Processes and automation

ELO Automation Services

Table of contents

Basics 4
How ELOas works 4
Rulesets 6
Basics 6
Create ruleset via ELO Administration Console 7
Options and error handling 12
Manual start of a ruleset 14
The rule structure 20
Programming 28
Programming with ELO Automation Services 28
ELOas JavaDoc 34
Debugging 35
Standard modules 40
Examples 68
Example – Moving a document 68
Example: e-mail folder monitoring 72
Example – migrating a document database 78
Example - Treewalk for ELOas 82
Example - Workflow processing 87
Filing via ELO Dropzone 90
ELOas filing via ELO Dropzone tiles 90
Barcode 94
Introduction 94
Reading barcodes with the Softek library 95
Reading barcodes with the ZXing library 97
Creating barcodes with the ZXing library 99
Debugger 100
ELOas debugger 100
Debugger (Java FX) 105
Opening the program 105
User interface 106
Starting an ELOas rule 109
Profiles 110
Keyboard shortcuts 116

Java libraries 117
ELOas debugger on Linux 120

Other topics 121

Manual installation of ELOas 121
Installing multiple ELOas instances 129
Installing ELOas libraries 133

Basics

How ELOas works

ELOas is a servlet that can post-process any number of ELO documents in a background process. This includes applying metadata from other data sources, moving documents, or setting up filing structures. Thanks to this flexibility, a number of other functions can also be created via the integrated JavaScript interface.

Forming the basis for processing, a ruleset consists of an XML configuration created via a graphical user interface in the ELO Administration Console. Multiple rulesets can be defined, which are executed in sequence with their own interval controls ("Every 10 minutes", "Daily at 1 PM", "Every 3rd day of the month"). Furthermore, the ruleset contains a search query and a sequence of rules for processing data.

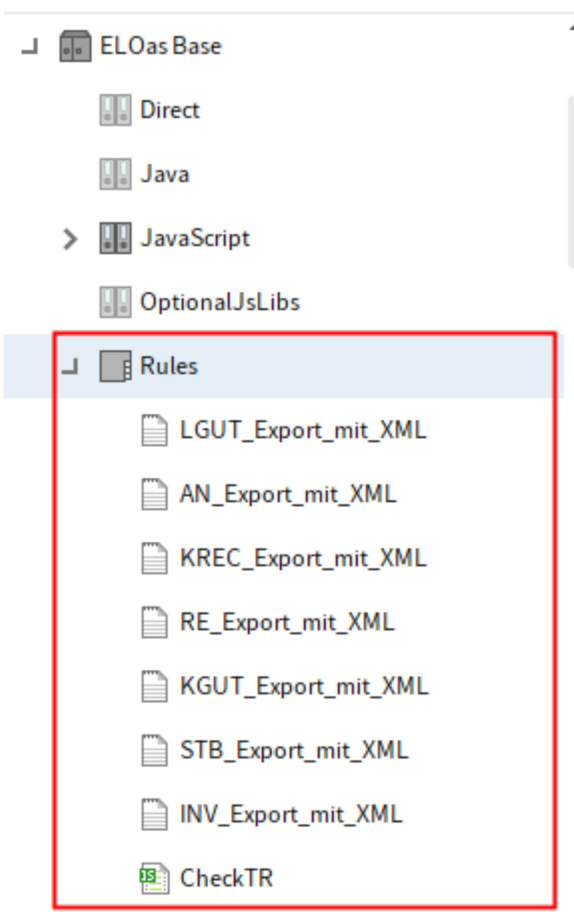


Fig.: Rules folder in ELO

ELOas activates every ruleset in its list within "ELOas\Rules" in a rotating sequence. For each ruleset, ELOas first checks whether the interval condition is met (has it been at least 10 minutes since it last was run?). If it isn't met, the next ruleset is processed. If, however, the execution time

has been reached, the specified search is performed. The ruleset now runs for each entry in the results list. You can change the target in ELO, enter metadata, or perform other actions here. Next, the document is saved and the following entry is processed until the end of the list of results has been reached. Finally, the new execution time is calculated and the server processes the next ruleset.

Additional rulesets can easily be added via the graphical user interface. Just as with changed rulesets, they only become active once the configuration is reloaded.

The XML configuration of the rules and the JavaScript code can be saved in a document file instead of on the Extra text tab of the metadata. In this case, you will see text files instead of folders for child entries in the tree.

An additional type of ruleset was introduced for ELOwf: the direct function call. These rulesets are created in their own folder called *Direct*, run in their own thread, and return a direct result. For this reason, they must not be defined with an interval, but must rather be created with a trigger (0 minutes: use 0M as interval). Additionally, you should only execute short actions here, as the called process has to wait for a result and cancels it after a specified timeout.

Search methods (index search, treewalk, task list, mailbox, timestamp)

ELOas is mainly designed to process a list of results from an index search. Over time, additional options have been added, which can be selected by naming the SEARCHNAME correspondingly.

TREEWALK: The object ID or ARCPATH to the start object is configured in SEARCHVALUE. It runs through the entire branch, and the ruleset is called for each entry with the corresponding metadata form.

WORKFLOW: All of the ELOas user's workflow tasks are read and the ruleset is called for each entry with the corresponding metadata form. The ruleset can also forward the workflow.

MAILBOX_<Profile name>: A connection to the e-mail server is established using the profile name, and the mailbox contents are read and processed. The ruleset is called for each e-mail message in the mailbox with an empty document.

DIRECT: This ruleset can be called via `http-get` and returns a direct result. Rulesets of this type can only be defined in the *Direct* folder and not in *Rules*, as they have to be executed in another thread.

TIMESTAMP: This call performs a search following the last change. Normally, you enter a range as the search term: "2012.01.01.00.0.00... 2012.01.31.23.59.59".

Rulesets

Basics

The program executes rules at regular intervals to carry out defined tasks in ELO.

Rules are written in JavaScript and are executed by the Indexserver at specified intervals.

Rule processing

1. The program first searches for objects in ELO that match the rule definition.
2. The rule is applied to all elements found in ELO.

> **Information**
>
> The setting for the documents to be searched is defined in the search fields.

1. If a rule does not match, it cannot be applied or executed. An error message appears.

   The defined rules are filed in ELO under *Administration > Rules*.

> **Information**
>
> Rules are split into two parts in the ELO Administration Console. In the *Rules* area, you will find the box *Rule 1*, where the steps to be executed are described. It contains information about what to do with each object found. The *Global error rule* box defines what should happen if errors occur.

Create ruleset via ELO Administration Console

In the ELO Administration Console, you create new rulesets in the *ELO Automation Services* area.

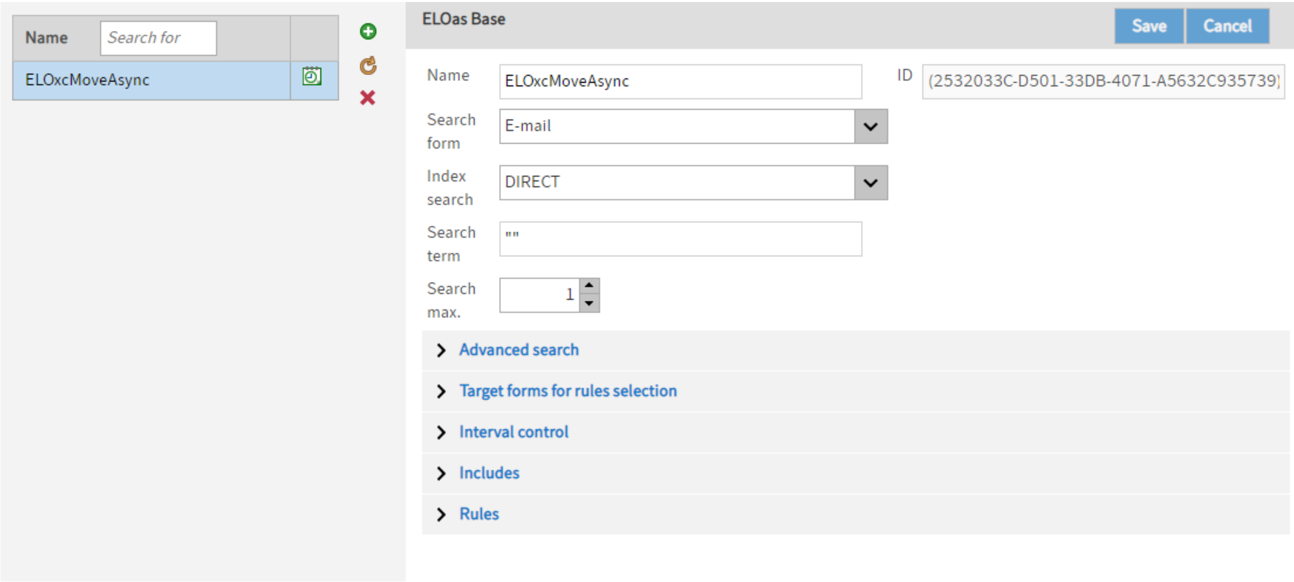


Fig.: 'ELO Automation Services' menu item

Add (green plus icon): Click the *Add* button to create a new ruleset.

Information

Once you have saved it, the new ruleset is stored in the *Rules* folder of ELO. You can configure the rulesets in the *Rules* area.

Reload data from server (yellow circle arrow icon): Click the *Reload data from server* button to reload the area.

Delete (red X icon): Click the *Delete* button to delete the selected ruleset.

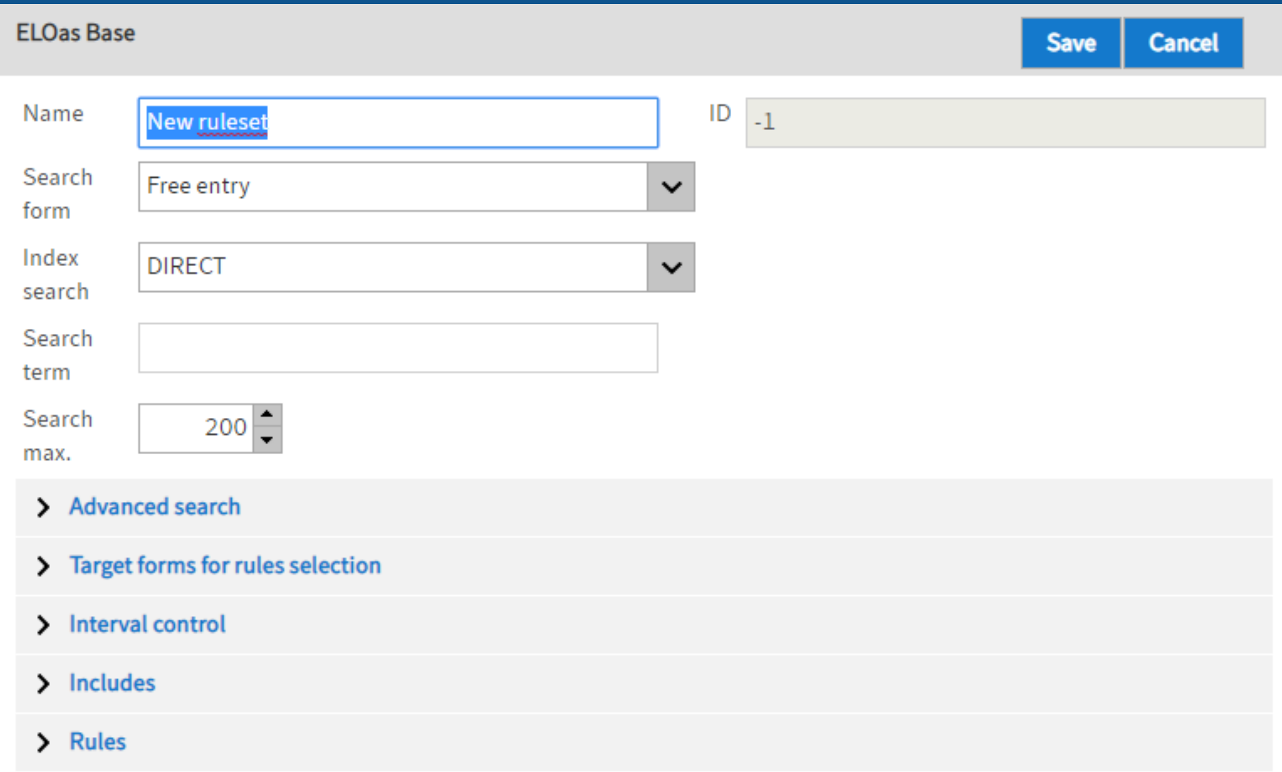


Fig.: New ruleset

Name: The name of the ruleset that you entered when you created the rule is shown here. The name can be modified later.

> **Please note**
>
> Not all characters are allowed. Refer to the following list.

- ¶
- "
- /
- \
- :
- ;
- ,

Search form: Select the search form that will be used to find documents to be processed.

Index search: Select a group field to search across different fields.

Search term: Enter the character string you want to search for here. All documents in ELO that correspond to the defined rules will be selected in accordance with the rules and criteria defined in the wizard. The character string must be entered in quotation marks.

Search max: Enter the maximum number of search results here.

Advanced search

From filing date ... to: You can narrow down the search here by selecting a specific filing date or a filing period.

From date ... to: You can narrow down the search here by selecting a specific date or period.

Target forms for rules selection

Add target form: This is where you can select the metadata form for the target folder that moved documents are filed to.

Interval control

In the *Interval control* area, you define how often you want to run ELO Automation Services.

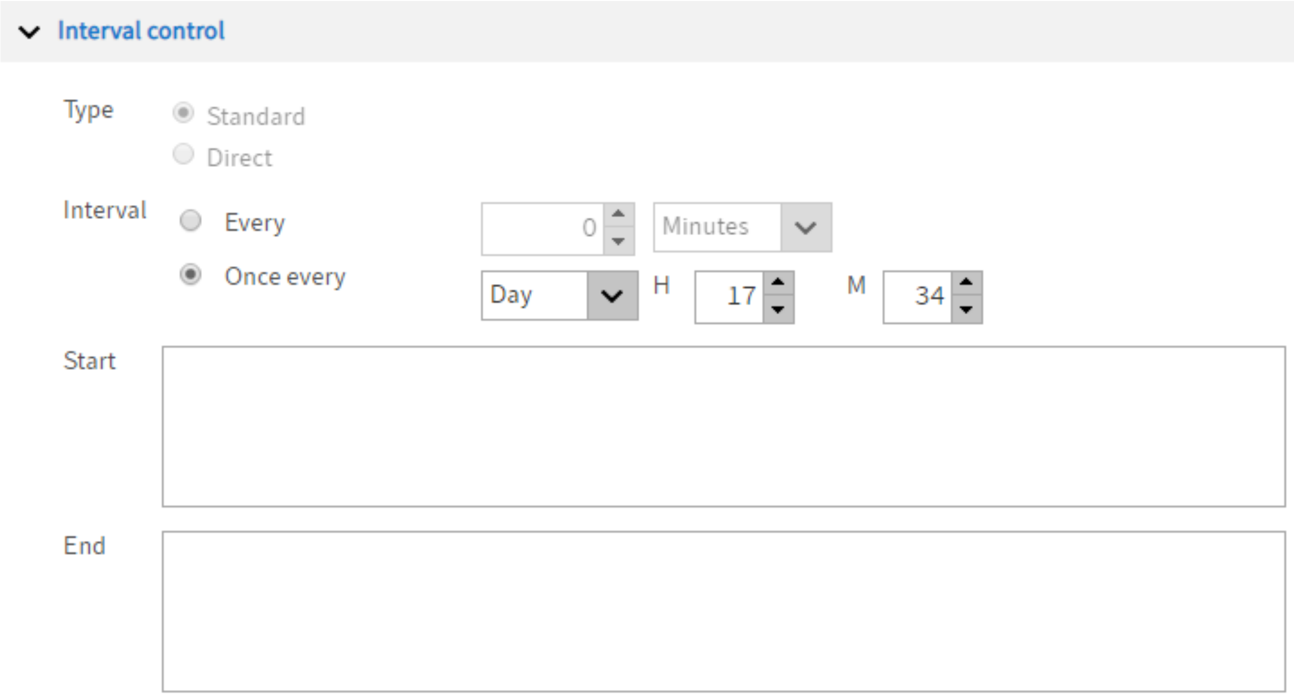


Fig.: 'Interval control' area

Start: This input box contains script code that will be executed before running the ruleset.

End: This box contains script code that will be executed after running the ruleset.

Includes

Add Include library: In the *Add Include library* field, you can add any script libraries that you need to the ELOas rule.

Rules

Fig.: 'Rules' area

This rule will be applied to all entries found in ELO. This is where you define settings such as where documents are moved.

Add (green plus icon): Add a new rule. The rules will be processed in sequence.

Name: Enter a name for the rule.

Condition: The query rule is defined here to check the status of a field, for example.

In this menu, you can select a script, e.g. to move files to the file system. Scripts are filed to the administration area of ELO Automation Services in ELO.

> **Please note**
>
> The configuration in your repository can differ from the illustrations shown here.

Filing path: Specify where you want to file the document to. Use the button at the end of the input field to enter separator characters for paths in ELO.

Target form: Select the metadata form for the document.

Fields: You can replace the content of fields in the metadata of the target documents here.

> **Information**
>
> If you switch to the *Script* tab, you make all the rule settings using a script. If you do so, you can no longer access the *Rule* tab.

Error handling

You can define the basic settings for handling errors in the *Global Error Rule* area. This rule is executed when an error occurs in a general rule.

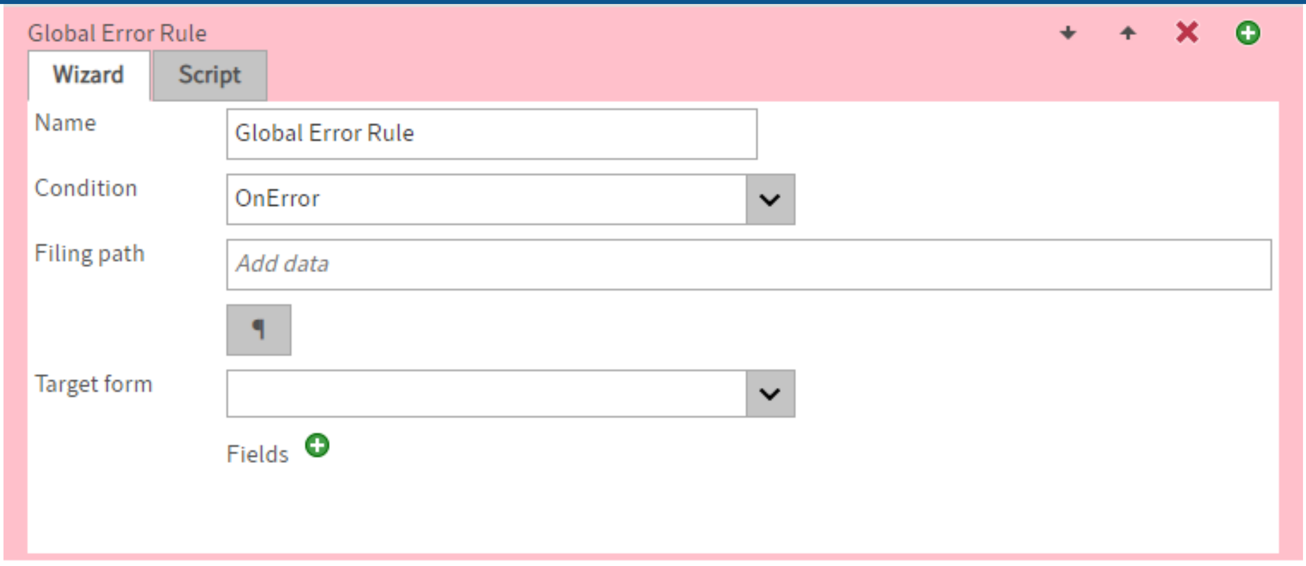


Fig.: Defining basic error handling

Name: The name of the error handling routine is entered here.

Condition: A condition for an error rule is defined here.

Filing path: Set the filing path of the error report in the repository here.

Target form: Select the metadata form of the document with the error log.

Fields: This option enables you to define individual fields that you can assign a specific character string on filing.

> **Please note**
>
> If a ruleset is invalid, it cannot be saved. The validity of a ruleset is verified when saving.

Options and error handling

Pause rules

You can trigger and stop individual rulesets via a link in the browser.

You can access the ELO Automation Services status page via the respective ELO Application Server manager or via the URL with the following structure:

```
http(s)://<server name>:<port>/as-<repository name>/?cmd=status
```

ELO Automation Services status report, Version 20.00.000 Build

No active ruleset, pausing					
Excecuted	**Name**	**Next run**	**Run**	**Action**	**Status**
154	Move newsletter	2020-04-06 11:34:31.295	Stop	Reload	Idle...
Direct Pool					1 / 1
Reload all					

Fig.: Deactivate a rule with a stop link

If the user clicks *Stop*, the running process is stopped for the corresponding ruleset. Click *Start* to restart the process.

Please note

The status page updates automatically every 10 seconds. After clicking *Start* or *Stop*, this command will be sent every 10 seconds. In normal operation, this does not cause any problems. However, if you have the same status page open in multiple browser windows, you may encounter problems or unexpected behavior if you enter different settings to the different windows.

Error message "Invalid Ruleset"

Problem 1

ELO Automation Services (ELOas) runs but the following error message is displayed in the log file: Invalid Ruleset suspended: org.xml.sax.SAXException: Invalid ruleset definition: Premature end of file.

Even though the rules are defined correctly (correct spelling, no error while parsing the xml document in the browser), the file cannot be read by ELO Automation Services, i.e. the request takes place but an empty document is triggered (as can be seen by the empty lines of the log file) and an error message appears. Check whether the ELOas user has sufficient rights. Check whether the ELO Indexserver and Document Manager of the repository in question have errors in their log files. You can test the rule with another client. Instead of saving the script in a .txt file in the *Rules* folder, save it as a separate folder in *Rules* (in the extra text of the folder) so that the script is regarded as a database entry.

Another approach is to check whether the latest version of ELO Automation Services is in use and whether the following entry exists in the XML file under *<installation path>\config\as-<repository name><name of server instance>\config.xml*:

```
<entry key="tempdir"> ... </entry>
```

This temporary directory must exist, as otherwise an error message will appear. The user must have write access to the temp directory at the system level.

Problem 2

When creating a second rule, not every group field is available in the drop-down menu on the first attempt.

In this case, you need to close and start the ELO Administration Console again. Warning: This does not apply for the first rule you created!

Manual start of a ruleset

Normally, ELOas executes the defined rulesets based on intervals. However, there are processes that are so complex in their execution that they cannot be run in short intervals. Still, they must become active as quickly as possible after a certain change has taken place. There is an option to manually execute a ruleset through a URL (or rather, via script).

If you execute rules via an "HTTP-GET" or "HTTP-RUN" command in ELOas 20, they need to be validated with a ticket. You need to attach a valid ticket to the corresponding ELOas URL, e.g:

http://localhost:9060/ELOas/actions?cmd=get&name=test&ticket=935666A2E27D8AB642C4C40AFAEAE2B9

You can turn off the internal ticket validation with the new ELOas configuration parameter `checkTicket`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Webapp properties</comment>
<entry key="password">elo</entry>
<entry key="tempdir">C:\ELOenterprise20\temp\as-LocalArchive\ELO-PCVLADOVM-3</entry>
<entry key="language">de</entry>
<entry key="user">ELO Service</entry>
<entry key="rootguid">(F6C173D7-3F71-4559-91E5-4886139B12CF)</entry>
<entry key="url">http://PCVLADOVM:9090/ix-LocalArchive/ix</entry>
<entry key="checkTicket">false</entry>
</properties>
```

Fig.: config.xml

Important

Using ELOas in proxy mode with disabled ticket validation poses a security risk, especially if the ELO Indexserver is available on the Internet.

Example

The following example shows how to call a ruleset from a client script, which then changes specific objects.

Warning: As the call takes place via http access, any user can trigger this action from the browser or via a script command. For this reason, you need to ensure that the function cannot be misused (such as by verifying the user number or through a fixed internal preset of the object ID).

First, the ruleset in use must be considered. By entering an interval of 0 minutes (`<interval>0H</interval>`), this ruleset will be defined as triggered manually. Thus, it will not be called cyclically, but will rather wait until a specific URL is received.

```
<ruleset>
    <base>
        <name>Expand Name</name>
        <search>
            <name>"OBJIDS"</name>
            <value></value>
            <mask>2</mask>
            <max>200</max>
        </search>
        <interval>0H</interval>
    </base>
<rule>
    <name>Expand Name</name>
    <condition></condition>
    <script>
        log.debug("Param1: " + EM_PARAM1);


        log.debug("UserId: " + EM_USERID);


        NAME = "Approved: " + NAME;


        EM_WRITE_CHANGED = true;

    </script>
</rule>
<rule>
    <name>Global Error Rule</name>
    <condition>OnError</condition>
    <script></script>
</rule>
</ruleset>
```

The interesting part is found in the script area:

```
<script>
    log.debug("Param1: " + EM_PARAM1);


    log.debug("UserId: " + EM_USERID);
```

The call can provide up to three parameters. These can be queried from the ruleset using the variables `EM_PARAM1`, `EM_PARAM2`, and `EM_PARAM3`. In addition, the script can optionally provide the ticket of the current login for authentication. In this case, the number of the logged on user is entered to the variable `EM_USERID`. If no user has been authenticated, the number -1 is entered. In the first parameter, one or more object IDs can be transferred. These will then overwrite the search value from the ruleset definition. In this case, "OBJIDS" must be specified as the name for the metadata field.

```
NAME = "Approved: " + NAME;
```

In the example, the short name of the selected object is preceded with the text "Approved". Any other changes to the SORD object can be made here.

```
EM_WRITE_CHANGED = true;
```

As the object has been changed, it should also be saved.

```
</script>
```

Activating the ruleset

After starting ELOas, this ruleset is started, but not yet active. It waits for an external trigger (visible by the text "Trigger" in the *Next run* field).

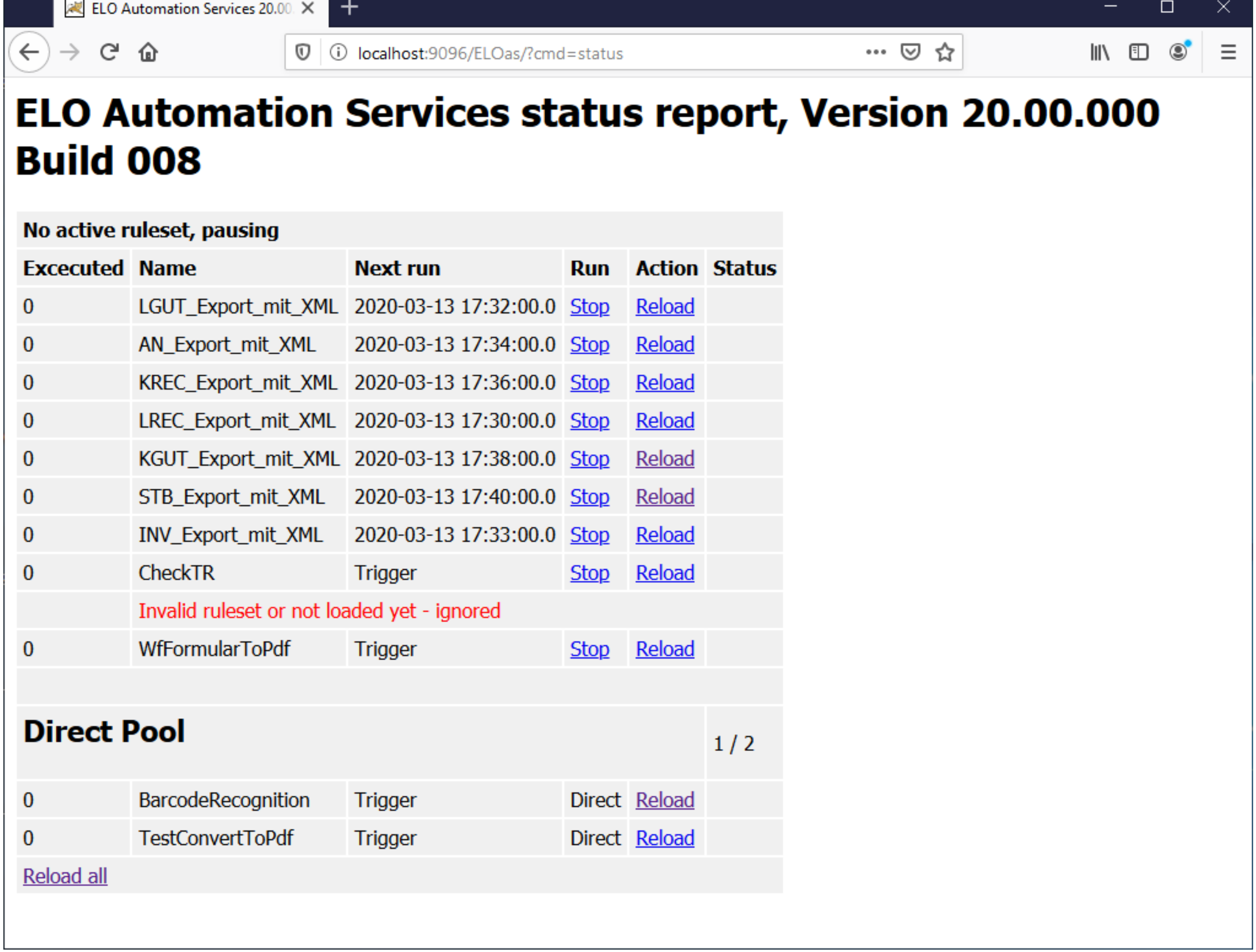


Fig.: ELOas status page

The trigger is either initiated from a URL, or from the Windows Client by using a script command (starting with version 7.00.056 of the client):

```
SendELOasRequest( <server name>, <port number>, <service name>, <with ticket>, <ruleset name>, <
```

The `SendELOasRequest` command performs an asynchronous call with `run`. Such rulesets are shown in the ELO Administration Console under *Rules*, instead of *Direct*.

Server name	Name or IP address of the ELOas server.
Port number	Port number of the ELOas server. Normally 8080, standard http port.
Service name	Service name of the ELOas server. In a standard installation, it is created by combining the prefix "as-" and the repository name (e.g. as-ELO). However, make sure to use the correct capitalization, as otherwise the Tomcat server will return an error.
With ticket	0. Do not send logon information

	1: Send current ticket as logon information. In this case, ELOas checks the ticket and identifies the user number. This information is provided to the ruleset. The ruleset can then decide whether and to what extent the action will be run. The logon information for SSO cannot be evaluated at present. This will be changed in the next version of the ELO Indexserver.
Ruleset Name	Name of the ruleset to be run. Only triggered rulesets can be called in this way. The call is ignored for interval-controlled rulesets.
Parameter1	First parameter. If this parameter is not empty, it is used as a search term when the ruleset is run.
Parameter2, Parameter3	Additional optional parameters. These can be queried by the ruleset and control how it is run.

The complete sample script for such a call could therefore look like the following. It calls the *Expand Name* ruleset for the objects with ObjId 7944 and 7945.

```
Set Elo=CreateObject("Elo.Professional")
MsgBox Elo.SendELOasRequest("localhost", 8084, "/ELOmover/as" , 1, "Expand Name", "7944,7945", "
```

The ruleset can also be triggered from any other application by calling a URL:

```
http://localhost:8084/ELOmover/as?
cmd=run&amp;name=Expand%20Name&amp;param1=7944,7945&amp;param2=TestParam2
```

> **Please note**
>
> In this case, you cannot transfer any authentication information. Ensure in your ruleset that the action cannot be misused.

Other notes

Triggering rulesets asynchronously

When a ruleset is triggered by a URL or a script call, ELOas runs it asynchronously. Thus, if another ruleset is currently active, script execution will not be delayed for as long as it takes for ELOas to become available again. Instead, the activation command is placed in a queue and then run at the next opportunity.

This has two consequences: first, the client script cannot assume that the operation has actually been performed just because the command has been processed. If this is important for the further course of the script, it must be checked by the script itself and integrated into the queue. However, please note that a situation may occur where ELOas is also processing other very complex actions at the same time. Generally, a script should therefore not wait for the completion of an ELOas action.

It is also possible that an impatient user may initiate the trigger multiple times. In this case, the ruleset will also be run multiple times. For this reason, it is necessary to ensure that repeat triggering does not lead to errors, such as by checking the object in advance and then canceling any repeated runs.

Triggering rulesets asynchronously also leads to another problem: Errors that occurred while processing the ruleset cannot be reported using the script call.

Triggering rulesets synchronously

With synchronous triggering, the ruleset is initiated directly and can also return a result. Synchronous triggering is used primarily by ELO workflows (form editor). With the call, `cmd=get` is required instead of `cmd=run`. In addition, the rulesets for the synchronous call must not be placed in the *Rules* folder, but rather in the *Direct* folder. Synchronous rulesets are run independently from the asynchronous ruleset in their own thread.

Permissions check

When being called from the ELO Windows Client, the client authentication ticket can be optionally transmitted as well. In this case, ELOas can check the login and ascertain the current user. For critical actions, a test to determine whether it is being run by a user with sufficient rights must be performed. If no user is logged in or the user does not have sufficient rights, execution should be canceled.

However, in certain cases, anonymous triggering may be completely acceptable, such as when a specific predefined object is being edited. This is the case, for example, when a specific predefined object is being edited. In this case, care should be taken to ensure the ObjectId cannot be changed by the call. This can most easily take place in an `onstart` event by setting the value `EM_SEARCHVALUE` in the script. In this case, the preset value from the ruleset script is used for the search instead of the parameter.

Order of operations

A manually triggered ruleset inserts itself quite normally to the order of operations for the rulesets. If multiple triggers have been activated for a ruleset, all triggers are processed first before the next ruleset is processed.

The rule structure

This section describes the XML rule structure in ELOas. Normally, this structure is maintained using a graphical user interface or GUI. If you have to make manual changes, you can use this description as a reference. At the same time, this description serves as a reference for implementing the GUI.

General structure

The complete structure is embedded in the <ruleset> tag. This consists of two parts: a <base> entry at the beginning, followed by any number of <rule> entries.

The <base> entry contains the information to search for the entries to be processed. These include the search rows, the search term, forms, and date restrictions.

The <rule> entries contain one processing instruction each. You can assign each rule a condition, change the filing target, or change the contents of the fields. Additionally, a rule can also have JavaScript contents. If defined as such, the other entries are ignored, but they can have values.

If the condition of a rule is "OnError", this rule is processed as an error handling rule. An error handling process can take place after every rule, and at the very end an error handling rule must be entered. These final error rules are called if an error occurs when moving or saving a file. If an error occurs during processing within a normal rule, the next possible error handling rule is called and processing is canceled.

Example of a simple ruleset:

```
<ruleset>
    <base>
        <name>Name of the ruleset</name>
        <search>
            <name>Metadata field name in JavaScript code</name>
            <value>Search term in JavaScript code.</value>
            <mask>Number of the metadata form for the search.</mask>
        </search>
        <interval>5M</interval>
    </base>
    <rule>
        <name>Name of the rule</name>
        <destination mask="Folder form"> New target in JavaScript Code</destination>
        <index>
            <name>Metadata field name in JavaScript code</name>
            <value>New contents of the metadata field in JavaScript code</value>
        </index>
    </rule>
    <rule>
```

```
        <name>Name of the error handling rule</name>
        <condition>OnError</condition>
    </rule>
</ruleset>
```

All entries in the <base> section

Tag	Function	Example
name	Name of the ruleset. This name is displayed on the status page, but is not processed further.	SAP processing
search	Parameter for searching for the documents to be processed. For a description, see the following section *All entries in the '<search>' section*	
masks	If you have to switch to another metadata form during the course of processing, all possible target form (mask) numbers have to be listed here. Each form number is framed with a `<mask>` tag.	`<mask>3</mask><mask>4</mask>`
interval	Repetition interval for processing the search. This interval can be entered in minutes (5M) or hours (1H). Further, it can also be run once a day at a specific time (15:30), once per week (17:20/SA), or once a month (22:00/31). If a day is specified for monthly execution that does not exist for the current month (e.g. the 31st of February), the last day of the month is used instead.	5M1H15:3017:20/SA22:00/31

All entries in the '<search>' section

The entries in the *<search>* section determine which documents are processed. At the start of each pass, a search is performed with these parameters. The list of results is processed according to the rules.

```
<search>
<name>Metadata field name in JavaScript code</name>
<value>Search term in JavaScript code</value>
<mask>Number of the metadata form for the search.</mask>
<max>Maximum number of documents per pass</max>
</search>
```

Day	Function	Example
name	Metadata field name in JavaScript code If the name is fixed, text can be entered directly in quotation marks. However, any JavaScript expression can be used as well.	"ELOOUTL2"
value	Search term in JavaScript code. If the value is fixed, text can be entered directly in quotation marks. However, any JavaScript expression can be used as well.	"ELO*"

Day	Function	Example
mask	Number of the metadata form for the search. Only a metadata form can be used here, and not a pure search form, as it is assumed that all matches have the same form definition during read-in.	2
max	Maximum number of documents per pass sending a search query to the ELO Indexserver. If more matches exist, they are processed in a later pass after all other rulesets have been run. This prevents an extensive ruleset from suppressing the processing of all other rulesets. A maximum of 1000 documents per pass are allowed.	200
`idatexdate`	The list of results can be restricted through a date range in the filing date (`idate`) or document date (`xdate`). This date can either be entered in absolute values in the ISO date format (YYYYMMDD) or in values relative to the current day (-5). The range consists of a start date in a `<from>` tag, and an end date in a `<to>` tag.	`<idate><from>-5</from><to>+0</to></idate>`

All entries in the '<rule>' section

After the `<base>` section, any number of `<rule\>` sections can follow. These are run in the order of the definition during processing.

A rule can exist in two different forms: as a normal rule and as an error rule. Such an error rule is simply skipped in the normal course of events. The next available error rule is only called in case of an error, and afterwards the processing of this document is canceled. This means that after an error rule is processed, no further rules will be processed.

The last rule in the `<rule\>` chain must always be an error rule. This ensures that error handling is always available in every case. Additionally, this rule is called if an error occurs while moving or saving a file.

Tag	Function	Example
name	Name of the rule, only be used for documentation and to better understand its function.	Additional indexing
condition	Processing condition for the rule. If this is an error rule, the fixed text "OnError" is entered here. Note that it must be written exactly in this way, as otherwise the rule will not be recognized as an error rule.	KDNR == "123"
	The execution condition is provided in the form of JavaScript code. The rule is only run if the condition is "true".	

Tag	Function	Example
destination	New filing target of the document as the repository path. This entry is optional and can remain empty. In this case, the document remains in its original position. If there are multiple destination rules, the first target is used as a new filing location, and all additional targets are entered as references to the first.	`<destination mask="1"> ¶ELO¶Mails¶" + ELOOUTL1</ destination>`
	If a filing target does not yet exist, it is created automatically.	
	The destination tag can also contain an additional "mask" attribute with the number of the folder metadata form for newly created folders. If this attribute is not available, "1" is used by default, which is the number of the folder form in a standard repository.	
mask	New document metadata form If this entry is not available or the form number is -1, the original metadata form is retained.	`<mask>20</mask>`
	If the form is changed, all entries are automatically applied with the same group name. This is also executed correctly if the metadata fields are divided up differently between forms.	
	If the original metadata form contained fields that the new form does not have, this data is automatically discarded without returning an error message.	
	ELOas cannot process documents with metadata forms that use the same group names for multiple entries, as the internal processing and structure of the rules assume a unique assignment.	
index	Any number of index entries can exist within a rule Each index entry contains the name of the relevant field and a JavaScript expression with the new value.	`<index><name>DOCDATE</ name> <value>"20070930"</ value></index>`
	Fields with an ISO date and fields for the filing and document date also require the in ISO date format.	
	In addition to the fields with the group names of the search form, all group names of the alternative metadata forms are available, as well as a number of pseudo-fields with standard values for metadata:	
	NAME: Short name	
	DOCDATE: Document date	
	ABLDATE: Filing date	
	ARCHIVINGMODE: Document status 0, 1, or 2 for "Version control disabled", "Version control enabled", or "Non-modifiable".	

Tag	Function	Example
	ACL: With "PARENT", apply the ACL of the new filing target. With `<rights>:<name>`, define any number of group rights. OBJCOLOR: Color number of the entry OBJDESC: Extra text OBJTYPE: Document or folder type of the entry. Information: An incorrect assignment can lead to disruptions in further processing. Documents can only have an OBJTYPE between 254 and 286.	
script	A rule can also contain JavaScript code to be run. In this case, all other parameters of this rule are ignored, but they can be retained, e.g. for documentation purposes.	

Changing permissions

Changed permissions can be configured in the ACL pseudo-metadata field. In the simplest case, enter "PARENT" here, which will then apply the rights of the target folder to this entry when it is saved. However, a complete list of rights can also be configured here. This list consists of a sequence of individual rights that are separated by a pilcrow symbol. Each individual right consists of the rights form (RWDELP - read, write, delete, edit, list, permissions), followed by a colon and the group name. For AND groups, enter a sequence of names, each separated by a colon, instead of the simple group name.

```
R:Everyone¶RW:Controlling¶RWDELP:Administration:Stuttgart:Management
```

In the example, the *Everyone* group has read access, the *Controlling* group has read and write access, and the AND group *Administration and Stuttgart and Management* has full access to the document.

If you want to set rights for a user instead of for a group, add "U" to the list of rights as well.

```
UR:Administrator
```

Notes

When generating the JavaScript code, all group names of the search form and the alternative metadata forms are entered as variables in all caps. This method minimizes the risk of group names overlapping with standard identifiers from JavaScript or the ELO runtime environment. In principle, however, it can lead to problems if one of the group names is identical to a standard identifier or one of the translation lists.

```
var NAME;
var ARCDATE;
var DOCDATE;
var OBJCOLOR;
var OBJDESC;
var OBJTYPE;
var ARCHIVINGMODE;
var ACL;
var EM_PARENT_ID;
var EM_PARENT_ACL;
var EM_SEARCHNAME;
var EM_SEARCHVALUE;
var EM_SEARCHCOUNT;
var EM_SEARCHMASK;
var EM_IDATEFROM;
var EM_IDATETO;
var EM_XDATEFROM;
var EM_XDATETO;
var EM_FOLDERMASK = "1";
```

Information

This list may be expanded in the course of project process. In particular, it can be expanded with additional entries through local customizing.

The number of the metadata form for the current document can be changed using a rule. However, if this results in an invalid form number or a number that does not exist in the list of alternative target forms, this will cause a runtime error when saving the document, and not when assigning the form.

If an error rule is called due to a runtime error, it will delete all already allocated filing targets of the previously processed rules. If the error rule does not have its own `<destination>`, the document remains at its original position. Otherwise, the target of the error rule is used.

Changed metadata is moved and saved at the end, after the last rule is processed. If this leads to an error, the last error rule is called, and not the error rule that belongs to the rule defining the target (which is, indeed, identical to what occurs when there is only a single error rule).

Sample structure

The following provides a sample definition, along with a list of the code generated from it. This information is for orientation purposes only.

```
<ruleset>
    <base>
```

```
        <name>Thiele e-mail form</name>

        <search>
            <name>"ELOOUTL2"</name>
            <value>"Thiele*"</value>
            <mask>2</mask>
            <max>2</max>
            <idate>
                <from>"-35"</from>

                <to>"+1"</to>
            </idate>
        </search>
        <masks>
            <mask>12</mask>
            <mask>13</mask>
            <mask>20</mask>
        </masks>
        <interval>1M</interval>
    </base>
    <rule>
        <name>Rule 1</name>
        <destination mask="5">"¶Thiele¶E-mails¶" + ELOOUTL1</destination>

        <mask>20</mask>
        <index>
            <name>ADDENTRY</name>
            <value>getObjShort(2)</value>
        </index>
        <index>
            <name>ELOOUTL2</name>
            <value>"!!" + ELOOUTL2</value>
        </index>
        <index>
            <name>DOCDATE</name>
            <value>"20070930"</value>
        </index>
        <index>
            <name>ARCHIVINGMODE</name>
            <value>2</value>
        </index>
        <index>
            <name>ACL</name>
            <value>"PARENT"</value>
        </index>
    </rule>
```

```
    <rule>
        <name>Journal copy</name>
        <destination mask="1">"¶Thiele¶Journals¶" + ELOOUTL1</destination>
    </rule>
    <rule>
        <name>Script rule</name>
        <script>
            moveTo(Sord, "¶Repository¶Targets1¶" + ELOOUTL1);
            moveTo(Sord, "¶Repository¶Targets2¶" + ELOOUTL2);
            moveTo(Sord, "¶Repository¶Targets3¶" + ELOOUTL3);
        </script>
    </rule>
    <rule>
        <name>Global Error Rule</name>
        <condition>OnError</condition>
        <destination>"¶Thiele¶Error"</destination>
        <index>
            <name>ELOOUTL2</name>
            <value>"!!" + ELOOUTL2</value>
        </index>
        <index>
            <name>ARCHIVINGMODE</name>
            <value>0</value>
        </index>
    </rule>
</ruleset>
```

Programming

Programming with ELO Automation Services

The chapter titled "Programming with ELO Automation Services" (ELOas) describes how to set up and use the JavaScript Runtime Environment. This module enables you to run additional functions in ELOas that are not available in the basic version.

Script execution

The XML configuration of the ruleset is not only interpreted by ELOas. It is also translated when imported to a JavaScript program and combined with the basic routines that are also available in JavaScript. This script will then be run later. This has various advantages:

The assignments in the XML configuration can contain the whole range of JavaScript expressions with any kind of function calls.

Any kind of JavaScript code sections with complex routines can be embedded in the XML configuration.

The basic routines can be extended with all types of functions. These can then also be used by administrators without programming knowledge, by simply calling the function within an expression. Take the DB Access and Document Export modules for example.

The extended basic routines can also use any external Java libraries to increase the range of functions (such as JDBC drivers, or even the IX client to directly control the ELO Indexserver).

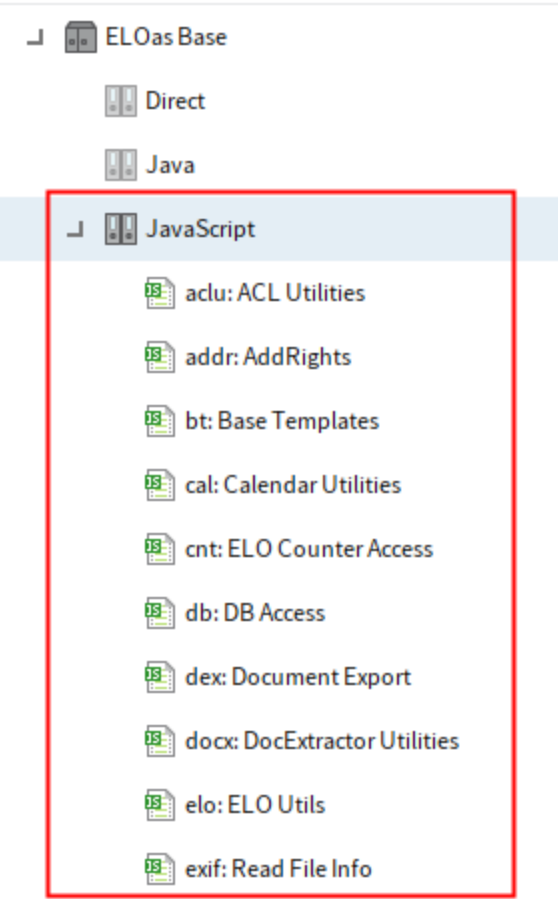


Fig.: 'JavaScript' directory

The major advantage of the basic functions in JavaScript is that these functions can be customized or (preferably) added to within the project, without requiring ELOas itself to be changed. Thus, you can work with a standard program, but adapt it to your requirements.

The basic installed version of ELOas includes the necessary basic functions to execute searches and process rules (base templates, imports, and ELO utilities). This part should normally remain unchanged. Only in special cases does it make sense to make changes here. Furthermore, it has two modules for database access (DB Access) and exporting document files (Document Export). In future versions, additional modules will be available. We also plan to set up a kind of online exchange in the SupportWeb for ELOas modules for business partners.

In order to run such modules without any conflicts, a namespace concept has been developed, which assigns each module its own namespace. Namespaces must always be written in lowercase, as this could otherwise lead to conflicts with group names from the form definitions. All 2- and 3-digit namespace names are reserved for ELO and are used for standard modules and released add-ons. For custom modules, partners can use namespace names of four or more digits. If you create a module that you only want to implement in one project, you can also use a one-digit name. The module name in ELO must start with the namespace name, followed by a colon and a short description (e.g. `dex: Document Export`). Internally, the namespace is implemented in a way that a JavaScript object is created with the name of the namespace, and then all required functions of the

module are assigned to this object. As this is ultimately a list, the individual functions are separated with a comma instead of a semicolon.

```
var dex = new Object();
dex = {
    command1: function(x,y) {



        …
    },

    command2: function() {



        …
    }
}
```

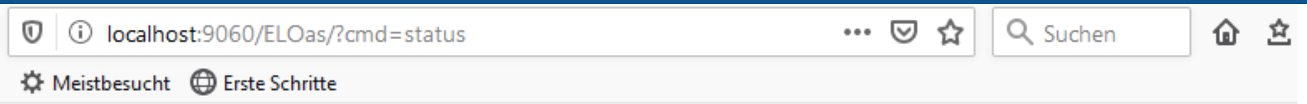
These functions can then be addressed by the JavaScript code with `dex.command1(x,y)` or with `dex.command2()`. As every module has its own unique identifier, these can be combined without the possibility of naming conflicts.

The *Imports* module has a special position among the basic modules. It is always placed at the very start of the chain in the JavaScript program. This is therefore where the required Java library imports should be placed. You can also configure global variables that are of general interest here. As this module is a global module, it does not have a namespace.

Creating custom modules

New custom modules can be created by the administrator by simply creating a new folder with the name of the module within the *ELOas\JavaScript* folder. The actual JavaScript code is entered to the folder's Extra text tab. Using permission controls in ELO, individual modules can be also enabled and disabled by setting an ACL, which controls access for the ELOas account.

In each case, newly created or released modules only become active once the service has been restarted or refreshed.



ELO Automation Services status report, Version 20.00.000 Build 005

No active ruleset, pausing					
Excecuted	**Name**	**Next run**	**Run**	**Action**	**Status**
0	DatevExportRule	Trigger	Stop	Reload	
2	FesteWerteKachel	2020-01-21 09:37:10.843	Stop	Reload	Idle...
2	Freie Eingabe	2020-01-21 09:37:10.843	Stop	Reload	Idle...
1	NotifyWf	2020-01-21 09:45:10.102	Stop	Reload	Idle...
2	PLANDATEN_AUTO_VS	2020-01-21 09:37:10.843	Stop	Reload	Idle...
2	RegExpExample	2020-01-21 09:37:10.843	Stop	Reload	Idle...
2	SendMail	2020-01-21 09:37:10.843	Stop	Reload	Idle...
2	TestIsoDate	2020-01-21 09:37:10.843	Stop	Reload	Idle...
0	TestSaveTiffAsPdf	Trigger	Stop	Reload	
2	TileExample	2020-01-21 09:37:10.843	Stop	Reload	Idle...
Direct Pool					1 / 2
0	CreateStdAsLibs	Trigger	Direct	Reload	
0	CreateStdAsLibsEN	Trigger	Direct	Reload	
0	TestActivateAsposeLicense	Trigger	Direct	Reload	
0	TestAsString	Trigger	Direct	Reload	
0	TestCallSignature	Trigger	Direct	Reload	
0	TestCanChangePermissions	Trigger	Direct	Reload	
0	TestConvertEmlToPdf	Trigger	Direct	Reload	

Fig.: ELOas status page

Custom modules can contain any number of functions or global variables. As all modules must be executed together within a JavaScript context, however, it is important to watch out for possible namespace conflicts when naming them. Unfortunately, such conflicts will not be seen as errors by the JavaScript interpreter and can therefore not be recognized automatically.

The objects of the custom module have an unlimited lifetime. After they are created, they remain active until the service is ended or refreshed. This can be very problematic in some cases, such as

with database connections. If a persistent connection is created at program start or first run and then remains active for an unlimited time, this can lead to limited resources becoming reserved for unnecessarily long times (such as when the ruleset only becomes active once a month). Or, even worse, the resource could become invalid (e.g. due to a database server restart). Recognizing an invalid service condition and initiating an automatic reconnect requires significant processor resources. This problem can be significantly moderated by only connecting such resources as needed, and by automatically releasing them at the end of the ruleset (see also the following chapter section *Lazy initialization*). To do this, every module must implement a function with a special name: `<namespace>ExitRuleset` (e.g. `dexExitRuleset`). After a ruleset has finished processing, this special function is invoked for each module. The script calls for deactivating the connection can be configured in this function.

Lazy initialization

If all external resources must immediately be connected and then disconnected at the end every time a ruleset is run, this can lead to significant unnecessary resource use. If a ruleset needs to react quickly and thus run once a minute, in many cases not a single active data set will be available for processing. Thus, unnecessary connections will frequently be created and then disconnected. For this reason, external resources should always be connected via "Lazy initialization". In this case, the connection will not be created at the same time as the search, but only once it will actually be used.

This formula is relatively easy to implement in practice. Let's use "Reader" as an example, and we want to use a resource that has the methods `Open()`, `Read()`, and `Close()`. The `Open()` should only be run upon the first `Read()`, and the `Close()` only when an `Open()` has also been performed. The ruleset reads a user name from this resource using `readUser`. The JavaScript code in Reader could then look like this:

```
var readerInitialized = false;

var reader = new Object();
reader = {

function readUser() {
    If (!readerInitialized) {
        Open();
        readerInitialized = true;
    }
    return Read();
}

}

function readerExitRuleset() {
    if (readerInitialized) {
        Close();
```

```
    };
};
```

Using the `readerInitialized` global variable, the module will recognize whether a connection has been opened using `Open()`. This is set to false when the program starts, and no contact exists yet.

If a rule from the ruleset then wants to identify the user name, the `readUser()` function is invoked. There it will first check whether a connection already exists. If not, a connection is opened with `Open()` and `readerInitialized` set to true. Then, for subsequent calls, no additional `Open()` will be executed. Only afterwards will `Read()` be used on the resource.

Once the ruleset is completed, the end function *readerExitRuleset* will be called for the `Reader` module. This will check whether an open connection still exists, and then close it with `Close()` if required.

ELOas JavaDoc

ELOas 21 provides a number of utility classes/functions for completing frequent tasks. The JavaDoc for the internal ELOas interface is available at http://www.forum.elo.com/javadoc/as/21/. In addition, the current ELO master contains a collection of example rules for calling functions from the ELOas interface.

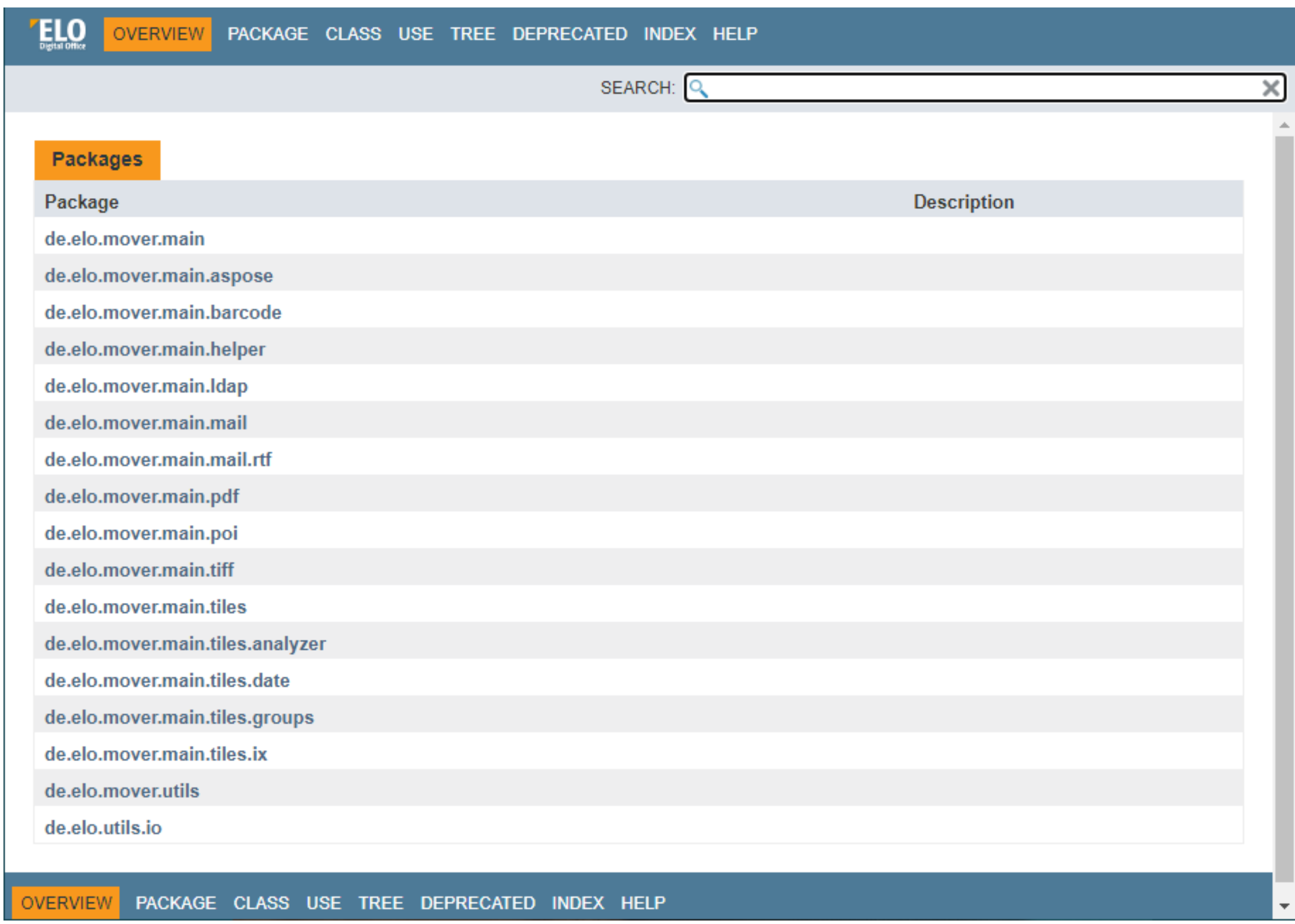


Fig.: Overview

Debugging

Starting with version 7.00.024, a debugger is also available for ELOas. The debug engine built into the Rhino Engine is used. It can be activated using a configuration parameter.

```
<entry key="debug">true</entry>
```

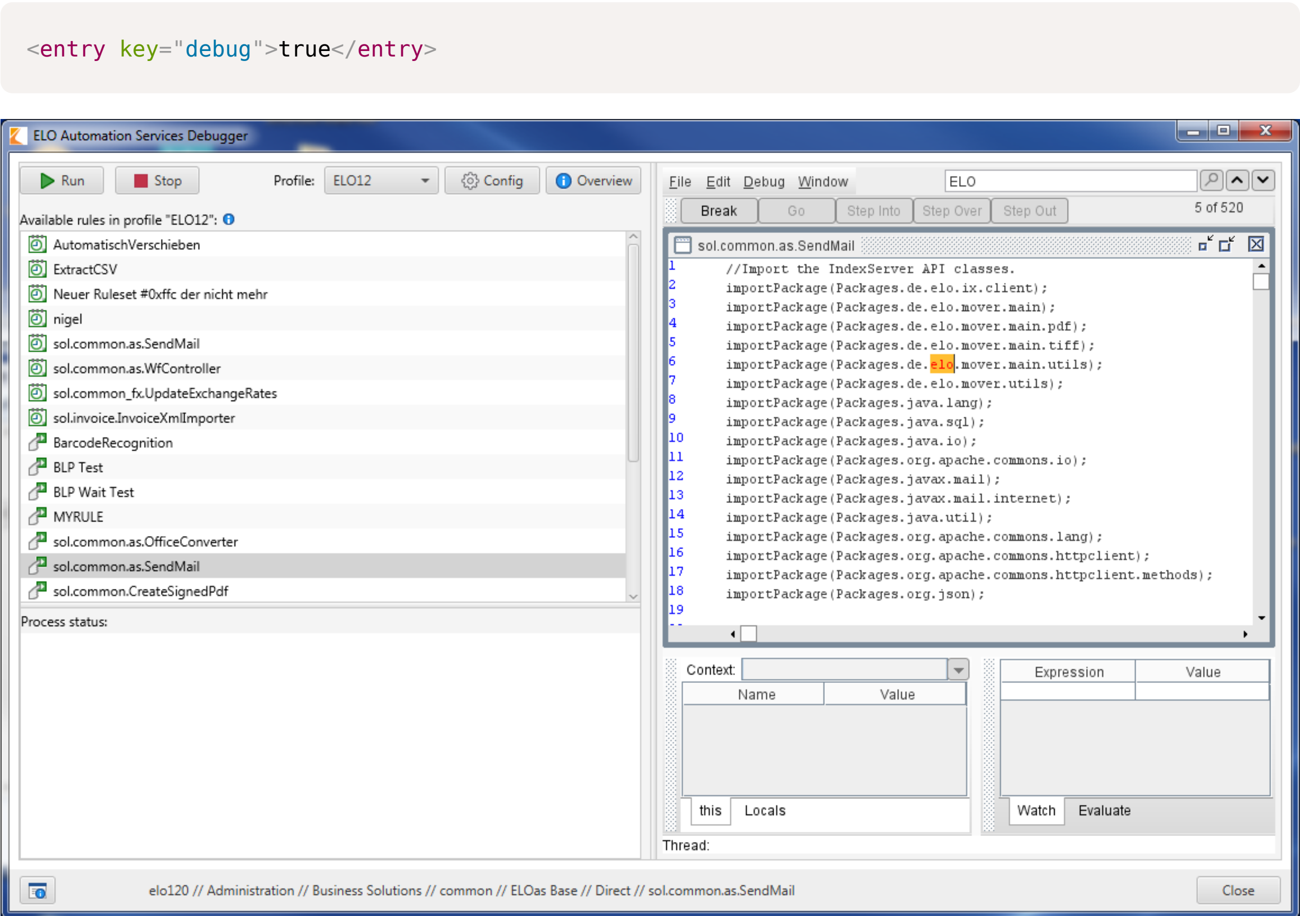


Fig.: ELO Automation Services Debugger

To operate the debugger, ELOas should be run locally from the developer's computer. In addition, it should be started as a console process and not as a Windows service. Otherwise, the debugger will not work on Windows Vista or Windows 7.

If you have multiple active rulesets in use, a separate debugger window opens for each. With the *Window* menu entry, you can switch between these individual windows.

In the debugger, you can set breakpoints for individual functions and inspect or change variable contents. You can then resume execution in individual steps or run mode.

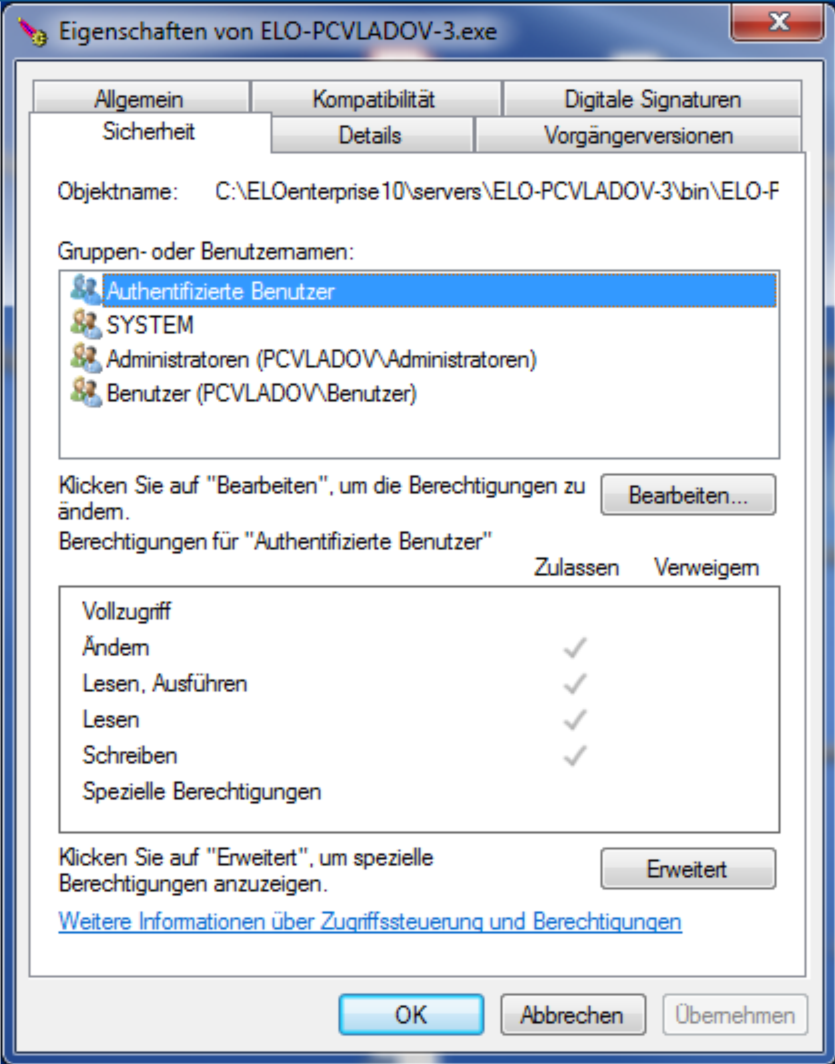


Fig.: Apache Tomcat properties

Syntax errors in the script

If the script contains a syntax error, JavaScript processing will not be able to start. The advantage of this kind of error is that you will them right when you start the program, shown in the ELOas status dialog box.

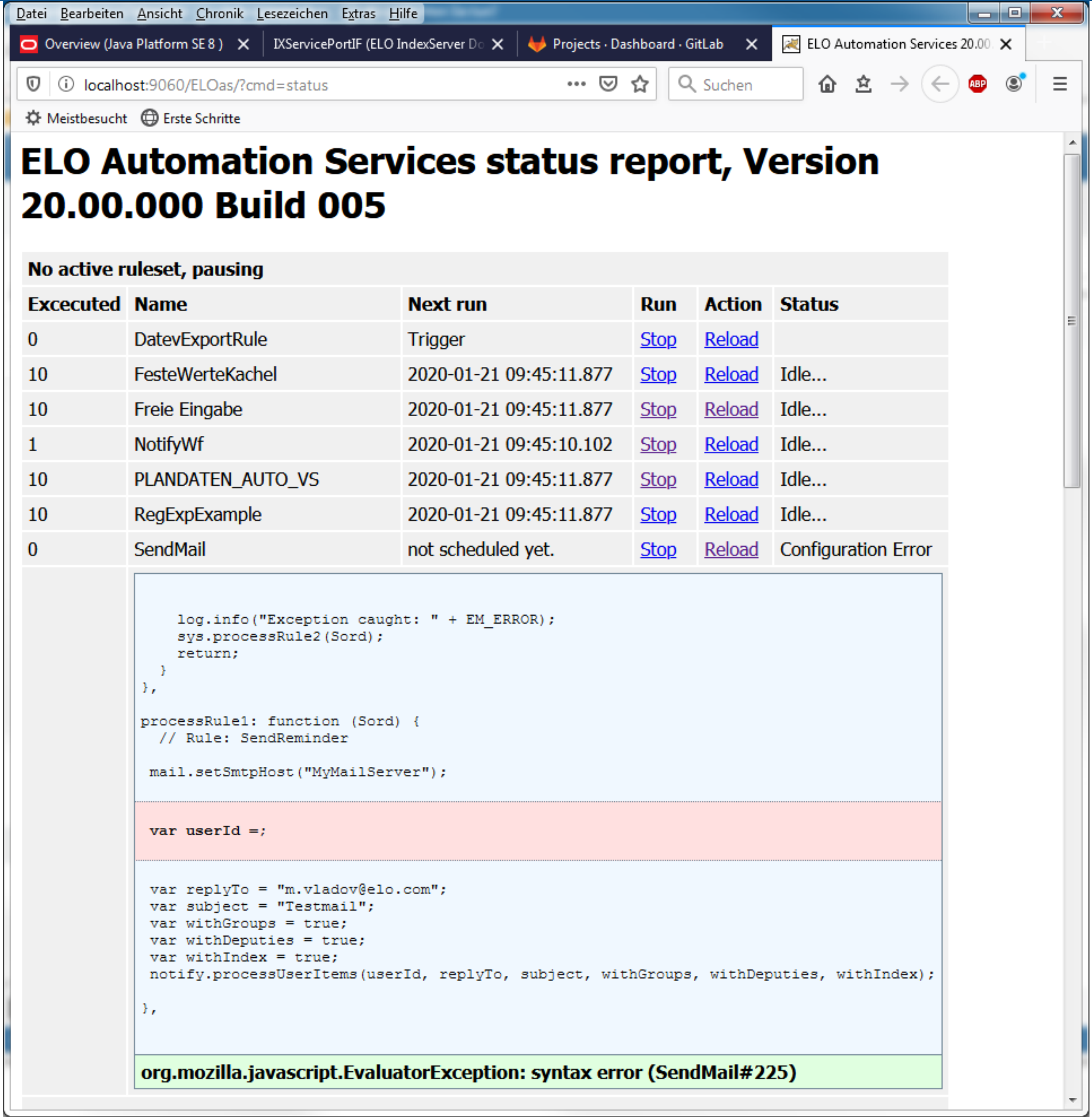


Fig.: Syntax errors in the script

The complete generated JavaScript program with all embedded modules is logged in the ELOas report on start-up to facilitate debugging. The error number listed refers to this section of the report (starting with the section "//Import the IndexServer API classes").

```
14:28:07,681 DEBUG (WorkingSet.java:368) - load JavaScript Templates,
        Parent GUID=(23594D10-4704-4FF9-938B-136792051D67)
14:28:07,744 DEBUG (WorkingSet.java:385) - Script file found: Base Templates
14:28:07,744 DEBUG (WorkingSet.java:385) - Script file found: Imports
14:28:07,744 DEBUG (WorkingSet.java:385) - Script file found: ELO Utils
```

```
14:28:07,759 DEBUG (WorkingSet.java:385) - Script file found: DB Access
14:28:07,759 DEBUG (WorkingSet.java:385) - Script file found: Document Export
14:28:07,759 DEBUG (WorkingSet.java:385) - Script file found: Dummy Modul mit
                                           Namenskonflikt
14:28:07,759 DEBUG (WorkingSet.java:276) - loadItems,
        Parent GUID=(9DAC7E8D-1467-4820-B53B-D27CCB5F06C0)
14:28:07,822 DEBUG (WorkingSet.java:286) - Number of Child entries: 1
14:28:07,822 DEBUG (WorkingSet.java:304) - Ruleset: MailRule1
14:28:08,025 DEBUG (WorkingSet.java:472) -
//Import the IndexServer API classes.
importPackage(Packages.de.elo.ix.client);
importPackage(Packages.java.lang);
importPackage(Packages.java.sql);
importPackage(Packages.sun.jdbc.odbc);
importPackage(Packages.java.io);

var NAME;
var ARCDATE;
var DOCDATE;
var OBJCOLOR;
var OBJDESC;
var OBJTYPE;
var ARCHIVINGMODE;
var ACL;

var EM_PARENT_ID;
var EM_PARENT_ACL;

var EM_NEW_DESTINATION = new Array();
var EM_FIND_RESULT = null;
…
```

Please note that this output will be repeated every restart and reload. Thus, a report file can contain multiple lists. The last list in the report is always the most current.

Logical or runtime errors

Runtime errors are somewhat more difficult to diagnose. The only option is to isolate the location of the error using log outputs. Such a log output is unfortunately much less transparent than an interactive debugger, but it also has significant advantages in batch processing. In ELOas, the Java logger is available on the JavaScript page under the name *log*. For this reason, the JavaScript code can also make entries there with `log.debug()`.

```
var cmd = "SELECT * FROM objekte where objid = 22"
var res = getLine(1, cmd)
log.debug(res.objshort)
```

```
log.debug(res.objidate)
log.debug(res.objguid)
```

The log outputs of the JavaScript code can be recognized by the lack of class name and missing line number in the report (?:?).

```
15:38:57,643 DEBUG (?:?) - Now init JDBC driver
15:38:57,659 DEBUG (?:?) - Get Connection
15:38:57,659 DEBUG (?:?) - Init done.
15:38:57,659 DEBUG (?:?) - createStatement
15:38:57,659 DEBUG (?:?) - executeQuery
15:38:57,659 DEBUG (?:?) - read result
15:38:57,659 DEBUG (?:?) - getLine done.
15:38:57,659 DEBUG (?:?) - Suchen geändert.
15:38:57,659 DEBUG (?:?) - 56666880
```

Standard modules

This document provides a detailed description of the following standard modules:

- cnt: ELO Counter Access
- db: DB Access
- dex: Document Export
- ix: Indexserver functions
- wf: Workflow Utils
- mail: Mail Utils
- fu: File Utils
- run: Runtime Utilities.

The standard modules *elo*, *tfer*, *addr*, *notify*, *exif*, and *www* are described in the JavaDoc for the internal ELOas interface available at http://www.forum.elo.com/javadoc/as/21/.

cnt: ELO Counter Access

The standard module *cnt* enables access to ELOam counter variables.

cnt: Available functions

Create counter: The `createCounter()` function creates a new counter with a start value that can be preset. If the counter already exists, it is reset.

```
createCounter: function (counterName, initialValue) {
    var counterInfo = new CounterInfo();
    counterInfo.setName(counterName);
    counterInfo.setValue(initialValue);

    var info = new Array(1);
    info[0] = counterInfo;

    ixConnect.ix().checkinCounters(info, LockC.NO);
},
```

Get counter value: The `getCounterValue()` function gets the current value of the specified counter. If the `autoIncrement` parameter is set to `true`, the counter value is automatically incremented as well.

```
getCounterValue: function (counterName, autoIncrement) {
    var counterNames = new Array(1);
    counterNames[0] = counterName;
    var counterInfo = ixConnect.ix().checkoutCounters(counterNames,
                                                      autoIncrement,
```

```
                                                           LockC.NO);
    return counterInfo[0].getValue();
},
```

Create tracking number from counter: You can use the `getTrackId()` function when you need a serial, automatically recognizable number. It reads the next counter value and codes a number with a prefix and a check digit. The generated string looks like this: `<prefix><sequential number>C<check digit>` ("ELO1234C0")

```
getTrackId: function (counterName, prefix) {


    var tid = cnt.getCounterValue(counterName, true);
    return cnt.calcTrackId(tid, prefix)
},
```

Create tracking number: You can use the `calcTrackId()` function when you need a serial, automatically recognizable number. It codes a number with a prefix and a check digit. The generated string looks like this: `<prefix><sequential number>C<check digit>` ("ELO1234C0")

```
calcTrackId: function (trackId, prefix) {


    var chk = 0;
    var tmp = trackId;

    while (tmp > 0) {
        chk = chk + (tmp % 10);
        tmp = Math.floor(tmp / 10);
    }

    return prefix + "" + trackId + "C" + (chk %10);
},
```

Search for tracking number in text: The `findTrackId()` function searches a text for a tracking number. The expected prefix and the length of the actual number can be controlled with a parameter. If the number has a variable length, the length parameter can be set to 0. If no appropriate result is found in the text, -1 is returned. Otherwise, the number value (and not the complete track ID) is returned.

```
findTrackId: function (text, prefix, length) {
    text = " " + text + " ";

    var pattern = "\\s" + prefix + "\\d+C\\d\\s";
```

```
    if (length > 0) {
        pattern = "\\s" + prefix + "\\d{" +
                  length + "}C\\d\\s";
    }

    var val = text.match(new RegExp(pattern, "g"));
    if (!val) {
        return -1;
    }

    for (var i = 0; i < val.length; i++) {
        var found = val[i];
        var number = found.substr(prefix.length + 1,
                            found.length - prefix.length - 4);
        var checksum = found.substr(found.length - 2, 1);
        if (checkId(number, checksum)) {
            return number;
        }
    }

    return -1;
}
```

db:DB Access

The DB Access standard module provides simple access to external databases. ODBC databases, as well as Microsoft SQL and Oracle SQL, are supported in the standard module. If other databases need to be accessed with a native JDBC driver, the corresponding JAR files must be copied to the LIB directory of the service, and the imports and access parameters saved to the *Imports* module. The order of database definitions in the imports module will then determine the value of the *Connection number* parameter in the following calls.

db: Available functions

```
getColumn( connection number, SQL query );
```

This call must be provided as a parameter for an SQL query, which requests a column and returns only one row as result.

For example:

```
"select USERNAME from CUSTOMERS where USERID = 12345"
```

The connection number will determine which database connection is used. The list of available connections is defined in the imports module.

Example with JavaScript code:

```
var cmd = "select USERNAME from CUSTOMERS where USERID = 12345"
var res = getColumn(1, cmd)
log.debug(res)
```

Example in the GUI Designer:

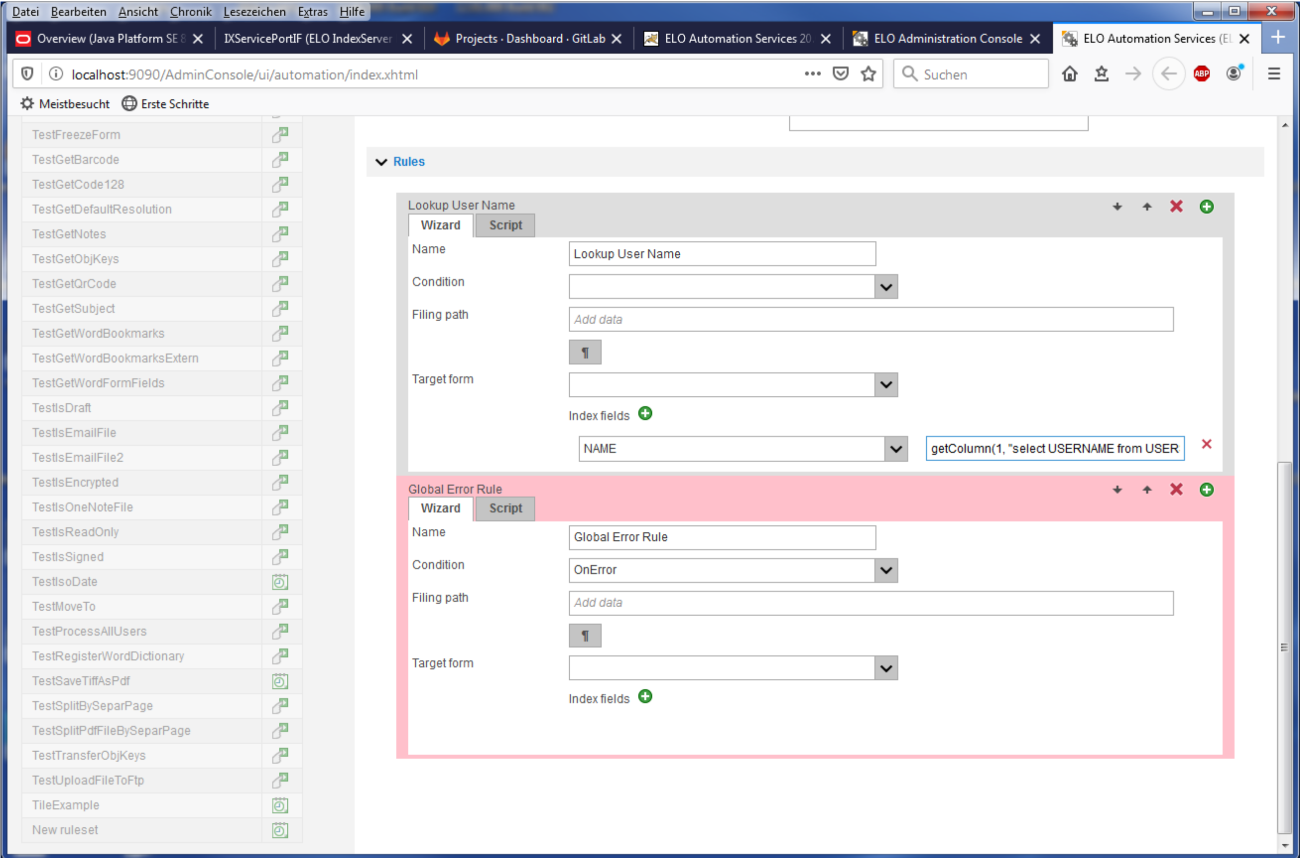


Fig.: GUI Designer

If the results list comprises multiple rows, only the first value is returned. All others are ignored without returning an error message.

```
getLine( connection number, SQL query );
```

This request returns a JavaScript object as the result with the values of the first row of the SQL query. The query can contain any number of columns, including an *. The column names, however, must be unique and valid JavaScript identifiers. Please note that the JavaScript identifiers are case sensitive.

For example:

```
"select USERNAME, STREET, CITY from CUSTOMERS where USERID = 12345"
```

The connection number will determine which database connection is used. The list of available connections is defined in the imports module.

Example with JavaScript code:

```
var cmd =
  "SELECT objshort, objidate, objguid FROM [elo20].[dbo].objekte where objid = 22"
var res = getLine(1, cmd)
log.debug(res.objshort)
log.debug(res.objidate)
log.debug(res.objguid)
```

If the results list contains multiple rows, only the values of the first row are returned. All other rows are ignored without returning an error message.

```
getMultiLine(connection number, SQL command, maximum number of rows)
```

This command works in a similar way to the `getLine` request. However, it returns an array of objects instead of a single object. Each row in the results list creates an entry in the array. To prevent buffer overflows in case of large databases and poorly formed queries, you can limit the maximum number of rows. Additional results are simply ignored.

Example:

```
var obj = db.getMultiLine(1, "select objshort, objid from [elo80].[dbo].objekte where objid < 10

    for (var lg = 0; lg < obj.length; lg++) {
        log.debug(obj[lg].objid + " : " + obj[lg].objshort);
    }

doUpdate(connection number, SQL command)
```

The `getLine` or `getColumn` calls cannot be "abused" to make changes to the database. This command uses the internal JDBC command `executeQuery`, which only permits `SELECT` queries.

In order to change an entry, the `doUpdate` call can be used. This transfers the entered SQL command to the JDBC command `executeUpdate`, which can be used to change existing entries or insert new entries.

Information

> As all parameters have to be transferred in text format, be careful to correctly code any quotation marks that may occur. Otherwise, error messages will occur, and in the worst case scenario, this could even lead to an SQL injection attack on the SQL server.

Imports

The type and scope of required imports depend on the database and can be found in the manufacturer's documentation. The JAR files in use may need to be copied to the LIB directory of the ELOas service.

The following shows an example of the necessary imports for the JDBC-ODBC bridge:

```
importPackage(Packages.sun.jdbc.odbc);
```

A standard system selector was introduced to the `Imports` module of the standard ELOas 12 libraries. For performance reasons, the standard system selector has the standard value `SordC.mbLean` and is used when processing available ELOas rules.

```
const EM_SYS_STDSEL = SordC.mbLean;
```

A system selector named `EM_SYS_SELECTOR` was also introduced to the `Imports` module. The system selector is set to the value of the set standard system selector in the `bt` module. In the `onStart` event of the ELOas rules, the system selector can use/process additional properties of an entry, besides the ID and name.

```
EM_SYS_SELECTOR=SordC.mbAll;
```

At the same time, a workflow selector named `EM_WF_SELECTOR` was added to the workflow constants:

```
var EM_WF_SELECTOR = SordC.mbLean;
```

Connection parameters

The database connection parameters must be saved in the Imports module. There you can find a list of connections that can be later addressed using their number (starting with 0) as connection number.

```
var EM_connections = [
    {
        driver: 'sun.jdbc.odbc.JdbcOdbcDriver',
        url: 'jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=C:\\Temp\\EMDemo.mdb',
        user: '',
```

```
        password: '',
        initdone: false,
        classloaded: false,
        dbcn: null
    },
    {
        driver: 'com.microsoft.sqlserver.jdbc.SQLServerDriver',
        url: 'jdbc:sqlserver://srvt02:1433',
        user: 'elodb',
        password: 'elodb',
        initdone: false,
        classloaded: false,
        dbcn: null
    }
];
```

The following information must be entered for each connection:

driver	JDBC class name for the database connection. You can get this information from the JDBC driver provider or from the database provider.
url	Access URL to the database. Database-dependent connection parameters are configured here, such as file paths for Access databases, or server names and ports for SQL databases. These connection parameters are manufacturer-dependent and can be found in the corresponding documentation.
user	Login name for database access. This parameter is not used by all databases (e.g. not by unprotected Access databases). In such cases, the parameter can remain empty.
password	Database password.
initdone	Internal variable for "lazy initialization".
classloaded	Internal variable to check whether the class file has already been loaded.
dbcn	Internal variable to save the database connection object.

JavaScript code

The `dbInit` routine is only called within the module. It is performed before each database access, and checks whether a connection has been established, establishing one if necessary.

```
function dbInit(connectId) {
  if (EM_connections[connectId].initdone == true) {
    return
  }

  log.debug("Now init JDBC driver")
  var driverName = EM_connections[connectId].driver
  var dbUrl = EM_connections[connectId].url
```

```
  var dbUser = EM_connections[connectId].user
  var dbPassword = EM_connections[connectId].password
  try {
    if (!EM_connections[connectId].classloaded) {
      Class.forName(driverName).newInstance()

      log.debug("Register driver ODBC")
      DriverManager.registerDriver(new JdbcOdbcDriver())
      EM_connections[connectId].classloaded = true
    }

    log.debug("Get Connection")
    EM_connections[connectId].dbcn = DriverManager.getConnection(
      dbUrl,
      dbUser,
      dbPassword
    )

    log.debug("Init done.")
  } catch (e) {
    log.debug("ODBC Exception: " + e)
  }

  EM_connections[connectId].initdone = true
}
```

The `exitRuleset_DB_Access()` function is called automatically once the ruleset is finished processing. It checks whether a connection exists, then closes it. This check must take place for all configured databases.

```
function exitRuleset_DB_Access() {
  log.debug("dbExit")

  for (i = 0; i < EM_connections.length; i++) {
    if (EM_connections[i].initdone) {
      if (EM_connections[i].dbcn) {
        try {
          EM_connections[i].dbcn.close()
          EM_connections[i].initdone = false
          log.debug("Connection closed: " + i)
        } catch (e) {
          log.info("Error closing database " + i + ": " + e)
        }
      }
    }
```

```
  }
}
```

The function `getLine()` reads a line from the database with any number of columns, then packs the results into a JavaScript object. This object then receives a member variable with the column name for each column.

```
function getLine(connection, qry) {
  // Sub-function: creates a JavaScript object with

  // the imported database contents
  function dbResult(connection, qry) {
    // First establish the connection
    dbInit(connection)

    // Now create a SQL statement object
    var p = EM_connections[connection].dbcn.createStatement()

    // And execute the query
    var rss = p.executeQuery(qry)

    // rss contains the list of results. Now the first
    // row is read
    if (rss.next()) {
      // The number of columns is identified via the metadata
      var metaData = rss.getMetaData()
      var cnt = metaData.getColumnCount()

      // A member variable is created for each column
      // It has the SQL column name as the name and
      // imported database contents as the variable.
      // Additionally, the first column can always be addressed
      // under the name 'first'.
      for (i = 1; i <= cnt; i++) {
        var name = metaData.getColumnName(i)
        var value = rss.getString(i)

        this[name] = value
        if (i == 1) {
          this.first = value
        }
      }
    }

    // Finally, the list of results and the SQL
```

```
    // statement are closed.
    rss.close()
    p.close()
  }

  // the actual function's start is here. A
  // JavaScript object with the database contents
  // is requested.
  var res = new dbResult(connection, qry)

  return res
}

// The getColumn function is a special variant
// of the getLine call. The SQL query can only
// show one column as a result. If there are more
// columns, these will be ignored, along with
// additional rows.
function getColumn(connection, qry) {
  var res = getLine(connection, qry)
  return res.first
}
```

dex: Document Export

The *Document Export* module can automatically export documents from the repository to the file system. This export is not a one-time process – if a new document version is created, the module automatically writes an updated file. Further, published files can be deleted. For security reasons, the files can only be placed in a preconfigured path.

To use this module, a metadata form must be defined that contains the document status and one or more filing targets in the file system. In addition, the document number of the most recent export will be saved in the form.

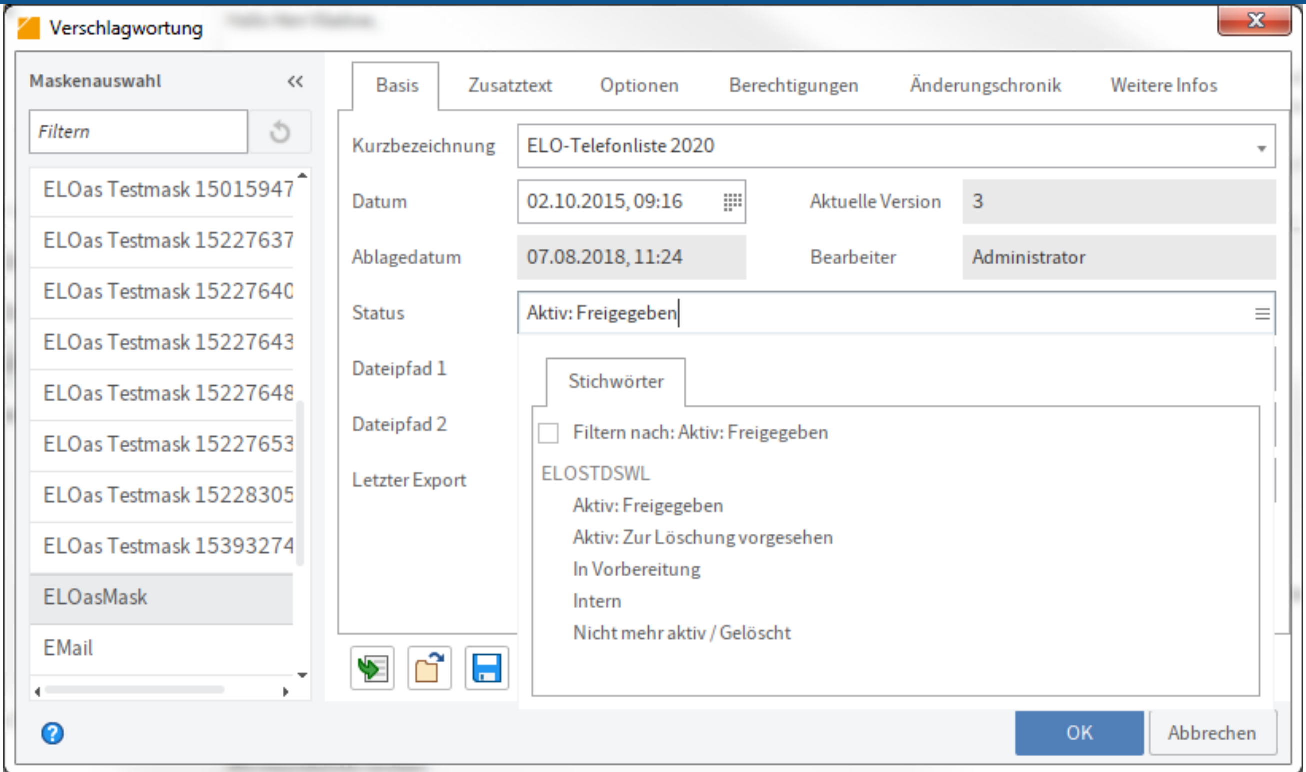


Fig.: Status field in the metadata form

The status field determines the actions to be performed. *Active: Released* registers the file for export. *Active: Set for deletion* deletes the file in the file system and sets the status to *No longer active/deleted*. All other status settings do not trigger an ELOas action and are intended for internal documents or documents that have not yet been released. As this status value is queried for internal processing, it is a good idea to only enter values to this line from a preconfigured keyword list.

The fields File path 1..5 contain the path and file name of the document in the file system. Note that this is a relative path, where the starting path is preset as a fixed value called `dexRoot` in the JavaScript module and can be changed there. This fixed value is designed for security purposes, as otherwise user error could lead to files being overwritten.

The *Last export* field contains the document number of the most recently exported file version. If a file is edited, creating a new version, the module recognizes this and writes a new copy to the file system. This field is then refreshed.

If an error occurs during processing, the error rule enters the text "ERROR" to the *Last export* field. This allows you to create a dynamic index in ELO, which then checks this field for the value ERROR and thus always shows a current list of all documents that cannot be exported.

Example for a dynamic index when the form has an ID of 22:

```
!+ , objkeys where objmask = 22 and objid = parentid and okeyname ='PDEXPORT
       and okeydata ='ERROR'
```

dex: Available functions

This module only provides one function: `processDoc`. It is assigned the ELO Indexserver SORD object as a parameter and, based on the status, checks whether the file should be exported or deleted, then performs the corresponding action. The new document ID is then transferred as return value. The current SORD object is available within a rule process in the JavaScript variable `Sord`.

Example in the XML ruleset code:

```
<rule>
    <name>Rule 1</name>
    <condition>(PDEXPORT != Sord.getDoc()) &amp;&amp; (PDEXPORT != "ERROR") || (PDSTATUS == "Act


    <index>
    <name>PDEXPORT</name>
    <value>dex.processDoc(Sord)</value>
    </index>
</rule>
```

dex: JavaScript code

First, the base path `docRoot` for the document repository is identified. The target path is always ascertained from this setting and the user input in the metadata form. In principle, it would be possible to leave the base path empty, allowing the user to enter any path they wish. However, this approach would present a great security risk, as every user could overwrite any file from the access area of ELOas.

```
var dexRoot = "c:\\temp\\"
```

The `processDoc` function is called from the rule definition. The status of the ELO Indexserver SORD object is checked and the required function is called.

```
function processDoc(Sord) {
  log.debug("Status: " + PDSTATUS + ", Name: " + NAME)

  if (PDSTATUS == "Active: Set for deletion") {
    return dex.deleteDoc(Sord)
  } else if (PDSTATUS == "Active: Released") {
    return dex.exportDoc(Sord)
  }

  return ""
}
```

If the status was set to "Delete", the `deleteDoc` function initiates the deletion of the files and changes the status to "Deleted".

```
function deleteDoc(Sord) {
  dex.deleteFile(PDPATH1)
  dex.deleteFile(PDPATH2)
  dex.deleteFile(PDPATH3)
  dex.deleteFile(PDPATH4)
  dex.deleteFile(PDPATH5)

  PDSTATUS = "No longer active / deleted"
  return Sord.getDoc()
}
```

The `deleteFile` function performs the actual deletion. It first checks whether a file name is configured and whether the file exists, and then removes it from the file system.

```
function deleteFile(destPath) {
  if (destPath == "") {
    return
  }

  var file = new File(docRoot + destPath)
  if (file.exists()) {
    log.debug("Delete expired version: " + docRoot + destPath)

    file["delete"]()
  }
}
```

The internal exportDoc function is called to write new file versions. The file is retrieved by the document manager and copied to the target folder.

```
function exportDoc(Sord) {
  var editInfo = ixConnect
    .ix()
    .checkoutDoc(Sord.getId(), null, EditInfoC.mbSordDoc, LockC.NO)
  var url = editInfo.document.docs[0].getUrl()
  dex.copyFile(url, PDPATH1)
  dex.copyFile(url, PDPATH2)
  dex.copyFile(url, PDPATH3)
  dex.copyFile(url, PDPATH4)
  dex.copyFile(url, PDPATH5)

  return Sord.getDoc()
}
```

The `copyFile` function executes the copying process on the target folder. It first checks whether a target file name already exists and if an older version exists that has to be deleted. The new version is then retrieved by the document manager and saved in the target folder.

```
function copyFile(url, destPath) {
    if (destPath == "") {
        return;
    }

    log.debug("Path: " + docRoot + destPath);


    var file = new File(docRoot + destPath);
    if (file.exists()) {
        log.debug("Delete old version.");
        file["delete"](#ELODOC-D50FBC7EA85D4A709D2C12762E1B9F300);
}
```

ix: IndexServer functions

The ELOix module contains a collection of various ELO Indexserver functions that are required frequently in scripting. However, most of these are simple wrappers to perform a similar ELO Indexserver command, and not complex new functions in themselves.

ix: Available functions

Delete a Sord entry : The object IDs of the SORD entry to be deleted and its parent entry must be passed as parameters to the `deleteSord()` function.

```
lookupIndex: function (archivePath) {
    log.info("Lookup Index: " + archivePath);
    var editInfo = ixConnect.ix().checkoutSord("ARCPATH:" + archivePath,
                                               EditInfoC.mbOnlyId, LockC.NO);
    if (editInfo) {
        return editInfo.getSord().getId();
    }   else {
        return 0;
    }
}
```

Search for an entry : The `lookupIndex()` function identifies the object ID of an entry found via the filing path. The `archivePath` parameter must start with a separator.

```
lookupIndex: function (archivePath) {



    log.info("Lookup Index: " + archivePath);



    var editInfo = ixConnect.ix().checkoutSord("ARCPATH:" + archivePath, EditInfoC.mbOnlyId, Loc
    if (editInfo) {
        return editInfo.getSord().getId();
    }   else {
        return 0;
    }
}
```

Search for an entry: The `lookupIndexByLine()` function identifies the object ID of an entry based on a metadata field search. If the Mask ID parameter is transferred with an empty string, all metadata forms are searched. The group name and the search term must be provided.

```
lookupIndexByLine : function(maskId, groupName, value) {
    var findInfo = new FindInfo();
    var findByIndex = new FindByIndex();
    if (maskId != "") {
        findByIndex.maskId = maskId;
    }

    var objKey = new ObjKey();
    var keyData = new Array(1);
    keyData[0] = value;
    objKey.setName(groupName);
    objKey.setData(keyData);
```

```
    var objKeys = new Array(1);
    objKeys[0] = objKey;

    findByIndex.setObjKeys(objKeys);
    findInfo.setFindByIndex(findByIndex);

    var findResult = ixConnect.ix().findFirstSords(findInfo, 1, SordC.mbMin);
    ixConnect.ix().findClose(findResult.getSearchId());

    if (findResult.sords.length == 0) {
        return 0;
    }

    return findResult.sords[0].id;
},
```

Read the full text information: The `getFulltext()` function returns the current full text information for a document. The full text data is returned as a string.

> **Please note**
>
> It is not possible to tell whether no full text exists, whether full text processing has been completed, or if it was canceled with errors. The text that exists at the time the query is performed is returned (which may be an empty string if no full text information exists).

```
getFulltext: function(objId) {
    var editInfo = ixConnect.ix().checkoutDoc(objId, null, EditInfoC.mbSordDoc, LockC.NO);
    var url = editInfo.document.docs[0].fulltextContent.url
    var ext = "." + editInfo.document.docs[0].fulltextContent.ext
    var name = fu.clearSpecialChars(editInfo.sord.name);

    var temp = File.createTempFile(name, ext);
    log.debug("Temp file: " + temp.getAbsolutePath());

    ixConnect.download(url, temp);
    var text = FileUtils.readFileToString(temp, "UTF-8");

    temp["delete"](#ELODOC-D50FBC7EA85D4A709D2C12762E1B9F301)` checks whether the entered folder
```

js createSubPath: function (startId, destPath, folderMask) {

```
log.debug("createPath: " + destPath);
```

```
try {
    var editInfo = ixConnect.ix().checkoutSord("ARCPATH:" + destPath,
                                               EditInfoC.mbOnlyId, LockC.NO);
    log.debug("Path found, GUID: " + editInfo.getSord().getGuid() +


              " ID: " + editInfo.getSord().getId());



    return editInfo.getSord().getId();;
}   catch (e) {
    log.debug("Path not found, create new: " + destPath +


              ", use foldermask: " + folderMask);
}

items = destPath.split("¶");
var sordList = new Array(items.length - 1);

for (var i = 1; i < items.length; i++) {
log.debug("Split " + i + " : " + items[i]);
var sord = new Sord();
sord.setMask(folderMask);
sord.setName(items[i]);

sordList[i - 1] = sord;
}

log.debug("now checkinSordPath");
var ids = ixConnect.ix().checkinSordPath(startId, sordList,
        new SordZ(SordC.mbName | SordC.mbMask));
log.debug("checkin done: id: " + ids[ids.length - 1]);

return ids[ids.length - 1];
}
```

wf: Workflow Utils

The *wf* module contains simplified access methods to workflow data. This is divided into two groups of functions:

The high level functions `changeNodeUser` and `readActiveWorkflow` are to be used for simple access from a running WORKFLOW process, and work with the currently active workflow. They are easy to use, but only perform a simple function.

The low level functions `readWorkflow`, `writeWorkflow`, `unlockWorkflow`, and `getNodeByName` can be used from any location. If you want to make multiple changes to the same workflow, you can ensure that the workflow will only be read and written once, and not multiple times for each operation.

wf: Available functions

Change user name of a person node: The `changeNodeUser():` function replaces the user in the current workflow node named `nodeName` with a new user - `nodeUserName`.

As this call reads, changes, and immediately rewrites the entire workflow, this simple call should only be used when only one node needs to be edited. If multiple changes are necessary, use the functions described later on to read, edit, and save a workflow.

As this function identifies the workflow ID from the currently active workflow, it can only be called from the "WORKFLOW" search. When using it in a TREEWALK or a normal search, a random workflow ID is used.

```
changeNodeUser: function(nodeName, nodeUserName) {
    var diag = wf.readActiveWorkflow(true);
    var node = wf.getNodeByName(diag, nodeName);
    if (node) {
        node.setUserName(nodeUserName);
        wf.writeWorkflow(diag);
    }   else {
        wf.unlockWorkflow(diag);
    }
}
```

Copy user name at a node: The `copyNodeUser()` function works in a similar way to `changeNodeUser`; however, it copies the user name from one node to another node.

```
copyNodeUser: function(sourceNodeName, destinationNodeName) {
    var diag = wf.readActiveWorkflow(true);
    var sourceNode = wf.getNodeByName(diag, sourceNodeName);
    var destNode = wf.getNodeByName(diag, destinationNodeName);

    if (sourceNode && destNode) {
        var user = sourceNode.getUserName();
        destNode.setUserName(user);
        wf.writeWorkflow(diag);
```

```
            return user;
        }   else {
            wf.unlockWorkflow(diag);
            return null;
        }
}
```

Read current workflow: The `readActiveWorkflow()` function reads the currently active workflow into a local variable for editing. At the end, it can be rewritten with `writeWorkflow`, or the lock can be removed with `unlockWorkflow`.

```
readActiveWorkflow: function(withLock) {
    var flowId = EM_WF_NODE.getFlowId();
    return wf.readWorkflow(flowId, withLock);
    },
```

Read workflow: The `readWorkflow()` function reads a workflow into a local variable. It can then be evaluated and changed. If you want to save the changes, it can be rewritten using `writeWorkflow`. If the workflow is locked and can be read but you do not want to save any changes, the lock can be removed with `unlockWorkflow`.

```
readWorkflow: function(workflowId, withLock) {
    log.debug("Read Workflow Diagram, WorkflowId = " + workflowId);
    return ixConnect.ix().checkoutWorkFlow(String(workflowId),
                                           WFTypeC.ACTIVE,
                                           WFDiagramC.mbAll,
                                           (withLock) ? LockC.YES : LockC.NO);
},
```

Rewrite workflow: The `writeWorkflow()` function writes the workflow from a local variable to the database. Any write lock set on it is reset automatically.

```
writeWorkflow: function(wfDiagram) {
    ixConnect.ix().checkinWorkFlow(wfDiagram, WFDiagramC.mbAll, LockC.YES);
},
```

Reset read lock: `unlockWorkflow()` function. If a workflow with a write lock has been read but cannot be changed, the lock can be reset with `unlockWorkflow`.

```
unlockWorkflow: function(wfDiagram) {
    ixConnect.ix().checkinWorkflow(wfDiagram, WFDiagramC.mbOnlyLock, LockC.YES);
},
```

Search workflow nodes: The `getNodeByName()` function searches the workflow node for a node name. The name must be unique, as otherwise the first node found will be returned.

```
getNodeByName: function(wfDiagram, nodeName) {
    var nodes = wfDiagram.getNodes();
    for (var i = 0; i < nodes.length; i++) {
        var node = nodes[i];
        if (node.getName() == nodeName) {
            return node;
        }
    }

    return null;
},
```

Start workflow from template: The startWorkflow() function starts a new workflow for an ELO object ID from a workflow template.

```
startWorkflow: function(templateName, flowName, objectId) {
    return ixConnect.ix().startWorkFlow(templateName, flowName, objectId);
}
```

mail: Mail Utils

This module is intended for sending e-mails. It requires an SMTP host, through which the e-mails can be sent. This host has to be made known before sending the first e-mail by using the `setSmtpHost` function. Messages can then be sent with `SendMail` or `SendMailWithAttachment`. The module consists of two parts: one for sending e-mails and one for reading e-mail mailboxes.

mail: Available functions for reading a mailbox

You can define a ruleset so that a search is performed on a mailbox and not the ELO repository or ELO task list. A logon routine has to be configured in the module for each type of mailbox. In this function, the mail server must be contacted, the desired e-mail folder searched through, and the list of messages read. Afterwards, ELOas continues to process the command normally. A document is prepared for each e-mail in the folder designated in SEARCHVALUE. Next, the ruleset is executed (the subject of the e-mail is automatically applied to the short name field). If the entry is not saved at the end, there will be nothing to find in the repository either. Only saved e-mails are transferred to the repository.

```
<search>
<name>"MAILBOX_GMAIL"</name>
<value>"ARCPATH:¶ELOas¶IMAP"</value>
<mask>2</mask>
```

In the ruleset, `MAILBOX_<connection name>` must be defined as the name, and the repository path or the number of the target folder as the value. A metadata form to be used for new documents also has to be defined.

The e-mail is then processed in the ruleset script. The *mail* module offers a few help routines to simplify this. In the following example, the body of the e-mail message will be copied to the Extra text tab in the metadata. Sender, recipient, and MailID will be applied to the corresponding fields of the e-mail form:

```
OBJDESC = mail.getBodyText(MAIL_MESSAGE);
ELOOUTL1 = mail.getSender(MAIL_MESSAGE);
ELOOUTL2 = mail.getRecipients(MAIL_MESSAGE, "¶");
ELOOUTL3 = msgId;
EM_WRITE_CHANGED = true;
```

If additional values or information are required, a complete Java e-mail (Mime) message object is available in the MAIL_MESSAGE variable.

To ensure that processed e-mail messages are not transferred to the repository multiple times, you should perform a search for the MailID before you start processing. If the e-mail message is already in the repository, the variable `MAIL_ALLOW_DELETE` is set to true. Otherwise, the e-mail message is processed. By setting the deletion flag, the e-mail is either removed from the mailbox or marked as processed during transfer.

```
var msgId = MAIL_MESSAGE.messageID;
if (ix.lookupIndexByLine(EM_SEARCHMASK, "ELOOUTL3", msgId) != 0) {
    log.debug("Mail bereits im Repository vorhanden, Ignorieren oder Löschen");
    MAIL_ALLOW_DELETE = true;
}   else {
    OBJDESC = mail.getBodyText(MAIL_MESSAGE);
    ELOOUTL1 = mail.getSender(MAIL_MESSAGE);
    ELOOUTL2 = mail.getRecipients(MAIL_MESSAGE, "¶");
    ELOOUTL3 = msgId;
    EM_WRITE_CHANGED = true;
}
```

This approach only reads an e-mail twice (once for normal processing, and once in the next pass for deletion), but it has the great advantage of ensuring the e-mail is only deleted from the mailbox if it definitely exists in the repository.

If you want to use a mailbox for monitoring, the following four functions are required in the JavaScript library 'mail':

Establish connection, open mailbox folder: `nextImap_<connection name>`

Next message in the list for processing: `finalizeImap_<connection name>`

Mark message as processed or delete: `finalizeImap_<connection name>`

Close connection: `closeImap_<connection name>`

In simple cases, only one of these four functions needs to be implemented: establish connection – `connectImap_<Verbindungsname>`. As a complete range of project-specific actions takes place here (login parameters, searching for target folder), there is no standard implementation. The three other functions already exist with a standard function in the system. You simply need to implement them to perform these additional functions.

Connect to IMAP server: `connectImap_<connection name>()`: This function must establish a connection with the e-mail server, search for the desired mailbox, and read it. Existing messages are saved to the variable `MAIL_MESSAGES`. The e-mail store must be saved to the variable `MAIL_STORE` and the folders that are read out to the variable `MAIL_INBOX`. Both of these values are required at the end of processing to close the connection. The variable `MAIL_DELETE_ARCHIVED` determines whether messages can be deleted from the mailbox. If set to false, deletion requests from the ruleset are ignored. This function will not be directly called up via a script, but rather activated internally in ELOas (in the MAILBOX search, in the example `MAILBOX_GMAIL`).

```
connectImap_GMAIL: function() {
    var props = new Properties();
    props.setProperty("mail.imap.host", "imap.gmail.com");
    props.setProperty("mail.imap.port", "993");
    props.setProperty("mail.imap.connectiontimeout", "5000");
    props.setProperty("mail.imap.timeout", "5000");
    props.setProperty("mail.imap.socketFactory.class",
                      "javax.net.ssl.SSLSocketFactory");
    props.setProperty("mail.imap.socketFactory.fallback", "false");
    props.setProperty("mail.store.protocol", "imaps");

    var session = Session.getDefaultInstance(props);
    MAIL_STORE = session.getStore("imaps");
    MAIL_STORE.connect("imap.gmail.com", "<<<USERNAME>>>@gmail.com",
                       "<<<PASSWORT>>>");
    var folder = MAIL_STORE.getDefaultFolder();
    MAIL_INBOX = folder.getFolder("INBOX");
    MAIL_INBOX.open(Folder.READ_WRITE);
    MAIL_MESSAGES = MAIL_INBOX.getMessages();
    MAIL_POINTER = 0;
    MAIL_DELETE_ARCHIVED = false;
},
```

Close connection: The `closeImap_<Verbindungsname>` function is optional and closes the current connection to the IMAP server. If no special tasks need to be performed on closing, you do not need to implement this function. Instead, the standard implementation `closeImap()` from the library is used. This closes the folder and the store.

```
closeImap_GMAIL: function() {
    // hier können eigene Aktionen vor dem Schließen ausgeführt werden

    // Standardaktion, Folder und Store schließen.
    MAIL_INBOX.close(true);
    MAIL_STORE.close();
},
```

Mark message as processed or delete: The `finalizeImap_<connection name>()` function is optional and deletes the current message, or otherwise marks it as processed. If it is not implemented, ELOam uses the standard implementation, which deletes a processed e-mail from the folder.

The following example does not delete the e-mail, but rather sets it to "read".

```
finalizeImap_GMAIL: function() {
    if (MAIL_DELETE_ARCHIVED && MAIL_ALLOW_DELETE) {
        message.setFlag(Flags.Flag.SEEN, true);
    }
},
```

Process next message in the list: The `nextImap_<connection name>` function is optional and returns the next message in the selected mailbox to the ruleset for processing. If the function is not implemented, ELOas will use the standard implementation, which sends every document for processing.

The example shows an implementation that only processes unread e-mails. They can be used in pairs with the finalizeImap implementation above, which sets e-mails as read rather than deleting them.

> **Please note**
>
> If you work with this method, you must use another way to ensure that the mailbox does not grow too large (such as by deleting automatically after a certain date).

```
nextImap_GMAIL: function() {
    if (MAIL_POINTER > 0) {
        mail.finalizePreviousMessage(MAIL_MESSAGE);
    }

    for (;;) {
        if (MAIL_POINTER >= MAIL_MESSAGES.length) {
            return false;
        }
```

```
        MAIL_MESSAGE = MAIL_MESSAGES[MAIL_POINTER];

        var flags = MAIL_MESSAGE.getFlags();
        if (flags.contains(Flags.Flag.SEEN)) {
            MAIL_POINTER++;
            continue;
        }

        MAIL_ALLOW_DELETE = false;
        MAIL_POINTER++;
        return true;
    }

    return false;
},
```

Read e-mail body text: The getBodyText() function transfers the message as a parameter (available in the script via the variable MAIL_MESSAGE) and returns the mail body as return parameter. It also searches for the first MIME part of type TEXT/PLAIN. If no corresponding part exists, an empty string is returned.

```
getBodyText: function(message) {
    var content = message.content;
    if (content instanceof String) {
        return content;
    }   else if (content instanceof Multipart) {
        var cnt = content.getCount();
        for (var i = 0; i < cnt; i++) {
            var part = content.getBodyPart(i);
            var ct = part.contentType;
            if (ct.match("^TEXT/PLAIN") == "TEXT/PLAIN") {
                return part.content;
            }
        }
    }

    return "";
},
```

Identify sender: The getSender() function returns the e-mail address of the sender.

```
getSender: function(message) {
    var adress = message.sender;
    return adress.toString();
},
```

Identify recipient: The getRecipients() function returns a list of all recipients (TO and CC). If there is more than one recipient, the list is provided in column index format, assuming that the ELO separator symbol ¶ is transferred in the 'delimiter' parameter.

```
getRecipients: function(message, delimiter) {
    var adresses = message.allRecipients;

    var cnt = 0;
    if (adresses) { cnt = adresses.length; }
    var hasMany = cnt > 1;

    var result = "";
    for (var i = 0; i < cnt; i++) {
        if (hasMany) { result = result + delimiter; }
        result = result + adresses[i].toString();
    }

    return result;
}
```

Available functions for sending e-mails

The send functions are not used directly by ELOas. They are utility functions for custom script programming to conceal the complexity of the Java mail API from the script developer.

Register SMTP server: The setSmtpHost() function registers the library of the SMTP host to be used. This library is used to send e-mails. This function must be activated before the first sendMail call.

```
setSmtpHost: function(smtpHost) {
    if (MAIL_SMTP_HOST != smtpHost) {
        MAIL_SMTP_HOST = smtpHost;
        MAIL_SESSION = undefined;
    }
},
```

Send e-mail: The `sendMail()` function sends an e-mail. The sender and recipient addresses are transferred as parameters, in addition to the subject and e-mail text.

```
sendMail: function(addrFrom, addrTo, subject, body) {
    mail.startSession();
    var msg = new MimeMessage(MAIL_SESSION);
    var inetFrom = new InternetAddress(addrFrom);
    var inetTo = new InternetAddress(addrTo);
    msg.setFrom(inetFrom);
    msg.addRecipient(Message.RecipientType.TO, inetTo);
    msg.setSubject(subject);
    msg.setText(body);
    Transport.send(msg);
},
```

Send e-mail with attachment: The `sendMailWithAttachment()` function sends an e-mail. The sender and recipient addresses are transferred as parameters, in addition to the subject, e-mail text, and the object ID of the attachment from ELO. The attachment is stored as a temporary file in a temporary path; sufficient space must be available at this location.

```
sendMailWithAttachment: function(addrFrom, addrTo, subject, body, attachId) {
    mail.startSession();
    var temp = fu.getTempFile(attachId);
    var msg = new MimeMessage(MAIL_SESSION);
    var inetFrom = new InternetAddress(addrFrom);
    var inetTo = new InternetAddress(addrTo);
    msg.setFrom(inetFrom);
    msg.addRecipient(Message.RecipientType.TO, inetTo);
    msg.setSubject(subject);

    var textPart = new MimeBodyPart();
    textPart.setContent(body, "text/plain");

    var attachFilePart = new MimeBodyPart();
    attachFilePart.attachFile(temp);

    var mp = new MimeMultipart();
    mp.addBodyPart(textPart);
    mp.addBodyPart(attachFilePart);
    msg.setContent(mp);
    Transport.send(msg);

    temp["delete"]();
}
```

fu: File Utils

The File Utils functions help ELOas users with file operations.

fu: Available functions

Clean up file name: If you want to create a file name from the short name, it may contain critical characters that can lead to problems in the file system (e.g. colon, backslash, and ampersand). The clearSpecialChars() function replaces all characters other than numbers and letters with an underscore (including umlauts and ß).

```
clearSpecialChars: function(fileName) {
    var newFileName = fileName.replaceAll("\\W", "_");
    return newFileName;
},
```

Load document file: The getTempFile() function downloads the document file for the specified ELO object to the local file system (in the ELOas temp folder). If the file is no longer required, it must be removed again by the script developer using the function deleteFile. Otherwise, it will remain on the hard drive.

> **Please note**
>
> This returns a Java file object, not a file name.

```
getTempFile: function(sordId) {
    var editInfo = ixConnect.ix().checkoutDoc(sordId, null,
                                          EditInfoC.mbSordDoc, LockC.NO);
    var url = editInfo.document.docs[0].url;
    var ext = "." + editInfo.document.docs[0].ext;
    var name = fu.clearSpecialChars(editInfo.sord.name);

    var temp = File.createTempFile(name, ext);
    log.debug("Temp file: " + temp.getAbsolutePath());

    ixConnect.download(url, temp);

    return temp;
},
```

Delete file: The deleteFile() function expects a Java file object as a parameter (not a string) and deletes this file.

```
deleteFile: function(delFile) {
    delFile["delete"]();
}
```

run: Runtime Utilities

This module contains routines for access to the Java runtime. This allows external processes to be started or the current memory status to be queried.

Start process: The execute(command) command starts an external process. ELOas waits for the this call to finish and only then does it continue processing. This allows actions in this process to be evaluated as well.

```
log.debug("Process: " + NAME );
run.execute("C:\\ Tools\\BAT\\dirlist.bat");
log.debug("Read Result");
var txt = dex.asString("dirlist.txt");
```

Query free and available memory: The freeMemory() and maxMemory() commands display the currently available free memory and the maximum available memory.

```
log.debug "freeMemory: " + run.freeMemory() +
        ", maxMemory: " + run.maxMemory());
```

Examples

Example – Moving a document

A document needs to be moved in ELO.

1. Open the *ELO Automation Services* in the ELO Administration Console.

2. Click *Add*.

3. Enter a new name for the rule, such as *Move newsletter*.

   The new rule is created but not yet saved.

4. Select a search metadata form.

   In this example, the *Marketing* metadata form is used.

5. In the *Index search* field, select the metadata form field that you want to use to select documents.

   In this example, the *Status* field is used. If the *Status* field contains the value `sent`, the document will be moved. Documents with other values will not be moved.

6. Enter `"sent"` as the search term.

> **Please note**
>
> If quotation marks are used in the example, they are necessary. If one or both quotation marks are missing, this leads to an error.

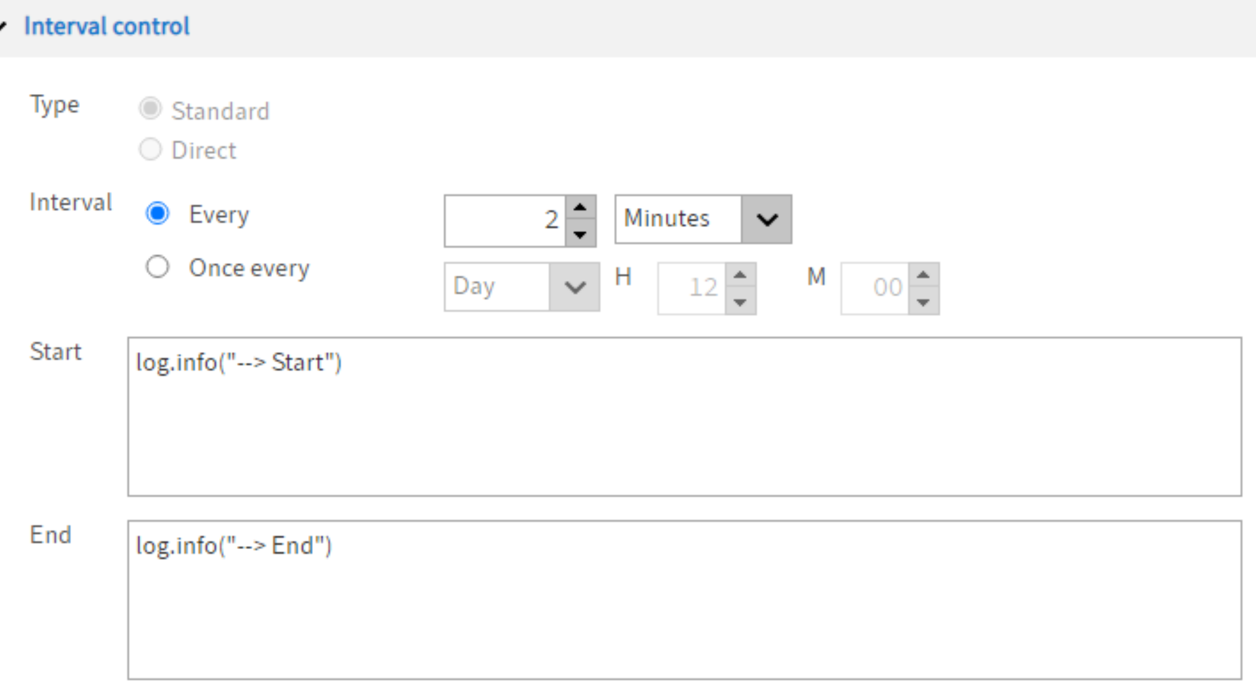


Fig.: Interval controls for rules

1. Define the interval that will pass before the rule is executed again.

   Optional: In the *Start* and *End* fields, you can enter script commands that are executed at the beginning or after the rule has been executed.

   This example uses the `log.info("<Any text>")` command to mark the beginning and end of the rule execution in the ELO Automation Services log file. This can be useful for troubleshooting.

   The path for the log file is as follows:

   ```
   <installation path>\logs\<name of server instance>\as-<repository name>.log
   ```

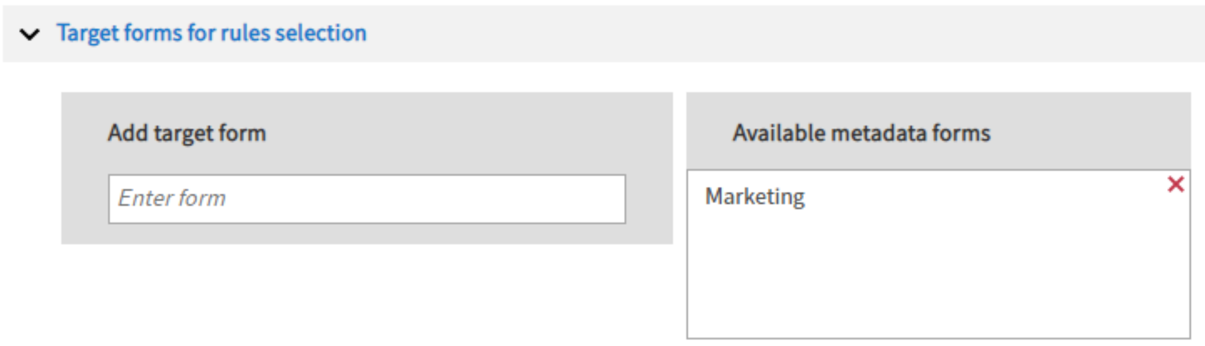
   

   Fig.: Selection of the available target forms

2. Under *Target forms for rules selection*, enter the metadata form that is used.

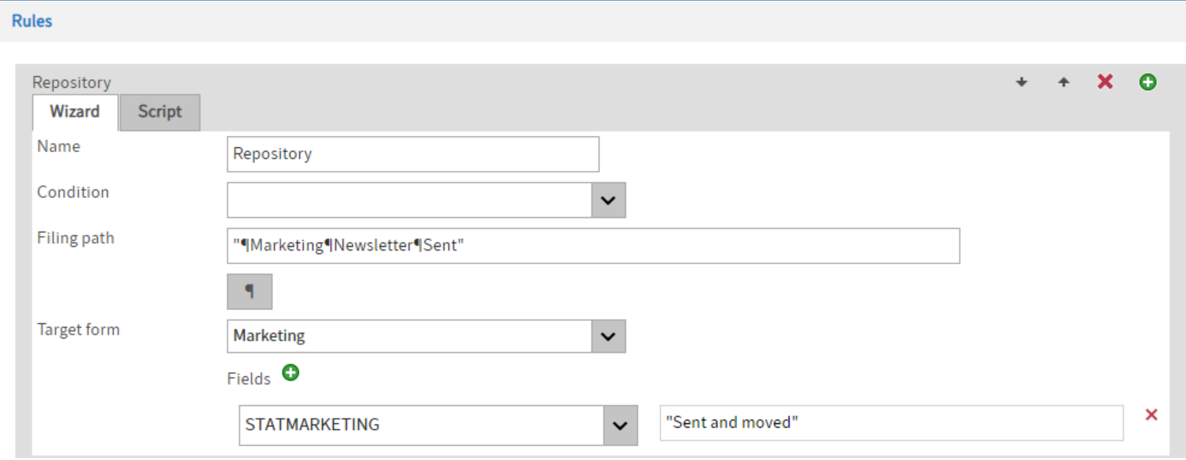


Fig.: Rule settings

3. Under *Rules*, enter a name for the first rule on the *Wizard* tab.

   In this example, the rule is named *Filing*.

4. Enter the target path to the *Filing path* field.

   This example uses the following path:

   `"¶Marketing¶Newsletter¶Sent"`

5. Under *Target form*, select the metadata form used above.

6. Click *Add field* (green plus icon) and select the field used above.

7. In the input field, enter the value that you want to apply to the field on the metadata form. This prevents the rule from entering into an infinite loop.

   This example uses the value: `"sent and moved"`.

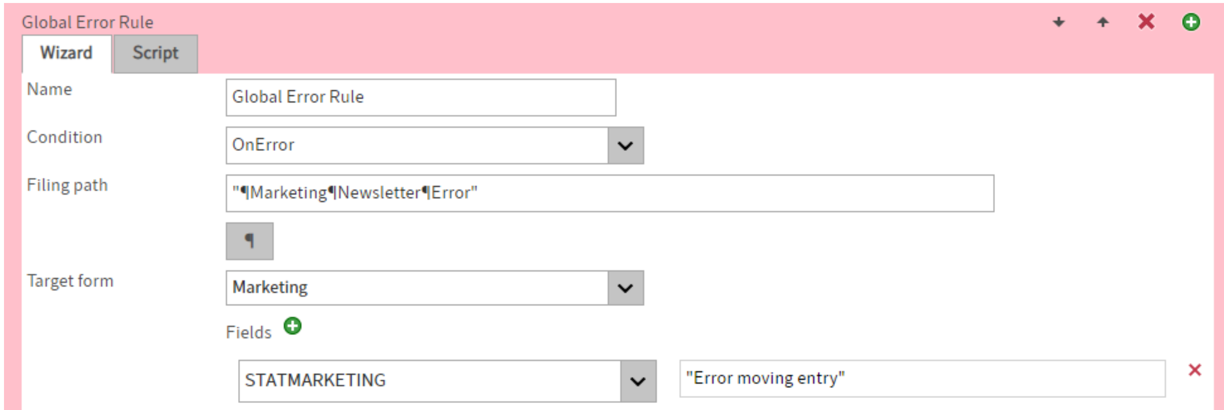


Fig.: Rules for errors

Optional: Under *Global Error Rule*, you can specify a rule that is triggered on errors.

This example uses the following path:

`"¶Marketing¶Newsletter¶Error"`

In this example, the value `"Move error"` is entered in the status field to prevent an infinite loop here as well.

8. Save the ruleset.

No active ruleset, pausing					
Excecuted	**Name**	**Next run**	**Run**	**Action**	**Status**
14	Move newsletter	2020-04-02 14:39:43.220	Stop	Reload	Idle...
Direct Pool					1 / 0
Reload all					

Fig.: ELO Automation Services status page, Reload

9. Go to the ELO Automation Services status page of and click *Reload* for the respective rule.

   You can access the ELO Automation Services status page via the respective ELO Application Server manager or via the URL with the following structure:

```
http(s)://<server name>:<port>/as-<repository name>/?cmd=status
```

The rule moves documents containing the character string "sent" to the *Sent* folder.

Example: e-mail folder monitoring

The ELO Automation Services JavaScript library contains a module for sending and receiving e-mails. This guide will explain how to use ELOas to monitor a mailbox.

> **Information**
>
> This example is not intended to simulate e-mail archiving. There are other modules in our product range that better accomplish this task. Instead, it is intended to serve as a basis for "autoresponders", i.e. programs that automatically trigger an action in response to an e-mail message (for example, a user sends a registration e-mail, after which their account is activated).

General approach

Before a ruleset can be created to process mailboxes, a mailbox connection must be created in the *mail* module. As there are many differences and options here, it is not possible to work from a simple configuration list. Instead, you have to create a `connect` method for each mailbox connection. This must establish a connection with the e-mail server, select the correct mailbox, and read the list of messages.

Every mailbox connection is given a simple, short name - e.g. GMAIL. This name is required at various locations and must be "identifier-compatible", i.e. it must start with a letter and can then contain additional letters or numbers (but no special characters, including letters with accent marks). This name is required at various places in the ruleset and the JavaScript implementation.

Establishing the connection

The JavaScript library already has a definition in the standard installation for a connection with the name GMAIL. We will use it for the example. As the connection name is used in special functions, you can also define multiple connections in parallel and use them in various rulesets.

The standard function for establishing the GMAIL connection looks like the following:

```
connectImap_GMAIL: function() {


    var props = new Properties();
    props.setProperty("mail.imap.host", "imap.gmail.com");
    props.setProperty("mail.imap.port", "993");
    props.setProperty("mail.imap.connectiontimeout", "5000");
    props.setProperty("mail.imap.timeout", "5000");
    props.setProperty("mail.imap.socketFactory.class",
                      "javax.net.ssl.SSLSocketFactory");
    props.setProperty("mail.imap.socketFactory.fallback", "false");
```

```
    props.setProperty("mail.store.protocol", "imaps");

    var session = Session.getDefaultInstance(props);
    MAIL_STORE = session.getStore("imaps");
    MAIL_STORE.connect("imap.gmail.com",
                       "<USER>@gmail.com",
                       "<PASSWORD>");
    var folder = MAIL_STORE.getDefaultFolder();
    MAIL_INBOX = folder.getFolder("INBOX");
    MAIL_INBOX.open(Folder.READ_WRITE);
    MAIL_MESSAGES = MAIL_INBOX.getMessages();
    MAIL_DELETE_ARCHIVED = false;
},
```

The example connects to the Googlemail server "imap.gmail.com" at port "993" via an encrypted connection (`mail.store.protocol` - `imaps`). This information is entered to a property object. Your own e-mail server may require other values - refer to the e-mail server's documentation for details.

> **Information**
>
> If you set up a Google e-mail account, you must first enable IMAP access to use this method. This is possible under *Settings* > *Forwarding and POP/IMAP* > *Activate IMAP*.

Logon is then performed using the command `MAIL_STORE.connect`. Enter the server name again, as well as the mailbox user with password.

After logon, the *Inbox* folder is searched for first. However, any other folders can be monitored, such as *Sent*:

```
MAIL_INBOX = folder.getFolder("[Google Mail]/Sent")
```

Using the command `MAIL_INBOX.getMessages()`, all e-mails in the folder are read and added to the internal message list. This list will be processed later by calling the ruleset once for each entry in this list.

The variable `MAIL_DELETE_ARCHIVED` determines whether the ruleset is allowed to delete messages or mark them as processed after successful processing. If it is set to "false", as is preconfigured, the message status is not changed. This is especially practical in the testing phase, as it is not necessary to constantly create new e-mails. In production, this entry is normally set to "true".

Create ruleset

A simple ruleset to process the mailbox content consists of two main parts: the definition of the search and the script to process the e-mails.

The search is defined as follows:

```
<search>
<name>"MAILBOX_GMAIL"</name>
<value>"ARCPATH:¶IMAP"</value>
<mask>2</mask>
<max>200</max>
</search>
```

The search name "MAILBOX_GMAIL" signalizes that this is not a normal repository search, but rather a search of a mailbox with the connection name GMAIL. The created ELO documents are filed to the "IMAP" folder (via `ARCPATH:¶IMAP`) and created with form 2 (e-mail in a standard ELO repository). Normally, the number of results is no longer relevant, but it should still be entered to prevent an error message in the designer.

The script to run is essentially determined by the required function. A simple script could look like the following:

```
<script>
    log.debug("Process Mailbox: " + NAME);


    OBJDESC = mail.getBodyText(MAIL_MESSAGE);
    ELOOUTL1 = mail.getSender(MAIL_MESSAGE);
    ELOOUTL2 = mail.getRecipients(MAIL_MESSAGE, "¶");
    EM_WRITE_CHANGED = true;
    MAIL_ALLOW_DELETE = true;
</script>
```

When the script is run, the message is available in the `MAIL_MESSAGE` variable. Standard values like e-mail text, sender, and recipient can be read from here. To simplify the process, the mail module provides the help routines `getBodyText`, `getSender`, and `getRecipients`.

The subject is automatically used as the short name (NAME). The body of the e-mail is entered to the extra text, and the sender and recipient are transferred to their corresponding metadata fields. Last, the message is marked as processed as deleted via `MAIL_ALLOW_DELETE`.

The complete example will then look like the following:

```
<ruleset>
<base>
<name>Mailbox</name>
<search>
<name>"MAILBOX_GMAIL"</name>
<value>"ARCPATH:¶IMAP"</value>
<mask>2</mask>
<max>200</max>
```

```
</search>
<interval>10M</interval>
</base>
<rule>
<name>List</name>
<condition></condition>
<script>
    log.debug("Process Mailbox: " + NAME);


    OBJDESC = mail.getBodyText(MAIL_MESSAGE);
    ELOOUTL1 = mail.getSender(MAIL_MESSAGE);
    ELOOUTL2 = mail.getRecipients(MAIL_MESSAGE, "¶");
    EM_WRITE_CHANGED = true;
    MAIL_ALLOW_DELETE = true;
</script>
</rule>
<rule>
<name>Global Error Rule</name>
<condition>OnError</condition>
<script></script>
</rule>
</ruleset>
```

Monitored processing

This simple example has a significant disadvantage: when an e-mail has already been marked as "processed" or deleted, and the process is canceled before the data could be saved in the repository, a data set will remain unprocessed. This problem can be completely avoided by working with a two-level approach: a new e-mail is initially only saved in ELO, but not deleted. The e-mail is only deleted if a later run finds that it already exists in ELO.

This approach has two requirements: the e-mail must be uniquely identifiable, and the method must check during processing whether the e-mail already exists in the repository. The first condition is easy to meet: every e-mail has an internal mail ID. This can be saved to a metadata field in ELO (such as in the default e-mail form in the field ELOOUTL3, which is intended for the mail ID). The second condition can easily be met with a help routine from the ELOix module: `ix.lookupIndexByLine`.

The changed script will then look like the following:

```
<script>
    log.debug("Process Mailbox: " + NAME);


    // if the message is already in the repository: then delete..
```

```
    var msgId = MAIL_MESSAGE.messageID;
    if (ix.lookupIndexByLine(EM_SEARCHMASK, "ELOOUTL3", msgId) != 0) {
        log.debug("E-mail already exists in repository,

                  ignore or delete");
        MAIL_ALLOW_DELETE = true;
    }   else {
        OBJDESC = mail.getBodyText(MAIL_MESSAGE);
        ELOOUTL1 = mail.getSender(MAIL_MESSAGE);
        ELOOUTL2 = mail.getRecipients(MAIL_MESSAGE, "¶");
        ELOOUTL3 = msgId;
        EM_WRITE_CHANGED = true;
    }
</script>
```

Marking instead of deleting

In the standard implementation, a processed e-mail is deleted from the mailbox. This is undesirable in some cases. However, a marker can be set for the messages instead. A possible candidate is the "read" flag. A processed e-mail message is set as "read" by ELOas and thus differs from a new e-mail. In this special case, additional methods have to be defined in the mail JavaScript library in addition to the `connectImap` method:

nextImap_GMAIL(): This function switches to the next message. In this example, you have to check whether an e-mail has already been marked as read, and can skip over it if needed.

finalizeImap_GMAIL(): This function marks the processed message. In the standard implementation, the message is deleted. In our example, however, it should only be marked as *read*.

nextImap_GMAIL

This function switches to the next message. It goes through the list of messages in sequence. The current position is saved in the `MAIL_POINTER` variable. If a message has already been marked as read, it is skipped. At the first unread message, it is activated (meaning, copied into the `MAIL_MESSAGE` variable) and the value "true" is returned. If there are no further messages, a "false" value is returned. ELOas finishes processing this ruleset and then switches to the next.

```
nextImap_GMAIL: function() {


    for (;;) {
        if (MAIL_POINTER >= MAIL_MESSAGES.length) {
            return false;
```

```
        }

        MAIL_MESSAGE = MAIL_MESSAGES[MAIL_POINTER];

        var flags = MAIL_MESSAGE.getFlags();
        if (flags.contains(Flags.Flag.SEEN)) {
            MAIL_POINTER++;
            continue;
        }

        MAIL_ALLOW_DELETE = false;
        MAIL_POINTER++;
        return true;
    }

    return false;
},
```

In addition to switching to the next message, initialization takes place: the variable `MAIL_ALLOW_DELETE` is set to false. This value should only be set to true when an object has been processed within the ruleset. In this case, the e-mail is marked as processed in the finalizeImap method.

finalizeImap_GMAIL

The `finalizeImap_GMAIL` function must mark an e-mail as processed. This is done by setting the `SEEN` flag. However, it may only be set if the `connect` method allows it at all (`MAIL_DELETE_ARCHIVED`), and the ruleset has marked the current e-mail message as archived (`MAIL_ALLOW_DELETE`).

```
finalizeImap_GMAIL: function() {


    if (MAIL_DELETE_ARCHIVED &amp;&amp; MAIL_ALLOW_DELETE) {
        message.setFlag(Flags.Flag.SEEN, true);
    }
},
```

Example – migrating a document database

For our internal "Improvement suggestion scheme", we have to migrate a database with about 1400 entries to ELO. The metadata and the documents are located in this database. In ELO, we want to create a folder every time metadata is entered, which then contains the actual document as a child entry. ELOas has been selected as primary tool for the migration.

Since ELOas currently cannot create documents, a dummy entry had to be initially created for each folder. Fortunately, the entries in the database have been numbered consecutively from 1 to 1440. For this reason, the dummy folders were relatively simple to create using a VBS script. All folders were created within another folder with the object ID of 274312.

```
Set ELO = CreateObject("ELO.professional")
Elo.CheckUpdate 0

for i=1 to 1440
    call Elo.PrepareObjectEx( 0, 4, 337 )
    Elo.ObjShort="TrackId " & i
    Elo.ObjIndex="#274312"
    call Elo.SetObjAttrib(2, i)
    call Elo.SetObjAttrib(0, "GilleM")
    call Elo.SetObjAttrib(3, "Produktverbesserung")
    Elo.UpdateObject
next
Elo.CheckUpdate 1
```

After this, ELOas is called. The data is collected from an SQL database:

```
"select Editor, Email, TheSubject, LTrim(BunField1) BunField1,
        ClassName, Task
                from [InetHelpDesk].[dbo].tblTasks a,
                     [InetHelpDesk].[dbo].tblBatch b,
                     [InetHelpDesk].[dbo].tblClass c,
                     [InetHelpDesk].[dbo].tblUser d
                        where a.BunID = b.BunId
                                and a.KlaID = c.KlaID
                                and a.UsrID = d.UsrID
                                and AufID = " + ETS_COUNT
```

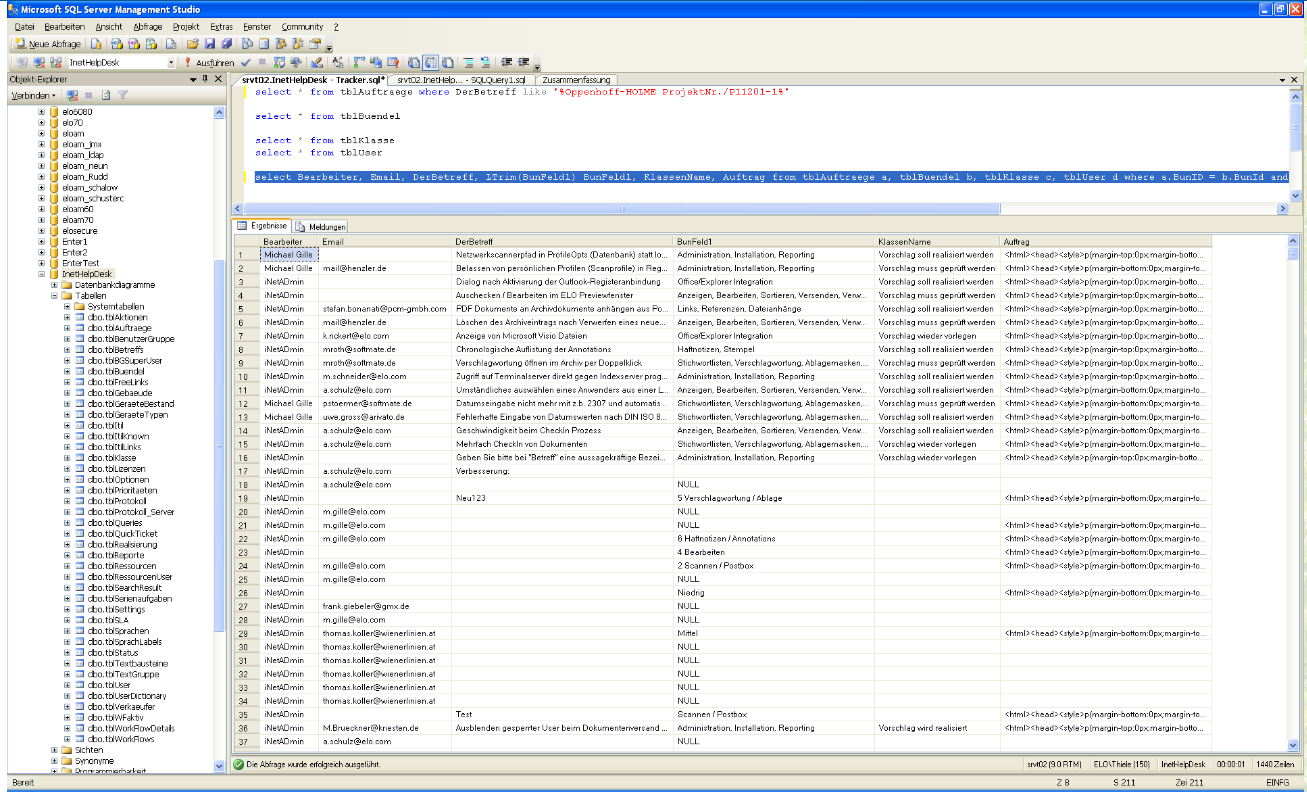


Fig.: SQL database

This is a somewhat extensive `SELECT` statement, which otherwise offers no special features. There is only one point worth mentioning: in the Select list, there is a column titled *LTrim(BunField1) BunField1*. In the database field *BunField1*, the data is partially saved with leading spaces, which we don't want. These are removed with LTrim. However, this means that the column would no longer have a name, which is why the column name is subsequently specified as *BunField1*. This technique should always be used when calculated values are to be used in the Select list.

The complete ruleset will look like this:

```
<ruleset>
    <base>
        <name>ImportTracker</name>
        <search>
            <name>"ETS_COUNT"</name>
            <value>"*"</value>
            <mask>337</mask>
            <max>200</max>
        </search>
        <interval>1H</interval>
    </base>
<rule>
    <name>Rule1</name>
    <condition></condition>
```

```
    <script>

        /* The data for the current folder is
        */ loaded from the database here

        var item = db.getLine(1, "select Editor, Email, TheSubject,
LTrim(BunField1) BunField1, ClassName, Task
from [InetHelpDesk][dbo].tblTasks a,
 [InetHelpDesk].[dbo].tblBatch b,
 [InetHelpDesk].[dbo].tblClass c,
 [InetHelpDesk].[dbo].tblUser d
 where a.BunID = b.BunId                                                       and a.KlaID = c.

        /* ETS_COUNT contains the record number, which is cleared after successful processing. *
        ETS_COUNT = "";

        /* The short name field is completed from the database, please note maximum field length
        NAME = item.DerBetreff;
        if (NAME == "") { NAME = "unknown"; }
        if (NAME.length() > 127) { NAME = NAME.substring(0, 126); }

        // The initiator is populated from the database.
        ETS_MAIL = item.Email;

        /* The subject field was assigned different keywords in the database than in the reposit
        /*  translation table is used here for this reason.
           A column index is used in ELO. */
        var thema = item.BunFeld1;
        if (thema == "Administration, Installation, Reporting") { thema = "Administration¶Instal
        if (thema == "Display, Sort, Edit, Send, Manage, Search") { thema = "Document Editing¶Vi
        if (thema == "Display, Edit, Sort, Send, Manage, Search") { thema = "Document Editing¶Vi
        if (thema == "User Interface, Design, Menus, Navigation") { thema = "Usability¶Interface
        if (thema == "Sticky notes, Stamps") { thema = "Annotations"; }
        if (thema == "Office / Explorer Integration") { thema = "Office Integration¶OS Integrati
        if (thema == "Offline availability") { thema = "Offline"; }
        if (thema == "Links, References, Attachments") { thema = "Links¶References"; }
        if (thema == "Scanning, Intray, Conversion, Printing") { thema = "Scanning¶Intray¶Conver
        if (thema == "Security, Login, Encryption, User rights") { thema = "User rights"; }
        if (thema == "Keyword lists, Metadata, Metadata forms, Versioning") { thema = "Metadata¶
        if (thema == "Workflow, Tasks") { thema = "Workflow¶Tasks"; }
        if (thema == "Interfaces, Scripts") { thema = "Scripting¶Interfaces"; }
        ETS_THEMA = thema;

        ETS_USER = "Product management";
        ETS_STATUS_INT = item.ClassName;
```

```
        EM_WRITE_CHANGED = true;

        //* The database information has now been entered. Only the document still remains unpro
           This will be created as an HTML file with an XML control file
        for the ELO XML Importer.. First, the HTML file is written:*/
        var id = Sord.getId();
        var dataFile = new File("d:\\temp\\trk\\" + id + ".htm");
        Utils.stringToFile(item.Order, dataFile, "ISO-8859-15");


        /* Next, the XML data stream is created.
           As the metadata is entered in the folder, only rudimentary metadata can be found here
        var xmlDesc = NAME.replace("\"", "'").
                           replace("&amp;", "&amp;amp;").
                           replace("<", "&amp;lt;").
                           replace(">", "&amp;gt;");
        var xmlFile = new File("d:\\temp\\trk\\" + id + ".xml");
        var xmlText = "<?xml version=\"1.0\" ?><eloobjlist
                      ver=\"1.0\"><obj><desc value=\"";
        xmlText = xmlText + xmlDesc;
        xmlText = xmlText +
                  "\"/><type value=\"0\"/><destlist><destination
                          type=\"1\" value=\"#";
        xmlText = xmlText + id;
        xmlText = xmlText + "\"/></destlist><docfile name=\"";
        xmlText = xmlText + id;
        xmlText = xmlText + ".htm\"/></obj></eloobjlist>";

        // The XML file is written last.
        Utils.stringToFile(xmlText, xmlFile, "UTF-8");


    </script>
</rule>
<rule>
    <name>Global Error Rule</name>
    <condition>OnError</condition>
    <script></script>
</rule>
</ruleset>
```

After ELOas has entered the metadata the database and created the HTML and XML document files, the ELO XML Importer imports the HTML files into the corresponding folders. This concludes the migration process. Time required for the complete project: about 4 hours.

Example - Treewalk for ELOas

There is a tree walk function available in ELO Automation Services to help process documents. This makes it possible to not only process search areas, but also to run through entire tree structures.

Introduction

Normally, ELOas performs a search for an index field to determine the list of documents to be processed. Alternatively, however, a "tree walk" can also be performed. With this tree walk, individual branches, or even the complete repository can be run through. Each entry is read twice: once a folder is entered, after which all child entries are run through, and then again when the folder is exited.

Example: We will use a filing cabinet metaphor, with the highest level folder titled "cabinet", then "folder", then "folder tab". The cabinet contains folders 1 and 2. Folder 1 contains folder tab 1.1. The following process then results:

```
Cabinet (enter)
Folder 1 (enter)
Folder tab 1.1 (enter)
Folder tab 1.1 (exit)
Folder 1 (exit)
Folder 2 (enter)
Folder 2 (exit)
Cabinet (exit)
```

A script can check whether the ruleset is called in the ascending branch (entering) or in the descending branch (exiting) using the `EM_TREE_STATE` variable. This contains 0 when entering and 1 when exit. Saving is only performed on exit. Changes that are performed upon entering the branch, however, will be retained until it is exited, even if a number of other objects were edited in the meantime.

A treewalk is initiated when the group name of the search index is entered as "TREEWALK", and as a search term the number of the starting node. No rules can be called on the start node. They are only performed on child entries.

Usage example

The following example runs through a branch and sets an internal ID (TrackId) for all objects of form type 6 (Track Item). The starting folder has the ID 3352.

In this simple example, no error handling has been provided, and for this reason the error rule is empty.

```
<ruleset>
    <base>
        <name>Create TrackId</name>
        <search>
            <name>"TREEWALK"</name>
            <value>3352</value>
            <mask>6</mask>
            <max>200</max>
        </search>
        <interval>10M</interval>
    </base>

<rule>
    <name>CreateId</name>
    <script>
        if ((EM_TREE_STATE == 1) &amp;&amp; (EM_ACT_SORD.getMask() == 6)) {
            // Only process TrackItems
            //cnt.createCounter("ETSTrackId", 10000);

            if (ETS_TICK == "") {
                log.debug("Create new TrackId: " + NAME);



                ETS_TICK = cnt.getTrackId("ETSTrackId", "V");
                EM_WRITE_CHANGED = true;
            }
        }
    </script>
</rule>

<rule>
    <name>Global Error Rule</name>
    <condition>OnError</condition>
    <script>
    </script>
</rule>
</ruleset>
```

The interesting part of the ruleset lies in the script area, which for this reason will be discussed for each line individually in the following:

```
if ((EM_TREE_STATE == 1) &amp;&amp; (EM_ACT_SORD.getMask() == 6)) {
```

The script should only be run when exiting the branch (`EM_TREE_STATE == 1`), and only on objects of type `TrackItem` (`EM_ACT_SORD.getMask() == 6`).

```
// Only process TrackItems
//cnt.createCounter("ETSTrackId", 10000);
```

The example uses a counter, which must be created in advance, for example through the command entered above. However, it can only be created once, as otherwise the TrackId will be continually reset.

```
if (ETS_TICK == "") {
```

A TrackId is only created if one does not exist yet (metadata field ETS_TICK is empty).

```
log.debug("Create new TrackId: " + NAME)

ETS_TICK = cnt.getTrackId("ETSTrackId", "V")
```

To create track IDs, there is a practical method in the counter module `cnt: getTrackId(` <CounterName>, <prefix>). This method takes a new counter value and supplements it with the prefix and a checksum. In the example, track ID V10001C2 is created from the counter value 10001.

```
EM_WRITE_CHANGED = true
```

The object is only saved if a new track ID has been created.

```
}
}
```

The ruleset is executed every 10 minutes and passes through the complete track item folder. All entries without a track ID are automatically supplemented, regardless the client they were created with.

Runtime environment variables

When the ruleset is executed, there are a large number of other variables that can be used for processing in addition to the `EM_TREE_STATUS` value.

Name	Content
`EM_TREE_STATUS`	Specifies whether the ruleset is executed in the ascending branch (0) or descending branch (1).
`EM_ACT_SORD`	Contains the SORD object with the current object data.

Name	Content
EM_PARENT_SORD	Contains the SORD object with the data of the parent node. This data can in principle also be changed. However, you have to make sure these changes are saved. In addition, the change must be recognized in the descending branch and the EM_WRITE_CHANGED flag set to true.
EM_ROOT_SORD	Contains the SORD object with the start node. As the ruleset is not applied to this entry, you will have to save your changes manually. This can take place by setting the variable EM_SAVE_TREE_ROOT.
EM_INDEX_LOADED	In contrast to processing after a search, it cannot be assumed with a treewalk that a loaded SORD object has a specific form type. In principle, any form can come up. The preset index variables from the metadata fields can, however, only be generated and filled that have been registered in the definition under <mask\ and under <masks>. In this case, the variable EM_INDEX_LOADED is set to true. If the form is unknown, the metadata fields can only be accessed via the EM_ACT_SORD object; EM_INDEX_LOADED is set to false.
	Information: when the index variables are filled, the metadata fields in EM_ACT_SORD should not be directly edited. These changes will then be lost before saving if the index variables are rewritten.
EM_TREE_LEVEL	With this variable, you can determine where you are within the treewalk (what level). The child entries in the start node are located at level 0 (for the start node, no rules are called).
EM_TREE_MAX_LEVEL	You can set a maximum depth with this rule. Child entries nested deeper than this will be ignored. Normally, this value is set to 32. If it must be changed, it can be set to the desired value before processing in the onstart routine.
EM_SAVE_TREE_ROOT	No rules can be called for the treewalk start node. If this has been changed through access via EM_TREE_ROOT or EM_PARENT_SORD, the variable EM_SAVE_TREE_ROOT must be set to register these changes.
	`<onend>var result = …var oldstate = …EM_SAVE_TREE_ROOT = result != oldstate;log.debug("now save root: " + EM_SAVE_TREE_ROOT);</onend>`
EM_TREE_EVAL_CHILDREN	If a run determines that a subarea should be excluded from processing, the variable EM_TREE_CHILDREN can be set to false. This value will only be evaluated for an ascending branch (with a descending branch, it would have been too late anyway, as the subarea would already have been processed) and it will be initiated for every object set to true (standard behavior: run through the entire subarea).
EM_TREE_ABORT_WALK	If you want to abort a run completely, you can set the flag EM_TREE_ABORT_WALK at any time. In this case, no more child entries are passed through. Additional entries at the same level that have not been processed will also remain unprocessed. This flag can be set to cancel processing after a fatal error.

Name	Content
	Information: In the `onstart` routine, necessary runtime controls can be performed to check whether the treewalk may be performed at all. If not, this flag can be used to cancel the run.

Example - Workflow processing

An extension exists to process workflow tasks. In a workflow, individual person nodes can be created for the ELOas account. If a workflow activates this node, an ELOas "WORKFLOW" search can be used to find and process a list of the active workflow tasks. This can be used to add the metadata and forward the workflow.

The necessary rulesets must be created at the XML level.

Collecting the workflow task list is essentially the same as a normal search. Enter "WORKFLOW" as the metadata field name. The search term itself will be ignored and should remain empty.

```
<base>
    <name>Workflow2</name>
    <search>
        <name>"WORKFLOW"</name>
        <value></value>
        <mask>13</mask>
        <max>1000</max>
    </search>
    <interval>1M</interval>
</base>
```

Even if a search form is not required to collect the list, a search form must still be specified. From the list of tasks, only those workflows will be processed that have this form. This is necessary in order for the index data to be able to be loaded in the local JavaScript variables. If a workflow should be able to use more than one form type, the ruleset must be entered multiple times.

Information

In the list of deadlines, no "FindFirst – FindNext" action will be performed. If there are a number of tasks that will not be processed, this can lead to no new tasks being found for actual processing.

When processing workflows, there are two activities in addition to changing the metadata: forwarding and changing the workflow. The following example shows how the workflow can be influenced depending on the current metadata. In addition, a simple approval workflow will be examined, for which the person processing it is not clear at the start. This person will be entered in the course of the workflow to the *PROCESSOR* index field by the mailroom department. In the template, the processor node is first initialized with *Owner*. The correct value is read from the *PROCESSOR* index field at runtime, then entered to the node. ELOas then runs under the ELO name *elowf* with a person node between the mailroom and the claims processor.

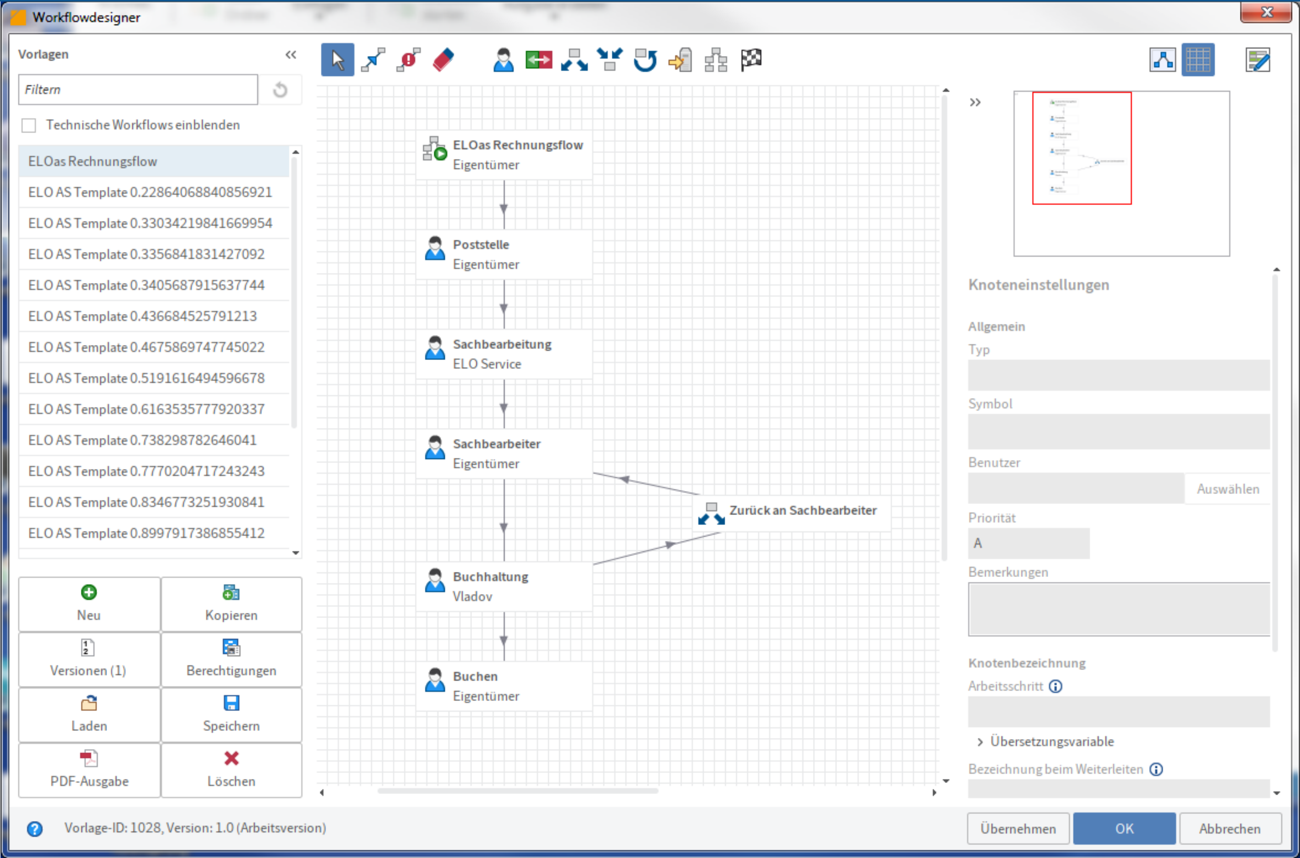


Fig.: Workflow designer, workflow template with ELOas

When the workflow arrives at ELOas, the mailroom will have defined the claims processor. ELOas reads the PROCESSOR metadata field and enters the value to the *Processor* successor node. This takes place through the following simple rule:

```
<rule>
    <name>Expand Name</name>
    <condition></condition>
    <script>
        log.debug("Process WF: " + NAME);


        wf.changeNodeUser("Employee", EMPLOYEE);

        EM_WF_NEXT = "0";
    </script>
</rule>
```

Changing the ELO user name is performed by the `wf.changeNodeUser` command. Enter the workflow name as the first parameter and the ELO user name as the second. The library *wf* takes care of the rest (locking the workflow, searching for the node, refreshing users, saving the workflow, releasing the lock).

After the user name has been set, the workflow has to be forwarded. This takes place by setting the variable `EM_WF_NEXT`. If left empty, nothing will be forwarded. The task remains the same (which should not remain as is forever, as at some point the list of deadlines will overflow). Once all conditions for forwarding have been met, then either the connection number or the name of the successor node can be specified. If there is only one successor, then the connection number can be entered: `EM_WF_NEXT = "0";`.

If there are multiple successors, the name of the successor node should be entered instead. It will also be assumed that the process will automatically be booked after the person processing it has forwarded it, meaning the accounting node will also be transferred to ELOas. This runs a script that audits the accounting data. If the script runs successfully, the function `ERPverify()` will return true and the workflow is forwarded to the *Charge account* node. If an error occurs, the workflow is returned to the claims processor. The script could then look like the following:

```
If (ERPverify()) {
    EM_WF_NEXT = "Book";
}   else {
    EM_WF_NEXT = "Employee";
}
```

Filing via ELO Dropzone

ELOas filing via ELO Dropzone tiles

ELOas 20.0 enables automatic filing via ELO Dropzone tiles. In an ELO Dropzone tile, you can define metadata for new documents. The following step-by-step guide will explain how to configure automatic filing.

> **Please note**
>
> Automatic filing via ELO Dropzone requires the latest ELOas standard libraries. You will find them on the official download page *http://www.forum.elo.com/script/20/eloinst.html*. You also need an ELO XML Importer license to automatically file documents using a Dropzone tile.

Step by step

1. Create the individual ELO Dropzone tiles using the ELO Dropzone module. The tiles are stored in *// Administration // Dropzone*. The tile definition is saved in the entry's extra text.

2. The "ELOas Base" folder contains a "Tiles" child folder. The required ELO Dropzone tiles are referenced to this folder. ELOas does not differentiate between personal and global tiles.

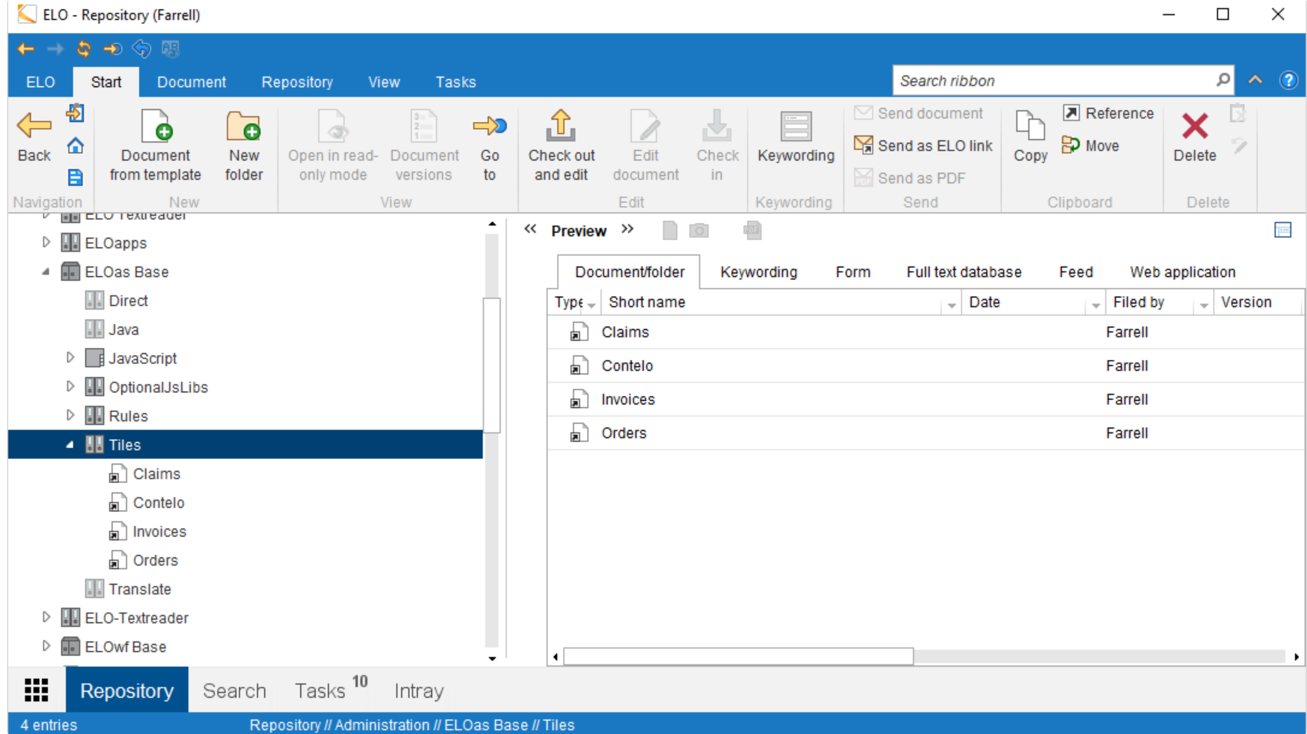


Fig.: Folder with referenced ELO Dropzone tiles

1.

A monitored directory is defined in the ELOas configuration file "config.xml".

```
<entry key="monitordir">C:\temp\ELOAsMonitor</entry>
```

One child folder is expected for each tile. Existing tiles must have a unique name for automatic filing to work.

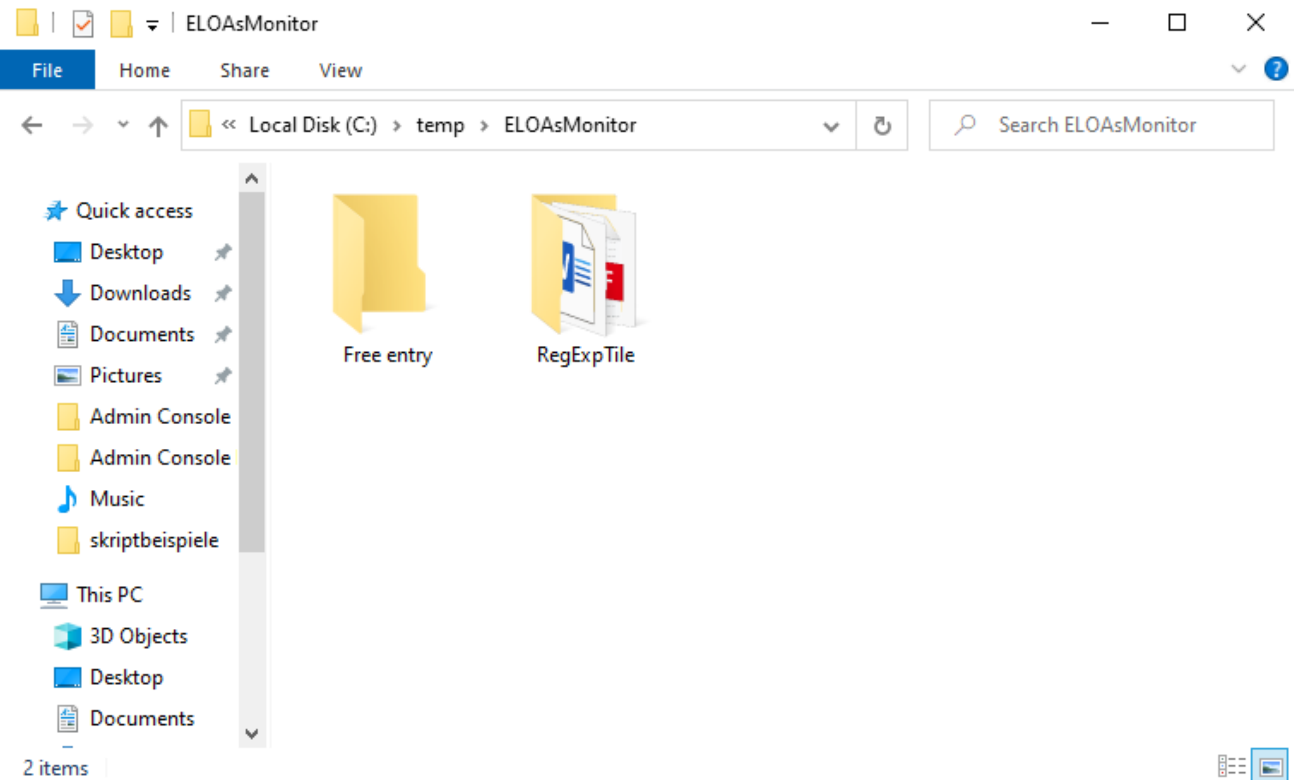


Fig.: Monitored directory

The files that are later automatically transferred to the repository by ELOas based on the tile definition land in these individual child folders.

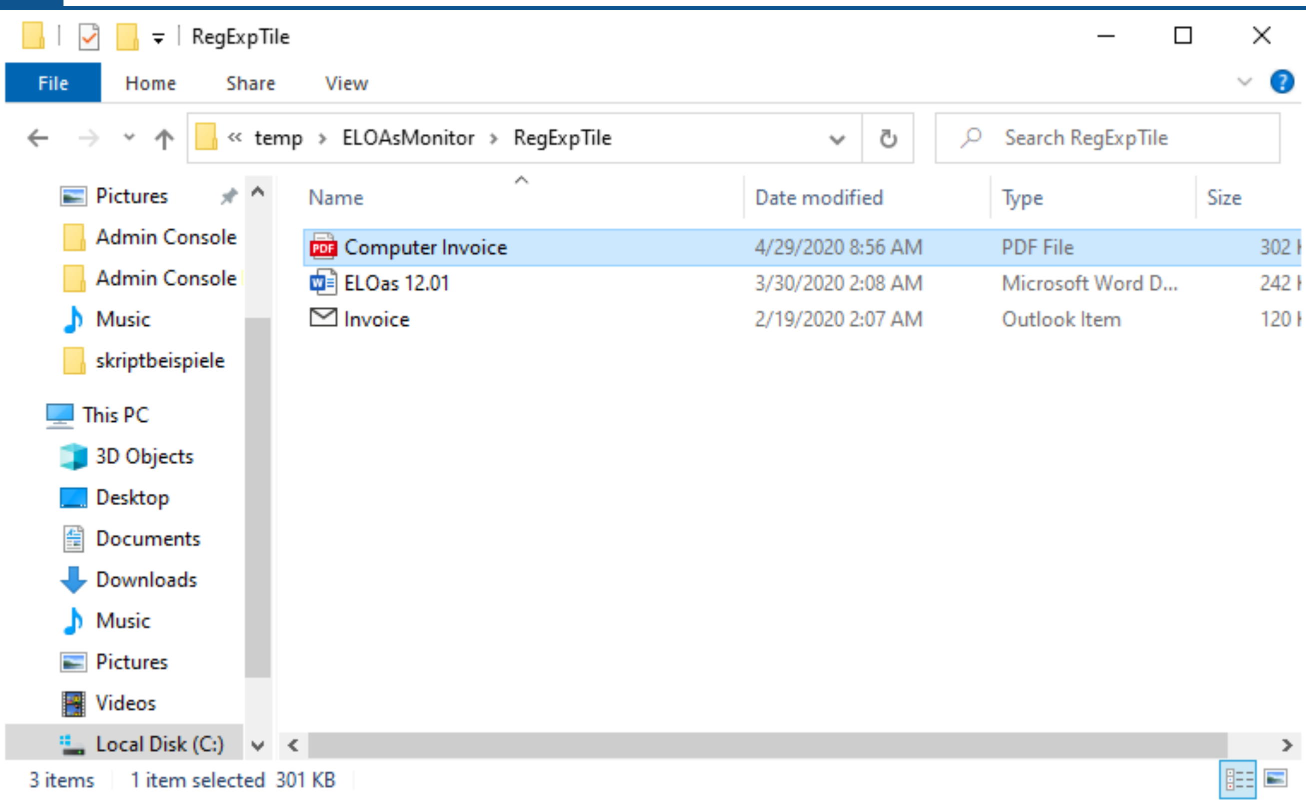


Fig.: Individual documents in the child folder before automatic filing

1. Regular expressions are configured in the metadata for the "Tiles" folder. These are then available for all tiles.

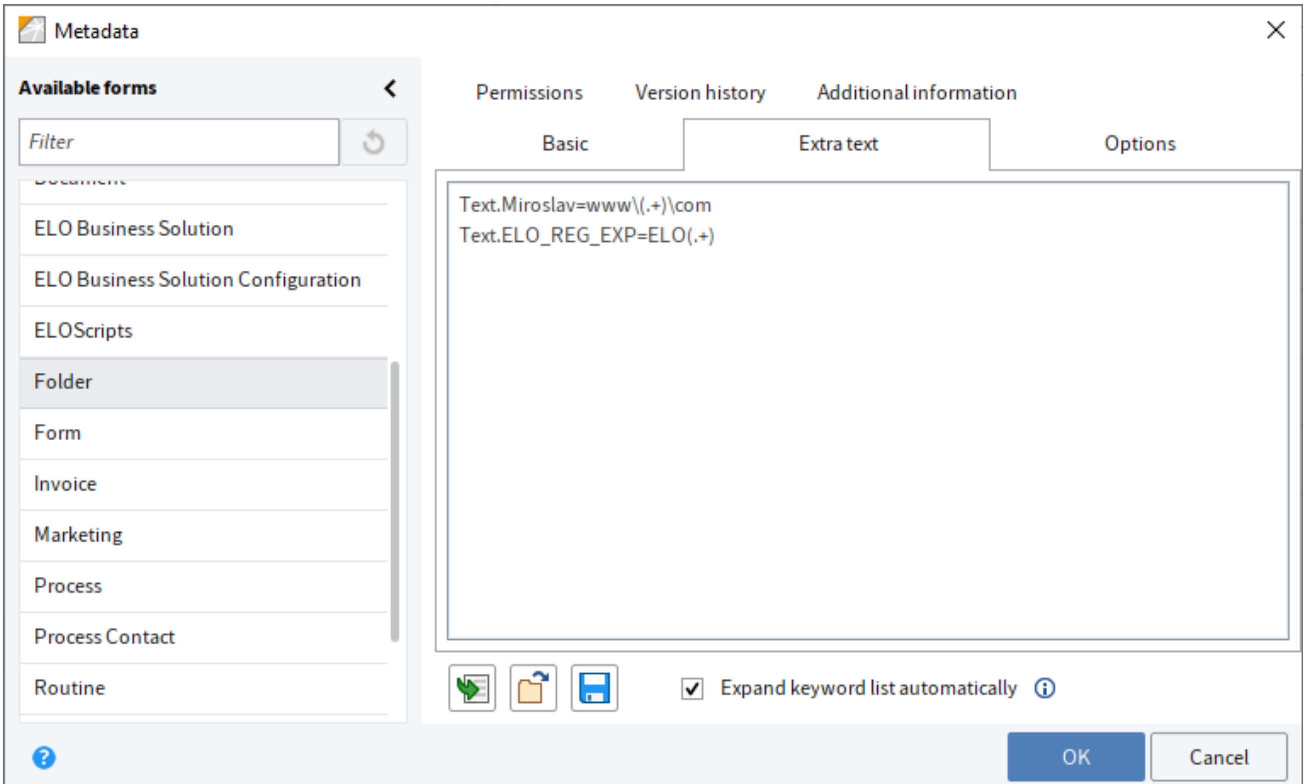


Fig.: Regular expression in the metadata for the 'Tiles' folder

1. With ELOas 11.0, a new rule type has been introduced for filing via an ELO Dropzone tile. The "<name>" section contains the "TILE" value and the "<value>" section contains the name of the referenced ELO Dropzone tile.

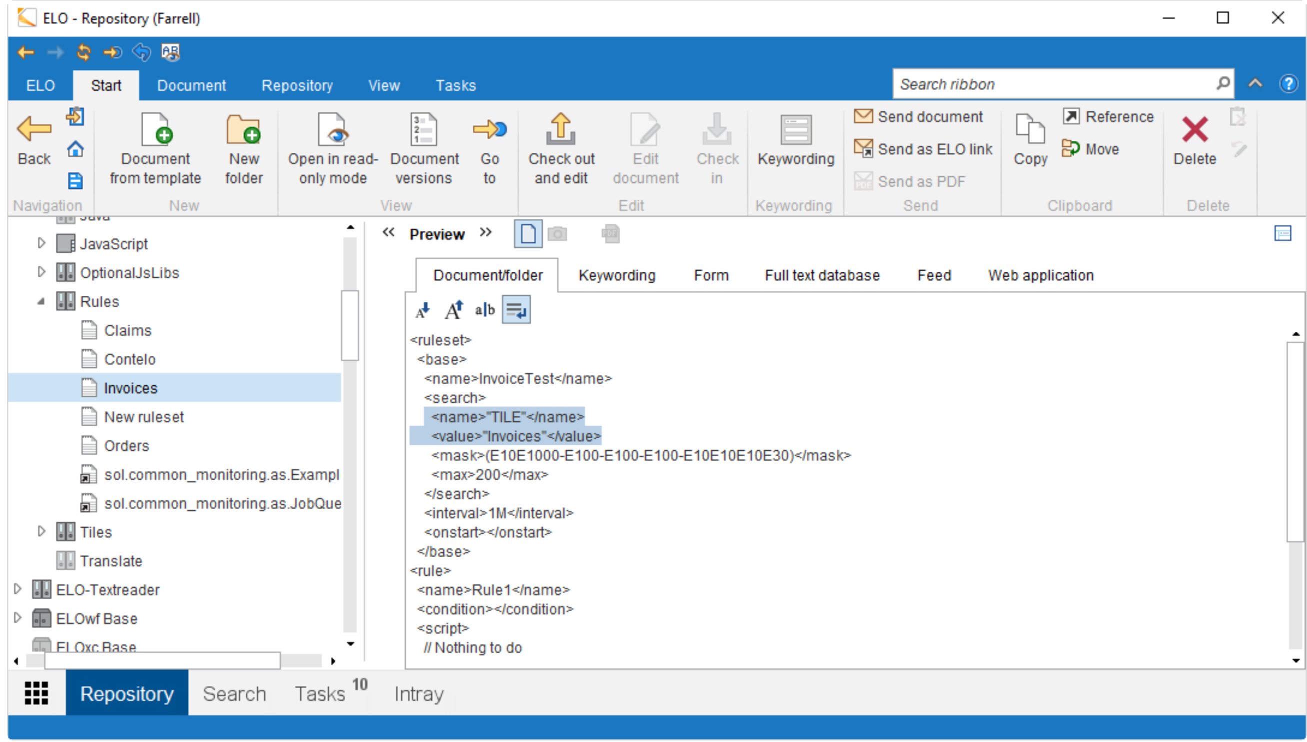


Fig.: ELOas rule for filing via ELO Dropzone

At defined intervals, ELOas checks the monitored tile directory to see whether it contains new documents to be filed. If this is the case, it files them. Existing folders are skipped. In a tile, you can define that local documents should be deleted after they are filed to the repository. If errors occur during filing, the problematic files are moved to a child folder named "Errors" so that ELOas does not repeatedly process them.

Barcode

Introduction

ELO Automation Services offer a utility class `ELOasUtils` with functions for reading and writing barcodes. The Softek library and ZXing library can be used for barcode functionality.

> **Information**
>
> For more information on the Softek library, go to: *http://www.bardecode.com/en1/app/barcode-reader-toolkit-for-windows/*.
>
> Further information on the ZXing library can be found at: *http://zxing.github.io/zxing/apidocs*.

Reading barcodes with the Softek library

From ELOas version 10 onward, you can read barcodes using the Softek library. Barcodes are recognized via the Softek barcode DLLs "SoftekBarcodeDLL.dll" or "SoftekBarcode64DLL.dll", depending on the operating system (32/64-bit). Barcode recognition is used in an ELOas script as follows:

```
var barcodeReader = Packages.de.elo.mover.utils.ELOAsUtils.
                    createBarcodeReader2(emConnect);
var barcodeFile = new File("C://temp//BarcodeFile.tif");
var barcodeCount = barcodeReader.ScanBarCode(barcodeFile.getPath());
log.info("barcodeCount=" + barcodeCount);
var barcodeDescr = barcodeReader.GetBarString(barcodeCount);
log.info("barcodeDescr=" + barcodeDescr);
```

Barcodes formats in the Softek library

Refer to the official Softek library documentation for supported barcode formats. The following formats are supported:

- Codabar 1D
- Code 128 1D
- Code 2 of 5 Datalogic 1D
- Code 2 of 5 Iata1 1D
- Code 2 of 5 Iata2 1D
- Code 2 of 5 Industrial 1D
- Code 2 of 5 Interleaved 1D
- Code 2 of 5 Matrix 1D
- Code 3 of 9 1 D
- Code 3 of 9 Extended 1D
- Code 93 1D
- EAN-8 1D
- EAN-13 1D
- GS1-128, UCC-128, EAN-128 1D
- GS1-Databar 2D
- Patch Code Symbols 1D
- UPC-A 1D
- UPC-E 1D
- QR-Code 2D
- Data Matrix ECC200 2D
- Micro-PDF-417 2D
- PDF417

Example for reading a QR code

You can enable the function to read QR codes with the ELOas instruction:

```
"barcodeReader.setReadQrCode(1);"
```

Information

The methods of the `BarcodeReader` class are described in the official *ELOas JavaDoc*.

Reading barcodes with the ZXing library

Barcodes are read via the static `getBarcode` method of the ELOas class `ELOAsUtils`. In this method, the file, the file page with the barcode, and the barcode configuration are transferred as parameters.

```
String barcode = ELOAsUtils.getBarcode(IXconnect ixConnect, File file,
                                        int page, String barcodeConfig);
```

The individual settings in the barcode configuration are separated by a pipe symbol. An example configuration could look like this.

Example

```
String barcodeConfig = "POSSIBLE_FORMATS:CODE_128,QR_CODE|
                        CHARACTER_SET:UTF8|ALLOWED_EAN_EXTENSIONS:2,5|
                        PURE_BARCODE:TRUE|RETURN_CODABAR_START_END:TRUE|
                        ASSUME_CODE_39_CHECK_DIGIT:TRUE|TRY_HARDER:TRUE";
```

In addition, the method `getBarcodeResult` is available in the class `ELOAsUtils`. This method returns the entire barcode result.

Example

```
Result barcodeResult = ELOAsUtils.getBarcodeResult(IXconnect ixConnect,
                                                    File file,
                                                    int page,
                                                    String barcodeConfig);
```

Barcodes formats in the ZXing library

Refer to the official ZXing library documentation for supported barcode formats. The individual formats are listed in the ZXing class `BarcodeFormat`. Further information can be found at: http://zxing.github.io/zxing/apidocs/.

The following formats are supported:

- Aztec 2D
- CODABAR 1D
- Code 128 1D
- Code 39 1D
- Code 93 1D
- Data Matrix 2D
- EAN-13 1D
-

EAN-8 1D
- ITF (Interleaved Two of Five) 1D
- MaxiCode 2D
- PDF417
- QR Code 2D
- RSS 14
- RSS EXPANDED
- UPC-A 1D
- UPC-E 1D
- UPC/EAN extension

Creating barcodes with the ZXing library

Creating barcodes on a document page takes place via the `writeBarcode` method. In this method, the target file, the barcode text, the size of the barcode, and the barcode configuration are transferred as parameters.

```
ELOasUtils.writeBarcode(IXconnect ixConnect, File targetFile,
                        String barcodeText, int width, int height,
                        String barcodeConfig)
```

Sample configuration

```
String barcodeConfig = "AZTEC_LAYERS:13|CHARACTER_SET:UTF8|
                       DATA_MATRIX_SHAPE:FORCE_RECTANGLE|
                       ERROR_CORRECTION:M|MARGIN:20|PDF417_COMPACT:TRUE|
                       PDF417_COMPACTION:NUMERIC|PDF417_DIMENSIONS:5,10,5,10";
```

A call to a static ELOas method in an ELOas rule looks like this:

```
var result = Packages.de.elo.mover.utils.ELOAsUtils.
                getBarcode(emConnect,barcodeFile, 1, barcodeConfig);
```

This enables you to use barcode information in ELO Automation Services.

Debugger

ELOas debugger

Searching for errors in an extensive ruleset can require a great deal of time and effort. The JavaScript must be adjusted for each pass. To do so, you must check out the document, edit it, check it back in, and then click *Reload.* Further, the use of the Rhino debugger under Apache Tomcat is problematic in a Windows 7 environment. You can save yourself at least this and the checkout/check-in process by using the ELOas debugger.

Information

For more information on the ELOas debugger, refer to the "ELO Automation Services Debugger (Java FX)" documentation.

Opening the program

The ELOas debugger 20 comes with OpenJDK 13. The debugger uses the supplied Java Runtime Environment and is started via the file "ELOAsDebug.bat" with the following content:

```
.\jdk-13.0.2\bin\java.exe -Xmx1000m --module-path=.\lib\modules

        --add-modules javafx.controls,javafx.base,javafx.graphics,

                javafx.web,javafx.swing
                        -classpath ".\*;.\lib\*"

                                de.elo.mover.eloasdbg.javafx.StartEloAs
```

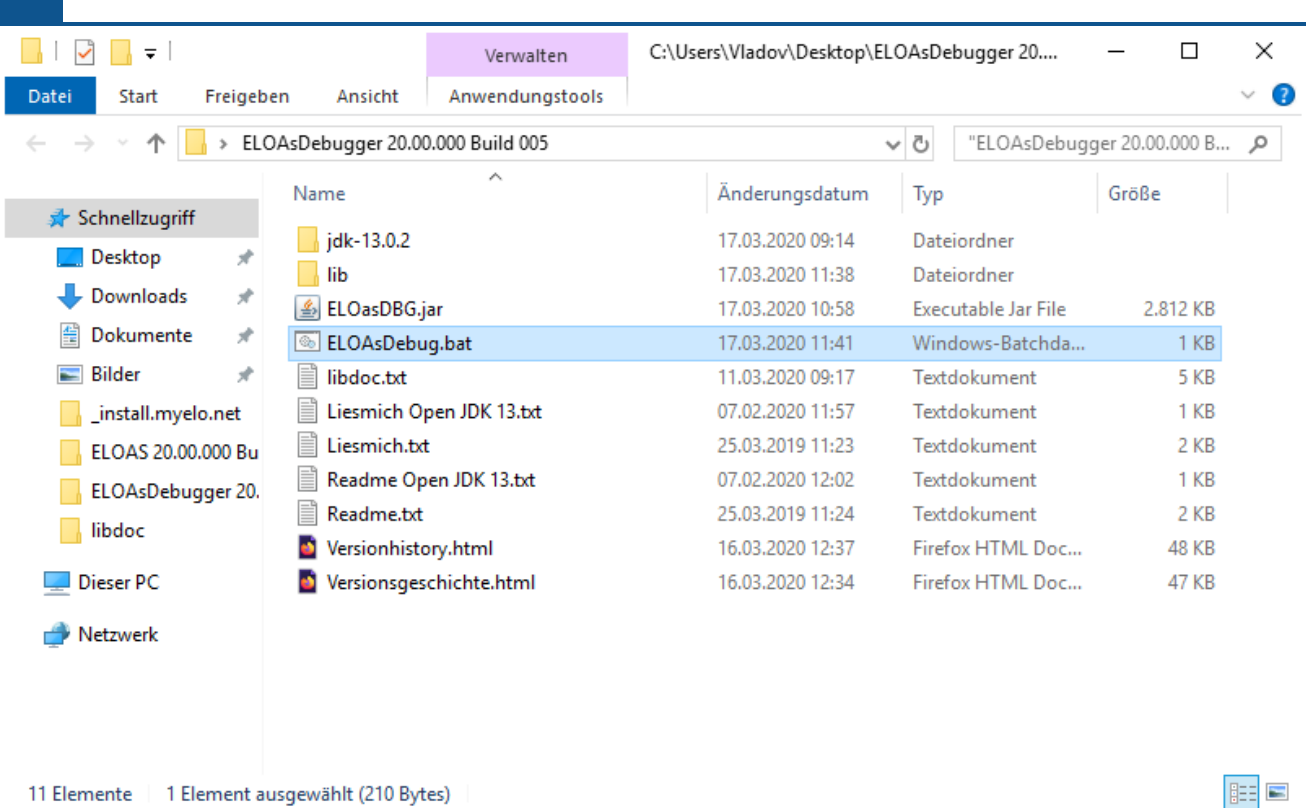


Fig.: 'ELOAsDebug.bat' file in the Windows file system

Configuration

Click the *Config* button. This will take you to the configuration dialog box.

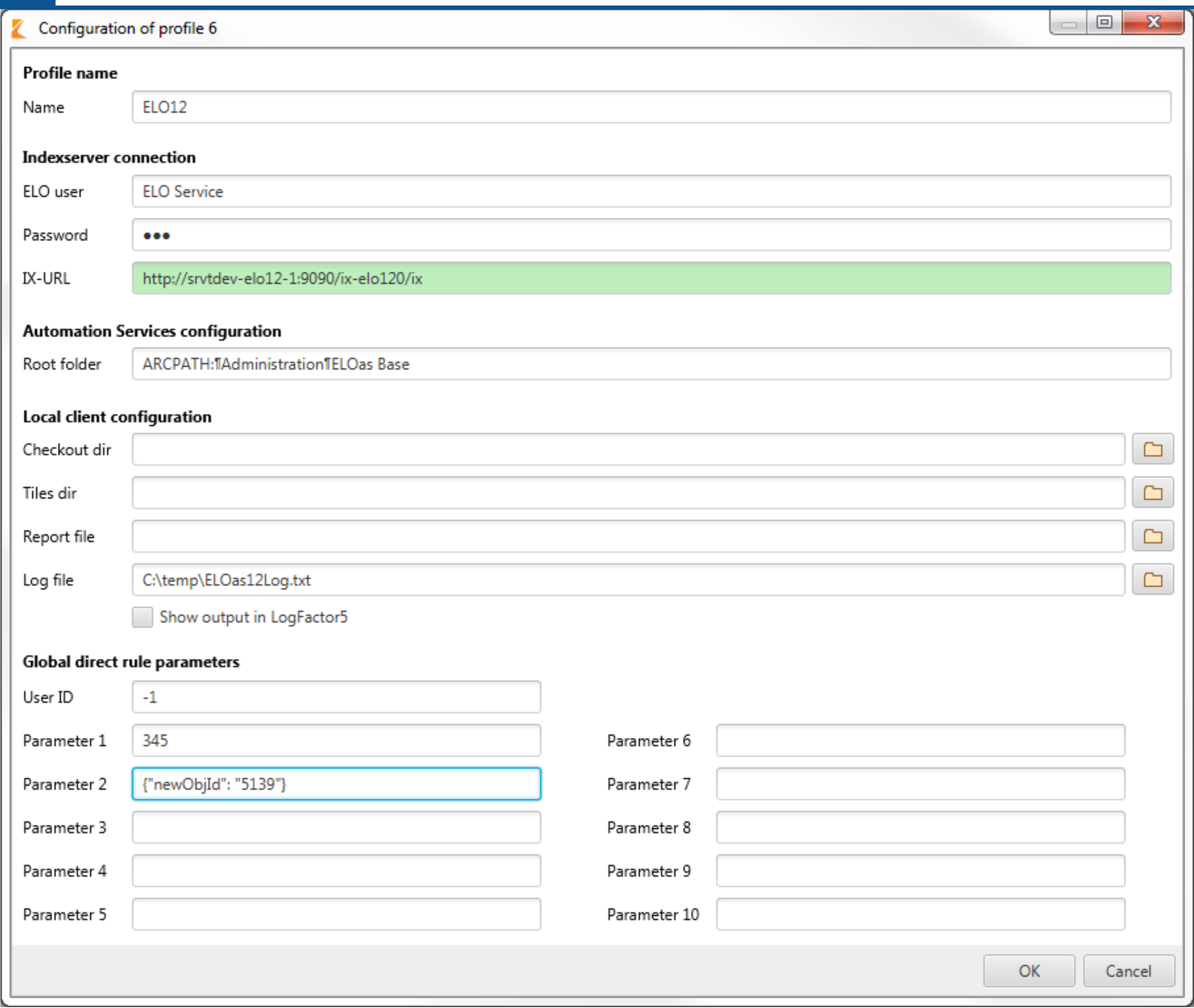


Fig.: Debugger configuration

The title of the dialog box displays the ID of the ELOas debugger profile being edited. The user password is hidden in the user password text field.

Name: The profile name may contain a maximum of 15 characters.

ELO user: The name of the ELO user.

Password: The password for the ELO Indexserver connection.

IX-URL: The URL of the ELO Indexserver. The text field contains a green background when the ELO Indexserver is available at the specified URL.

Root folder: The path where the ELOas configuration is saved.

Checkout dir: Clicking the button next to the *Checkout dir* field allows you to select the ELO Java Client checkout directory.

Tiles dir: Clicking the button next to *Tiles dir* allows you to select the monitored directory for the referenced ELO Dropzone tiles.

Report file: Clicking the button next to the *Report file* field allows you to select an ELOas debugger report file.

Log file: Clicking the button next to the *Log file* field allows you to select the log file.

Global direct rule parameters: Here, you can configure the global parameters for direct ELOas rules. You can edit the user ID and ten parameters.

The profile configuration dialog box has a scroll bar that is shown when the dialog box is reduced beyond a certain size.

Click the OK button to save your changes in the system registry. The settings for the current ELOas debugger profile (ID: 1) are saved to the following location in the system registry:

"HKEY_CURRENT_USER\Software\JavaSoft\Prefs\elo digital office\eloas.1".

Click *Cancel* to discard your changes and close the dialog box. You can also press the ESC key to close the ELOas debugger profile configuration dialog box. The dialog box has a minimum size setting. When you enlarge the dialog box, the individual components of the dialog box are enlarged proportionally. This allows you to display long profile inputs.

Editing a ruleset

After starting, all rulesets for the current configuration are loaded automatically. However, they are not run right away, allowing you to enter breakpoints to the JavaScript code.

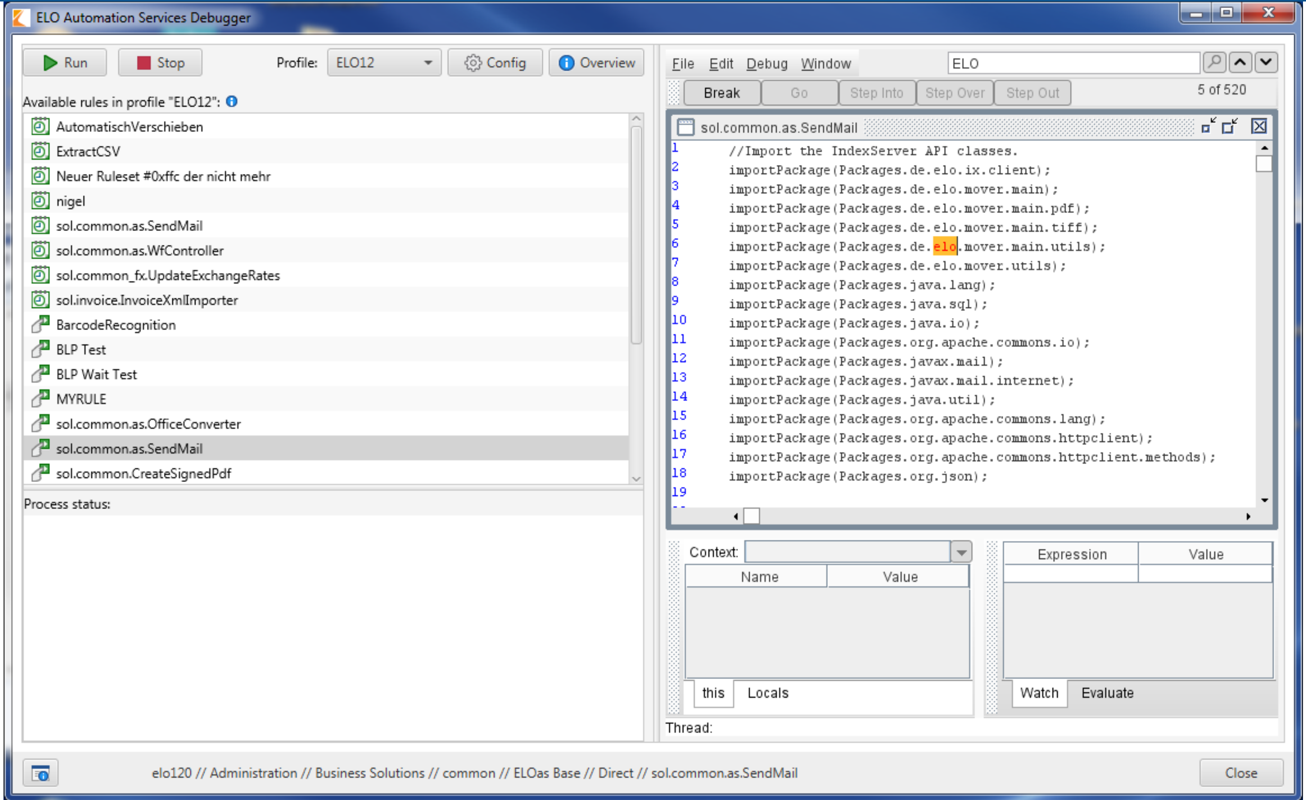


Fig.: ELOas Automation Services, edit ruleset

If you have multiple rulesets, please ensure that you have selected the correct one in the Rhino debugger window under *Window*. Now you can set breakpoints wherever you wish and start the ruleset.

Start a ruleset by clicking the corresponding ruleset entry in the list and then clicking *Run*. Please note that the ruleset is now activated, but will naturally still be subject to the interval control. If you have set the ruleset start for midnight, it will also only become active in the debugger at this point in time. For debugging purposes, the setting "1M" - i.e. once a minute - is a good setting for recurring rulesets, and "10H" - i.e. every 10 hours - good for rulesets that should be run once.

If you want to edit a ruleset or a JavaScript file, then you can check them out or call the already checked out file directly in a text editor of your choice. Perform your desired changes and then save the data. As long as the editor does not open the file exclusively (which is somewhat uncommon for text editors), you will not need to close the editor as well. Simply click *Run* in the debugger again. The ruleset will now be automatically reloaded and restarted from the repository and the checkout directory. A new log file is created so you do not have to deal with old logs.

Debugger (Java FX)

Opening the program

The ELOas debugger starts via the file EloAsDebug.exe and requires a computer with JRE 1.7 or higher installed.

You can also run the ELOas debugger via the command line using the following command:

```
"C:\Program Files\Java\jre1.8.0_152\bin\javaw.exe"
        -classpath ".\*;lib\*" de.elo.mover.eloasdbg.javafx.StartEloAs
```

Use with OpenJDK

The ELOas debugger can also be used with OpenJDK. The ELOas debugger 20 comes as a complete package with OpenJDK 13. Java no longer has to be installed separately.

In the program directory of the ELOas debugger, click the file "ELOasDebug.bat".

Displaying the debugger on high-resolution screens

For the Rhino debugger embedded in the ELOas debugger to be displayed properly with a horizontal resolution of 4000 pixels (4K) on Windows 10 (from version 1703), you have to configure the option for scaling the application. Follow the steps below to configure this option:

1. On the file system, right-click the file "EloAsDebug.exe" and select the "Properties" menu item.
2. Open the tab "Compatibility".
3. Click the "Change high DPI settings" button.
4. Check the box next to "Overwrite behavior for high DPI scaling".
5. Select "System" from the drop-down menu.
6. Click *OK* to save your changes.

ELOas debugger 20 includes the Rhino scripting engine "rhino-1.7.12.jar".

User interface

The ELOas debugger user interface looks like this:

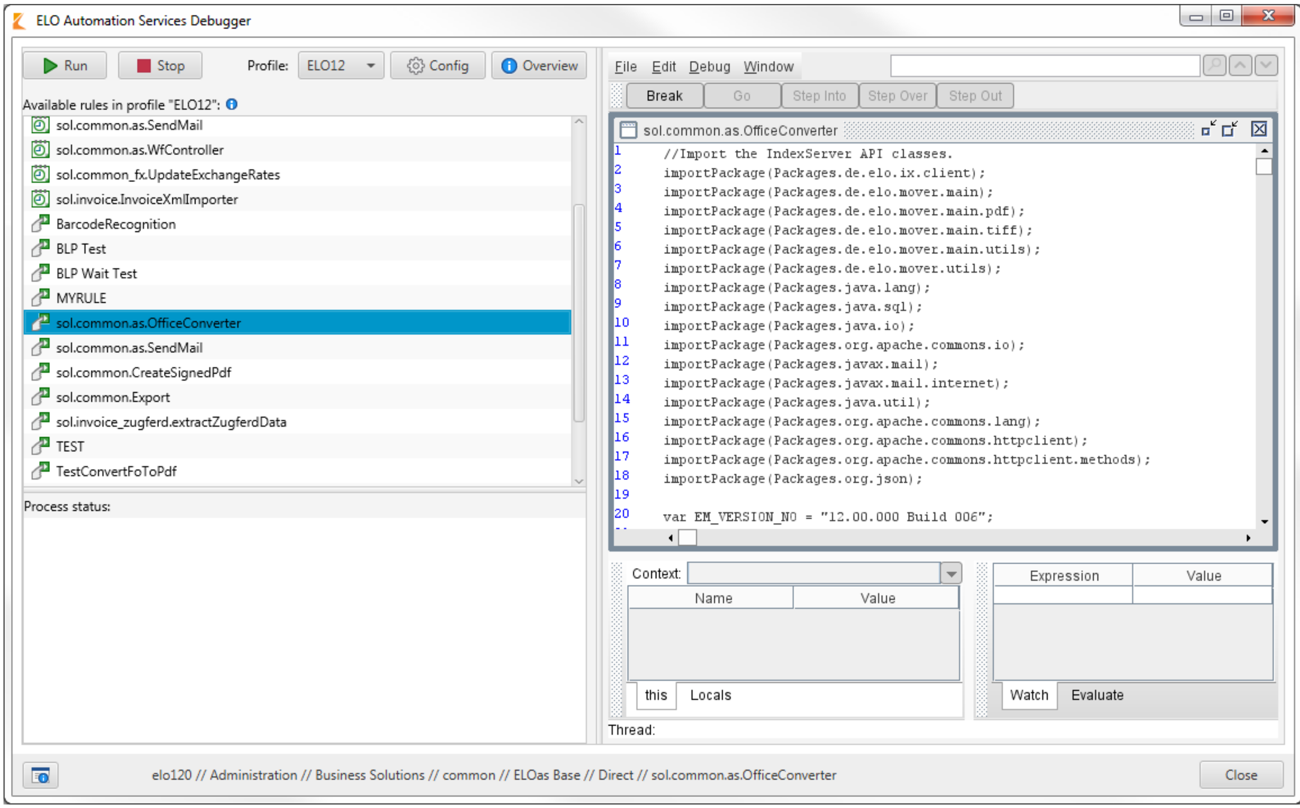


Fig.: ELOas debugger user interface

When you start the ELOas debugger, the first rule is selected by default. The rule contents are shown on the right-hand side.

> **Information**
>
> The ELOas debugger is only available in English.

The size and position of the individual program dialog boxes are saved in the registry and then restored the next time the program starts. The dimensions are saved for the main ELOas debugger window, the *LogFactor5* window, the profiles overview dialog box, the profile and parameter configuration windows, and the *About this program* dialog box.

The column arrangement in the *LogFactor5* dialog box is also saved to the registry and restored the next time the program starts.

The ELOas debugger contains a split bar between the list of existing rulesets and the status area. The split bar position is saved to the registry and restored the next time the program starts.

Searching rule contents

The rule contents are shown on the right-hand side of the ELOas debugger interface. You can search the contents using the search field located above this space.

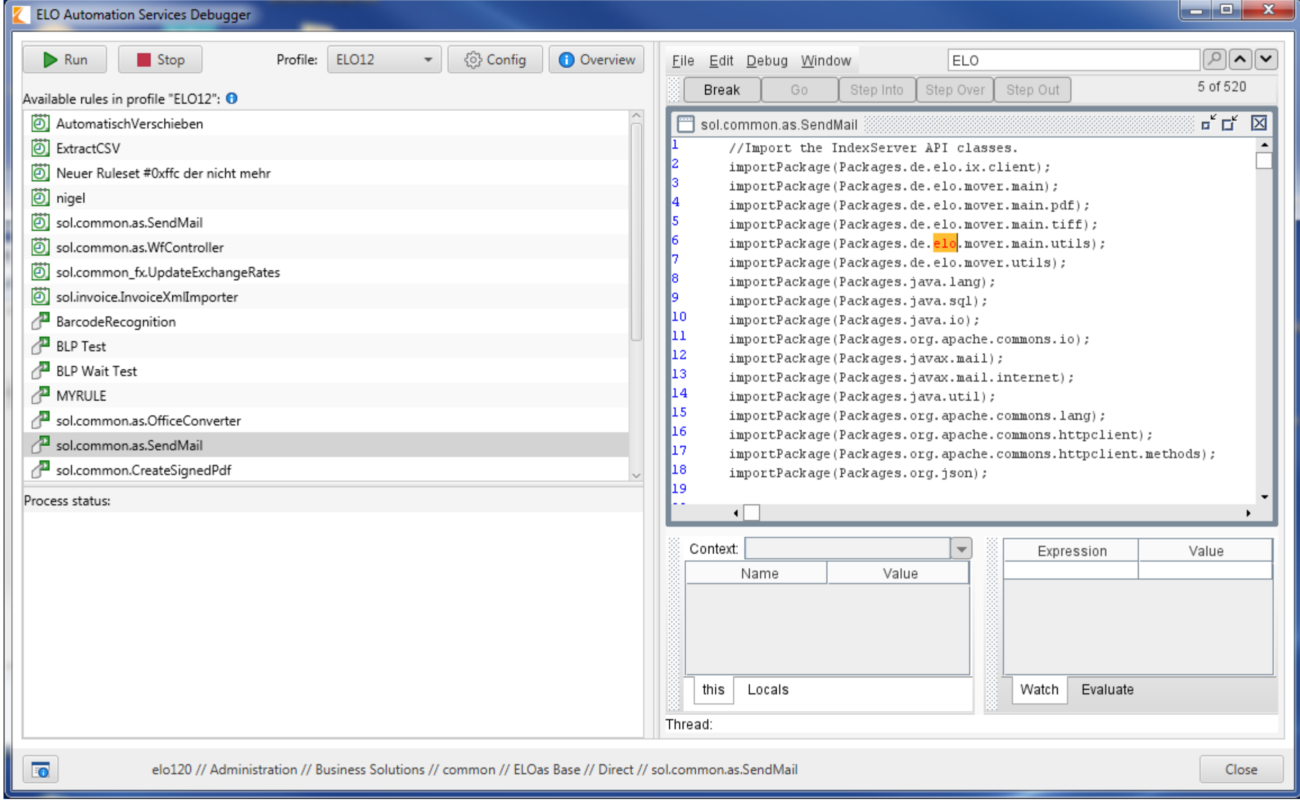


Fig.: Searching the rule contents

After entering your search term to the search field, it will be highlighted in the rule contents if found.

Additional options are available in the search field context menu.

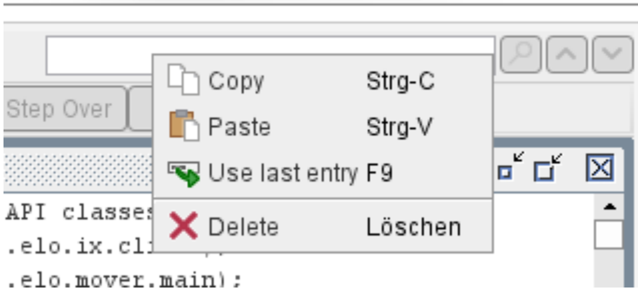


Fig.: Search field context menu

Copy copies the text to the clipboard and *Paste* inserts your text from the clipboard. Clicking *Use last entry* enters the last search term to the search field. Clicking *Delete* removes the search term.

Status reports

The main window of the ELOas debugger features a multi-line text field for background process status reports. This field cannot be edited, but its contents can be copied to the clipboard.

Program information

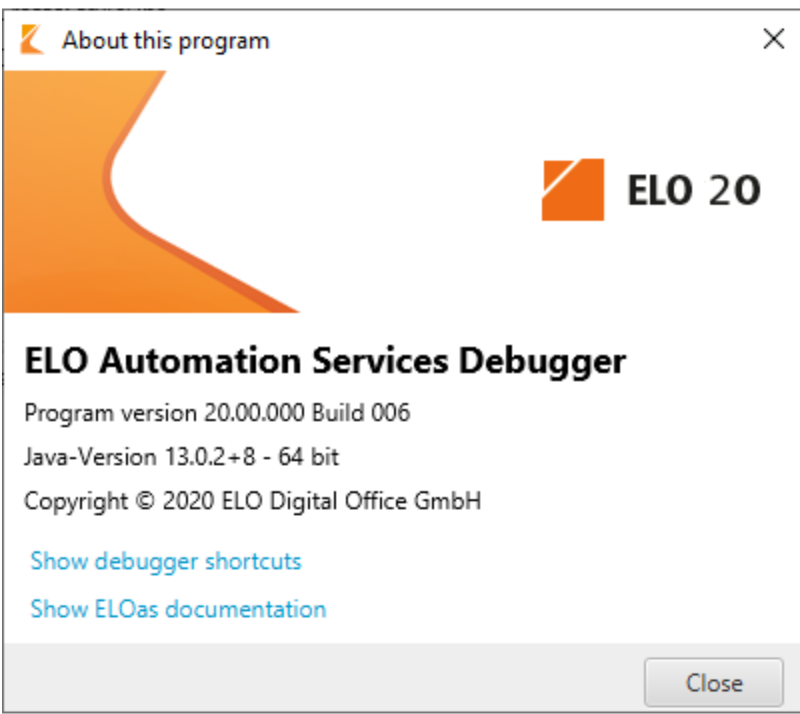


Fig.: 'About this program' dialog box

The lower left of the dialog box contains a button for program information. Clicking the button opens the 'About this program' dialog box, where you will find the program version and Java version. You will also find links to the keyboard shortcuts for the ELOas debugger and ELOas documentation. Click *Close* to exit this dialog box.

Starting an ELOas rule

Click *Run* to begin debugging an ELOas rule. The debug process is run in the embedded Rhino debugger. If the option for LF5 output is enabled in the current ELOas debugger profile, this program also starts in a separate window. This window displays the ELOas debugger outputs. The different log levels can be marked in a suitable color, as needed. Click *Stop* to stop debugging an ELOas rule. The buttons for starting and stopping a rule are always active in the current version of the ELOas debugger.

If you click Run without selecting an ELOas rule first, the following warning appears:

Fig.: Error message when starting debugging without selecting an ELOas rule

Profiles

Using ELOas debugger profiles

The upper right area of the main ELOas debugger window contains a drop-down menu with the ELOas debugger profiles. This drop-down menu contains 10 profiles. Each profile can be uniquely identified based on the profile name.

When you select a profile, the relevant graphic dialog components are refreshed. The list of existing ELOas rules is also refreshed when you change profiles. This list displays the rules from the ELOas debugger profile currently in use. Double-click an ELOas rule to start the debug process for that rule.

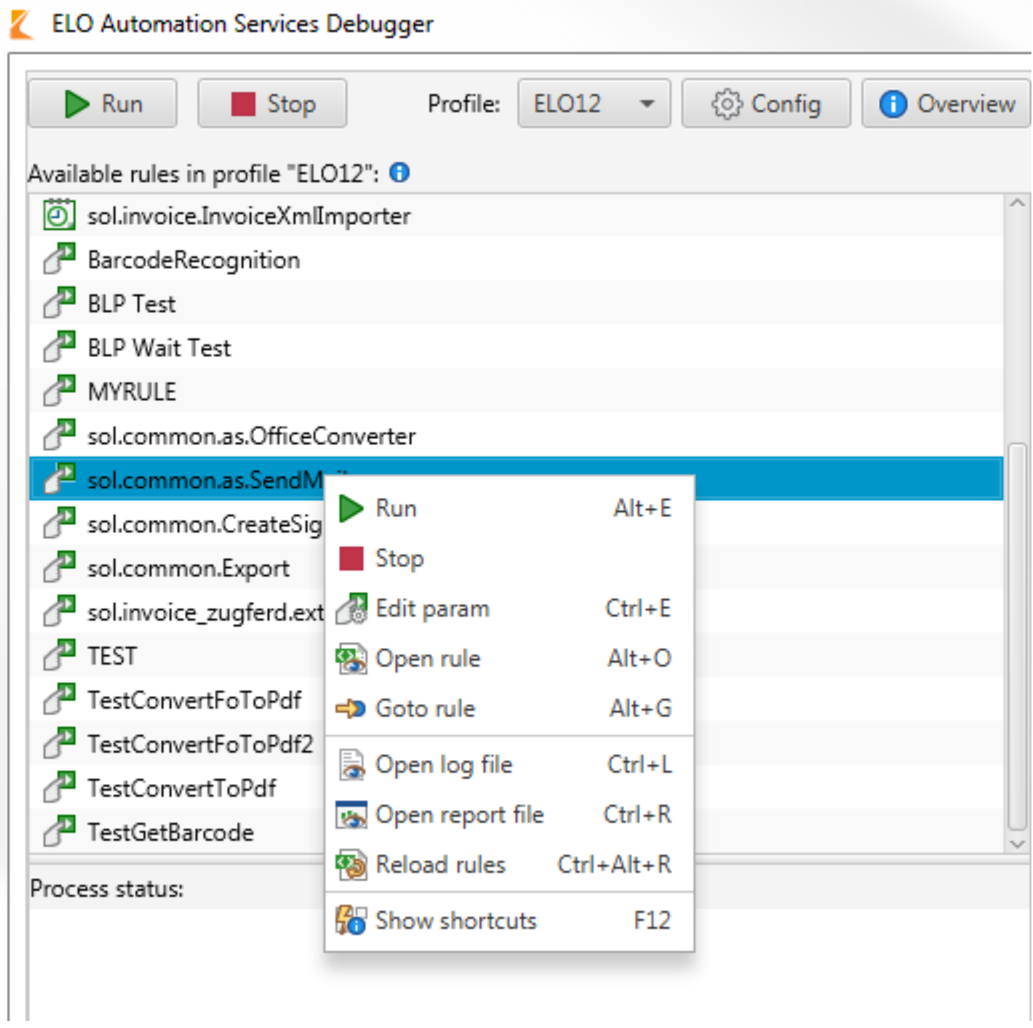


Fig.: Context menu in the ELOas debugger

Right-click a rule to open the list context menu. The following actions are available in the context menu:

Run: Start the debug process for the selected ELOas rule.

Stop: Stop the debug process.

Edit param: Modify the rule parameters. The specific parameters for the direct rule are applied instead of the global rule.

Open rule: Open the rule as a text file.

Goto rule: Open the filing location of the rule in ELO.

Open log file: Open the configured ELOas debugger log file.

Open report file: Open the configured ELOas debugger report file.

Reload rules: Reload the existing ELOas rules.

Show shortcuts: Shows the keyboard shortcuts in the ELOas debugger.

The list of available rulesets also shows the direct rulesets. The individual rulesets are distinguished from one another based on their icon. The type icon of a ruleset contains a corresponding description text.

Profile overview

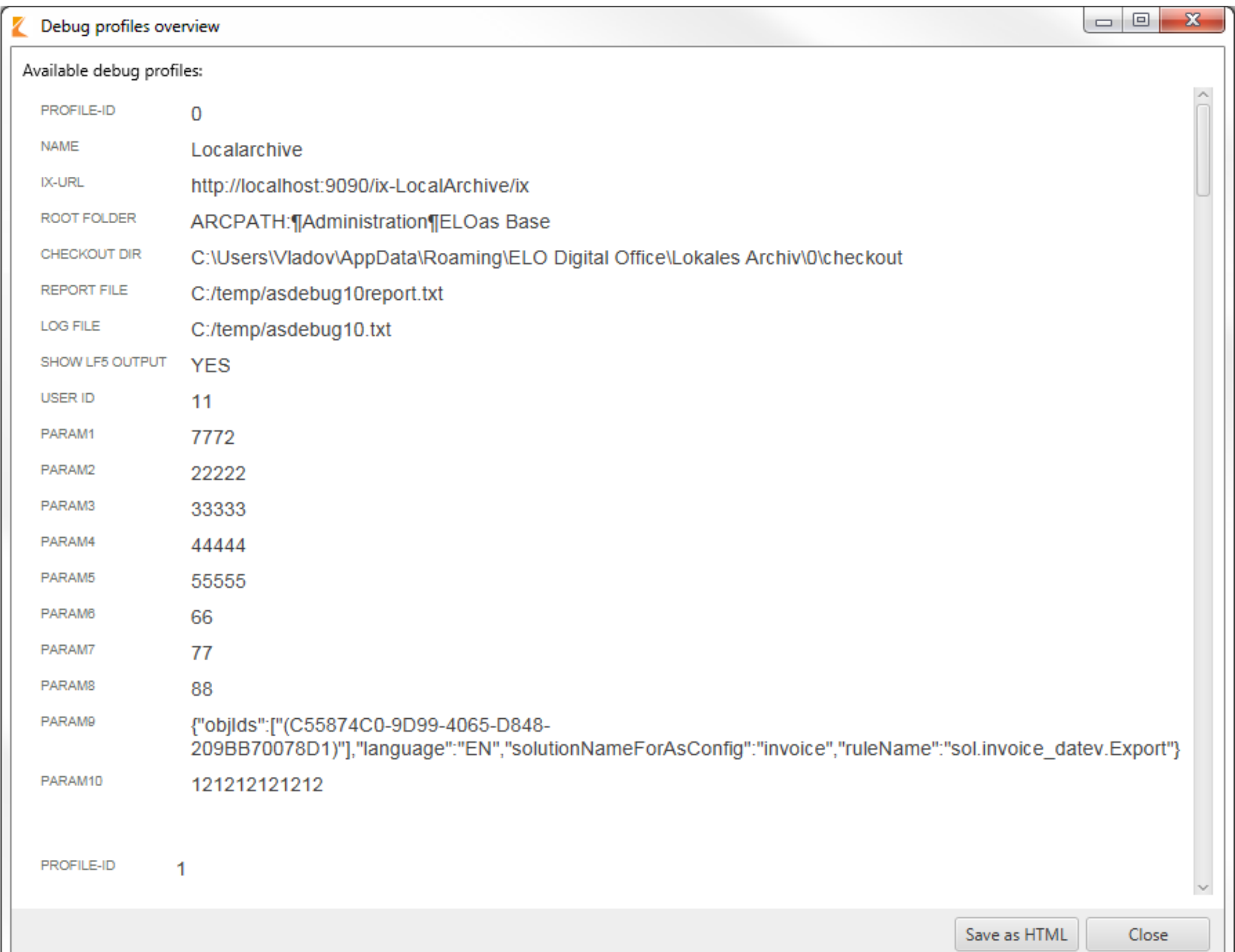


Fig.: ELOas debugger profile overview

Click *Overview* to open the overview of existing ELOas debugger profiles. The existing profiles are shown on an HTML page. The overview shows the most important properties of a profile. Save the profile overview to the local file system as an HTML file by clicking *Save as HTML*.

Information on the selected profile

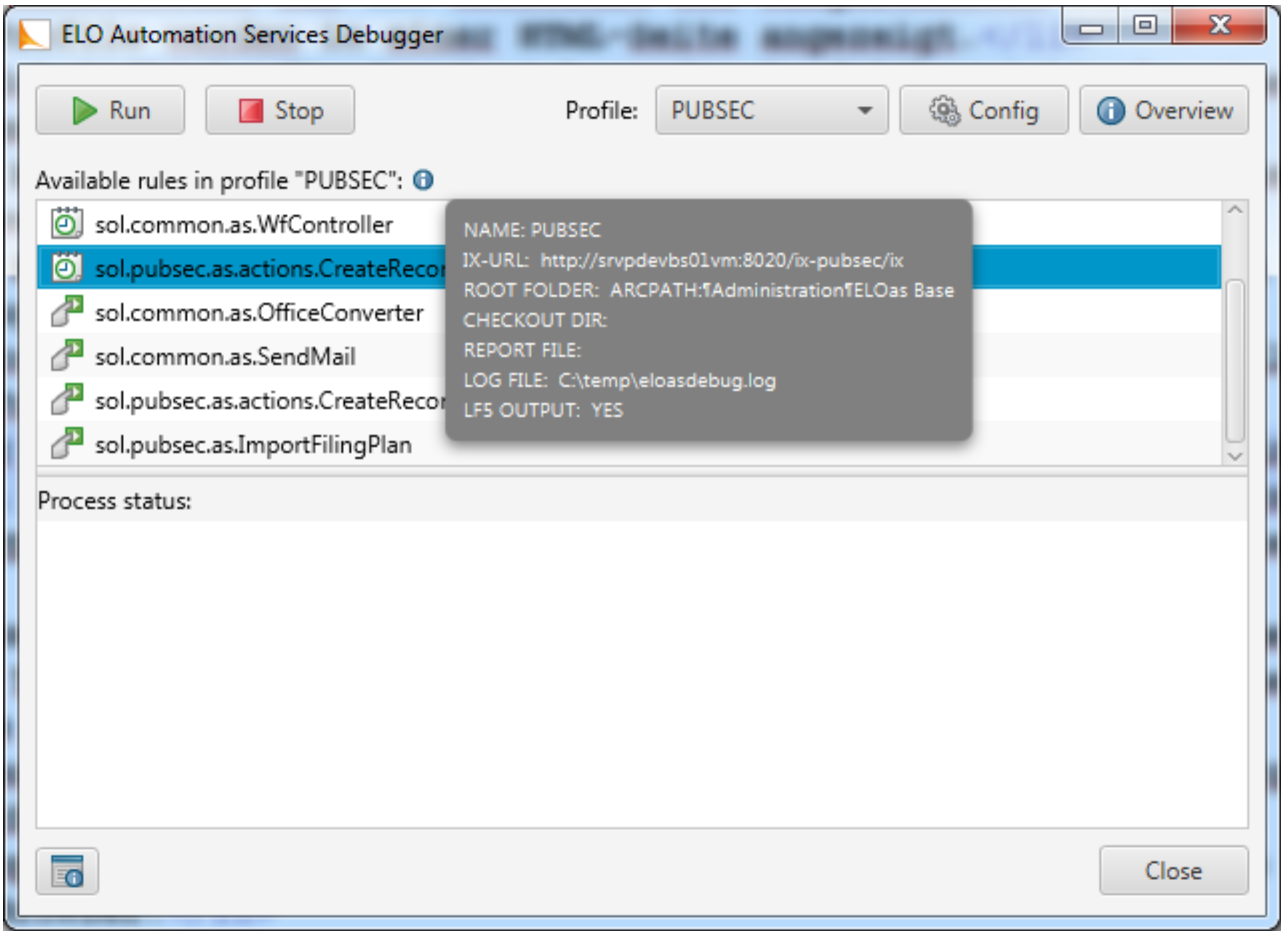


Fig.: Information on the selected profile

The main dialog box of the ELOas debugger contains an info icon that you can click to see the most important data on the currently selected ELOas debugger profile.

Editing profiles

Edit the active ELOas debugger profile by clicking *Config*.

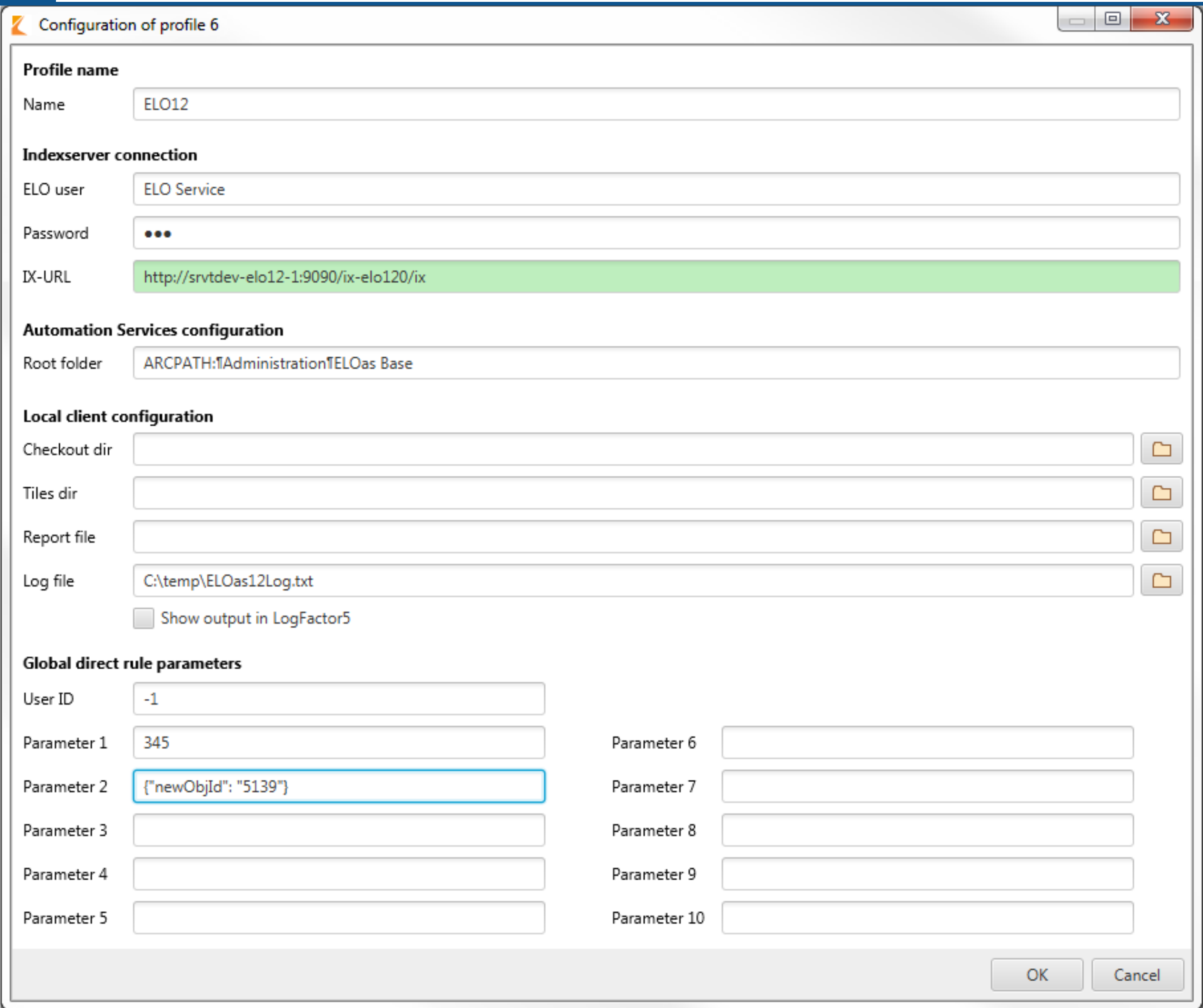


Fig.: Profile configuration dialog box

The title of the dialog box displays the ID of the ELOas debugger profile being edited. The user password is hidden in the user password text field.

Name: The profile name may contain a maximum of 15 characters.

ELO user: The name of the ELO user.

Password: The password for the ELO Indexserver connection.

IX-URL: The URL of the ELO Indexserver. The text field contains a green background when the ELO Indexserver is available at the specified URL.

Root folder: The path where the ELOas configuration is saved.

Checkout dir: Clicking the button next to the *Checkout dir* field allows you to select the ELO Java Client checkout directory.

Tiles dir: Clicking the button next to *Tiles dir* allows you to select the monitored directory for the referenced ELO Dropzone tiles.

Report file: Clicking the button next to the *Report file* field allows you to select an ELOas debugger report file.

Log file: Clicking the button next to the *Log file* field allows you to select the log file.

Global direct rule parameters: Here, you can configure the global parameters for direct ELOas rules. You can edit the user ID and ten parameters.

The profile configuration dialog box has a scroll bar that is shown when the dialog box is reduced beyond a certain size.

Click the OK button to save your changes in the system registry. The settings for the current ELOas debugger profile (ID: 1) are saved to the following location in the system registry:

"HKEY*CURRENT*USER\Software\JavaSoft\Prefs\elo digital office\eloas.1".

Click *Cancel* to discard your changes and close the dialog box. You can also press the ESC key to close the ELOas debugger profile configuration dialog box. The dialog box has a minimum size setting. When you enlarge the dialog box, the individual components of the dialog box are enlarged proportionally. This allows you to display long profile inputs.

Editing direct rules

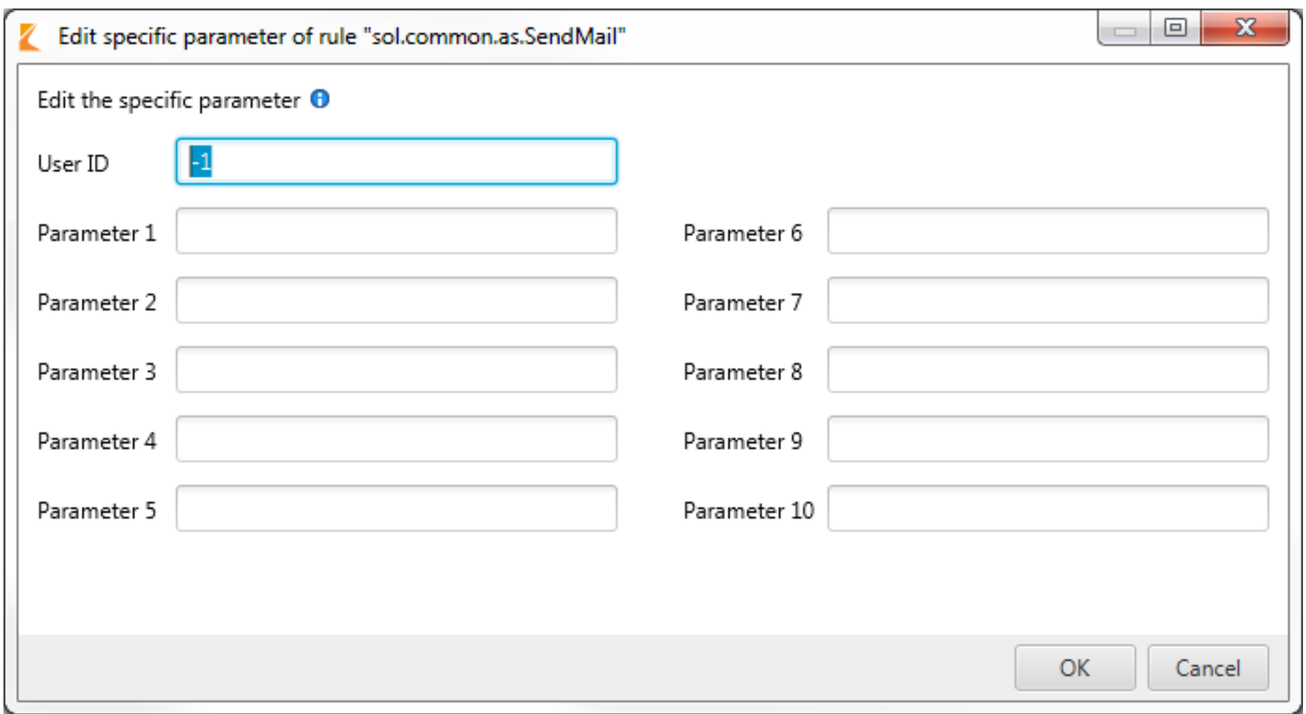


Fig.: Dialog box for editing direct rules

In this dialog box, you can edit the specific parameters of a direct rule. You can reach this dialog box via the context menu by selecting a rule in the list of available rules. In this dialog box, you can edit the user ID and the ten available parameters.

Changing profiles

If you have edited the profile and want to switch to another profile, you will have to restart the ELOas debugger.

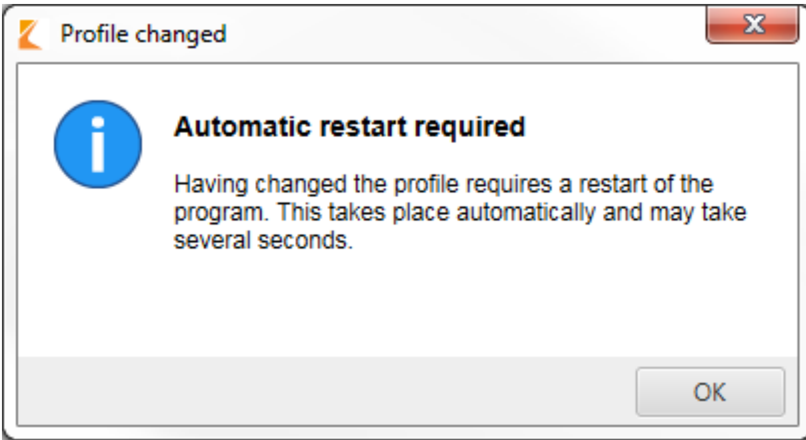


Fig.: Message dialog box indicating automatic restart after changing profiles

Keyboard shortcuts

Individual ELOas debugger functions are also assigned keyboard shortcuts.

F1 shows the *About this program* dialog box.

F2 opens the profile overview.

F3 jumps to the next search result.

SHIFT + F3 jumps to the previous search result.

F9 uses the last entry.

F12 opens the overview of keyboard shortcuts.

CTRL+P shows the profile configuration dialog box.

CTRL+O opens the profiles overview.

CTRL+E opens the dialog box for editing parameters.

CTRL + I shows the program version of the ELOas debugger.

CTRL+S brings the script contents into the foreground.

CTRL + G opens the filter function.

CTRL+L opens the log file.

CTRL+R opens the configured report file.

CTRL + F brings the "LogFactor5" window to the foreground.

CTRL + J opens the dialog box for selecting page jumps.

CTRL + W evaluates the selected text.

CTRL + ALT + L opens the "Go to line" dialog box. Enter the lines you want to navigate to here.

CTRL + ALT + R reloads the rules.

ALT + E opens the configured report file.

ALT + O opens the selected rule.

ALT + G jumps to the location where the rule is stored in ELO.

Java libraries

The following chapter contains a list of Java libraries included with the ELOas debugger.

No.	Library	Description
1.	EloixClient.jar	Library for accessing the ELO Indexserver (Indexserver interface)
2.	eloserverutils.jar	General ELO server utility classes
3.	commons-lang-2.6.jar	Utility methods for frequent string operations, serialization, and object reflection
4.	commons-lang3-3.9.jar	Utility methods for frequent string operations, serialization, and object reflection
5.	aspose-cad-19.7.jar	Library for creating and managing AutoCAD documents
6.	aspose-cells-19.8.jar	Library for creating and managing Microsoft Excel documents
7.	aspose-diagram-19.8-jdk16.jar	Library for creating and managing Microsoft Visio documents
8.	aspose-email-19.8-jdk16.jar	Library for reading e-mail messages
9.	aspose-words-19.9-jdk17.jar	Library for creating and managing Microsoft Word documents
10.	aspose-slides-19.9-jdk16.jar	Library for creating and managing Microsoft PowerPoint documents
11.	aspose.pdf-19.8.jar	Library for creating and editing PDF files
12.	aspose-barcode-19.8.jar	Library for creating and reading barcodes
13.	bcprov-jdk15on-1.52.jar	Library for accessing encrypted documents
14.	httpclient-4.4.jar	Library for sending HTTP requests
15.	httpcore-4.4.jar	Library for sending HTTP requests
16.	jai_codec.jar	Library for image processing in Java
17.	jai_core.jar	Library with the main functions for image processing in Java
18.	jai_imageio patch.jar	Library for image processing in Java
19.	log4j-1.2.17.jar	Library for log outputs in Java applications
20.	slf4j-log4j12-1.7.25.jar	Library for log outputs
21.	slf4j-api-1.7.25.jar	SLF4J logger interface
22.	jcl-over-slf4j-1.7.25.jar	Library for migration to the SLF4J logger
23.	rhino-1.7.12.jar	Library for running JavaScript scripts
24.	commons-io-2.7.jar	Library with utility methods for frequent file operations
25.	bcpkix-jdk15on-1.59.jar	Library for editing encrypted Microsoft office documents
26.	bcmail-jdk15on-1.59.jar	Library for editing encrypted e-mails
27.	bcprov-jdk15on-1.59.jar	Library for editing encrypted Microsoft office documents
28.	mlibwrapper_jai.jar	Additional library for image processing in Java
29.	jna.jar	Library for accessing system resources

No.	Library	Description
30.	platform.jar	Additional library for accessing system resources
31.	forms-1.1.0.jar	Library for creating layouts for graphic components
32.	commons-codec-1.9.jar	Library with general encoder/decoder classes for base 64, hex, and URLs
33.	pdfbox-2.0.18.jar	Library for accessing PDF files
34.	fontbox-2.0.18.jar	Library for PDF file fonts
35.	xmpbox-2.0.18.jar	Additional library for working with PDF documents
36.	javax.mail-1.6.2.jar	Library for sending e-mails
37.	activation-1.1.1.jar	Utility library for sending e-mails
38.	metadata-extractor-2.13.0.jar	Library for reading metadata from image files
39.	xmpcore-6.1.10.jar	Library for editing, printing, and converting documents
40.	db2jcc4.jar	Library for the DB2 database driver
41.	json-20190722.jar	Library for creating JSON strings
42.	gson-2.8.6.jar	Additional library for creating JSON strings
43.	sqljdbc4.jar	Library for the Microsoft SQL Server database driver.
44.	ojdbc6.jar	Library for the Oracle database driver.
45.	imgscalr-lib-4.2.jar	Library for image scaling.
46.	poi-4.1.0.jar	Library for accessing Microsoft Office documents
47.	poi-scratchpad-4.1.0.jar	Additional library for accessing Microsoft Office documents
48.	poi-ooxml-4.1.0.jar	Additional library for accessing Microsoft Office documents
49.	poi-ooxml-schemas-4.1.0.jar	Additional library for accessing Microsoft Office documents
50.	xmlbeans-3.0.1.jar	Additional library for accessing Microsoft Office documents
51.	commons-compress-1.18.jar	Utility library for accessing Microsoft Office documents
52.	commons-collections4-4.3.jar	Utility library for accessing Microsoft Office documents
53.	core-3.4.0.jar	Library for barcode recognition
54.	javase-3.4.0.jar	Another library for barcode recognition
55.	fop.jar	Library for converting XML files to PDF
56.	xmlgraphics-commons-2.3.jar	Library for editing XML files
57.	batik-all-1.10.jar	Library for applications that use images in SVG format
58.	avalon-framework-impl-4.3.1.jar	Library for creating and configuring components
59.	avalon-framework-api-4.3.1.jar	Interface to the library for creating and configuring components
60.	serializer-2.7.2.jar	Library for serialization
61.	xalan-2.7.2.jar	Library for converting XML documents to HTML

No.	Library	Description
62.	xercesImpl-2.9.1.jar	Library for an XML parser
63.	xml-apis-1.3.04.jar	Java API for XML operations
64.	xml-apis-ext-1.3.04.jar	Library for a DOM, SAX, and JAXP interface
65.	jsch-0.1.55.jar	Library for the Java implementation of SSH2
66.	jacob.jar	Library for accessing COM objects from a Java application
67.	jacob-1.19-Lib.jar	Library with the Jacob DLLs for accessing COM objects from a Java application
68.	postgresqljdbc4.jar	JDBC driver for the PostgreSQL database
69.	quartz-2.3.0.jar	Library for running processes at specific times
70.	quartz-jobs-2.3.0.jar	Additional library for running processes at specific times

ELOas debugger on Linux

The ELOas debugger 20 also comes as a complete package that is copied to the necessary position in the file system. The debugger is run from the "ELOasDebug.sh" file.

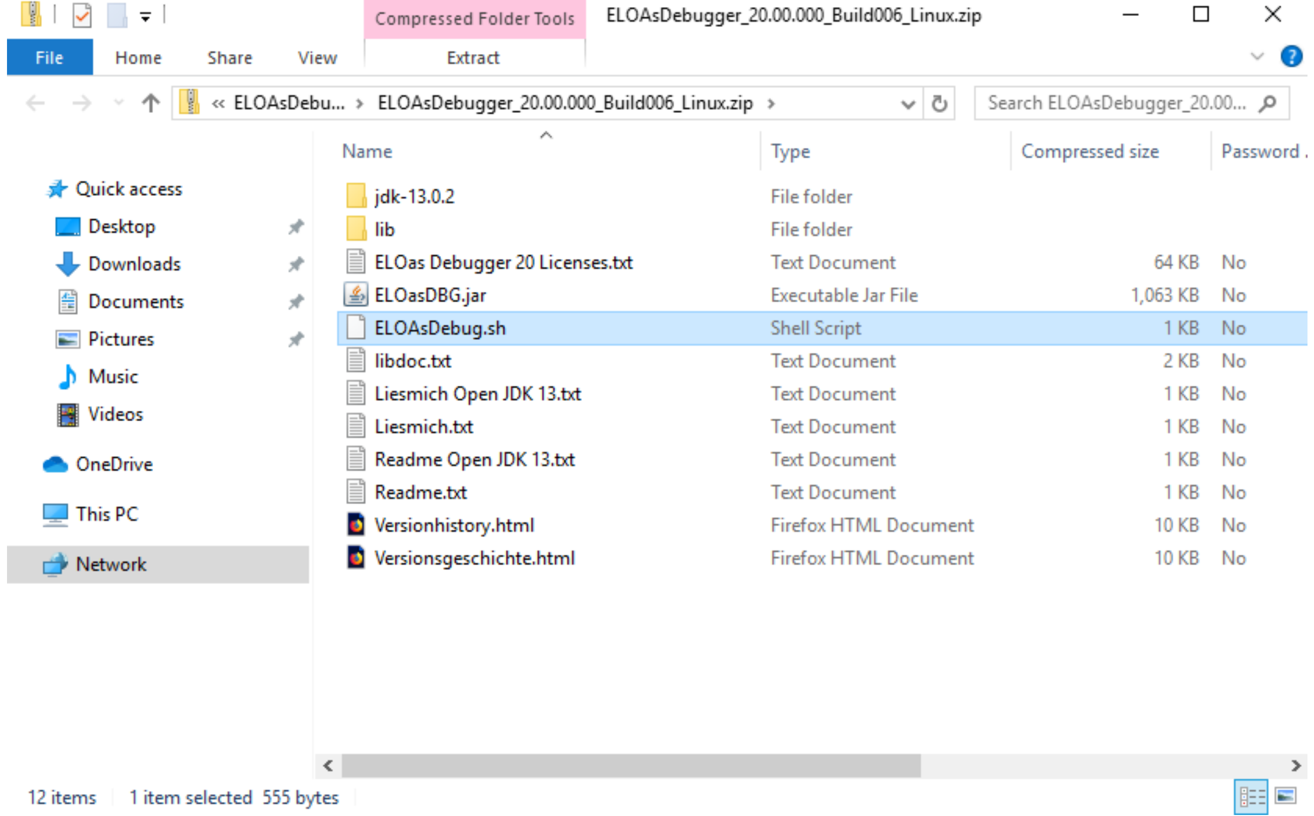


Fig.: ELOas debugger for Linux package

Other topics

Manual installation of ELOas

This document describes the manual installation of ELO Automation Services (ELOas). Under ELOprofessional, the module is created automatically by with the server installation. When installing later, or in a distributed environment, however, it must be installed manually.

Like almost all modules in the ELOenterprise server line, ELOas is programmed as a servlet and requires a Java Runtime Environment and an application server to run, such as Tomcat 9.0. Java version 11 or higher is required.

The configuration is stored in the XML file *config.xml* in the default ELO configuration directory. This allows updates to be performed without difficulty while retaining the original configuration. The default language for ELOas is automatically set during the ELO Server server setup based on the installation language selected and entered in the file *config.xml* in the `language` parameter. In the following example, it is "en" for English:

```
<entry key="language">en</entry>
```

The execution instructions of ELOas with the rulesets, translation lists, and basic scripts are located in a folder in the repository. You need to define the connection to the ELO Indexserver and this base folder in the configuration.

Required files

You will find the following files in the ZIP archive for the manual installation:

ELOas.war

ELOas.xml

logback.xml

config.xml

ELO Automation Services Konfiguration.zip

Installation.pdf

JavaScriptCode.pdf

Regeldefinition.pdf

Preparing for installation

Running ELOas requires the standard ELOas libraries in the "JavaScript" folder. Newer ELOas versions automatically install the standard ELOas libraries on program start-up if they are not already installed. The latest standard ELOas libraries can be downloaded and installed at any time from the *official scripting site*:

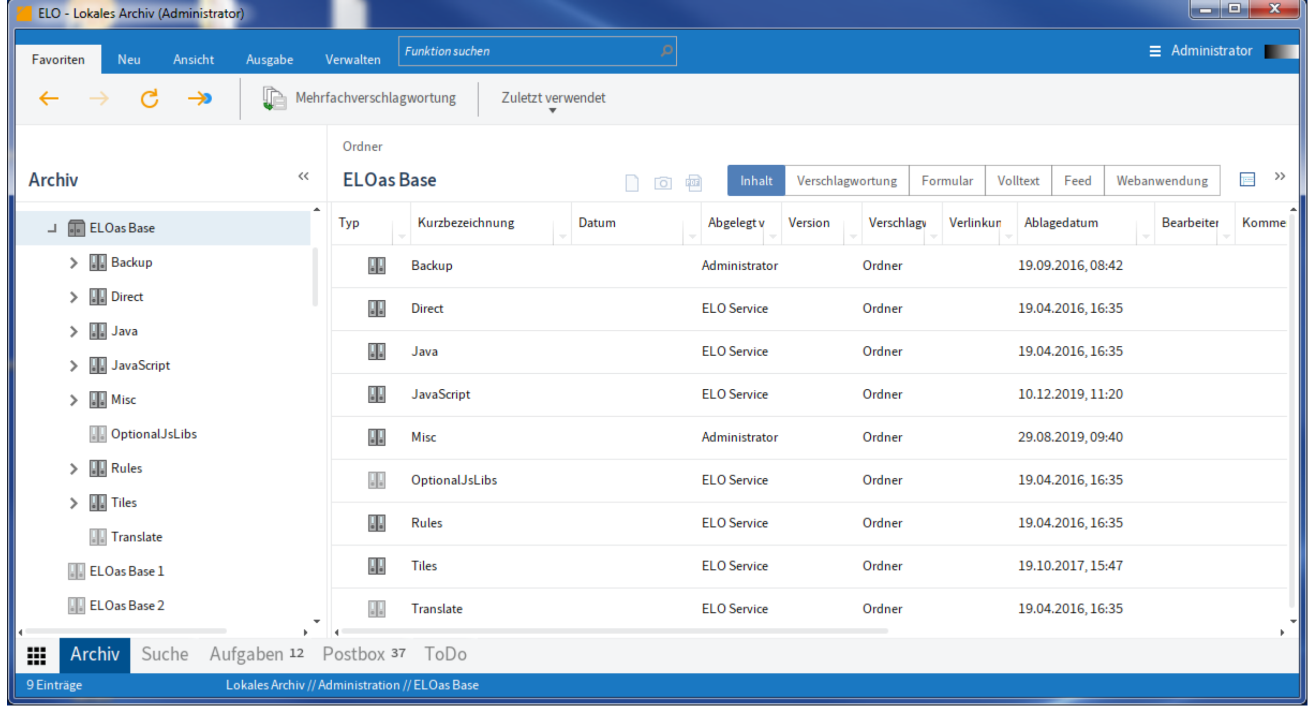


Fig.: Rules folder in ELO

The *Rules* child folder contains the user-defined rulesets. A sample has been placed here that can be used as a template for custom solutions.

The files *ELOas.war* and *ELOas.xml* should be renamed according to the repository name and the ELO standard convention for service names, into *as-<Name of repository>.war* and *as-<Name of repository>.xml.*

- Thus, for the repository "elo20", they should be renamed "as-elo20.war" and "as-elo20.xml". Pay attention to capitalization here, as this is important for later access. Both of these files are copied to a temporary directory on the computer running the application server (such as C:\TEMP).

In the *ELOas.xml* file, the path must be entered for the configuration directory of your ELO environment:

```
<?xml version='1.0' encoding='UTF-8'?>
<Context path="/as-elo20">
    <Environment name="webappconfigdir"
                 value="G:\ELOprofessional\config\as-elo20"

                 type="java.lang.String" override="false"/>
</Context>
```

For the files *logback.xml* and *config.xml*, a child directory is created in the ELO configuration directory for this ELOas configuration, and both of these files are copied there.

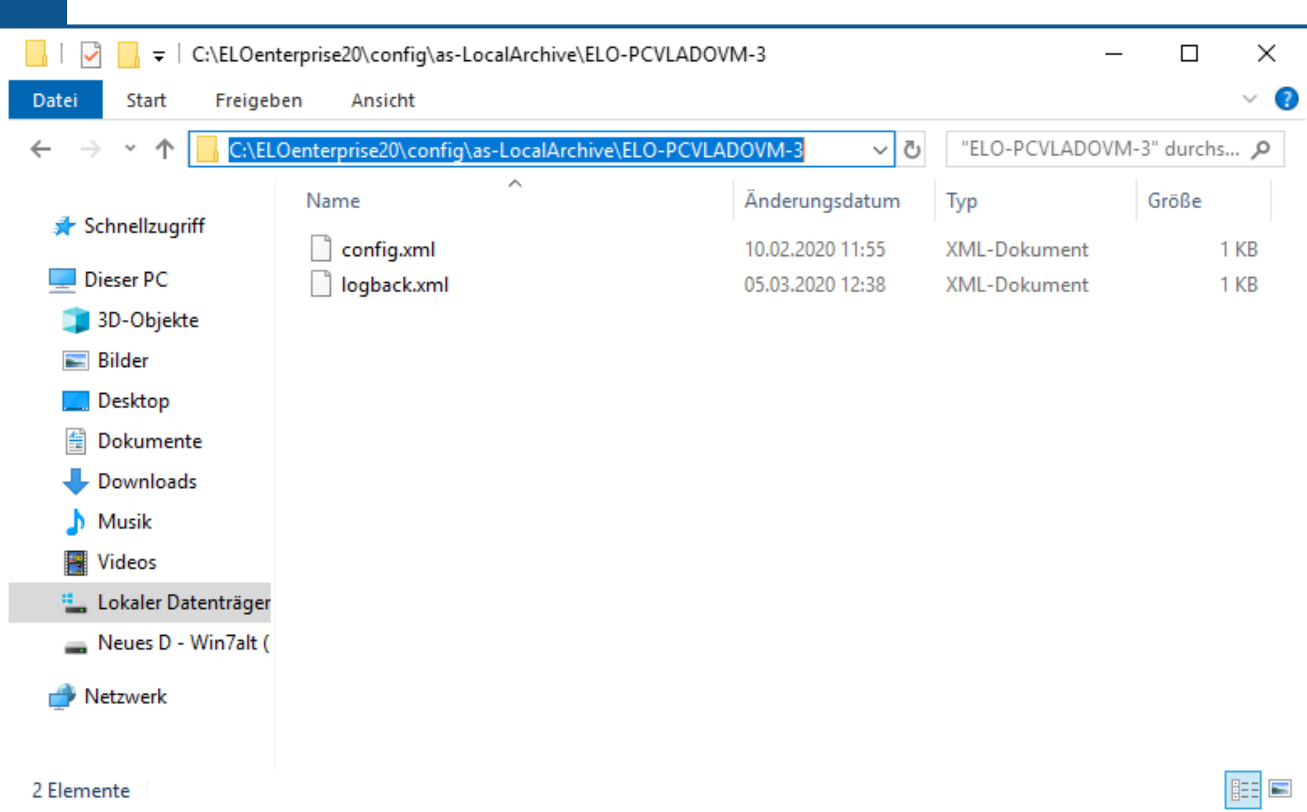


Fig.: ELOas configuration directory

The name of the configuration directory should start with "as-" and then contain the repository name. Thus, for the repository "elo20", it should have the name "as-elo20". In the *logback.xml* file, the path for the output directory must be adjusted for the local installation.

<file>C:/Programs/Tomcat 9.0/logs/as-elo20.log</file>

In the *config.xml* file, the parameters for Indexserver access must be adjusted:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>parameters for this web application</comment>
<entry key="url">http://testserver:8080/ix-elo20/ix</entry>
<entry key="user">Services</entry>
<entry key="password">130-167-2-31-129-121-203-174-234-167-21-87-88-80-78-122</entry>
<entry key="rootguid">(F6C173D7-3F71-4559-91E5-4886139B12CF)</entry>
</properties>
```

The `url` key contains the access path to the ELO Indexserver. Once again, pay attention to capitalization, as the ELO Indexserver will not be found if entered incorrectly.

The `user` key contains the ELO login name for ELOas on the ELO Indexserver. Normally, you should create a separate account for additional services. This account should not be used by interactive users.

The `password` key contains the ELO password. You can make this entry in plain text for testing purposes. After starting the service, the report will then contain a notice of how the corresponding encryption will appear. You can then apply this text from the log report to the configuration by using cut and paste.

The `rootguid` key contains the GUID of the home folder of ELOas. The default value is the GUID of the sample folder from the import data set. If you created your own folder for this data, you can easily get the GUID by running the following script in the ELO Windows Client (GetGuid.vsb file in the ZIP archive):

```
Set Elo=CreateObject("ELO.professional")
if Elo.SelectView(0)=1 then
    Id=Elo.GetEntryId(-1)

    if Id>1 then
        if Elo.PrepareObjectEx( Id, 0, 0 ) > 0 then
            call Elo.ToClipboard(Elo.ObjGuid)
            MsgBox Elo.ObjGuid
        end if
    end if
end if
```

This script finds the GUID of the currently selected entry and copies the GUID to the Windows Clipboard. From there, you can apply it to the configuration by opening it in a text editor and pressing CTRL-V.

The `rootguid` key is also used to configure several ELOas instances. You can run up to ten ELOas instances. For each instance, create an "ELOas Base" folder in the repository under *<Name of repository> // Administration*. For each individual "ELOas Base" folder in the *config.xml* file, create separate `rootguid` parameters as in the following example:

```
<entry key="rootguid">(F6C173D7-3F71-4559-91E5-4886139B12CF)</entry>
<entry key="rootguid1">(D6EF1F0B-ADE4-C3E2-74F9-3658ED55449A)</entry>
<entry key="rootguid2">(2CFDEA54-3DA9-E567-F335-6F3D223C9BAF)</entry>
```

The ELOas rules in the individual "ELOas Base" folders are then executed separately. If you are running multiple instances, the logs are also written to a log file. The path of the log file is defined in the configuration file *logback.xml*.

The `tempdir` key contains an optional directory for temporarily downloading the text files if the XML and JavaScript data has been placed in text files instead of the extra text. If `tempdir` is empty or does not exist, the extra text version is used; otherwise the text file version is preferred.

```
<entry key="tempdir">C:\Temp\ELOas</entry>
```

Please note

When creating a new user for this service, the ELO Indexserver does not respond to the change immediately. To ensure that it works, you can clear the user cache on the status page of the ELO Indexserver to force an immediate update.

Deploying the files

In the Application Server, now enter the parameters for the deployment. The context path (which is not optional, even if it says so in the Tomcat configuration) contains the name of the web application. The two file paths point to the configuration and program file. Clicking *Install* will install the application.

Installieren

Verzeichnis oder WAR Datei auf Server installieren

Kontext Pfad (optional):	/as-elo20
Version (für parallele Installationen):	
XML Konfigurationsdatei URL:	C:\ELOenterprise20\config\as-LocalArchive\ELO-PCVLADOV-3\config.xml
WAR oder Verzeichnis URL:	C:\ELOenterprise20\prog\webapps\as.war
	Installieren

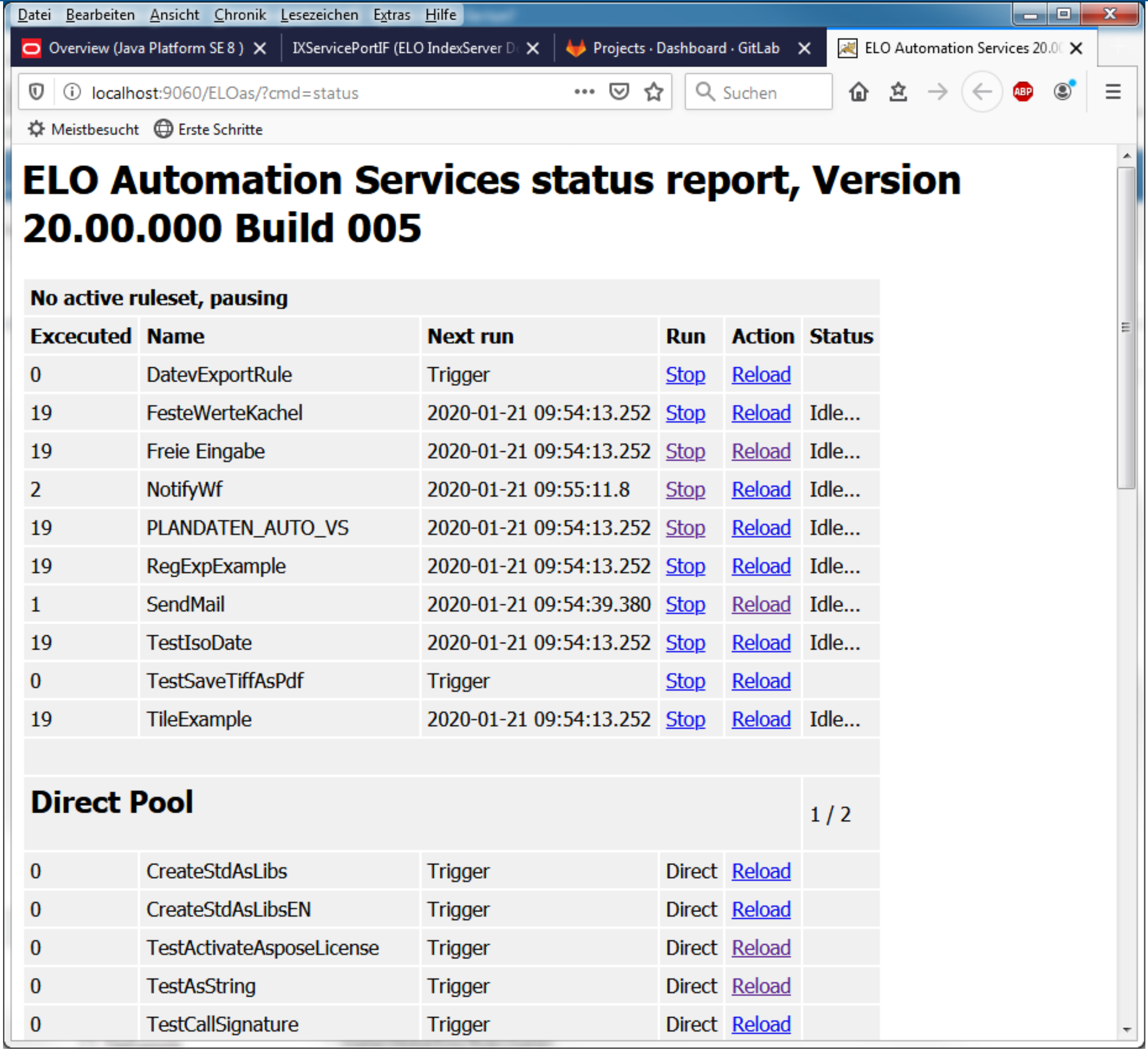
Fig.: Entering parameters for the deployment

The "ELOas.war" file in the current ELOas version contains a text file named "version.txt". This file contains extensions for the individual ELOas versions.

Displaying the status page

ELOas has its own status page, which can be reached via the following URL:

http://<SERVERNAME>:9070/as-<NAME OF REPOSITORY>/as?cmd=status



ELO Automation Services status report, Version 20.00.000 Build 005

No active ruleset, pausing					
Excecuted	Name	Next run	Run	Action	Status
0	DatevExportRule	Trigger	Stop	Reload	
19	FesteWerteKachel	2020-01-21 09:54:13.252	Stop	Reload	Idle...
19	Freie Eingabe	2020-01-21 09:54:13.252	Stop	Reload	Idle...
2	NotifyWf	2020-01-21 09:55:11.8	Stop	Reload	Idle...
19	PLANDATEN_AUTO_VS	2020-01-21 09:54:13.252	Stop	Reload	Idle...
19	RegExpExample	2020-01-21 09:54:13.252	Stop	Reload	Idle...
1	SendMail	2020-01-21 09:54:39.380	Stop	Reload	Idle...
19	TestIsoDate	2020-01-21 09:54:13.252	Stop	Reload	Idle...
0	TestSaveTiffAsPdf	Trigger	Stop	Reload	
19	TileExample	2020-01-21 09:54:13.252	Stop	Reload	Idle...
Direct Pool					1 / 2
0	CreateStdAsLibs	Trigger	Direct	Reload	
0	CreateStdAsLibsEN	Trigger	Direct	Reload	
0	TestActivateAsposeLicense	Trigger	Direct	Reload	
0	TestAsString	Trigger	Direct	Reload	
0	TestCallSignature	Trigger	Direct	Reload	

Fig.: ELOas status page with active rules

The status page lists all active rulesets together with information about how often they have already been run and when the next planned run takes place.

If a JavaScript error occurs, it will be displayed on the status page as well, together with the line number of the error and the program code in this area.

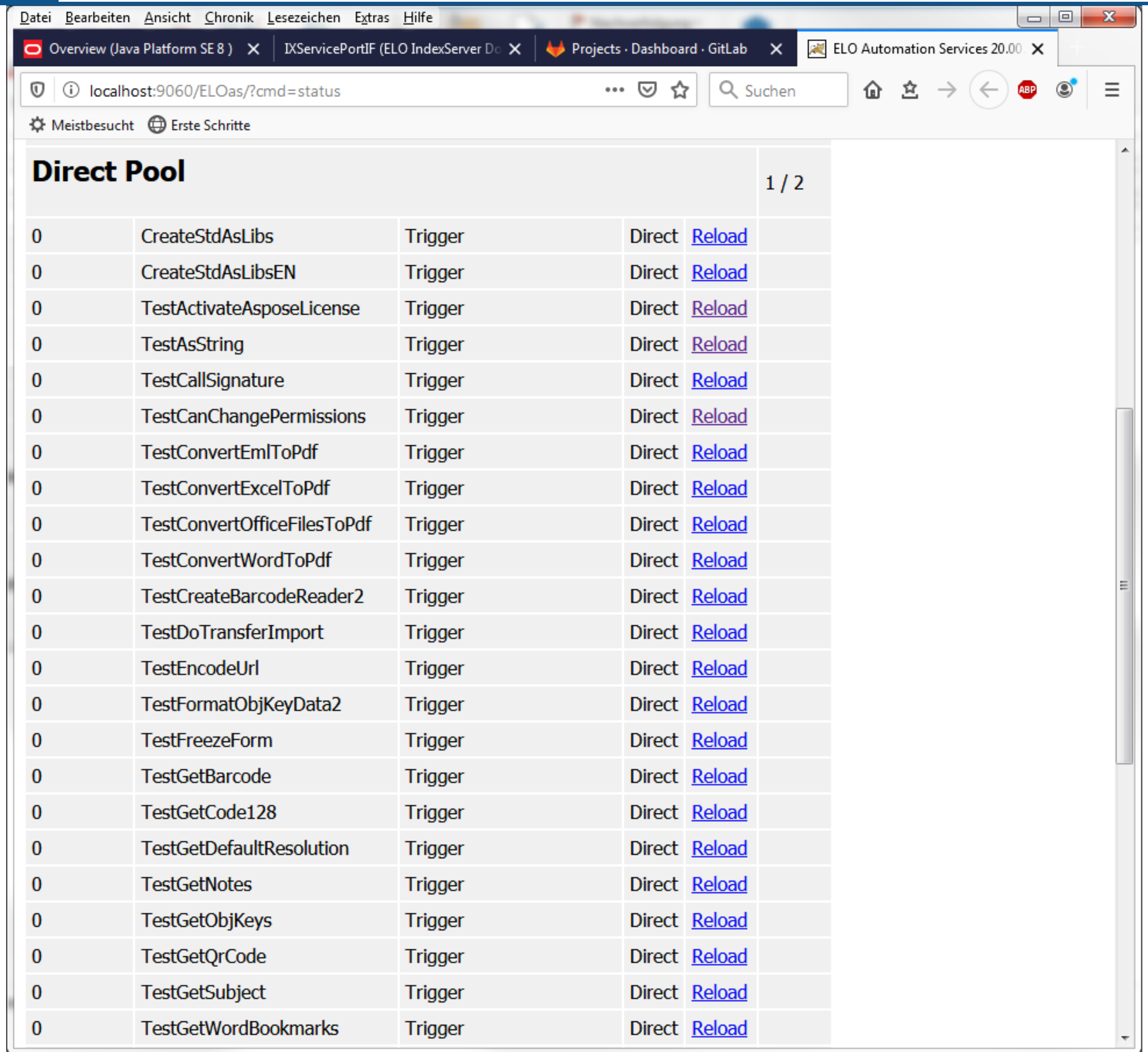


Fig.: Applying changes with 'Reload'

Changes to rules or enclosure scripts in the repository can be applied by clicking *Reload* without restarting the server.

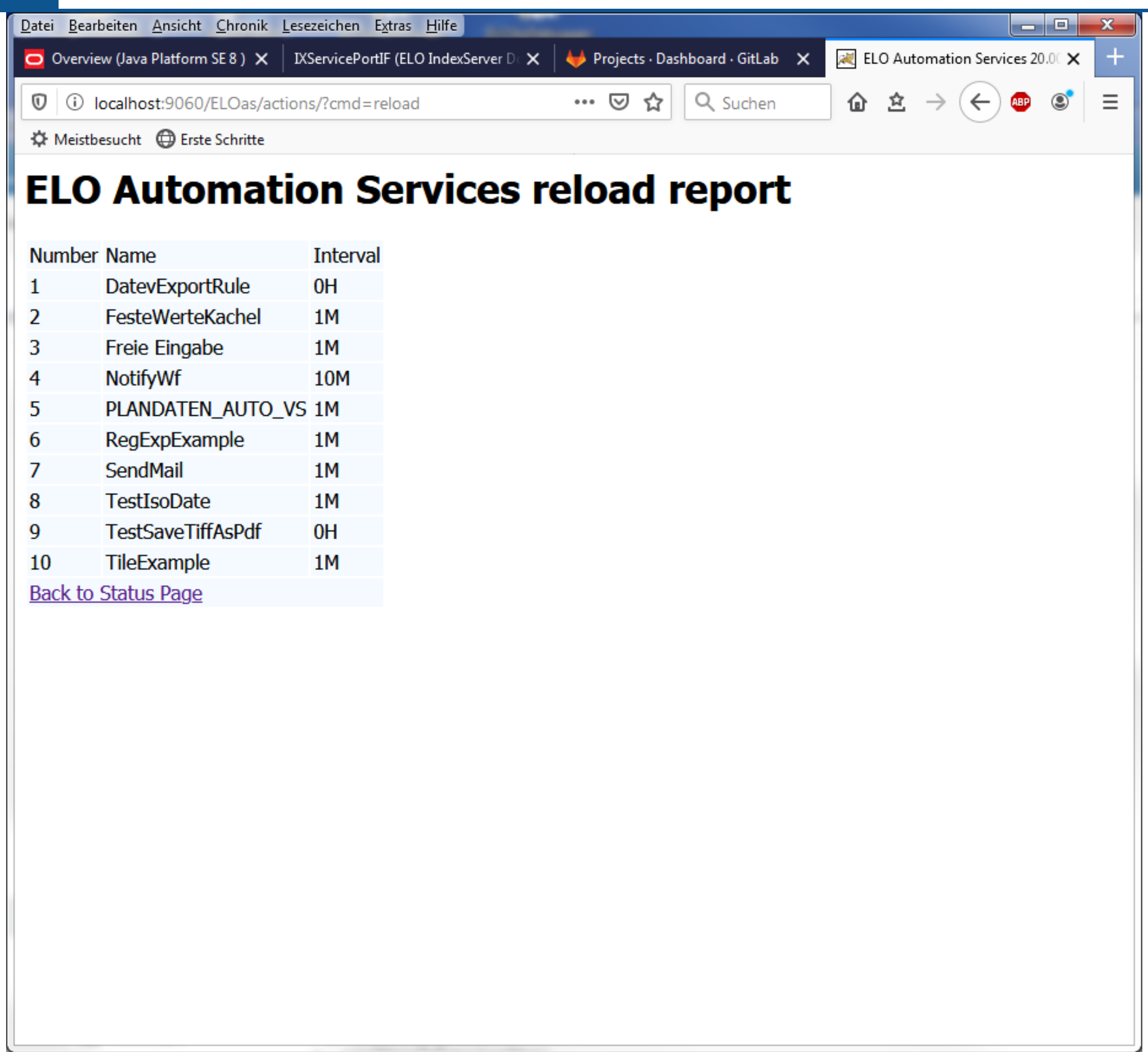


Fig.: ELO Automation Services reload report

Clicking *Back to Status Page* returns to the normal status display.

On the *Insert* tab, the catalogs contain elements that should be coordinated with the general document layout. With the help of these catalogs, you can insert tables, headers, footers, lists, cover sheets, and miscellaneous other document boilerplates.

Installing multiple ELOas instances

It is possible to install multiple instances of ELO Automation Services for a single repository (ELO Document Manager) in ELOenterprise environments. However, since ELOas is designed to use a fixed GUID for its ruleset folder in the repository, the normal ELO server setup program cannot be used to install multiple instances of ELOas in the same repository. It is not possible for multiple ELOas instances to share the same base folder.

To install additional ELO Automation Services instances for a repository, proceed as follows.

1. Create a copy of the *ELOas Base* folder.

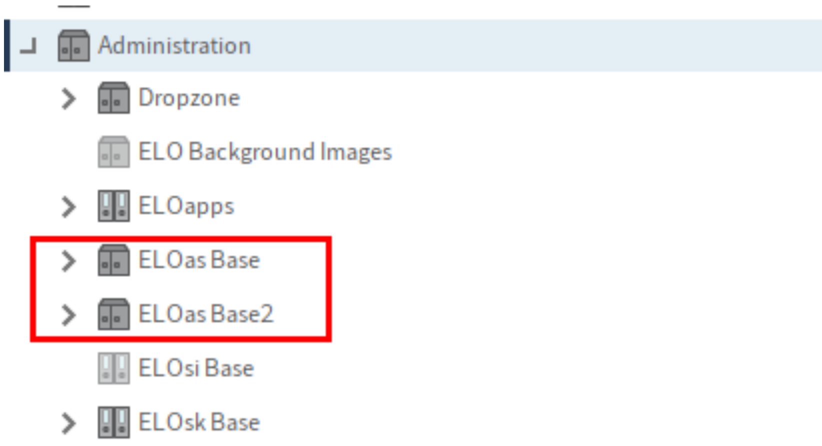


Fig.: Second ELOas Base folder

2. Copy the GUID of the new *ELOas Base* folder to a text editor.

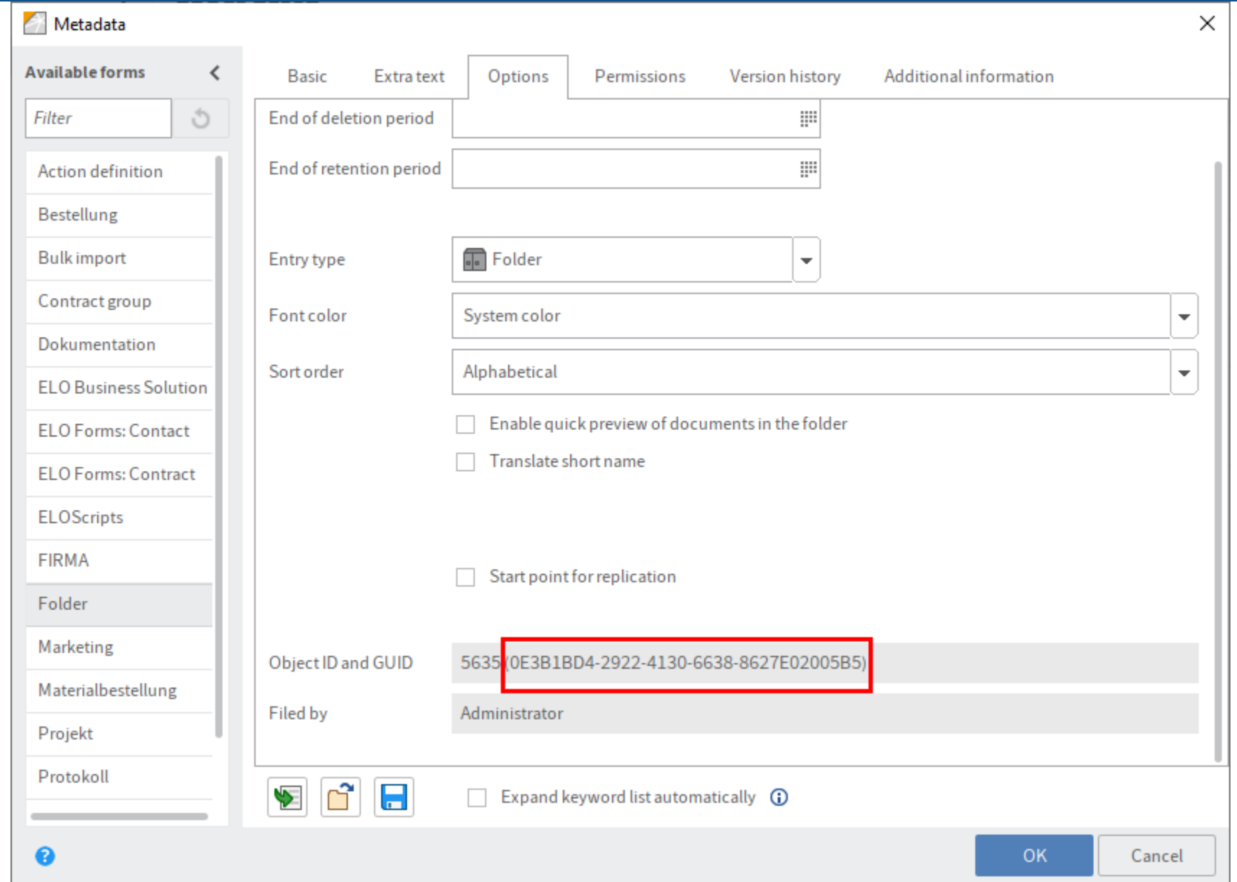


Fig.: GUID of the second ELOas Base folder

3. Stop the instance of Tomcat where you want to install the new ELOas instance.
4. Go to the web application configuration directory (`<tomcat install>\conf\Catalina\localhost`) and copy the application's XML file. In this example, `as-EXTEN01.xml` is copied to `as-EXTEN02.xml`.

Information

It is also possible to copy the ELOas configuration file to a different Tomcat server.

1. Open the copied .XML file in a text editor and change the entry for `webappconfigdir` and `Context path` to accommodate the new ELOas instance. In this example, it would be:

```
<Context docBase="E:\ELO\prog\webapps\as.war" path="/as-EXTEN2" unpackWAR="true">
<Environment name="webappconfigdir" override="false" type="java.lang.String" value="E:\ELO
</Context>
```

2. Now navigate to the *config* directory as defined in the previous step. Copy the original configuration directory for ELOas to create a new configuration directory for the new instance.

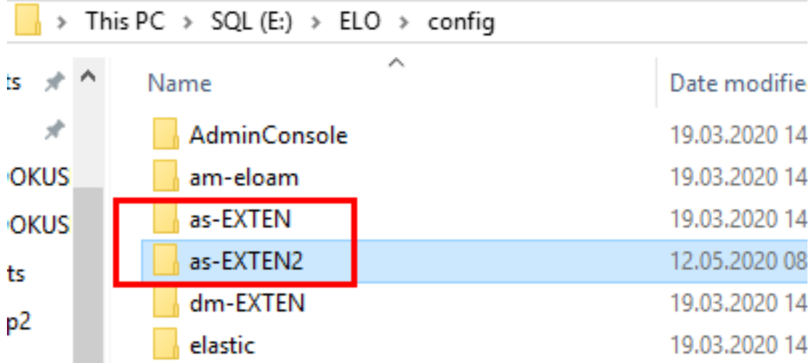


Fig.: Copied and renamed directory

3. Open the new directory, then open the *config.xml* file in a text editor.

config.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3  <properties>
4  <comment>Webapp properties</comment>
5  <entry key="password">52-247-139-10-8-11-59-34</entry>
6  <entry key="tempdir">E:\ELO\temp\as-EXTEN2\ELO-ELODOKUSRV-3</entry>
7  <entry key="language">en</entry>
8  <entry key="user">ELO Service</entry>
9  <entry key="rootguid">(0E3B1BD4-2922-4130-6638-8627E02005B5)</entry>
10 <entry key="url">http://ELODOKUSRV:9090/ix-EXTEN/ix</entry>
11 </properties>
12
```

Fig.: Modified GUID for the second ELOas instance

4. Change the `rootguid` entry so that the GUID is identical to that of the repository folder named in step 2:

5. Open the *logback.xml* file in the same directory and define a different name for the log file.

6. Start the ELO Application Server (Tomcat).

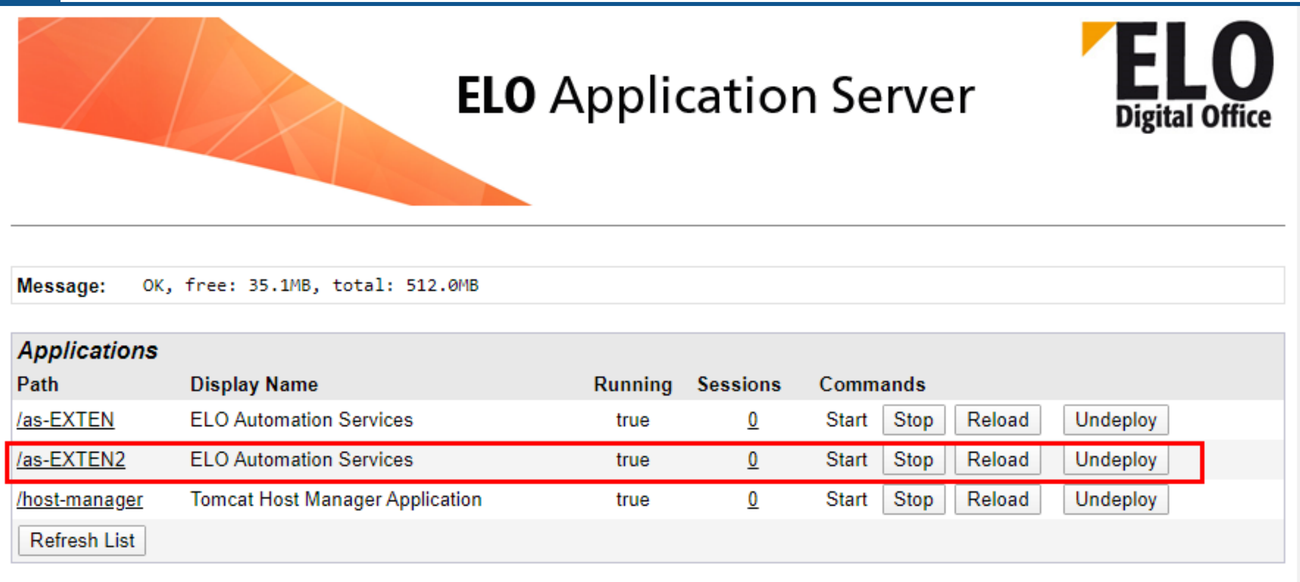


Fig.: Second ELOas instance on the ELO server

1. In the Tomcat Server Manager, check that the new ELOas instance is running properly.

Installing ELOas libraries

ELO Automation Services contains a number of libraries in the default configuration. However, it is recommended to install multiple JavaScript libraries to ensure maximum functionality. These libraries are available separately and are updated regularly.

First, you need to import these libraries into your repository.

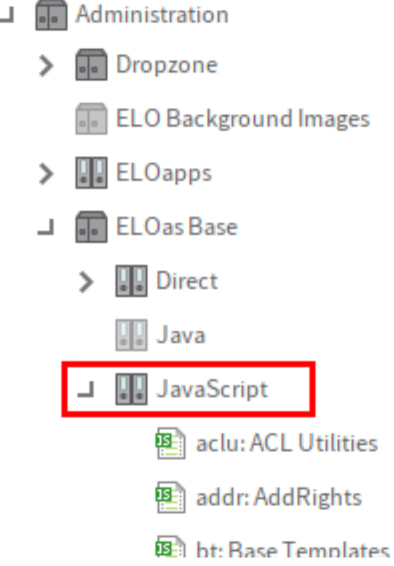


Fig.: ELOas Base folder in the tree

Make sure that the libraries are not already installed in your repository. These JavaScript files are stored in the following folder:

```
Administration//ELOas Base//JavaScript
```

If there are already JavaScript files in that folder, first make sure that they have not been customized for your environment. If not, delete them before performing the update.

Install the ELOas libraries from the ELO SupportWeb at: http://www.forum.elo.com/script/20/eloinst.html.