

# EMu Documentation

## Scheduled Operations

Document Version 1.1

**EMu version 4.3**

**EMu**  
Museum  
Management  
System



kesoftware.com  
©2014 KE Software  
All rights reserved



# Contents

<b>SECTION 1</b>	<b>Overview</b>	<b>1</b>
<b>SECTION 2</b>	<b>How to schedule an operation</b>	<b>3</b>
	The Operation tab	3
	Delete Operation: the Delete tab	6
	Image Import Operation: the Image Import tab	8
	Merge Operation: the Merge tab	10
	Examples	12
	Scenario 1	12
	Scenario 2	14
<b>SECTION 3</b>	<b>Viewing Operation Results</b>	<b>17</b>
	View Result Files	17
	Save all Result Files	18
	Save a Result File	18
<b>SECTION 4</b>	<b>How to create an additional type of Scheduled Operation</b>	<b>19</b>
	Storage of Scheduled Operations scripts	20
	Invoking a scheduled operation	22
	Accessing information from a Scheduled Operations record	23
	An example operation	25
	Useful functions that may be called from within an operation	30
	OpenLogFile	31
	FileLog	31
	GetStartPosition	32
	AddToProcessed	32
	GetAttachmentFields	33
<b>SECTION 5</b>	<b>emuoperations</b>	<b>35</b>
	Using emuoperations	35
	Configuring emuoperations	36
	<b>Index</b>	<b>37</b>



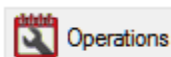
## SECTION 1

## Overview

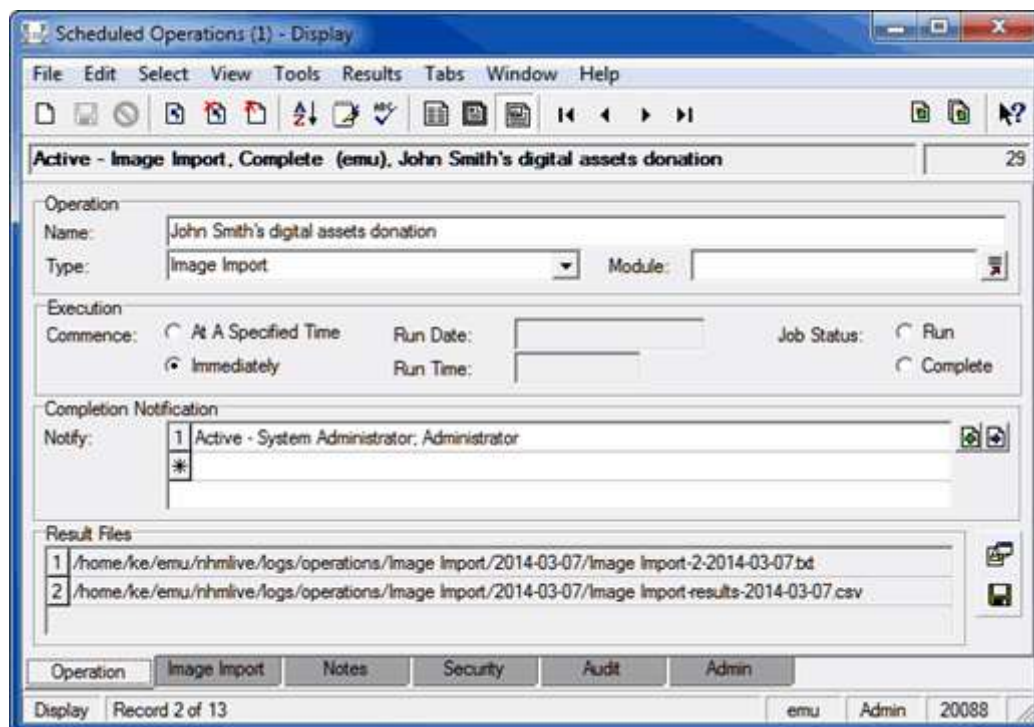


In order to use the Scheduled Operations facility, a user must have (or be a member of a group that has) Table Access to the Operations module (operations) and the daInsert operations permission.

The Scheduled Operations facility introduced with EMu 4.3 enables the scheduling of operations to be run immediately or at a specified date and time. Operations are scheduled in the Scheduled Operations module, which is accessed by selecting



in the Command Center:



With the Scheduled Operations facility it is possible to define:

- The type of operation to run
- The module to apply the operation to
- A time to commence the operation
- People to notify when the operation is complete

A scheduled operation is defined and stored as a record in the Scheduled Operations module.

When a scheduled operation is run, any files created during the operation are listed in the *Result Files* table on the Operation tab. Result files can be viewed and saved.

Audit logs are produced for all scheduled operations, allowing suitably authorized users to search / view the results of all operations performed by all users.

EMu 4.3 supports three types of scheduled operation:

- Merge Records
- Delete Records
- Image Import




System Administrators may define additional types of Scheduled Operation as required. See How to create an additional type of Scheduled Operation (page 19) for details.

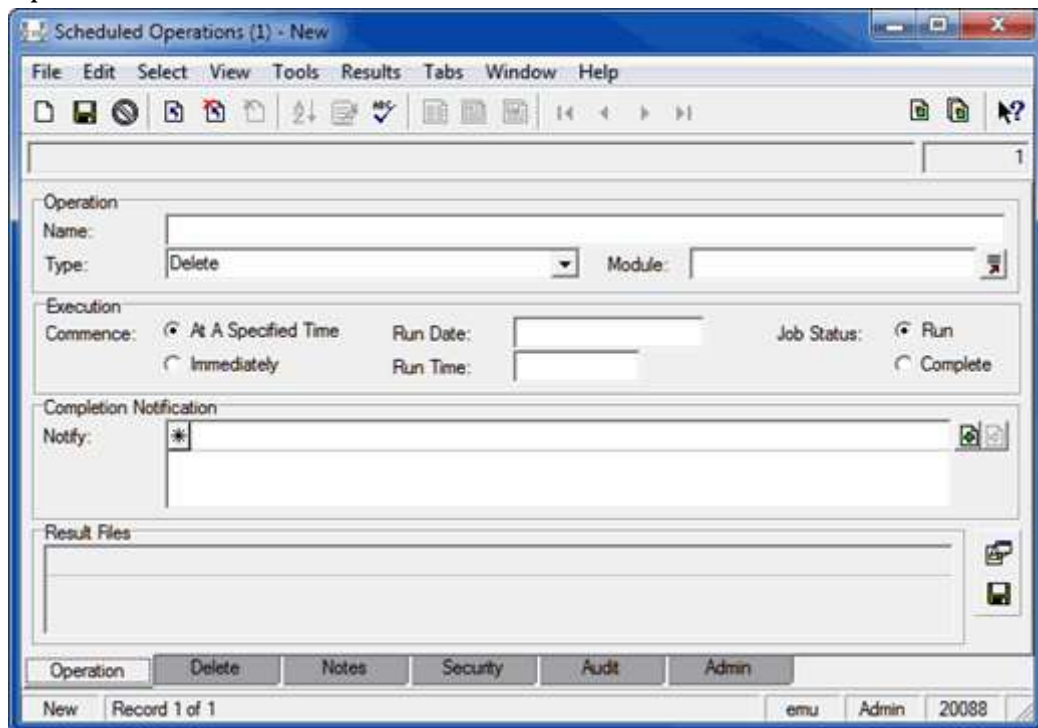
## SECTION 2

## How to schedule an operation

Scheduling an operation is similar to creating any other record.

### The Operation tab

1. Select  Operations in the Command Center to display the Scheduled Operations module:



2. Enter a descriptive name for the operation in the *Name: (Operation)* field.
3. Select the type of operation to be performed from the *Type: (Operation)* drop list. By default, there are three types of operation to choose from:
  - **Delete (page 6)**  
Delete a series of IRNs from a module.
  - **Image Import (page 8)**  
Import images from a directory into the Multimedia module.
  - **Merge (page 10)**  
Merge one or more records with a Target record in a module.



System Administrators may define additional operations as required. See How to create an additional type of Scheduled Operation (page 19) for details.

4. In the *Module: (Operation)* field, select the module in which the operation is to be performed.



When scheduling an `Image Import` it is not necessary to specify a module as `emultimedia` (the `Multimedia` module) is implicit to the operation (images are imported into the `emultimedia` table).

5. In the *Execution* group of fields specify the time that the operation will be executed. There are two options:

- At A Specified Time

With this option selected it is possible to specify a *Run Date* and *Run Time* for the operation to commence its processing. This allows operations to be run outside of normal business hours or at the weekend.

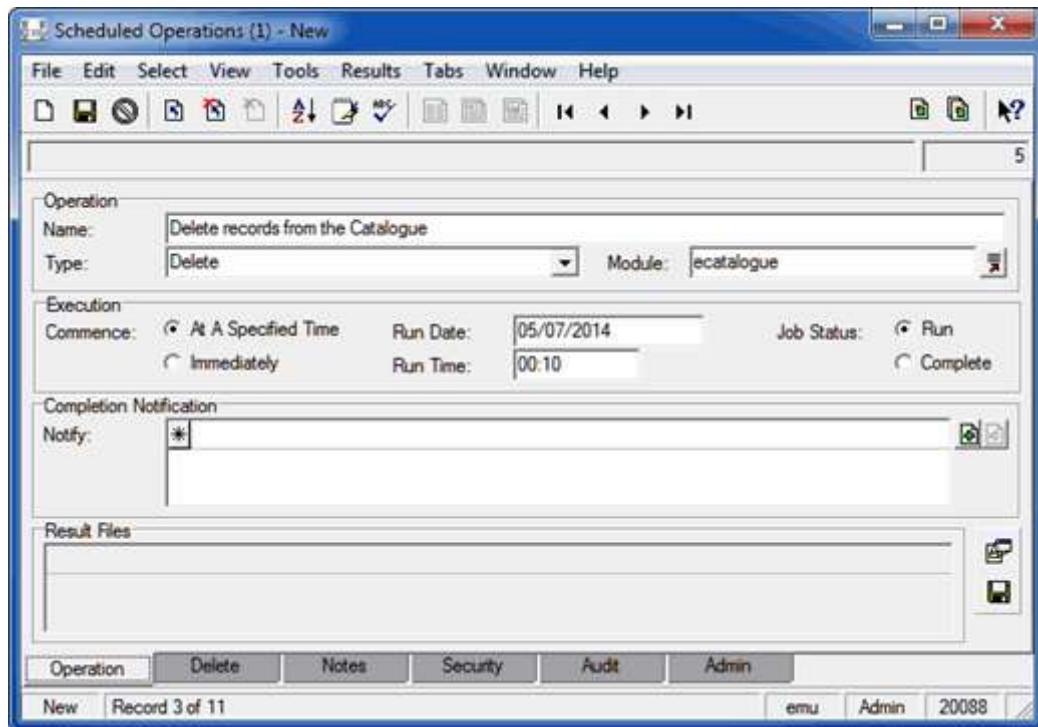


A date and time specified here is the **earliest** that the operation will be run. The actual time at which an operation is run will depend on when the `emuoperations` script is scheduled to run (page 22): `emuoperations` is the script used to execute an operation that has been scheduled in a record in the `Schedule Operations` module (page 35). When `emuoperations` is run, it looks for any operations that were scheduled to run prior to the current date and time and commences them. Thus, if `emuoperations` is scheduled to run once per day, it will commence any operation scheduled to run in the previous 24 hours (in theory an operation could have been scheduled to run 23 hours and 59 minutes earlier). If `emuoperations` is to be run once per day, it probably makes sense therefore to schedule operations close to the time at which `emuoperations` is run. Alternatively, `emuoperations` can be run at various times throughout the day.

- Immediately

With this option the operation will commence as soon as the record is saved.





6. In *Notify: (Completion Notification)* attach the Parties record for anyone who is to be notified by email when the scheduled operation has completed.

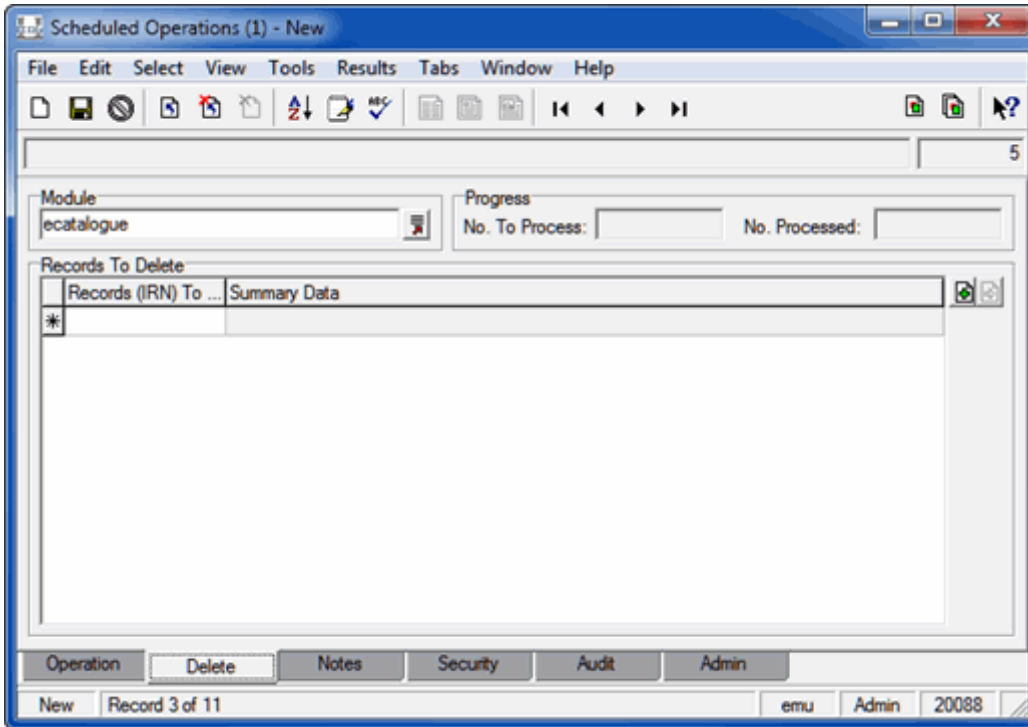





Email notifications will only be received by parties added to the *Notify: (Completion Notification)* table if their Parties record includes a valid email address in the *Email: (Internet Details)* field.

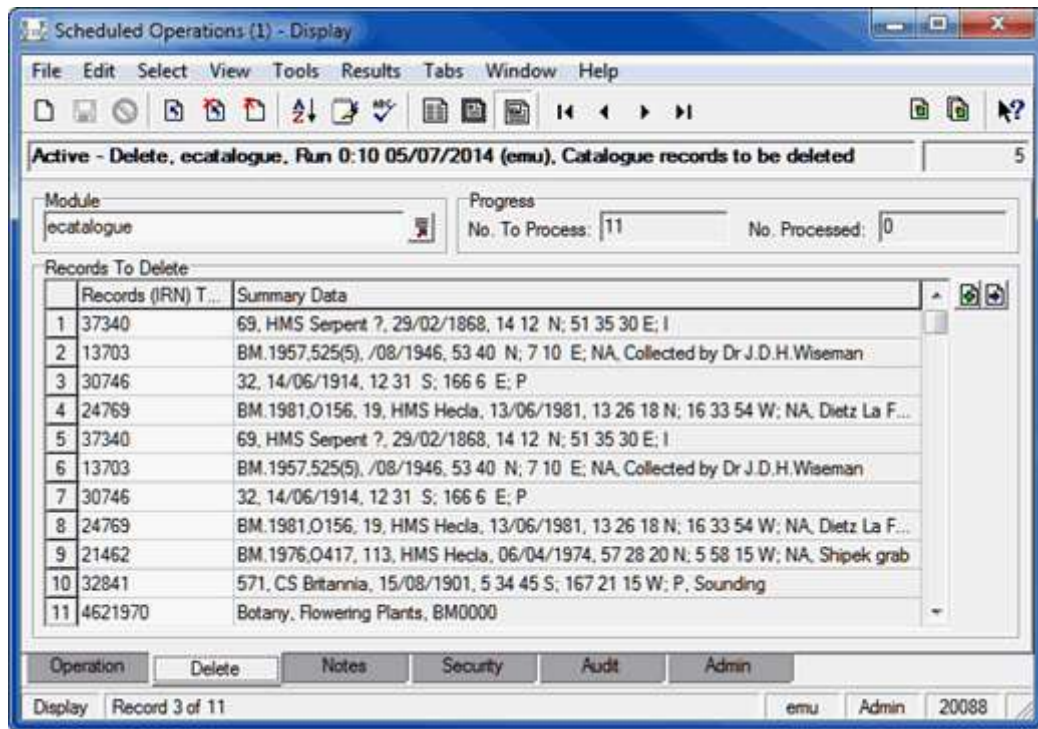
*Job Status: (Execution)* indicates that the operation is waiting to be run, or that it has been run and is complete. Note that if an operation terminates unexpectedly, the status will remain as *Run* until the operation is restarted and it completes.

## Delete Operation: the Delete tab

When **Delete** is selected from the *Type: (Operation)* drop list on the Operation tab, the Delete tab displays:

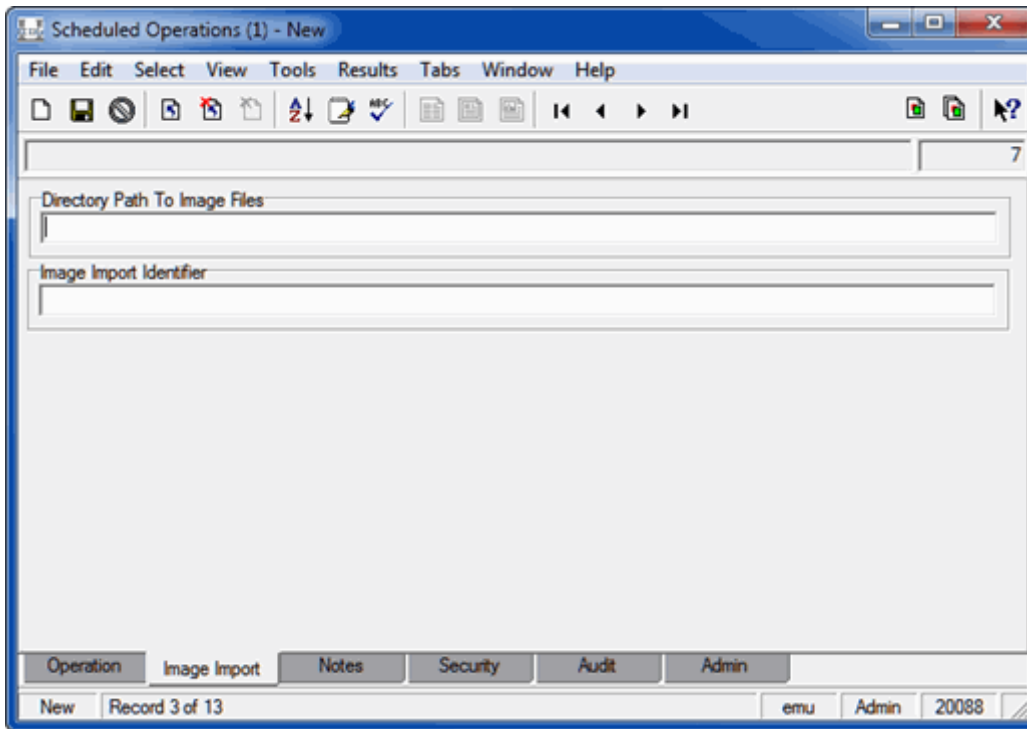


1. The *Module* field will list the module from which records will be deleted if a module was specified (Step 4) on the Operation tab (page 3).  
If a module was not selected on the Operation tab, specify in the *Module* field which module the records are to be deleted from.
2. In the *Records To Delete* table add the records that are to be deleted from the module specified in the *Module* field.  
Records can be added through the attachment or drag and drop process:
  - 2.1. Click  beside the *Records To Delete* table to open the module specified in the *Module* field.
  - 2.2. Search the module for the record or records to delete and click **Attach Current Record**  or **Attach Selected Records**  in the Tool bar to add the record(s) to the *Records To Delete* table in the Scheduled Operations module.  
-OR-
  - 2.3. Open the module specified in the *Module* field and search for the record or records to be deleted.
  - 2.4. Select the record or records in List View and drag and drop them to the *Records To Delete* table in the Scheduled Operations module.
3. Save the record:

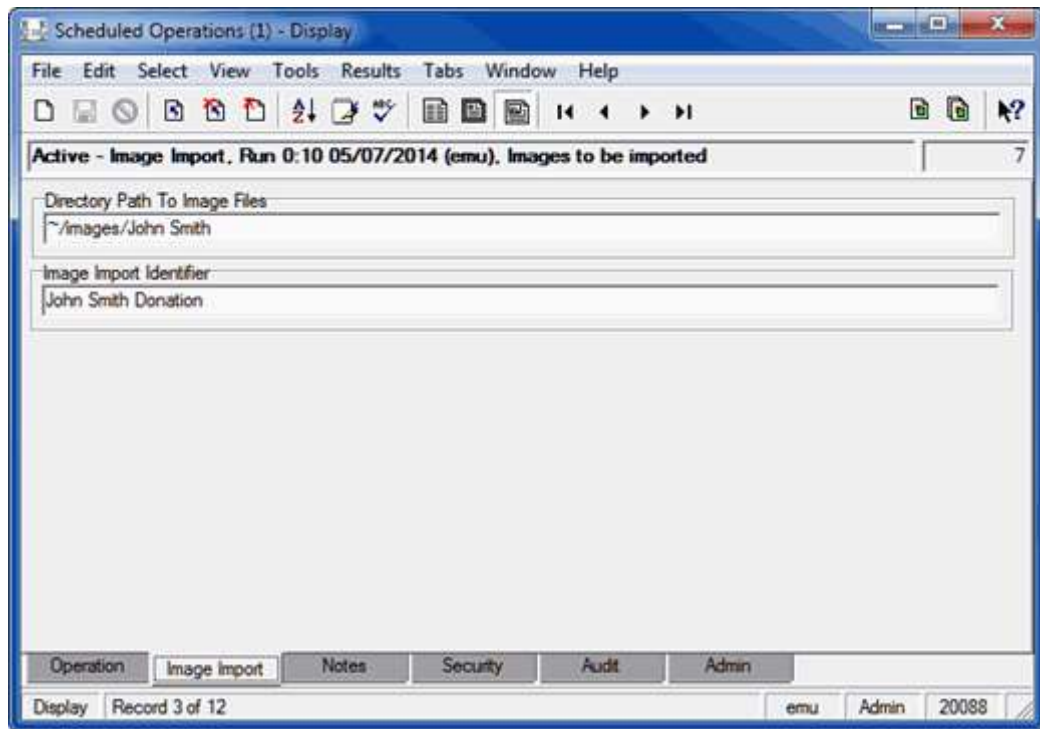


## Image Import Operation: the Image Import tab

When **Image Import** is selected from the *Type: (Operation)* drop list on the Operation tab, the Image Import tab displays:

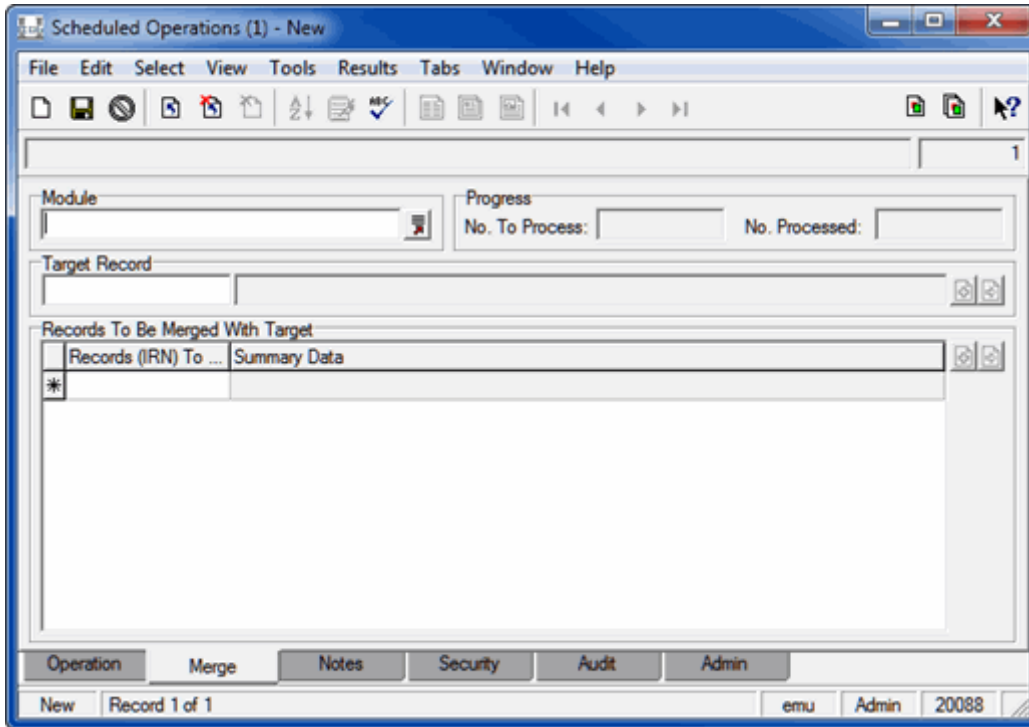




1. In *Directory Path To Image Files*, enter the pathway to the image files to be imported. The path may be a full path:  
/home/emu/..  
or a relative path:  
~/.../.. or .../..
2. If required, enter an identifier in the *Image Import Identifier* field. The value entered here will be stored in the *Import Identifier* field on the Admin tab of all Multimedia records created through this scheduled import.
3. Save the record:



## Merge Operation: the Merge tab



When **Merge** is selected from the *Type: (Operation)* drop list on the Operation tab, the Merge tab displays:

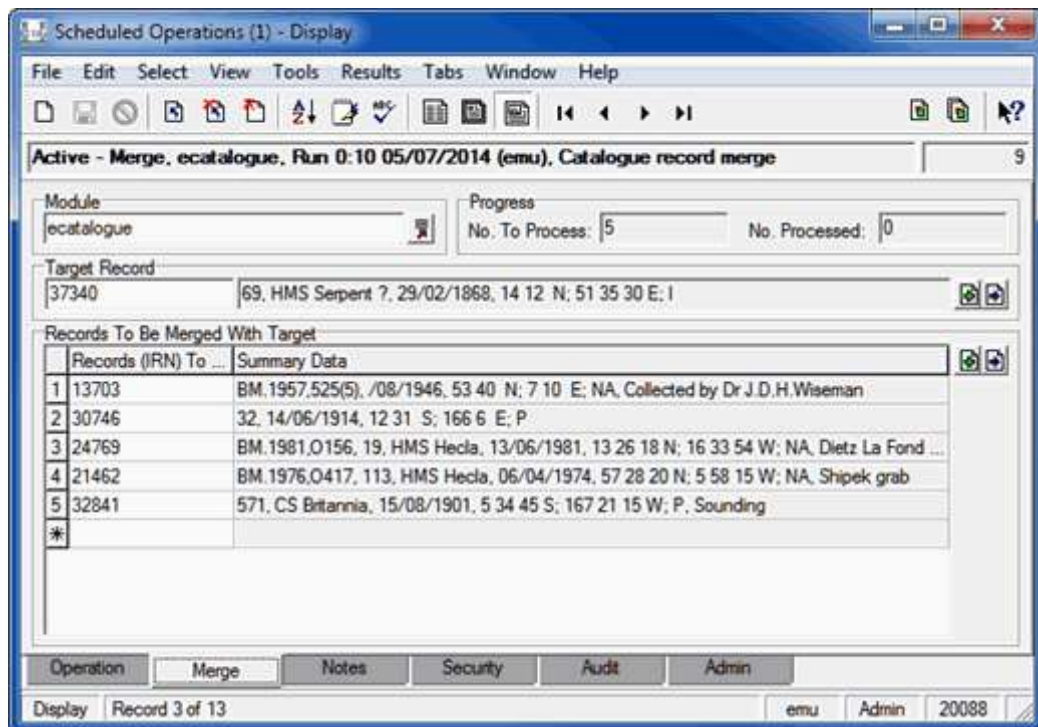


1. The *Module* field will list the module in which records will be merged if a module was specified (Step 4) on the Operation tab (page 3).  
If a module was not selected on the Operation tab, specify in the *Module* field in which module the merge will take place.
2. In the *Target Record* field add the record that will be the target of the merge (i.e. the record with which one or more records will be merged).  
Records can be added through the attachment or drag and drop process:
  - 2.1. Click  beside the *Target Record* field to open the module specified in the *Module* field.
  - 2.2. Search the module for the Target Record and click **Attach Current Record**  in the Tool bar to add the record to the *Target Record* field in the Scheduled Operations module.

-OR-

  - 2.3. Open the module specified in the *Module* field and search for the Target Record.
  - 2.4. Drag and drop the Target Record to the *Target Record* field in the Scheduled Operations module. There are various ways to do this:
    - In List View click the record to drag and drop it on the *Target Record* field in the Scheduled Operations module.

- Select the record in List View and drag the Drag Current Record button  in the Tool bar to the *Target Record* field in the Scheduled Operations module.
  - Display the record in Details View and drag the Drag Current Record button  in the Tool bar to the *Target Record* field in the Scheduled Operations module.
3. In the *Records To Be Merged With Target* table add the records that are to be merged with the Target Record  
Records can be added through the attachment or drag and drop process described earlier (page 6).
  4. Save the record:





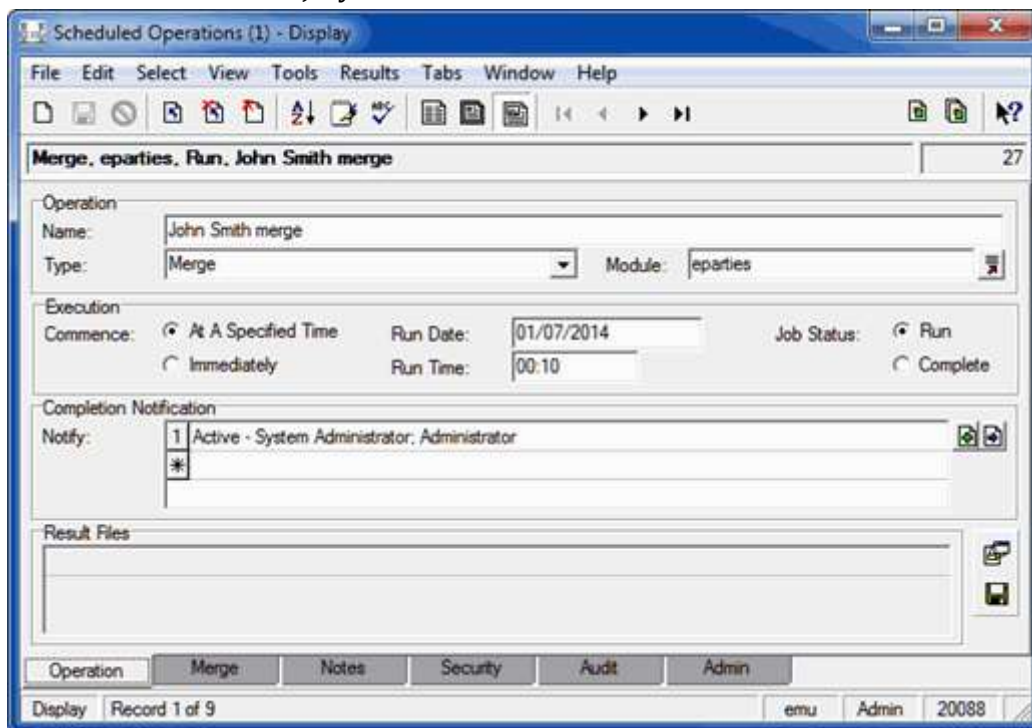
## Examples

### Scenario 1

A record clean up project is under way. As part of the clean up we wish to merge five variations of John Smith's Parties record into one. As users are still entering records, we need to wait until 1 July before we can run the Merge.

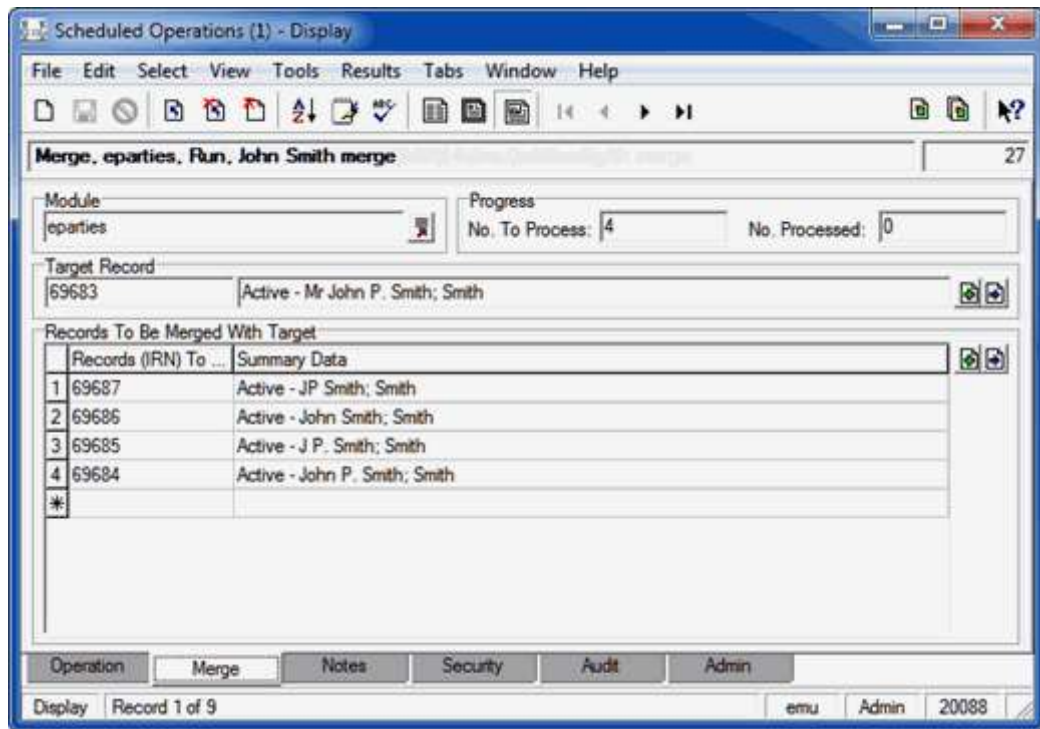
#### Solution

1. Add a Scheduled Operations record with a *Type* of Merge for eparties, scheduled to run at 12:10 AM on 1 July:



2. Identify one of the five John Smith Parties records as the Target Record and attach it to the *Target Record* field on the Merge tab of the Scheduled Operation record.
3. Add the remaining four Parties records for John Smith to the *Records To Be Merged With Target* table:



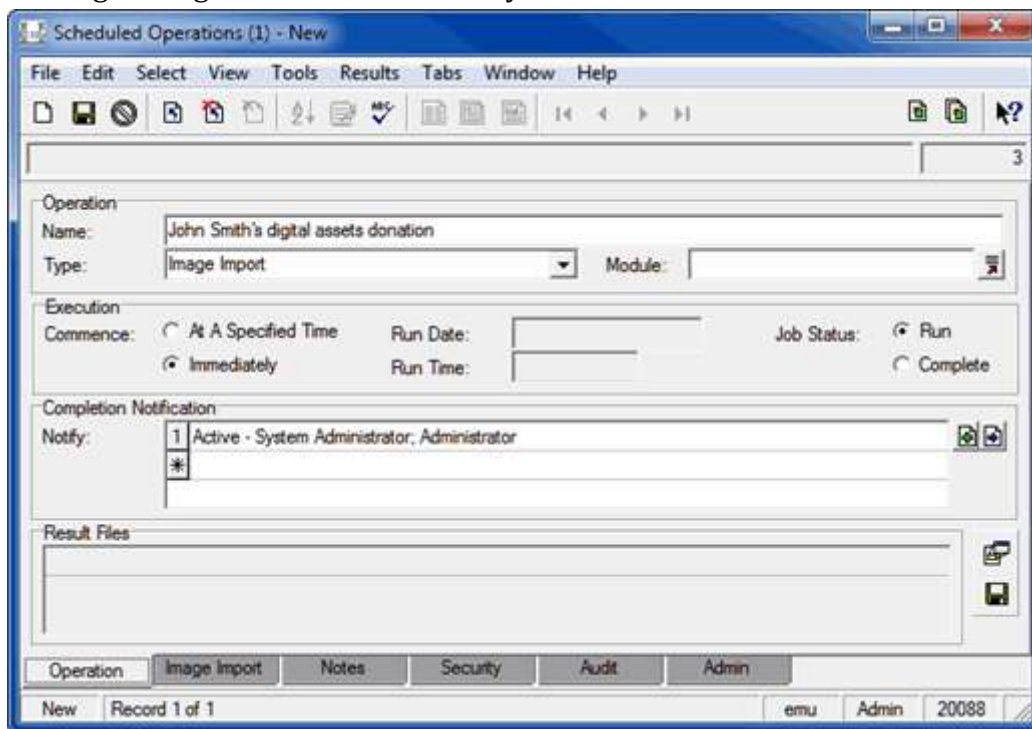


## Scenario 2

A large number of digital assets have been donated to your institution. Rather than load them individually, you would like to have them loaded automatically commencing immediately.

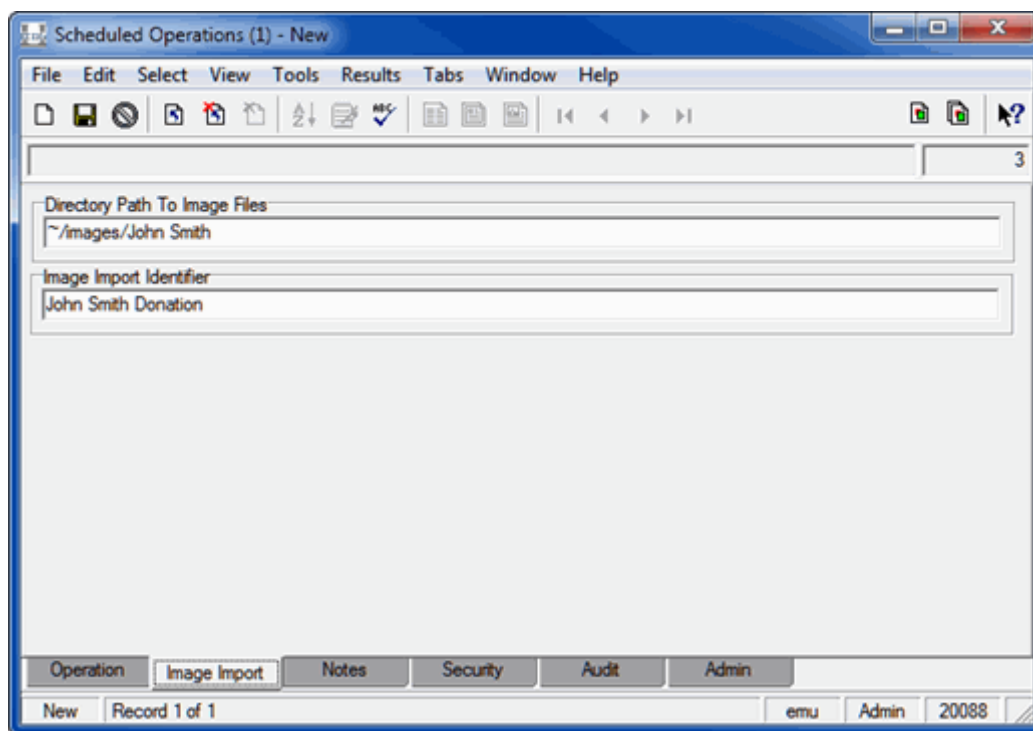
### Solution

1. Add a Scheduled Operations record with a *Type* of Image Import to commence loading the digital assets immediately:



When scheduling an Image Import it is not necessary to specify a module as multimedia (the Multimedia module) is implicit to the operation (images are imported into the multimedia table).

2. On the Image Import tab specify the directory where the digital assets are stored and an identifier for the created records:



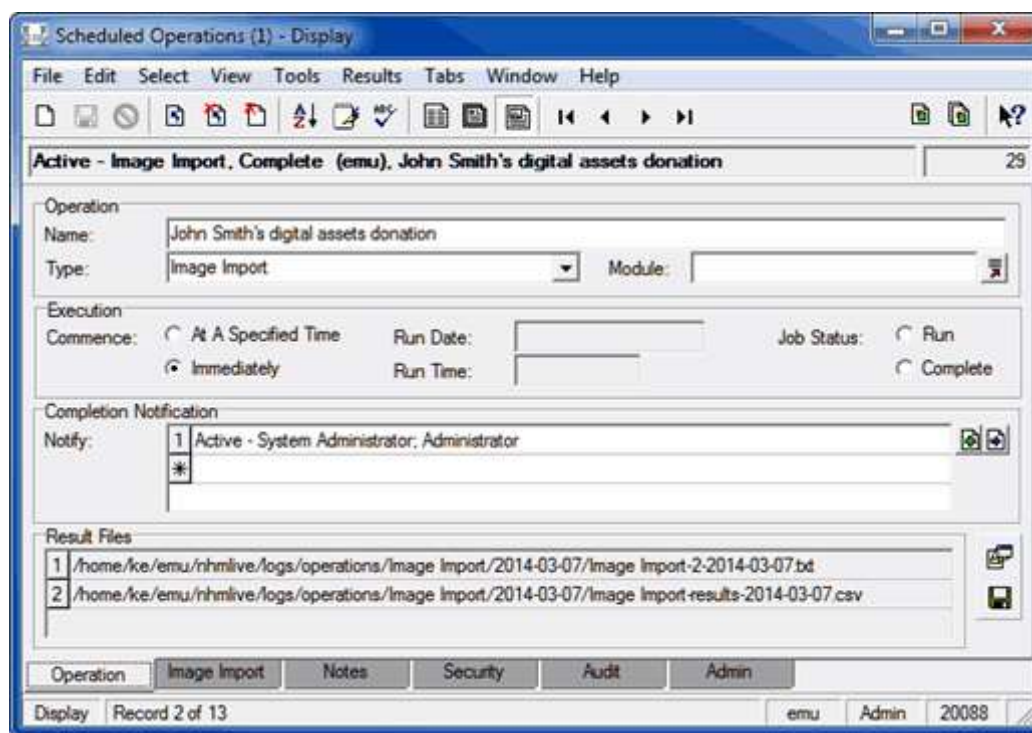
When this record is saved the digital asset import will commence without the need for any further action from the user, who will be able to continue with their other work.



## SECTION 3


## Viewing Operation Results

Scheduled operations are run automatically by EMu. For each operation executed a Results File is created and added to the *Result Files* table on the Operation tab of the Scheduled Operations record. The files are stored on the EMu server:


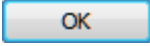


## View Result Files

1. Select **Results>Launch Viewer>[Result File]** in the Menu bar.  
-OR-

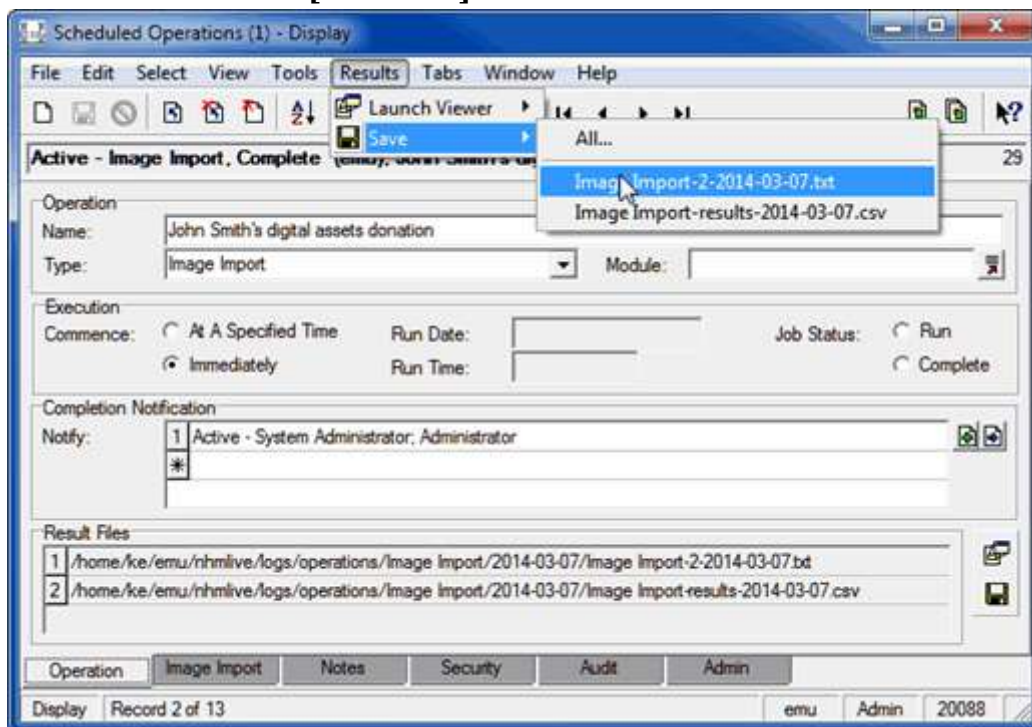
Select the row in the *Result Files* table with the file to be viewed and click . The application / viewer associated with the file extension is invoked to display the file.

## Save all Result Files

1. Select  beside the *Result Files* table  
-OR-  
Select **Results>Save>All** in the Menu bar.  
The Browse for Folder dialog displays.
2. Choose the directory into which all Result Files will be saved.
3. Select .

## Save a Result File

1. Select **Results>Save>[Result File]** in the Menu bar:



The Save As dialog displays.

2. Choose the location to save the Result File and click .

## SECTION 4

# How to create an additional type of Scheduled Operation

EMu provides three Scheduled Operations functions by default:

- Delete
- Image Import
- Merge

In this section we examine how System Administrators can create an additional type of Scheduled Operation.

## Storage of Scheduled Operations scripts

Each type of Scheduled Operation (e.g. Delete, Merge, etc.) is defined by a script which resides under the `etc/operations` or `local/etc/operations` directory on the EMu server.



When adding a script for an additional type of Scheduled Operation for your EMu system, place it under `local/etc/operations` to avoid the risk of having it overwritten during EMu upgrades.

The script includes the name of the operation which will be listed in the *Type: (Operation)* drop list on the Operation tab of the Scheduled Operations module.

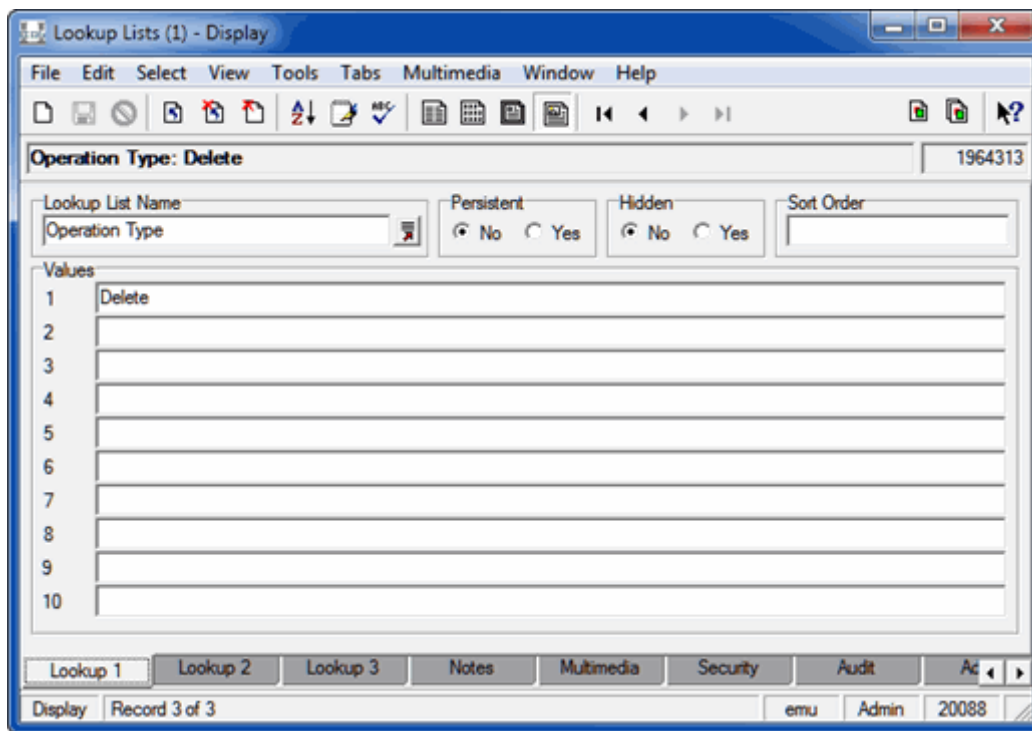
When the `emuoperations` process runs it scans the `etc/operations` and `local/etc/operations` directories to locate the scripts for all types of Scheduled Operations (files that end with a `.pl` extension) and registers a name for each type of operation found. The following example registers the `Delete` Scheduled Operation:

```
sub
Register
{
    my $plugins = shift;

    #
    # We handle the "Delete" method.
    #
    $plugins->{"Delete"} = \&Delete;
}
```

When a new type of Scheduled Operation is added to EMu, a Lookup List entry needs to be added to the *Operation Type* Lookup List. For the above example a Lookup List record was added to the *Operation Type* Lookup List with a value of `Delete`:





---

## Invoking a scheduled operation

When scheduling an operation in a record in the Scheduled Operations module, the operation can be scheduled to commence:

- At A Specified Time
- OR-
- Immediately

If *Commence: (Execution)* is set to `Immediately`, the operation will be invoked as soon the Scheduled Operations record is saved. The operation will commence running on the EMu server and control returned to the user to continue with their work.

If *Commence: (Execution)* is set to `At A Specified Time`, the operation will be invoked by the `emuoperations` script on the EMu server at the appropriate time.



The execution of each pending operation consumes a licence in the same way that a user would consume a licence to complete the task. Similarly to users performing tasks, multiple operations can be run simultaneously up to the system licence limit.

The `emuoperations` script is designed to be run from cron with an entry similar to the following:

```
30 17 * * * /home/ke/emu/client/bin/emurun emuoperations 2>&1 |
/home/ke/emu/client/bin/emurun emulogger -t "KE EMu Operations" -z
operations
```

The script is typically run once per day but can be configured to run any number of times during the day. When the `emuoperations` script runs it looks for any operations that were scheduled to run prior to the current date and time and commences them.



A date and time specified in a Scheduled Operations record is thus the **earliest** that the operation will be run. The actual time at which an operation is run will depend on when the `emuoperations` script is scheduled to run (page 22): `emuoperations` is the script used to execute an operation that has been scheduled in a record in the Schedule Operations module (page 35). When `emuoperations` is run, it looks for any operations that were scheduled to run prior to the current date and time and commences them. Thus, if `emuoperations` is scheduled to run once per day, it will commence any operation scheduled to run in the previous 24 hours: in theory an operation could have been scheduled to run 23 hours and 59 minutes earlier. If `emuoperations` is to be run once per day, it probably makes sense therefore to schedule operations close to the time at which `emuoperations` is run. Alternatively, `emuoperations` can be run at various times throughout the day.

`emuoperations` will also re-run any previous operations that did not complete.

---

## Accessing information from a Scheduled Operations record

Each type of Scheduled Operation registers a function that is called to process the operation. For example, the Delete Scheduled Operation is performed by a registered function called `Delete`.

```
sub
Register
{
    my $plugins = shift;

    #
    # We handle the "Delete" method.
    #
    $plugins->{"Delete"} = \&Delete;
}
```

The function is passed two parameters:

- An IMu session which allows access to EMu records to perform the operation.
- A hash of data from a Scheduled Operations record with details about this particular operation (i.e. when, what records are affected, what module, etc.).

```
sub
Delete
{
    my $imusection = shift;
    my $record = shift;

    #
    # Run the "Delete" operation.
    #
    ...
}
```

The list of keys available in the hash are:

<code>Irnr</code>	The IRN of a record in the Scheduled Operations module with details about this scheduled operation.
<code>Name</code>	The name of the operation.
<code>Type</code>	The type of operation.
<code>Module</code>	The module the operation is to be performed on.
<code>ActionIrnr</code>	The target IRN for the Merge operation.
<code>IrnrnToProcess</code>	The list of IRNs that the operation needs to process.
<code>IrnrnProcessed</code>	The list of IRNs that the operation has already processed.



Typically this would be an empty list except when an operation failed to complete.

<code>Directory</code>	The directory which contains files / information required by an operation to process.
<code>Identifier</code>	An identifier to add to records updated as part of running the operation.

The values for the keys are accessed through the `$record` parameter, e.g.:

```
$record->{Module}
```

-OR-

```
@{$record->{IrnrnToProcess}}
```

## An example operation

In this example a list of IRNs is deleted:

```
#!/usr/bin/perl

use strict;
use warnings;
use lib "$ENV{EMUPATH}/utils/imu/lib";
use IMu::Module;

#
# Registration function.
#
no warnings 'redefine';

sub
Register
{
    my $plugins = shift;

    #
    # We handle the "Delete" method.
    #
    $plugins->{"Delete"} = \&Delete;
}

use warnings 'redefine';

#
# The handler for the "Delete" operation
#
sub
Delete
{
    my ($imusection, $record) = @_;
    my ($attachments, $start, @deleteirns, $irn, $i);

    #
    # Check that we have the required information
    #
    if (! defined($record->{IrnsToProcess}) || @{$record->{IrnsToProcess}} == 0)
    {
        FileLog("Error: no irns supplied for
deletion");
        return(1);
    }
    elsif (! defined($record->{Module}) or $record->{Module} eq "")
```

```

        {
            FileLog("Error: delete module is not
defined");
            return(1);
        }

        #
        # Get the other information that we need to process
        #
        $attachments = GetAttachmentFields($record->{Module});
        @deleteirns = @{$record->{IrnsToProcess}};
        $start = GetStartPosition($record);
        FileLog("Running DELETE plugin for $record-
>{Module}");
        FileLog("%d records scheduled for deletion, starting
at position $start", scalar(@deleteirns));

        #
        # Now delete each record in turn
        #
        for ($i = $start; $i < @deleteirns; $i++)
        {
            $irn = $deleteirns[$i];
            FileLog("Deleting irn $irn...");
            last if (! ProcessDeletion($imusection,
$attachments, $irn, $record));
            AddToProcessed($irn);
        }
        return($i != @deleteirns);
    }

    #
    # Do the actual deletion work
    #
    sub
    ProcessDeletion
    {
        my ($imusection, $attachments, $irn, $record) = @_ ;
        my ($table, $colname, $module, @matches, $hits,
%found, $key, $column);

        eval
        {
            %found = ();
            foreach $key (keys %{$attachments})
            {
                #
                # The assignment here is unusual but
it gets around an
                # odd foreach scoping problem after an

```

```

exception is thrown.
                                #
                                $table = $key;
                                $module = IMu::Module->new($table,
$imusection);
                                foreach $column (keys %{$attachments-
>{$table}})
                                {
                                #
                                # Find records which match
                                #
                                # $colname = $column;
                                $hits = $module-
>findTerms([$colname, $irn]);
                                next if ($hits <= 0);
                                #
                                # Add records to found hash
                                #
                                FileLog("Found $hits matches
for $colname in $table");
                                push(@{$found{$table}-
>{$colname}}, GetMatches($module));
                                }
                                }
                                };
                                if ($?)
                                {
                                FileLog("Error: failed to process $colname in
$table for irn $irn: $?");
                                return(0);
                                }
                                @matches = keys %found;
                                if (@matches)
                                {
                                #
                                # Log that we cannot delete the record
                                #
                                FileLog("Unable to delete irn $irn because it
is attached in the following places:");
                                foreach $table (@matches)
                                {
                                foreach $colname (keys
%{$found{$table}})
                                {
                                FileLog("\tModule: $table,
Column: $colname, Record(s): " . join(", ", @{$found{$table}-
>{$colname}}));
                                }
                                }
                                }

```

```
        }
    }
    else
    {
        #
        # Delete the record
        #
        DeleteRecord($imusection, $irn, $record);
    }
    #
    # Add irn to processed
    #
    return(1);
}

#
# Delete the record
#
sub
DeleteRecord
{
    my ($imusection, $irn, $record) = @_ ;
    my ($module, $hits, $result);

    eval
    {
        $module = IMu::Module->new($record->{Module},
$imusection);
        $hits = $module->findKey($irn);
        if ($hits > 0)
        {
            $result = $module->remove("start", 0,
1);
            if ($result == 0)
            {
                FileLog("Failed to delete irn
$irn from $record->{Module}");
            }
        }
        else
        {
            FileLog("Failed to find irn $irn in
$record->{Module}");
        }
    };
    if ($?)
    {
        FileLog("Failed to delete $irn from $record-
>{Module}: $?");
    }
}
```



```
}

#
# Get all the records that match the attachment query
#
sub
GetMatches
{
    my ($module) = @_;
    my ($result, @matches, $row);

    #
    # Get all of the records at once
    #
    @matches = ();
    $result = $module->fetch("start", 0, -1, "irn");
    if ($result->{count})
    {
        #
        # Get the irn for each row and push it to the
list of matches
        #
        foreach $row (@{$result->{rows}})
        {
            push(@matches, $row->{irn});
        }
    }
    return(@matches);
}

1;
```

---

## Useful functions that may be called from within an operation

The following functions are available to be called for use within an operation:

<code>OpenLogFile</code> (page 31)	Opens a results log file and adds it to the list of Result Files.
<code>FileLog</code> (page 31)	Writes a message to the standard operation Result File.
<code>GetStartPosition</code> (page 32)	Determines from what position to start processing the <code>IrnsToProcess</code> list.
<code>AddToProcessed</code> (page 32)	Adds the processed IRN to the <code>IrnsProcessed</code> list.
<code>GetAttachmentFields</code> (page 33)	Returns a hash of all attachment fields for a module.

## OpenLogFile

Input parameters:   Filename

Returns:            File Handle for writing and an indication if the file already exists  
(from a previous attempt to run the operation)

```
sub
DoSomething
{
    my $handle;
    my $exists;

    #
    #  Open a file for logging results.
    #
    ($handle, $exists) = OpenLogFile("results.csv");
    if ($exists)
    {
        print $handle "...Resuming processing...";
    }
    ...
    close($handle);
}
```

## FileLog

Input parameters:   Format string and parameters

Returns:            Nothing

```
sub
DoSomething
{

    #
    #  Log a message.
    #
    FileLog("Error: no irns supplied for deletion");
    ...
    #
    #  Log a formatted message.
    #
    FileLog("%d records scheduled for deletion, starting
at position $start", scalar(@deleteirns));
}
```

## GetStartPosition

Input parameters: Record hash passed to operation

Returns: Index into `IrnsToProcess`

```
sub
Operation
{
    my $imusection = shift;
    my $record = shift;
    my $start;

    #
    # Get the start position for processing the
operation.
    #
    $start = GetStartPosition($record);
    ...
}
```

## AddToProcessed

Input parameters: IRN

Returns: Nothing

```
sub
Operation
{
    my $imusection = shift;
    my $record = shift;
    my $irn;

    ...

    #
    # Finished processing the operation on an irn.
    #
    AddToProcessed($irn);
    ...
}
```

## GetAttachmentFields

Input parameters: Module

Returns: A hash of modules with attachment columns to the requested module

```

sub
Operation
{
    my $imuseession = shift;
    my $record = shift;
    my $attachments;
    my $module;
    my $column;

    ...

    #
    # Get the attachment fields for the operation module.
    #
    $attachments = GetAttachmentFields($record->{Module});
    ...

    #
    # Process the attachment fields.
    #
    foreach $module (keys %{$attachments})
    {
        foreach $column (keys %{$attachments->
>{$module}})
        {
            ...
        }
    }
    ...
}

```



## SECTION 5

## emuoperations

`emuoperations` is a script used to execute scheduled operations.



A date and time specified in a Scheduled Operations record is the **earliest** that the operation will be run. The actual time at which an operation is run will depend on when the `emuoperations` script is scheduled to run (page 22). When run, `emuoperations` looks for any operations that were scheduled to run prior to the current date and time and commences them. Thus, if `emuoperations` is scheduled to run once per day, it will commence any operation scheduled to run in the previous 24 hours (in theory an operation could have been scheduled to run 23 hours and 59 minutes earlier). If `emuoperations` is to be run once per day, it probably makes sense therefore to schedule operations close to the time at which `emuoperations` is run. Alternatively, `emuoperations` can be run at various times throughout the day.

---

## Using emuoperations

`emuoperations` may be used in two ways:

1. **Run all Scheduled Operations**

Usage: `emuoperations`

Any Scheduled Operations required to be run will be executed. The current date and time is used to determine what operations are required. This form of the command is used by cron on a daily basis to ensure all Scheduled Operations for the given day are performed.

3. **Run a specific Scheduled Operation**

Usage: `emuoperations -iirn`

The `irn` argument is the IRN of a Scheduled Operations record to be executed. This form of `emuoperations` is useful for testing new operations as it allows a specific operation to be run without waiting for the Scheduled Operations date and time to arrive.

## Configuring emuoperations

The `emuoperations` script connects to an `imuserver` to perform the scheduled operations. This connection needs to be made on a specific port. By default the standard EMu configuration port for IMu is the port number 20,000 greater than EMu's client connection port. For example, if the standard EMu client connection port is 20000, the standard `imuserver` connection port is 40000.

The `emuoperations imuserver` must run on a different port to perform the scheduled operations. The `eoperations` load starts the `imuserver` for handling operation requests. The port for `emuoperations` to connect on is defined by the `EMUSERVERPORT` environment variable plus 30000. `EMUSERVERPORT` is the port the EMu client uses to connect to the EMu server and corresponds to the Service value entered in the EMu Client login box.

It is recommended that the Administrator sets the `EMUSERVERPORT` environment variable in the `etc/config` file on the EMu server. Add the following text to the end of the `etc/config` file (if it does not exist already):

```
#
# EMUSERVERPORT is the port the EMu client uses to connect to the
# EMu server.
# The port corresponds to the "Service" value entered in the EMu
# Client Login box.
#
EMUSERVERPORT=port
export EMUSERVERPORT
```

where *port* is the service name used to connect to this EMu server.



# Index

## A

- Accessing information from a Scheduled Operations record • 24
- AddToProcessed • 31, 33
- An example operation • 26

## C

- Configuring emuoperations • 36

## D

- Delete Operation  
the Delete tab • 3, 6, 11

## E

- emuoperations • 4, 22, 35
- Examples • 12

## F

- FileLog • 31, 32

## G

- GetAttachmentFields • 31, 34
- GetStartPosition • 31, 33

## H

- How to create an additional type of Scheduled Operation • 2, 4, 19
- How to schedule an operation • 3

## I

- Image Import Operation  
the Image Import tab • 3, 8
- Invoking a scheduled operation • 4, 22, 35

## M

- Merge Operation  
the Merge tab • 3, 10

## O

- OpenLogFile • 31, 32
- Overview • 1

## S

- Save a Result File • 18
- Save all Result Files • 18
- Scenario 1 • 12
- Scenario 2 • 14
- Storage of Scheduled Operations scripts • 20

## T

- The Operation tab • 3, 6, 10

## U

- Useful functions that may be called from within an operation • 31
- Using emuoperations • 35

## V

- View Result Files • 17
- Viewing Operation Results • 17