `../lib` directory.

5. This will also generate the API and user documentation, to be found in the `migrate/apidoc` and `migrate/userdoc` directories, respectively.

For details on how to work with Adempiere source code, consult the Adempiere documentation [http://www.adempiere.com/index.php/Compile].

Building and Running Migrate in Eclipse

Consult the Adempiere documentation [http://www.adempiere.com/index.php/Create_your_ADempiere_development_environment] on how to compile and run Adempiere from within ECLIPSE.

Note that the JDBC drivers for installed databases must be in the classpath.

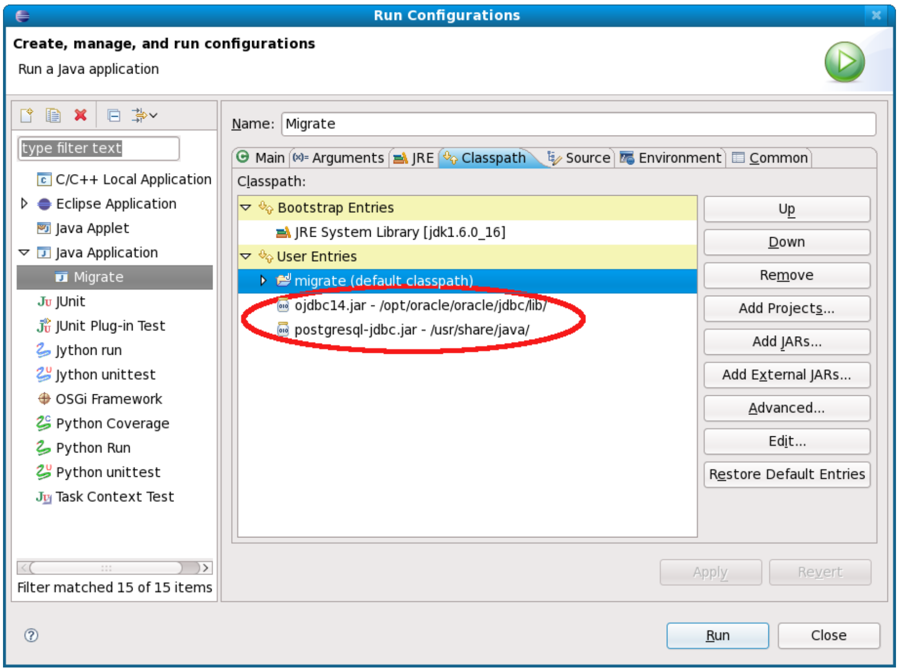
If you have installed Adempiere, they can be found in `$ADEMPIERE_HOME/lib`:

- `$ADEMPIERE_HOME/lib/oracle.jar` for Oracle
- `$ADEMPIERE_HOME/lib/postgresql.jar` for PostgreSQL

Otherwise they can be found in subdirectories of your local database installation, for example

- `$ORACLE_HOME/jdbc/lib/ojdbc14.jar` for Oracle
- `/usr/share/java/postgresql-jdbc.jar` for PostgreSQL

To add files or directories to the `classpath` in Eclipse (version 3.4.1), in the Run menu select `Run Configurations`..., select the `Classpath` tab and click the `Add External JARs`... button.



Name: Migrate

Classpath:
- Bootstrap Entries
  - JRE System Library [jdk1.6.0_16]
- User Entries
  - migrate (default classpath)
  - ojdbc14.jar - /opt/oracle/oracle/jdbc/lib/
  - postgresql-jdbc.jar - /usr/share/java/

Figure 4.1. JDBC drivers must be set in the `classpath` for Migrate to run in Eclipse

Extending MIGRATE

Source Files

Being open-source, MIGRATE has the advantage that you can modify the source code to fit your particular needs.

More than that, MIGRATE is designed to be easily extendable for localization and for handling additional database vendors, and you are invited to help and contribute your solutions to ADEMPIERE.

To help you navigate the source files, they are listed here by category:

Table 4.1. Source Files

Category	Source Files
Main class	`Migrate.java`
Parameters and constants	`Parameters.java`

Graphical User Interface	`Gui.java`
	`HelpAbout.java`
	`HelpInfo.java`
	`images/*`

Logging	`MigrateLogger.java`
	`MigrateLogger_Formatter.java`
	`MigrateLogger_Filter.java`
	`PreparedStatementWrapper.java`

Localization	`Messages.java`
User Documentation	`manual.xml` `images/doc_`
JDBC connection to database	`DBConnection.java`

Vendor-specific SQL-generation and database rules and conventions	`DBEngine.java`
	`DBEngineInterface.java`
	
	`DBEngine_Oracle.java`
	`DBEngine_Postgresql.java`

Database objects	`DBObject.java`
	`DBObjectInterface.java`
	`DBObjectDefinition.java`
	
	`DBObject_Table.java`
	`DBObject_Table_Column.java`
	
	`DBObject_PrimaryKey.java`
	`DBObject_PrimaryKey_Table.java`
	`DBObject_PrimaryKey_Column.java`
	
	`DBObject_ForeignKey.java`
	`DBObject_ForeignKey_Table.java`
	`DBObject_ForeignKey_Column.java`
	
	`DBObject_Check.java`
	`DBObject_Check_Table.java`

Category	Source Files

	`DBObject_Check_Rule.java`
	`DBObject_Unique.java`
	`DBObject_Unique_Table.java`
	`DBObject_Unique_Column.java`
	`DBObject_Index.java`
	`DBObject_Index_Table.java`
	`DBObject_Index_Column.java`
	`DBObject_View.java`
	`DBObject_View_Definition.java`
	`DBObject_Sequence.java`
	`DBObject_Sequence_Counter.java`
	`DBObject_Function.java`
	`DBObject_Function_Argument.java`
	`DBObject_Function_Body.java`
	`DBObject_Operator.java`
	`DBObject_Operator_Signature.java`
	`DBObject_Operator_Definition.java`
	`DBObject_Trigger.java`
	`DBObject_Trigger_Table.java`
	`DBObject_Trigger_Definition.java`

Application Dictionary Objects	`ADObject_TreeNode.java`

Adding Languages and Locales

All messages are contained in the resource file `Messages.java`, which contains US-English text as default locale.

To add additional languages or locales, copy `Messages.java` to a new file following Java's Resource Bundle [http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/] naming convention.

For example, to create a French resource file, name it `Messages_fr.java`.

To differentiate between French as spoken in France and French as spoken in Canada, create two resource files named `Messages_fr_FR.java` and `Messages_fr_CA.java`.

Of course the class declaration must be changed to match the file name, for example `public class` *`Messages`* `extends ListResourceBundle {` … would become `public class` *`Messages_fr_FR`* `extends ListResourceBundle {` ….

The file contains an array of `{“`*`key`*`”, “`*`localized String`*`”}` pairs. The keys should not be modified, as they are used to look up the localized string by the Resource Bundle. The localized string should be translated to the required language.

Note that while Resource Bundles generally accept `{"key", Object}` pairs, MIGRATE can only handle String values such as in `{"key", "String"}` pairs[2].

Adding Database Vendors

To be able to communicate with different database vendors and follow their conventions and rules, MIGRATE uses a layer of "database engines" which answer to specific predefined requests and provide vendor-specific SQL statements.

These database engines are implemented as JAVA Interfaces and can therefore easily be extended to other database vendors. In this case, "easily" just means that interfaces for additional database vendors can easily be added, but the actual programming and debugging of such interfaces will still be a laborious task.

The interface definition, manifested in source file `DBEngineInterface.java`, defines which functions a vendor-specific database engine must contain, what arguments those functions will be given, and what MIGRATE expects as return values. Consult the DBEngineInterface API [../apidoc/com/kkalice/adempiere/migrate/DBEngineInterface.html] for details (it is generated by javadoc during compilation).

Two database engines are included with the original distribution of MIGRATE: one for ORACLE and one for POSTGRESQL.

To add a new database engine, it is probably easiest to make a copy of the file which most closely matches the vendor you want to implement, name it according to the new vendor (for example, `DBEngine_MySql.java`, or `DBEngine_AdabasD.java`), and rename the class declaration inside the file (`public class DBEngine_MySql implements DBEngine_Interface {…`, or `public class DBEngine_AdabasD implements DBEngine_Interface {…`).

Then go through the methods step by step, compare the difference between `DBEngine_Oracle.java` and `DBEngine_Postgresql.java`, and figure out what your database vendor requires. After you are done programming the interface, extensive testing and debugging will follow.

To Do

The following are some features which would be nice for MIGRATE to have, but which have not been implemented yet.

The community is invited to submit contributions:

Identify Renamed Tables

In: `Migrate.synchronizeTables()`

---

[2] For this reason, to translate keyboard codes for mnemonic highlighting of menu items, labels, or buttons, the keyboard code, which is an int, is converted to an Integer which is converted to a String, as in:

```
…
{"guiMenuHelp", "Help"},
{"guiMenuHelpMnemonic", new Integer(KeyEvent.VK_H).toString()},
…
```

MIGRATE drops tables not existing in the reference database and adds tables not existing in the target. So if a table has been renamed, the data contained in that table will be lost. It is therefore necessary to identify tables which have been renamed.

The obvious solution would be to check the `AD_Element_ID` of the table's primary key, but that method will fail:

In the past, when `C_Allocation` was renamed to `C_AllocationLine`, the primary key `C_Allocation_ID` (element 1380) became `C_AllocationHdr_ID`, and a new primary key `C_AllocationLine_ID` (element 2534) was created for the renamed table.

A different solution must be found.

Preserve Parent Links

In: `Migrate.preserveParentLinks()`

If a table in the live database does not contain a column existing in the reference database, that column will be created with a default value. But if the new column is used as part of a foreign key constraint in the reference database, the default value will not reference any parent record in the target database, which will result in an error when the foreign key is created.

Such "unlinked" fields should be linked to the correct parent, and it must be deduced from other data in the table what the correct parent is.

Currently the hints how to find the correct parent are hard-coded.

At some time, a `C_Dunning_ID` column was added to the `C_DunningRun` table, which was used as a foreign key to `C_Dunning`. When running an upgrade migration, the column is added and filled with `0` as default value. But `0` does not point to any parent in the `C_Dunning` table, and would thus result in an error when the foreign key is created.

It turns out that `C_DunningRun` contains a column called `C_DunningLevel_ID`, which links to the table `C_DunningLevel`. And `C_DunningLevel` has a link to the `C_Dunning` Table. So the correct target for the new `C_Dunning_ID` column can be deduced by following the link to `C_DunningLevel_ID` and from there to `C_Dunning`.

This hint is currently hard-coded.

MIGRATE should be able to find out by itself how to deduce the correct parent.

As long as that can not be done, such hints must continue to be hard-coded as additional situations of this type are encountered.

Populate New Parents

In: `Migrate.populateNewParents()`

If new tables exist in the reference database but not in the target, they might be parent tables which must be filled with data from already existing child records.

Originally there was only a table `C_Allocation`. At some point, that table was renamed `C_AllocationLine`, and a new parent table `C_AllocationHdr` was introduced.

At that time, `C_AllocationHdr_ID` had to be set to the value of `C_AllocationLine_ID`, and columns in `C_AllocationHdr` that also existed in `C_AllocationLine` had to be filled with the values from `C_AllocationLine`, using

```
INSERT INTO … SELECT …;
```

The link from the child to the new parent record had to be set, and since the parent record's `C_AllocationHdr_ID` now had the same value as the child's `C_AllocationLine_ID`, it could easily be done with:

```
UPDATE C_AllocationLine SET C_AllocationHdr_ID = C_AllocationLine_ID WHERE C_AllocationHdr_ID IS NULL;
```

Finally, any references from other tables pointing to the old child table had to be re-directed to point to the new parent table, for example

```
UPDATE Fact_Acct SET AD_Table_ID=735 WHERE AD_Table_ID=390;
```

(`C_AllocationHdr` has `AD_Table_ID 735`, `C_AllocationLine` has `AD_Table_ID 390`)

Above is actually not so difficult to implement, but the problem is how to find the primary child table.

For example, if `C_InvoiceLine` and `C_InvoiceTax` exist, and a new table `C_Invoice` is created, how do we know that `C_InvoiceLine` is the table from which `C_Invoice` should be populated, not `C_InvoiceTax`?

Another problem arises from inconsistent table naming:

`C_Invoice` - `C_InvoiceLine` (the short name is the parent, the long name is the child)
`C_AllocationHdr` - `C_AllocationLine` (both parent and child names are long)
`GL_JournalBatch` - `GL_Journal` - `GL_JournalLine` (the parent has a long name, the child has a short name, and the grandchild has a long name again)

Translation of Functions

In: `DBEngine_`*`vendor`*`.translateFunctionBodyFull()`

MIGRATE can more or less successfully translate views using regular expressions, but the translation of functions is much more difficult.

Any help to translate functions between the different procedural languages native to each database vendor would be highly appreciated.

Fail-Safe / Safe-Fail

MIGRATE requires the migration process not to be interrupted.

If it does get interrupted, for example because of a power outage, you need to restore the live database from your backup and start the migration process again from scratch. That is because MIGRATE drops views, functions, constraints, indexes etc. before starting the migration process. If the migration process is interrupted before those objects are recreated, they will be lost forever.

It would be nice if MIGRATE saved the meta-data it gathered and then used that saved meta-data to resume migrations which were interrupted.

Delete Client / Delete Transactions

The original COMPIERE migration tool had a facility to delete transactions (in effect “resetting” a client) or to delete a client entirely. It is probably better not to include such functionality in MIGRATE but rather have a specialized tool for such kind of task.

However, if anybody sees the need to add such functionality to MIGRATE, there already is a private `dropClient()` function in the main `Migrate` class which can be made public and used for such purpose. (It is currently used to drop the *GardenWorld* client).

There is no function yet to delete only transactions.