ELO Automation Services

[Stand: 17.03.2020 | Programmversion: 12.00.000]

Diese Dokumentation fasst die einzelnen Dokumente zu den ELO Automation Services zusammen. Die Bezeichnung „ELO Automation Services“ wird mit „ELOas“ abgekürzt.



Information: In den Abbildungen werden die ELO Automation Services noch mit der vorherigen Bezeichnung „ELO Mover“ benannt. Gemeint sind die ELO Automation Services.

Inhalt

1 Programmieren mit den ELO Automation Services 4
1.1 Funktionsweise des ELOas 4
1.1.1 Überblick 4
1.1.2 Suchmethoden (Indexsuche, Treewalk, Aufgabenliste, Mailbox, Timestamp) 5
1.1.3 Skriptausführung 5
1.1.4 Anlegen eigener Module 7
1.1.5 Lazy Initialization 9
1.2 Fehlersuche 10
1.2.1 Syntaxfehler im Skript 11
1.2.2 Logische- oder Laufzeitfehler 13
1.3 Standardmodule 14
1.3.1 cnt: ELO Counter Access 14
1.3.2 db:DB Access 16
1.3.3 dex: Document Export 22
1.3.4 ix: IndexServer Functions 25
1.3.5 wf: Workflow Utils 28
1.3.6 mail: Mail Utils 30
1.3.7 fu: File Utils 36
1.3.8 run: Runtime Utilities 37
2 Manueller Start eines Rulesets 39

2.1 Beispiel ... 39
2.2 Aktivierung des Rulesets ... 41
2.3 Weitere Hinweise ... 43
2.3.1 Asynchrones Triggern des Rulesets ... 43
2.3.2 Synchrones Triggern des Rulesets ... 43
2.3.3 Berechtigungsprüfung ... 43
2.3.4 Ausführungsreihenfolge ... 44
3 Aufbau der Rule-Struktur ... 45
3.1 Allgemeiner Aufbau ... 45
3.1.1 Alle Einträge im <base> Abschnitt ... 46
3.1.2 Alle Einträge im Abschnitt ‚<search>' ... 46
3.1.3 Alle Einträge im Abschnitt ‚<rule>' ... 48
3.1.4 Berechtigungsänderungen ... 51
3.2 Anmerkungen ... 51
3.3 Beispielhafte Struktur ... 52
4 Beispiel - Mailpostfach überwachen ... 55
4.1 Allgemeine Vorgehensweise ... 55
4.1.1 Verbindungsaufbau ... 55
4.1.2 Ruleset erstellen ... 56
4.2 Überwachte Bearbeitung ... 58
4.3 Markieren statt löschen ... 59
4.3.1 nextImap_GMAIL ... 59
4.3.2 finalizeImap_GMAIL ... 60
5 Beispiel - Migration einer Dokumenten-Datenbank ... 61
6 Beispiel - Treewalk für ELOas ... 66
6.1 Einleitung ... 66
6.2 Anwendungsbeispiel ... 66
6.3 Variablen der Laufzeitumgebung ... 68
7 Beispiel - Workflowbearbeitung ... 71
8 ELOas Debugger ... 74
8.1 Programm öffnen ... 74
8.2 Konfiguration ... 75

8.3 Ruleset bearbeiten ... 76
9 Manuelle Installation der ELOas ... 78
9.1 Benötigte Dateien ... 78
9.2 Installationsvorbereitung ... 79
9.3 Deployment der Dateien ... 82
9.4 Statusseite anzeigen ... 82

1 Programmieren mit den ELO Automation Services

Das Kapitel Programmieren mit den ELO Automation Services (ELOas) beschreibt den Aufbau und die Verwendung der JavaScript-Runtime-Umgebung. Über diese Module kann der ELOas erheblich erweitert werden und zusätzliche Funktionen ausführen, die in der Basisversion nicht vorhanden sind.

1.1 Funktionsweise des ELOas

1.1.1 Überblick

Der ELOas ist ein Servlet welches in einem Hintergrundprozess beliebige ELO Dokumente nachbearbeiten kann. Dazu gehört die Nachverschlagwortung aus anderen Datenquellen, das Verschieben von Dokumenten oder der Aufbau von Ablagestrukturen. Aufgrund der Flexibilität kann über die integrierte JavaScript-Schnittstelle aber auch eine Vielzahl anderer Funktionen erstellt werden.

Als Basis der Abarbeitung dient ein Ruleset. Dieser besteht aus einer XML-Konfiguration, welche mit einem grafischen User Interface aus der ELO Administration Console heraus erstellt wurde. Es können mehrere Rulesets definiert werden, diese werden dann nacheinander mit einer jeweils eigenen Intervallsteuerung ausgeführt („alle 10 Minuten“, „einmal täglich um 13:00 Uhr“, „jeden 3. Tag im Monat“). Das Ruleset enthält weiterhin eine Suchanfrage und eine Abfolge von Regeln zur Bearbeitung der Daten.

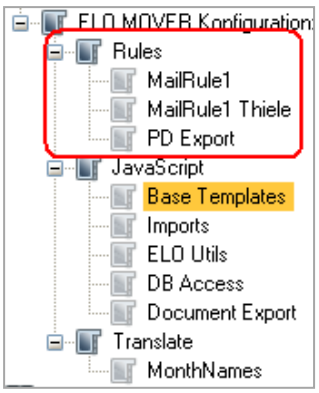


Abb. 1: Rules-Ordner im ELO Archiv

Der ELOas aktiviert im Betrieb in einer rotierenden Abfolge jedes Ruleset aus seiner Liste unterhalb von „ELOas\Rules“. Für jedes Ruleset wird zuerst geprüft, ob die Intervallbedingung erfüllt ist (sind seit dem letzten Lauf 10 Minuten vergangen?). Wenn diese noch nicht erfüllt ist, wird das nächste Ruleset abgearbeitet. Wenn aber die Zeit zur Ausführung erreicht ist, wird nun die spezifizierte Suche durchgeführt. Aus der daraus gewonnenen Trefferliste wird nun für jeden Eintrag die Regelliste abgearbeitet. Hier kann das Archivziel geändert werden, die Verschlagwortung ergänzt oder andere Aktionen ausgeführt werden. Anschließend wird das Dokument gespeichert und der folgende Eintrag bearbeitet bis das Ende der Trefferliste erreicht ist. Abschließend wird dann der neue Ausführungszeitpunkt errechnet und die Bearbeitung mit dem nächsten Ruleset fortgeführt.

Weitere Rulesets können leicht über das grafische User Interface hinzugefügt werden. Sie werden aber ebenso wie veränderte Rulesets erst nach einem neuen Laden der Konfiguration aktiv.

Die XML-Konfiguration der Regeln und der JavaScript-Code können alternativ auch in einer Dokumentendatei statt im Zusatztext gespeichert werden. In diesem Fall sieht man in der Struktur Textdateien statt Ordner für die Untereinträge.

Für den ELOwf wurde eine weitere Ruleset-Art eingeführt: der direkte Aufruf. Diese Rulesets werden in einem eigenen Ordner *Direct* angelegt, laufen in einem eigenen Thread und liefern direkt ein Ergebnis zurück. Aus diesem Grund dürfen sie nicht mit einem Intervall definiert werden, sondern müssen als Trigger angelegt werden (0 Minuten: 0M als Intervall verwenden). Zudem sollte man unbedingt darauf achten, dass man hier nur kurze Aktionen ausführt, da der aufrufende Prozess auf das Ergebnis warten muss und dieses nach einer Timeout Zeit abbricht.

1.1.2 Suchmethoden (Indexsuche, Treewalk, Aufgabenliste, Mailbox, Timestamp)

Der ELOas ist primär für die Abarbeitung einer Trefferliste gedacht, die aus einer Indexsuche resultiert. Im Laufe der Zeit sind aber weitere Optionen hinzugekommen, die durch eine passende Benennung des SEARCHNAME gewählt werden können:

TREEWALK: Im SEARCHVALUE wird die Objekt-ID oder ARCPATH zum Startobjekt hinterlegt. Es wird dann der komplette Teilbaum durchlaufen und das Ruleset zu jedem Eintrag mit der passenden Verschlagwortungsmaske aufgerufen.

WORKFLOW: Es werden alle Workflow Termine des ELOas Benutzers eingelesen und das Ruleset zu jedem Eintrag mit der passenden Verschlagwortungsmaske aufgerufen. Im Ruleset kann dann eine Workflowweiterleitung ausgelöst werden.

MAILBOX_<Profilname>: Es wird eine Verbindung zum Mailserver unter dem Profilnamen aufgenommen und ein Postfachinhalt eingelesen und abgearbeitet. Das Ruleset wird zu jeder Mail in dem Postfach mit einem leeren Dokument aufgerufen.

DIRECT: Dieses Ruleset kann per http-get aufgerufen werden und liefert direkt ein Ergebnis. Rulesets dieser Art dürfen nur im Ordner *Direct* und nicht in *Rules* definiert werden, da sie in einem anderen Thread ausgeführt werden müssen.

TIMESTAMP: Dieser Aufruf führt eine Suche nach der letzten Änderung durch. Im Normalfall wird man als Suchbegriff einen Bereich angeben: „2012.01.01.00.0.00... 2012.01.31.23.59.59“.

1.1.3 Skriptausführung

Die XML-Konfiguration des Regelsets wird vom ELOas nicht einfach nur interpretiert. Sie wird stattdessen beim Einlesen in ein JavaScript-Programm übersetzt und mit den Basisroutinen, die auch im JavaScript vorliegen, verbunden. Dieses Skript wird dann später ausgeführt. Das hat verschiedene Vorteile:

Die Zuweisungen in der XML-Konfiguration dürfen vollwertige JavaScript-Ausdrücke mit beliebigen Funktionsaufrufen enthalten.

In die XML-Konfiguration können beliebige JavaScript-Code Abschnitte mit komplexen Routinen eingebettet werden.

Die Basisroutinen können um beliebige Funktionen erweitert werden. Diese können dann auch von Administratoren verwendet werden, die keine eigenen Programmierkenntnisse haben, indem einfach die Funktion innerhalb eines Ausdrucks aufgerufen wird. Als Beispiele hierzu kann man sich die Module DB Access und Document Export ansehen.

Die erweiterten Basisroutinen können beliebige externe Java Bibliotheken zur weiteren Funktionsergänzung verwenden (z. B. JDBC Treiber oder aber auch den IX-Client zur direkten Indexserveransteuerung).

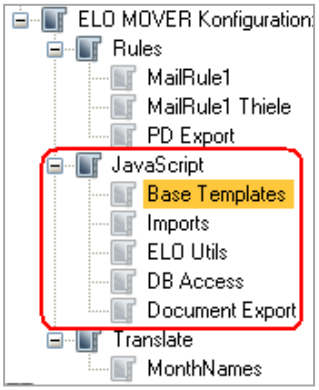


Abb. 2: Verzeichnis 'JavaScript'

Der große Vorteil bei den Basisfunktionen in JavaScript liegt darin, dass diese Funktionen im Projekt angepasst oder vorzugsweise ergänzt werden können, ohne dass der ELOas selbst verändert werden muss. Man kann also mit einen Standardprogramm arbeiten, welches aber weit an die Anforderungen angepasst werden kann.

Der ELOas bringt zur Installation die notwendigen Basisfunktionen für die Ausführung der Suche und die Abarbeitung der Regeln mit (Base Templates, Imports und ELO Utils). Dieser Teil sollte im Normalfall unverändert bleiben, nur in Sonderfällen ist es sinnvoll, hier Änderungen durchzuführen. Darüber hinaus bringt er noch zwei Module für den Datenbankzugriff (DB Access) und den Export von Dokumentendateien (Document Export) mit. In zukünftigen Versionen wird es hier weitere Module geben. Es ist auch geplant im Supportbereich so eine Art Tauschbörse für ELOas Module für Business Partner einzurichten.

Damit solche Module konfliktfrei betrieben werden können, ist ein Namespace Konzept vorgesehen, welches jedem Modul einen eigenen Namespace zuordnet. Namespaces sind immer komplett klein zu schreiben, andernfalls kann es zu Konflikten mit Gruppennamen aus der Maskendefinition kommen. **Alle 2- und 3-stelligen Namespacenamen sind für ELO reserviert** und werden für Standardmodule und freigegebene Erweiterungen verwendet. Für eigene Module können die Partner dann 4-stellige oder längere Namespacenamen verwenden. Falls Sie ein Modul erstellen, welches nur in einem Projekt eingesetzt werden soll, können Sie dafür auch einstellige Namen verwenden. Der Modulname im ELO muss dann mit der Benennung des Namespace beginnen, gefolgt von einem Doppelpunkt und einer kurzen Beschreibung (z. B. „dex: Document Export“). Intern wird der Namespace so implementiert, dass ein JavaScript-Objekt mit dem Namen des Namespaces angelegt wird und diesem Objekt werden dann alle benötigten Funktionen des Moduls zugeordnet. Da es sich um eine Aufzählung handelt, werden die einzelnen Funktionen mit einem Komma statt einem Semikolon getrennt:

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

  command2: function() {
    …
  }
}
```

Diese Funktionen können dann vom JavaScript-Code mit dex.command1(x,y) oder dex.command2() angesprochen werden. Dadurch, dass jedes Modul eine eigene eindeutige Kennung besitzt, können diese beliebig kombiniert werden, ohne dass es zu Namenskonflikten kommen kann.

Unter den Basismodulen kommt dem Modul *Imports* eine Sonderstellung zu. Es wird im erzeugten JavaScript-Programm immer ganz an den Anfang der Kette gestellt. Hier ist also der Platz für die notwendigen Imports von Java-Bibliotheken. Zusätzlich kann man hier globale Variablen hinterlegen, die von allgemeinem Interesse sind. Da dieses Modul ein globales Modul ist, besitzt es keinen Namespace.

1.1.4 Anlegen eigener Module

Neue eigene Module können vom Administrator einfach angelegt werden indem im Ordner *ELOas\JavaScript* ein neuer Ordner mit dem Namen des Moduls angelegt wird. Der eigentliche JavaScript-Code wird im Zusatztext des Ordners hinterlegt. Über die ELO Berechtigungssteuerung können zudem einzelne Module ein- und ausgeschaltet werden indem eine ACL gesetzt wird, die dem ELOas Konto Lese-Zugriff erlaubt oder nicht.

In jedem Fall werden neu angelegte oder freigeschaltete Module erst dann aktiv, wenn der Service neu gestartet oder aktualisiert wurde.



Abb. 3: ELOas Statusseite

Das eigene Modul darf beliebige eigene Funktionen oder globale Variablen mitbringen. Da zur Laufzeit alle Module innerhalb eines JavaScript-Kontexts gemeinsam ausgeführt werden, ist es jedoch wichtig, dass man bei der Benennung auf mögliche Namenskonflikte achtet. Unglücklicherweise werden solche Konflikte vom JavaScript-Interpreter nicht als Fehler angesehen und können somit nicht automatisch erkannt werden.

Die Objekte des eigenen Moduls haben eine unbegrenzte Lebensdauer. Nachdem sie erzeugt wurden bleiben sie aktiv bis der Service beendet oder aktualisiert wird. Das kann in einigen Fällen sehr problematisch sein, z. B. bei Datenbankverbindungen. Wenn so eine persistente Verbindung beim Programmstart oder beim ersten Lauf angelegt wird und danach unbegrenzt aktiv bleibt, kann das dazu führen, dass knappe Ressourcen unnötig lange belegt werden (z. B. falls das Ruleset nur einmal im Monat aktiv wird). Noch schlimmer wirkt sich aus, dass die Ressource möglicherweise ungültig wird (z. B. durch einen Neustart des Datenbankservers). Um einen ungültigen Zustand des Services zu erkennen und ein automatisches Reconnect zu veranlassen muss erheblicher Aufwand betrieben werden. Dieses Problem kann man erheblich abmildern, indem solche Ressourcen nur bei Bedarf verbunden werden und am Ende des Rulesets automatisch frei gegeben werden (siehe hierzu auch nachfolgenden Abschnitt *1.1.5 Lazy Initialization*). Hierfür muss jedes Modul eine Funktion mit einem speziellen Namen implementieren: <Namespace>ExitRuleset (z. B. dexExitRuleset). Am Ende der Bearbeitung eines Rulesets wird für jedes Modul diese spezielle Funktion aufgerufen. In dieser Funktion können dann die Aufrufe für den Verbindungsabbau hinterlegt werden.

1.1.5 Lazy Initialization

Wenn bei jeder Ruleset-Ausführung sofort alle externen Ressourcen verbunden werden und am Ende wieder getrennt werden, kann das zu einem erheblichen unnötigen Leistungsverbrauch führen. Wenn ein Ruleset schnell reagieren soll und deshalb einmal pro Minute ausgeführt wird, dann wird in vielen Fällen kein einziger aktiver Datensatz zur Bearbeitung vorliegen. Es werden also häufig nicht benötigte Verbindungen erzeugt und getrennt. Aus diesem Grund sollten externe Ressourcen immer per „Lazy Initialization" angebunden werden. In diesem Fall wird die Verbindung nicht gleich mit der Suche aufgebaut, sondern erst dann, wenn sie tatsächlich verwendet werden soll.

Dieses Schema ist in der Praxis relativ einfach zu implementieren. Nehmen wir für ein Beispiel „Reader" an, dass wir eine Ressource verwenden wollen, die die Methoden Open(), Read() und Close() besitzt. Das Open() soll nur beim ersten Read() ausgeführt werden, das Close nur dann, wenn auch ein Open ausgeführt wurde. Das Ruleset liest aus dieser Ressource per „readUser" einen Benutzernamen ein. Der JavaScript-Code in Reader könnte dann so aussehen:

```
var readerInitialized = false;

var reader = new Object();
reader = {

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

}

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

Über eine globale Variable *readerInitialized* merkt sich das Modul, ob schon eine Verbindung mittels Open() geöffnet wurde. Diese wird zum Programmstart auf false gesetzt, es besteht noch kein Kontakt.

Wenn eine Regel aus dem Ruleset dann den Benutzernamen ermitteln möchte, wird die Funktion readUser() aufgerufen. Dort wird zuerst geprüft, ob bereits eine Verbindung besteht. Wenn nicht, wird sie mit Open() geöffnet und das readerInitialized auf true gesetzt. Bei nachfolgenden Aufrufen wird also kein erneutes Open() ausgeführt. Erst danach wird per Read() aus der Ressource gelesen.

Wenn das Ruleset fertig abgearbeitet ist, wird zu dem Modul *Reader* die Endefunktion *readerExitRuleset* aufgerufen. Hier wird geprüft, ob überhaupt eine geöffnete Verbindung vorliegt und diese dann bei Bedarf mit Close() geschlossen.

1.2 Fehlersuche

Ab der Version 7.00.024 gibt es auch einen Debugger für den ELOas. Es wird die in der Rhino Engine vorhandene Debug Engine verwendet. Diese kann über einen Konfigurationsparameter aktiviert werden.

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

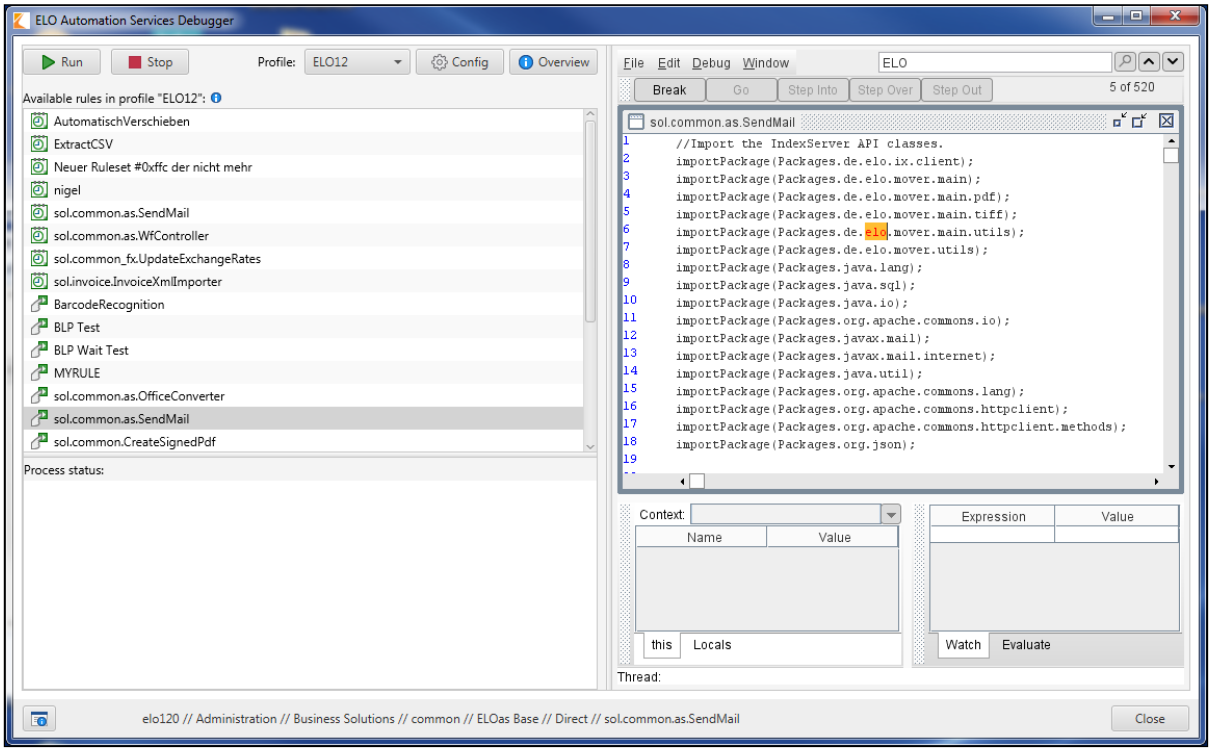


Abb. 4: Debugging

Zum Betrieb des Debuggers sollte der ELOas lokal auf der Entwicklermaschine ausgeführt werden. Zudem sollte er als Konsolenprozess und nicht als Windows Dienst gestartet werden. Andernfalls funktioniert der Debugger unter Windows Vista oder Windows 7 nicht mehr.

Falls Sie mehrere aktive Rulesets haben, gibt es für jeden ein eigenes Debuggerfenster. Sie können über den Menüeintrag *Window* zwischen den einzelnen Ansichten umschalten.

Im Debugger können Sie Breakpoints auf einzelne Funktionen setzen und Variableninhalte inspizieren oder verändern. Die Ausführung können Sie dann im Einzelschritt oder Ausführungsmodus fortsetzen.

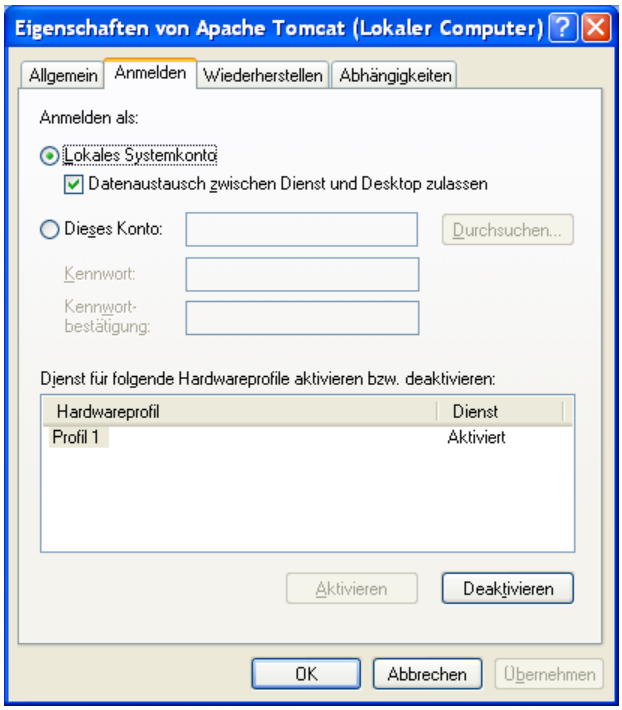
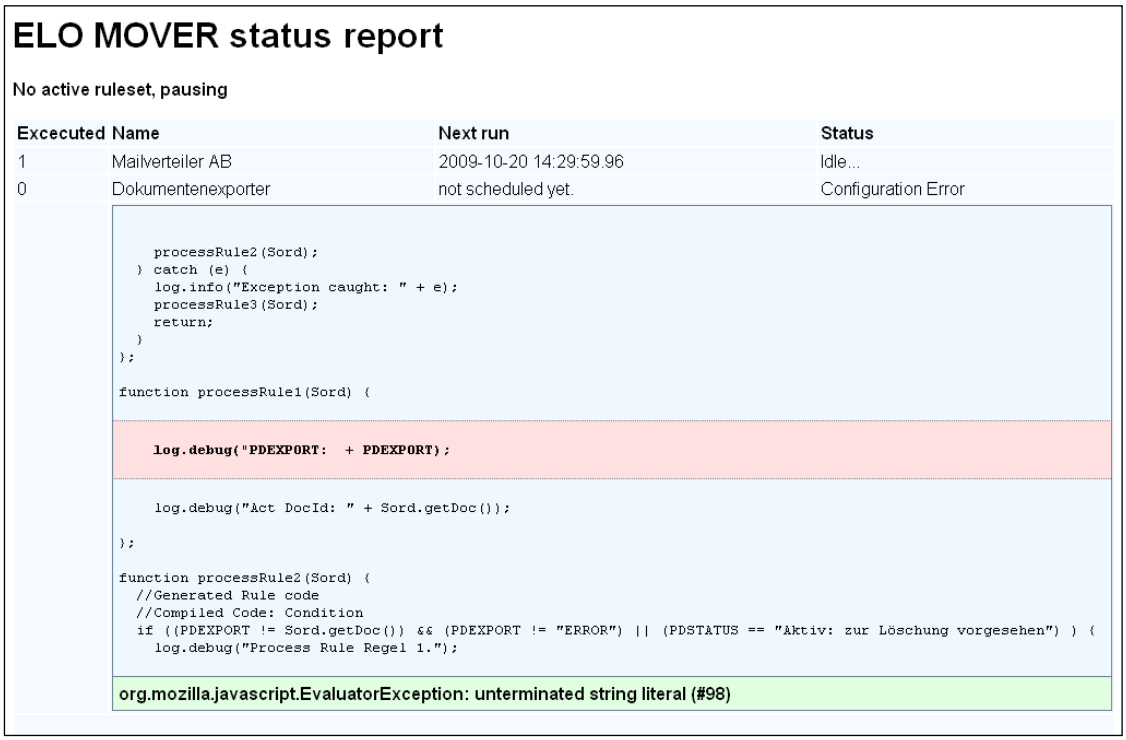


Abb. 5: Eigenschaften von Apache Tomcat

1.2.1 Syntaxfehler im Skript

Wenn das Skript einen Syntaxfehler aufweist, dann kann die JavaScript-Verarbeitung nicht gestartet werden. Solche Fehler haben den Vorteil, dass sie direkt beim Programmstart sichtbar werden und im Status-Dialog des ELOas angezeigt werden können.



ELO MOVER status report

No active ruleset, pausing

Excecuted	Name	Next run	Status
1	Mailverteiler AB	2009-10-20 14:29:59.96	Idle...
0	Dokumentenexporter	not scheduled yet.	Configuration Error

```
    processRule2(Sord);
  } catch (e) {
    log.info("Exception caught: " + e);
    processRule3(Sord);
    return;
  }
);

function processRule1(Sord) {

    log.debug("PDEXPORT:  + PDEXPORT);

    log.debug("Act DocId: " + Sord.getDoc());

);

function processRule2(Sord) {
  //Generated Rule code
  //Compiled Code: Condition
  if ((PDEXPORT != Sord.getDoc()) && (PDEXPORT != "ERROR") || (PDSTATUS == "Aktiv: zur Löschung vorgesehen") ) {
    log.debug("Process Rule Regel 1.");
```

org.mozilla.javascript.EvaluatorException: unterminated string literal (#98)

Abb. 6: Syntaxfehler im Skript

Zur Unterstützung bei der Fehlersuche wird im ELOas Report beim Start das komplette generierte JavaScript-Programm mit allen eingebundenen Modulen protokolliert. Die in der Anzeige aufgelistete Fehlernummer bezieht sich auf diesen Abschnitt des Reports (ab der Stelle „//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;
…
```

Achten Sie darauf, dass diese Ausgabe bei jedem Neustart und auch beim Reload wiederholt wird. In einer Reportdatei können sich also mehrere Auflistungen befinden. Aktuell ist immer die letzte Liste im Report.

1.2.2 Logische- oder Laufzeitfehler

Etwas schwieriger wird der Fall bei Laufzeitfehlern. Hier gibt es nur die Möglichkeit mittels Log-Ausgaben die Fehlerstelle einzugrenzen. So eine Log-Ausgabe ist zwar deutlich weniger komfortabel als ein interaktiver Debugger, hat aber bei der Massenverarbeitung durchaus auch Vorteile. Der Java Logger des ELOas ist auf der JavaScript-Seite unter dem Namen *log* erreichbar. Deshalb kann der JavaScript-Code dort auch mit log.debug() Einträge vornehmen.

```
   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);
```

Die Log-Ausgaben des JavaScript-Codes erkennt man an dem fehlenden Klassennamen und der fehlenden Zeilennummer im 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
```

1.3 Standardmodule

1.3.1 cnt: ELO Counter Access

Das Standardmodul cnt stellt den Zugriff auf ELOam Countervariablen zur Verfügung.

1.3.1.1 Verfügbare Funktionen

Counter anlegen: createCounter(). Diese Funktion erzeugt einen neuen Counter und initialisiert ihn mit einem Startwert. Falls der Counter bereits existiert, wird er zurückgesetzt.

```
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);
},
```

Counterwert ermitteln: getCounterValue(). Diese Funktion ermittelt den aktuellen Wert des angegebenen Counters. Wenn der Parameter autoIncrement auf true gestellt wird, wird der Counterwert zusätzlich automatisch hochgezählt.

```
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();
},
```

Tracking-Nummer aus Counter bilden: getTrackId(). Wenn man eine fortlaufende und automatisch erkennbare Nummer benötigt, kann man diese Funktion verwenden. Sie liest den nächsten Counterwert und codiert eine Zahl mit einem Präfix und einer Prüfziffer. Der erzeugte String sieht dann so aus <Präfix><Fortlaufende Zahl>C<Prüfziffer> („ELO1234C0“)

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

Tracking-Nummer bilden: calcTrackId(). Wenn man eine fortlaufende und automatisch erkennbare Nummer benötigt, kann man diese Funktion verwenden. Sie codiert eine Zahl mit einem Präfix und einer Prüfziffer. Der erzeugte String sieht dann so aus <Präfix><Fortlaufende Zahl>C<Prüfziffer> („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);
},
```

Tracking-Nummer im Text suchen: findTrackId(). Diese Funktion sucht in einem Text nach einer Tracking-Nummer. Das erwartete Präfix und die Länge der eigentlichen Zahl können über einen Parameter gesteuert werden. Falls die Zahl eine variable Länge besitzt, kann der Längenparameter auf 0 gestellt werden. Wenn im Text kein passender Treffer vorhanden ist, wird eine -1 zurück geliefert. Andernfalls wird der Zahlenwert (und nicht die komplette Track-ID) geliefert.

```
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;
}
```

1.3.2 db:DB Access

Das Standardmodul DB Access stellt einen einfachen Zugriff auf externe Datenbanken zur Verfügung. Im Standard werden ODBC Datenbanken sowie Microsoft SQL und Oracle SQL unterstützt. Falls auf andere Datenbanken mit einem native JDBC driver zugegriffen werden soll, müssen die entsprechenden JAR Dateien in das LIB Verzeichnis des Services kopiert werden und die Imports und Zugriffsparameter im Modul *Imports* hinterlegt werden. Die Reihenfolge der Datenbankdefinitionen in dem Imports Modul bestimmt dann den Wert des Parameters *Verbindungsnummer* in den nachfolgenden Aufrufen.

1.3.2.1 Verfügbare Funktionen

```
getColumn( Verbindungsnummer, SQL Abfrage );
```

Dieser Aufruf muss als Parameter eine SQL-Abfrage mitgeben, welche eine Spalte abfragt und als Ergebnis nur eine Zeile zurückliefert.

Beispiel:

```
„select USERNAME from CUSTOMERS where USERID = 12345“
```

Über die Verbindungsnummer wird bestimmt, welche Datenbankverbindung verwendet wird. Die Liste der verfügbaren Verbindungen ist im Modul Imports definiert.

Beispiel mit JavaScript-Code:

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

Beispiel im GUI Designer:

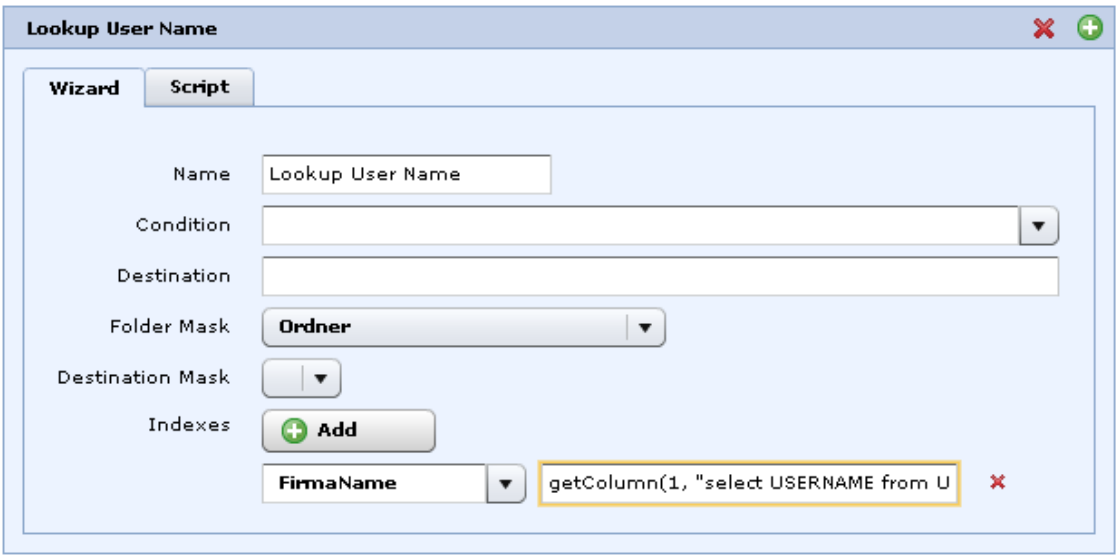


Abb. 7: GUI-Designer

Falls die Trefferliste mehrere Zeilen umfasst, wird nur der erste Wert geliefert. Alle weiteren werden ohne Fehlermeldung ignoriert.

```
getLine( Verbindungsnummer, SQL Abfrage );
```

Dieser Aufruf gibt als Ergebnis ein JavaScript-Objekt mit den Werten der ersten Zeile der SQL Abfrage zurück. Die Abfrage kann beliebig viele Spalten enthalten, auch ein *. Die Spaltennamen müssen aber eindeutig und zulässige JavaScript-Bezeichner sein. Achten Sie auf die Groß- und Kleinschreibung (case sensitive) bei den JavaScript-Bezeichnern.

Beispiel:

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

Über die Verbindungsnummer wird bestimmt, welche Datenbankverbindung verwendet wird. Die Liste der verfügbaren Verbindungen ist im Modul Imports definiert.

Beispiel mit JavaScript-Code:

```
var cmd = "SELECT objshort, objidate, objguid FROM  [elo70].[dbo].objekte
where objid = 22";
var res = getLine(1,cmd);
log.debug(res.objshort);
log.debug(res.objidate);
log.debug(res.objguid);
```

Falls die Trefferliste mehrere Zeilen umfasst, werden nur die Werte der ersten Zeile zurück geliefert. Alle weiteren Zeilen werden ohne Fehlermeldung ignoriert.

```
getMultiLine(Verbindungsnummer, SQL Kommando, Maximale Anzahl Zeilen)
```

Dieses Kommando arbeitet ähnlich wie der getLine-Aufruf. Allerdings wird nicht ein Objekt, sondern ein Array von Objekten zurückgegeben. Jede Zeile aus der Trefferliste erzeugt ein Eintrag in dem Array. Damit es bei großen Datenbanken und unglücklichen Abfragen nicht zu einem Speicherüberlauf kommt, kann man die maximale Anzahl von Zeilen begrenzen. Zusätzliche Treffer werden einfach ignoriert.

Beispiel:

```
  var obj = db.getMultiLine(1, "select objshort, objid from
[elo80].[dbo].objekte where objid < 100 order by objshort", 50);

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

doUpdate(Verbindungsnummer, SQL Kommando)
```

Man kann die Aufrufe getLine oder getColumn nicht dafür „missbrauchen“ um eine Veränderung in der Datenbank auszuführen. Dieses Kommandos verwenden intern den JDBC Befehl executeQuery – und dieser lässt nur SELECT Abfragen zu.

Zum Verändern eines Eintrags kann man den Aufruf doUpdate verwenden. Dieser übergibt das eingetragene SQL Kommando an den JDBC Befehl executeUpdate – damit kann man auch bestehende Einträge verändern oder neue Einträge einfügen.



Information: Da alle Parameter in Text Form übergeben werden müssen, muss man selbst darauf achten, dass eventuell vorkommende Anführungsstriche korrekt codiert werden. Andernfalls führt es mindestens zu Fehlermeldungen, im schlimmsten Fall zu einem SQL Injection Angriff auf den SQL Server.

1.3.2.2 Imports

Die Art und der Umfang der benötigten Imports sind abhängig von der Datenbank und müssen der Herstellerdokumentation entnommen werden. Die verwendeten JAR-Dateien müssen bei Bedarf in das LIB-Verzeichnis des ELOas Services kopiert werden.

Im Folgenden ein Beispiel für die notwendigen Imports der JDBC-ODBC Bridge:

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

Im Modul „Imports“ der Standard ELOas Libraries 12 wurde ein Standardsystem-Selektor eingeführt. Der Standardsystem-Selektor hat aus Performancegründen den Standardwert „SordC.mbLean“ und wird bei der Verarbeitung der vorhandenen ELOas Regeln verwendet.

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

Außerdem wurde im Modul „Imports“ ein System-Selektor namens „EM_SYS_SELECTOR“ eingeführt. Der System-Selektor wird im „bt“-Modul auf den Wert des eingestellten Standardsystem-Selektors gesetzt. Der System-Selektor kann im „onStart“-Event der ELOas Regeln die weiteren Eigenschaften eines Eintrags außer ID und Name verwenden bzw. verarbeiten.

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

Analog dazu wurden die Workflow-Konstanten um einen Workflow-Selektor namens „EM_WF_SELECTOR“ erweitert:

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

1.3.2.3 Verbindungsparameter

Die Datenbankverbindungsparameter werden im Modul Imports hinterlegt. Dort gibt es eine Liste von Verbindungen, die dann später durch ihre Nummer (mit 0 beginnend) als Verbindungsnummer angesprochen werden können.

```
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
  }
];
```

Für jede Verbindung müssen folgende Informationen hinterlegt werden:

driver	JDBC Klassenname für die Datenbankverbindung. Diese Information erhalten Sie vom JDBC Treiberhersteller oder vom Datenbankhersteller.
url	Zugriffs-URL auf die Datenbank. Hier werden datenbankabhängige Verbindungsparameter hinterlegt, z. B. Dateipfade bei Access Datenbanken oder Servernamen und Ports bei SQL Datenbanken. Diese Verbindungsparameter sind Herstellerabhängig und müssen der jeweiligen Dokumentation entnommen werden.
user	Login-Name für den Datenbankzugriff. Dieser Parameter wird nicht von allen Datenbanken verwendet (z. B. nicht von ungeschützten Access Datenbanken). In diesem Fall kann der Parameter leer bleiben.
password	Passwort für die Datenbankanmeldung.
initdone	Interne Variable für die “lazy initialization”.
classloaded	Interne Variable zur Kontrolle ob die Klassendatei bereits geladen wurde.
dbcn	Interne Variable zur Speicherung des Datenbank Verbindungsobjekts.

1.3.2.4 JavaScript-Code

Die dbInit Routine wird nur Modulintern aufgerufen. Sie wird vor jedem Datenbankzugriff aufgerufen und prüft nach, ob schon eine Verbindung aufgebaut wurde und erstellt sie bei Bedarf.

```
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;
}
```

Die Funktion exitRuleset_DB_Access() wird automatisch nach der Beendung der Ruleset-Verarbeitung aufgerufen. Sie prüft nach, ob eine Verbindung existiert und schließt sie dann. Diese Kontrolle muss für alle konfigurierten Datenbanken erfolgen.

```
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);
        }
      }
    }
  }
}
```

Die Funktion getLine() liest eine Zeile aus der Datenbank mit beliebigen Spalten und packt die Ergebnisse in ein JavaScript-Objekt. Dieses Objekt enthält für jede Spalte eine Member Variable mit dem Spaltennamen.

```
function getLine(connection, qry) {
  // Unterfunktion: erzeugt ein JavaScript Objekt mit
  // dem eingelesenen Datenbankinhalt
  function dbResult(connection, qry) {
    // Zuerst die Verbindung aufbauen
    dbInit(connection);

    // nun ein SQL Statement Objekt erzeugen
    var p = EM_connections[connection].dbcn.createStatement();

    // und die Abfrage ausführen
    var rss = p.executeQuery(qry);

    // rss enthält die Trefferliste, es wird nun die erste
    // Zeile eingelesen
    if (rss.next()) {
      // über die Metadaten wird die Zahl der Spalten ermittelt
      var metaData = rss.getMetaData();
      var cnt = metaData.getColumnCount();

      // Zu jeder Spalte wird nun eine Member Variable erzeugt
      // Diese hat als Namen den SQL Spaltenname und als Wert
      // den gelesenen Datenbankinhalt.
      // Die erste Spalte ist zusätzlich immer unter dem Namen
      // first ansprechbar.
      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;
        }
      }
    }

    // Abschließend wird die Trefferliste und das SQL
    // Statement geschlossen.
    rss.close();
    p.close();
  }

  // hier ist der eigentliche Funktionsstart. Es wird
  // ein JavaScript Objekt mit dem Datenbankinhalt
  // angefordert.
  var res = new dbResult(connection, qry);
```

```
  return res;
}

// Die Funktion getColumn ist eine spezielle Variante
// des getLine Aufrufs. Die SQL Abfrage darf nur eine
// Spalte als Ergebnis aufweisen. Falls es weitere
// Spalten gibt, werden diese, genauso wie zusätzliche
// Zeilen, ignoriert.
function getColumn(connection, qry) {
  var res = getLine(connection, qry);
  return res.first;
}
```

1.3.3 dex: Document Export

Das Modul Document Export kann automatisiert Dokumente aus dem Archiv in das Dateisystem exportieren. Dieser Export ist kein einmaliger Vorgang – wenn eine neue Dokumentenversion erzeugt wird, schreibt das Modul automatisch eine aktualisierte Datei. Weiterhin können bereits veröffentlichte Dateien wieder gelöscht werden. Die Dateien können aus Sicherheitsgründen nur in einem vorkonfigurierten Pfad liegen.

Zur Verwendung muss eine Ablagemaske definiert werden, welche den Dokumentenstatus und ein oder mehrere Ablageziele im Dateisystem enthält. Zusätzlich wird in der Maske die Dokumentennummer des letzten Exports gespeichert.

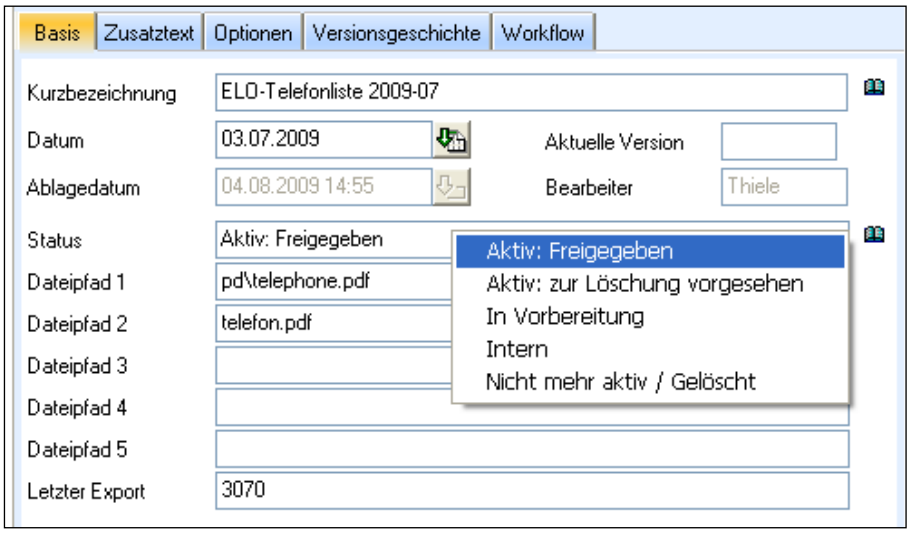


Abb. 8: Statusfeld in der Verschlagwortungsmaske

Das Statusfeld bestimmt die durchzuführenden Aktionen. Mit *Aktiv: Freigegeben* wird die Datei zum Export angemeldet. *Aktiv: zur Löschung vorgesehen* bewirkt, dass die Datei im Dateisystem gelöscht wird und der Status auf *Nicht mehr aktiv / Gelöscht* gesetzt wird. Alle anderen Statuseinstellungen bewirken keine Aktion des ELOas und sind für interne Dokumente oder noch nicht freigegebene Dokumente gedacht. Da dieser Statuswert von der internen Verarbeitung abgefragt wird, ist es sinnvoll diese Zeile nur aus einer vorkonfigurierten Stichwortliste heraus zu füllen.

Die Felder Dateipfad 1..5 enthalten den Pfad und Dateiname des Dokuments im Dateisystem. Dabei handelt es sich um einen relativen Pfad, der Startanteil ist als „dexRoot“ fest im JavaScript-Modul vorgegeben und kann dort angepasst werden. Dieser feste Anteil ist zur Sicherheit eingeplant, denn sonst können durch Fehleingaben der Benutzer beliebige Dateien überschrieben werden.

Das Feld *Letzter Export* enthält die Dokumentennummer der zuletzt exportierten Dateiversion. Wenn nach einer Bearbeitung eine neue Dateiversion erzeugt wurde, wird das durch das Modul erkannt und eine Kopie der Datei neu in das Dateisystem geschrieben. Anschließend wird dieses Feld aktualisiert.

Wenn bei der Bearbeitung ein Fehler aufgetreten ist, wird durch die Error Rule in dem Feld Letzter Export der Text „ERROR“ hinterlegt. Man kann somit im ELO ein dynamisches Register erstellen, welches dieses Feld auf den Wert ERROR überprüft und somit immer eine aktuelle Liste aller nicht exportierbaren Dokumente anzeigt.

Beispiel für ein dynamisches Register, wenn die Maske die ID 22 besitzt:

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

1.3.3.1 Verfügbare Funktionen

Das Modul stellt nur eine Funktion zur Verfügung: *processDoc.* Diese bekommt als Parameter das Indexserver-SORD-Objekt und prüft anhand des Status ob die Datei exportiert oder gelöscht werden soll und führt die entsprechende Aktion aus. Als Rückgabewert wird die neue Dokumenten-ID übergeben. Das aktuelle SORD-Objekt steht innerhalb einer Regelabarbeitung in der JavaScript-Variablen *Sord* zur Verfügung.

Beispiel im XML-Ruleset-Code:

```
<rule>
  <name>Regel 1</name>
  <condition>(PDEXPORT != Sord.getDoc()) &amp;&amp; (PDEXPORT != "ERROR")
|| (PDSTATUS == "Aktiv: zur Löschung vorgesehen") </condition>
  <index>
   <name>PDEXPORT</name>
   <value>dex.processDoc(Sord)</value>
  </index>
 </rule>
```

1.3.3.2 JavaScript-Code

Als erstes wird der Basispfad *docRoot* für die Dokumentenablage bestimmt. Der Zielpfad wird immer aus dieser Einstellung und der Benutzereingabe in der Ablagemaske ermittelt. Prinzipiell wäre es möglich, diesen Basispfad leer zu lassen, so dass der Benutzer beliebige Pfade angeben kann. Diese Vorgehensweise würde aber ein großes Sicherheitsrisiko darstellen, da dann jeder Benutzer beliebige Dateien aus dem Zugriffsbereich des ELOas überschreiben kann.

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

Die Funktion processDoc wird von der Regeldefinition heraus aufgerufen. Hier wird der Status des Indexserver-SORD-Objekts geprüft und die benötigte Funktion aufgerufen.

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

  if (PDSTATUS == "Aktiv: zur Löschung vorgesehen") {
    return dex.deleteDoc(Sord);
  } else if (PDSTATUS == "Aktiv: Freigegeben") {
    return dex.exportDoc(Sord);
  }

  return "";
}
```

Falls der Status auf „Löschen“ eingestellt war, wird in der Funktion deleteDoc die Löschung der Dateien veranlasst und der Status auf „Gelöscht“ umgestellt.

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

  PDSTATUS = "Nicht mehr aktiv / Gelöscht";
  return Sord.getDoc();
}
```

Die Funktion deleteFile führt die eigentliche Löschung durch. Sie prüft zuerst, ob ein Dateiname konfiguriert ist und ob die Datei vorhanden ist und entfernt diese dann aus dem Dateisystem.

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

  var file = new File(docRoot + destPath);
  if (file.exists()) {
    log.debug("Delete expired version: " + docRoot + destPath);
    file["delete"]();
  }
}
```

Falls eine neue Dateiversion geschrieben werden soll, wird die interne Funktion exportDoc aufgerufen. Hier wird die Datei vom Dokumentenmanager abgeholt und in die Zielordner kopiert.

```
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();
}
```

Die Funktion copyFile führt den Kopiervorgang in den Zielordner aus. Es wird zuerst geprüft, ob ein Zieldateiname vorliegt und ob eine eventuell vorhandene Altversion gelöscht werden muss. Anschließend wird die neue Version vom Dokumentenmanager geholt und im Zielordner gespeichert.

```
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"]();
  }

  ixConnect.download(url, file);
}
```

1.3.4 ix: IndexServer Functions

Das Modul ix enthält eine Sammlung verschiedener Indexserver Funktionen, die häufiger mal in Skripten benötigt werden. Dabei handelt es sich im Wesentlichen aber nur um einfache Wrapper um den gleichartigen Indexserverbefehl und keine neue komplexe Funktionalität.

1.3.4.1 Verfügbare Funktionen

Löschen eines Sord Eintrags: deleteSord(). Dieser Funktion werden als Parameter die Objekt-IDs des zu löschenden SORD-Eintrags und des Parent-Eintrags mitgegeben.

```
deleteSord: function (parentId, objId) {
  log.info("Delete SORD: ParentId = " + parentId + ",  ObjectId = " +
objId);
  return ixConnect.ix().deleteSord(parentId, objId, LockC.NO, null);
},
```

Suchen eines Eintrags: lookupIndex(). Diese Funktion ermittelt die Objekt-ID eines Eintrags welche über den Archivpfad gefunden wird. Der Parameter archivePath muss mit einem Trennzeichen beginnen.

```
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;
  }
}
```

Suchen eines Eintrags: lookupIndexByLine(): Diese Funktion ermittelt die Objekt-ID eines Eintrags anhand einer Indexzeilensuche. Wenn der Parameter Mask-ID mit einem Leerstring übergeben wird, wird eine maskenübergreifende Suche durchgeführt. Der Gruppenname und der Suchbegriff müssen übergeben werden.

```
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;
},
```

Lesen der Volltextinformation: getFulltext(). Diese Funktion liefert die aktuelle Volltextinformation zu einem Dokument. Der Volltext wird als String zurückgegeben.



Beachten Sie: Sie können nicht erkennen, ob kein Volltext vorliegt oder die Volltextbearbeitung vollständig abgeschlossen ist oder mit Fehler abgebrochen wurde. Es wird der Text zurück geliefert, der zum Abfragezeitpunkt vorhanden ist (evtl. auch eben ein Leerstring – wenn keine Volltextinformation vorliegt).

```
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"]();

  return text;

}
```

Erzeugen einer Ordnerliste: createSubPath(). Diese Funktion prüft nach, ob der angegebene Ordnerpfad im Archiv vorhanden ist und legt die fehlenden Teile bei Bedarf automatisch an.

```
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];
}
```

1.3.5 wf: Workflow Utils

Das Modul *wf* enthält vereinfachte Zugriffe auf Workflowdaten. Dabei gibt es zwei Gruppen von Funktionen.

Die High Level Funktionen *changeNodeUser* und *readActiveWorkflow* sind für den einfachen Zugriff aus einer laufenden WORKFLOW Abarbeitung heraus zu verwenden und arbeiten mit dem aktuell aktiven Workflow. Sie sind leicht zu verwenden, führen aber nur eine einfache Funktion aus.

Die Low-Level Funktionen *readWorkflow*, *writeWorkflow*, *unlockWorkflow* und *getNodeByName* können von beliebiger Stelle aus verwendet werden. Wenn mehrere Änderungen im gleichen Workflow durchgeführt werden müssen, kann man hierüber sicherstellen, dass der Workflow nur einmal gelesen und geschrieben wird und nicht x-mal für jede Operation.

1.3.5.1 Verfügbare Funktionen

Benutzernamen eines Personenknoten ändern: changeNodeUser(). Diese Funktion tauscht im aktuellen Workflow im Workflowknoten mit dem Namen *nodeName* den Benutzer gegen einen neuen Benutzer *nodeUserName* aus.

Da dieser Aufruf stets den kompletten Workflow einliest, verändert und sofort wieder zurückschreibt, sollte dieser einfache Aufruf nur dann verwendet werden, wenn nur ein Knoten bearbeitet werden soll. Falls mehrere Änderungen notwendig sind, sollten die später beschriebenen Funktionen zum Lesen, Bearbeiten und Speichern eines Workflows verwendet werden.

Da diese Funktion die Workflow-ID aus dem aktuell aktiven Workflow ermittelt, darf er nur aus der Suche „WORKFLOW“ heraus aufgerufen werden. Bei der Verwendung in einem TREEWALK oder einer normalen Suche heraus wird eine zufällige Workflow Id verwendet.

```
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);
  }
}
```

Benutzernamen eines Knotens kopieren: copyNodeUser(). Diese Funktion arbeitet ähnlich wie changeNodeUser, allerdings kopiert sie den Benutzernamen aus einem Knoten in einen anderen Knoten.

```
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;
  }
}
```

Aktuellen Workflow einlesen: readActiveWorkflow(). Diese Funktion liest den aktuell aktiven Workflow in eine lokale Variable zur Bearbeitung ein. Er kann am Ende mit writeWorkflow zurückgeschrieben werden oder die Sperre kann per unlockWorkflow wieder freigegeben werden.

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

Workflow einlesen: readWorkflow(). Diese Funktion liest einen Workflow in eine lokale Variable ein. Dieser kann dann ausgewertet und verändert werden. Wenn die Änderungen gespeichert werden sollen, dann können diese per writeWorkflow zurückgeschrieben werden. Wenn der Workflow mit Lock gelesen wurde aber keine Änderungen gespeichert werden sollen, kann die Sperre mit unlockWorkflow zurückgenommen werden.

```
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);
},
```

Workflow zurückschreiben: writeWorkflow(). Diese Funktion schreibt den Workflow aus einer lokalen Variablen in die Datenbank. Eine eventuell vorhandene Schreibsperre wird automatisch zurückgesetzt.

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

Lesesperre zurücksetzen: unlockWorkflow(). Wenn ein Workflow mit Schreibsperre gelesen wurde aber nicht verändert werden soll, kann man die Sperre mittels unlockWorkflow zurücksetzen.

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

Workflowknoten suchen: getNodeByName(). Diese Funktion sucht den Workflowknoten zu einem Knotennamen. Der Namen muss eindeutig sein, andernfalls wird der erste gefundene Knoten geliefert.

```
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;
},
```

Workflow aus Vorlage starten: startWorkflow(). Diese Funktion startet einen neuen Workflow zu einer ELO Objekt-Id aus einer Workflowvorlage.

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

1.3.6 mail: Mail Utils

Dieses Modul ist zum Senden von Emails gedacht. Es benötigt dafür einen SMTP Host, über den die Mails verschickt werden können. Dieser muss vor dem ersten Mailversand über die Funktion setSmtpHost bekannt gemacht werden. Anschließend kann man per SendMail oder SendMailWithAttachment Nachrichten versenden. Das Modul besteht aus zwei Teilen, zum Versenden von Mail und zum Lesen von Mail Postfächern.

1.3.6.1 Verfügbare Funktionen zum Lesen eines Postfachs

Im Ruleset kann definiert werden, dass als Basis nicht eine Suche im ELO Archiv oder in der ELO Aufgabenliste durchgeführt wird, sondern ein Postfach durchlaufen wird. Für jeden Postfachtyp muss im Modul mail eine Anmelderoutine hinterlegt werden. In dieser Funktion muss der Mail Server kontaktiert werden, der gewünschte Mail Folder gesucht werden und die Liste der Messages eingelesen werden. Danach übernimmt die normale ELOas Verarbeitung das Kommando. Für jede Mail wird ein Dokument in dem Ordner vorbereitet, der im SEARCHVALUE definiert wurde und anschließend wird darauf das Ruleset ausgeführt (der Betreff der Mail wird automatisch in die Kurzbezeichnung übernommen). Wenn der Eintrag am Ende nicht gespeichert wird, dann findet sich dazu auch nichts im Archiv. Nur gespeicherte Mails werden ins Archiv übertragen.

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

Im Ruleset muss als Name *MAILBOX_<Verbindungsname>* definiert werden und als Wert der Archivpfad oder die Nummer des Zielordners. Zudem muss auch die Maske definiert werden, die für die neuen Dokumente verwendet wird.

Im Skript des Rulesets wird dann die Mail verarbeitet. Auch hier bietet das Modul mail ein paar Hilfsroutinen, die das Leben etwas einfacher machen. In dem folgenden Beispiel wird der Mailkörper in den Zusatztext übertragen, Absender, Empfänger und MailID in die entsprechenden Indexzeilen der ELO Mailmaske:

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

Falls zusätzliche Werte oder Informationen benötigt werden, steht in der Variablen MAIL_MESSAGE ein vollständiges Java Mail (Mime)Message Objekt zur Verfügung.

Damit bereits bearbeitete Mails nicht doppelt ins Archiv übertragen werden, sollte vor der Verarbeitung eine Suche nach der MailID durchgeführt werden. Wenn die Mail bereits im Archiv ist, wird einfach die Variable MAIL_ALLOW_DELETE auf true gesetzt, andernfalls wird die Mail verarbeitet. Durch das Setzen des Löschflags wird die Mail beim Weiterschalten aus dem Postfach entfernt oder als Bearbeitet markiert.

```
  var msgId = MAIL_MESSAGE.messageID;
  if (ix.lookupIndexByLine(EM_SEARCHMASK, "ELOOUTL3", msgId) != 0) {
    log.debug("Mail bereits im Archiv 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;
  }
```

Diese Vorgehensweise liest eine Mail zwar doppelt ein (einmal zur normalen Verarbeitung und einmal im nächsten Durchlauf zum Löschen), hat aber den großen Vorteil, dass sichergestellt ist, dass die Mail erst dann aus dem Postfach gelöscht wird, wenn sie im Archiv auch wirklich vorhanden ist.

Wenn Sie ein Postfach zur Überwachung verwenden wollen, werden in der JavaScript-Library mail die folgenden vier Funktionen benötigt:

Verbindung aufnehmen, Postfachfolder öffnen: connectImap_<Verbindungsname>

Nächste Nachricht der Liste zur Bearbeitung: nextImap_<Verbindungsname>

Nachricht als bearbeitet markieren oder löschen: finalizeImap_<Verbindungsname>

Verbindung schließen: closeImap_<Verbindungsname>

Von diesen vier Funktionen muss in einfachen Fällen nur eine einzige Implementiert werden: Verbindung aufnehmen – connectImap_<Verbindungsname>. Da hier eine Vielzahl von projektspezifischen Aktionen stattfindet (Anmeldeparameter, Zielordner aufsuchen), gibt es keine Standardimplementierung. Die drei anderen Funktionen sind bereits mit einer Standardfunktion im System vorhanden. Sie müssen nur implementiert werden, wenn man zusätzliche Funktionen ausführen möchte.

Mit IMAP Server verbinden: connectImap_<Verbindungsname>(): Diese Funktion muss eine Verbindung zum Mailserver aufnehmen und das gewünschte Postfach aufsuchen und auslesen. Die vorhandenen Nachrichten werden in der Variablen MAIL_MESSAGES hinterlegt. Der Mail Store muss in der Variablen MAIL_STORE gespeichert werden und der ausgelesene Ordner in der Variablen MAIL_INBOX. Diese beiden Werte werden am Ende der Bearbeitung zum Schließen der Verbindung benötigt. Über die Variable MAIL_DELETE_ARCHIVED wird bestimmt, ob aus dem Postfach gelöscht werden darf. Wenn es auf false gesetzt wird, werden Löschanforderungen aus dem Ruleset ignoriert. Diese Funktion wird nicht direkt über ein Skript aufgerufen, sie wird ELOas-intern aktiviert (bei der MAILBOX Suche, im Beispiel *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;
},
```

Verbindung schließen: closeImap_<Verbindungsname>: Diese Funktion ist Optional und schließt die aktuelle Verbindung zum Imap Server. Wenn es keine speziellen Aufgaben beim Schließen gibt, müssen Sie diese Funktion nicht implementieren. Es wird stattdessen dann die Standardimplementierung closeImap() aus der Library verwendet. Diese schließt den Folder und den 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();
},
```

Message als bearbeitet markieren oder löschen: finalizeImap_<Verbindungsname>(): Diese Funktion ist Optional und löscht die aktuelle Nachricht oder markiert sie anderweitig als bereits bearbeitet. Wenn sie nicht implementiert wird, verwendet ELOam die Standardimplementierung, welche eine bearbeitete Mail aus dem Folder löscht.

Das Beispiel löscht die Mail nicht, sondern setzt sie nur auf „Gelesen“.

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

Nächste Message aus der Liste bearbeiten: nextImap_<Verbindungsname>: Diese Funktion ist Optional und liefert die nächste Nachricht aus dem ausgewählten Postfach zur Bearbeitung an das Ruleset. Wenn die Funktion nicht implementiert wird, verwendet ELOas die Standardimplementierung, welche jedes Dokument in die Bearbeitung gibt.

Das Beispiel zeigt eine Implementierung, welche nur ungelesene Mails bearbeitet. Sie kann im Paar in der oben aufgeführten finalizeImap Implementierung verwendet werden, welche bearbeitete Mails nicht löscht, sondern nur als gelesen markiert.

Achtung: Wenn Sie mit dieser Methode arbeiten, müssen Sie auf einen anderen Weg sicherstellen, dass das Postfach nicht über allen Maßen anwächst (z. B. durch eine automatische Löschung nach einem Zeitraum).

```
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;
},
```

Mailkörper Text lesen: getBodyText(): Dieser Funktion wird die Nachricht als Parameter übergeben (im Skript über die Variable MAIL_MESSAGE verfügbar) und liefert als Rückgabeparameter den Mailkörper. Dazu wird der erste MIME Part vom Typ TEXT/PLAIN gesucht. Wenn kein entsprechender Part vorhanden ist, wird ein Leerstring geliefert.

```
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 "";
},
```

Absender ermitteln: getSender(): Diese Funktion liefert die E-Mail-Adresse des Absenders.

```
getSender: function(message) {
```

```
  var adress = message.sender;
  return adress.toString();
},
```

Empfänger ermitteln: getRecipients(): Diese Funktion liefert eine Liste aller Empfänger (TO und CC). Wenn es mehr als einen Empfänger gibt, wird die Liste im Spaltenindex Format geliefert, wenn man im Parameter „delimiter" das ELO Trennsymbol ¶ übergibt.

```
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;
}
```

1.3.6.2 Verfügbare Funktionen zum Versenden von Mails

Die Versende-Funktionen werden nicht direkt vom ELOas verwendet. Es sind Hilfsfunktionen zur eigenen Skriptprogrammierung, um die Komplexität des Java Mail APIs vor dem Skriptentwickler zu verdecken.

SMTP Server anmelden: setSmtpHost(): Diese Funktion macht der Library den zu verwendenden SMTP Host bekannt. Er wird für den Mailversand verwendet. Diese Funktion muss vor dem ersten „sendMail"-Aufruf aktiviert werden.

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

Mail versenden: sendMail(): Diese Funktion sendet eine E-Mail. Als Parameter werden die Absender- und Empfängeradresse mitgegeben sowie der Betreff und der Mailtext.

```
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);
},
```

Mail mit Attachment versenden: sendMailWithAttachment(): Diese Funktion sendet eine E-Mail. Als Parameter werden die Absender- und Empfängeradresse mitgegeben sowie der Betreff, der Mailtext und die Objekt-ID für das Attachment aus dem ELO Archiv. Das Attachment wird als temporäre Datei im Temp-Pfad zwischengespeichert, dort muss also ausreichend Platz verfügbar sein.

```
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"]();
}
```

1.3.7 fu: File Utils

Die Funktionen aus dem Bereich File Utils unterstützen den ELOas Benutzer bei Dateioperationen.

1.3.7.1 Verfügbare Funktionen

Dateiname bereinigen: clearSpecialChars(): Wenn ein Dateiname aus der Kurzbezeichnung erzeugt werden soll, dann kann diese kritischen Zeichen enthalten, die im Filesystem zu Problemen führen können (z. B. Doppelpunkt, Backslash \, &). Diese Funktion ersetzt deshalb alle Zeichen außer Ziffern und Buchstaben durch einen Unterstrich (auch Umlaute und ß).

```
clearSpecialChars: function(fileName) {
```

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

Dokumentendatei laden: getTempFile(): Diese Funktion lädt die
Dokumentendatei des angegebenen ELO Objekts in das lokale Filesystem (in
den Temp Ordner des ELOas). Wenn die Datei nicht mehr benötigt wird, muss
sie durch den Skriptentwickler über die Funktion deleteFile wieder entfernt
werden. Andernfalls bleibt sie auf der Festplatte zurück.
Achtung: es wird nicht ein Dateiname, sondern ein Java File Objekt
zurückgegeben.

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;
},
```

Datei löschen: deleteFile(): Diese Funktion erwartet als Parameter ein Java File Objekt (kein String) und löscht diese Datei.

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

1.3.8 run: Runtime Utilities

Dieses Modul enthält Routinen zum Zugriff auf die Java Runtime. Hierüber können externe Prozesse gestartet werden oder der aktuelle Speicherzustand abgefragt werden.

Prozess starten: execute(command): Dieser Befehl dient zum Starten eines externen Prozesses. Der ELOas warten dann auf den Abschluss dieses Aufrufs und fährt erst dann mit der Bearbeitung fort. Somit können also auch Aktionen dieses Prozesses ausgewertet werden.

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

Freien und verfügbaren Speicher abfragen: freeMemory() und maxMemory(): Diese Befehle dienen zur Anzeige des aktuell verfügbaren freien Speichers und des maximal verfügbaren Speichers.

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

2 Manueller Start eines Rulesets

Normalerweise führt der ELOas die definierten Rulesets intervallgesteuert aus. Es gibt aber auch Vorgänge, die in der Abarbeitung so umfangreich sind, dass sie nicht in kurzen Intervallen ausgeführt werden können, auf der anderen Seite aber auch möglichst schnell nach einer bestimmten Veränderung aktiv werden müssen. Hierzu gibt es die Möglichkeit, dass die Ausführung eines Rulesets durch eine URL manuell (bzw. per Skript) gestartet wird.

2.1 Beispiel

Das nachfolgende Beispiel zeigt, wie man aus einem Client Skript heraus ein Ruleset aufrufen kann, der dann bestimmte Objekte verändert.

Achtung: Da der Aufruf über einen http Zugriff erfolgt, kann prinzipiell jeder Benutzer auch per Browser oder per Skript Befehl diese Aktion auslösen. Es muss deshalb sichergestellt werden, dass die Funktion nicht missbraucht werden kann (z. B. durch Kontrolle der Benutzernummer oder durch eine feste interne Vorgabe der Objekt IDs).

Zuerst soll das verwendete Ruleset betrachtet werden. Durch die Angabe eines Intervalls von null Minuten (<interval>0H</interval>) wird dieses Ruleset als manuell getriggert definiert. Er wird also nicht zyklisch aufgerufen, sondern wartet auf den Empfang einer bestimmten URL.

```
<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 = "Freigegeben: " + NAME;
    EM_WRITE_CHANGED = true;

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

```
</rule>
</ruleset>
```

Der interessante Teil liegt im Skript Bereich:

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

Der Aufruf kann bis zu drei Parameter mitgeben. Diese können vom Ruleset über die Variablen EM_PARAM1, EM_PARAM2 und EM_PARAM3 abgefragt werden. Zudem kann das Skript optional das Ticket der aktuellen Anmeldung zur Authentifizierung mitgeben. In diesem Fall ist die Variable EM_USERID mit der Nummer des angemeldeten Benutzers gefüllt. Wenn keine Authentifizierung vorliegt, ist die Benutzernummer mit -1 belegt. Im ersten Parameter können eine oder mehrere Objekt-IDs übergeben werden, diese überschreiben dann den Search-Value aus der Ruleset-Definition. Als Indexzeilenname muss in diesem Fall OBJIDS angegeben werden.

```
    NAME = "Freigegeben: " + NAME;
```

Im Beispiel wird der Kurzbezeichnung der ausgewählten Objekte einfach der Text „Freigegeben“ vorangestellt. Hier können aber auch beliebige andere Veränderungen des SORD-Objekts durchgeführt werden.

```
    EM_WRITE_CHANGED = true;
```

Da das Objekt verändert wurde, soll es auch gespeichert werden.

```
  </script>
```

2.2 Aktivierung des Rulesets

Nach dem Start des ELOas wird dieses Ruleset mit gestartet aber noch nicht aktiv. Er wartet auf einen externen Trigger (sichtbar an dem Text „Trigger“ im Feld *Next run*).

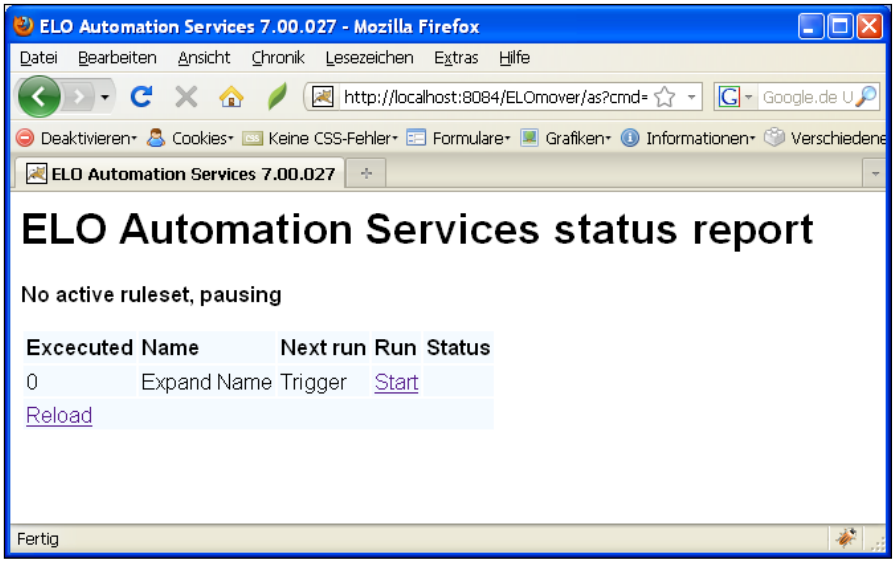


Abb. 9: ELOas Statusseite

Das Triggern kann durch den Aufruf einer URL, oder aus dem Windows Client heraus durch einen Skript Befehl erfolgen (ab Client Version 7.00.056):

```
SendELOasRequest( <Servername>, <Portnummer>, <Servicename>, <mit Ticket>,
<Ruleset Name>, <Parameter 1>, <Parameter 2>, <Parameter 3>)
```

Der Befehl *SendELOasRequest* macht einen asynchronen Aufruf mittels *run*. Ein solcher Regelsatz wird in der Administration Console nicht unter *Direct*, sondern unter *Regeln* angezeigt.

Servername	Name oder IP Adresse des ELOas Servers.
Portnummer	Portnummer des ELOas Servers. Im Normalfall 8080, Standard http Port.
Servicename	Servicename des ELOas Servers. In einer Standardinstallation setzt er sich aus dem Präfix as- und dem Archivnamen zusammen (z. B. as-ELO). Dabei ist unbedingt auf die korrekte Groß/ Kleinschreibweise zu achten, andernfalls meldet der Tomcat Server einen Fehler.
Mit Ticket	0: keine Anmeldeinformation mit senden

	1: aktuelles Ticket als Anmeldeinformation mit senden. In diesem Fall prüft der ELOas das Ticket und ermittelt hieraus die Benutzernummer. Diese Information wird dann dem Ruleset zur Verfügung gestellt. Im Ruleset kann dann entschieden werden, ob und in welchen Umfang die Aktion ausgeführt wird. Im Augenblick kann die Anmeldeinformation bei der SSO-Anmeldung nicht ausgewertet werden. Das wird in der nächsten Indexserverversion geändert.
Ruleset Name	Name des auszuführenden Rulesets. Es können nur getriggerte Rulesets aufgerufen werden. Bei intervallgesteuerten Rulesets wird der Aufruf ignoriert.
Parameter1	Erster Parameter. Dieser Parameter wird, wenn er nicht leer ist, als Suchbegriff für die Ruleset-Ausführung verwendet.
Parameter2 Parameter3	Weitere optionale Parameter. Diese können vom Ruleset abgefragt werden und die Ausführung steuern.

Das komplette Beispielskript für einen Aufruf kann dann so aussehen. Es ruft das Ruleset *Expand Name* für die Objekte mit der ObjId 7944 und 7945 auf.

```
Set Elo=CreateObject("Elo.Professional")
MsgBox Elo.SendELOasRequest("localhost", 8084, "/ELOmover/as" , 1, "Expand
Name", "7944,7945", "TestParam2", "")
```

Das Ruleset kann auch aus beliebigen anderen Applikationen durch Aufruf einer URL getriggert werden:

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



Beachten Sie: In diesem Fall können Sie keine Authentifizierungsinformation mitgeben. Stellen Sie im Ruleset sicher, dass kein Missbrauch erfolgen kann.

2.3 Weitere Hinweise

2.3.1 Asynchrones Triggern des Rulesets

Wenn ein Ruleset durch eine URL oder einen Skriptaufruf getriggert wird, führt der ELOas diesen asynchron aus. Falls also gerade ein anderes Ruleset aktiv ist, wird die Skriptausführung nicht so lange verzögert, bis der ELOas verfügbar ist. Stattdessen wird der Aktivierungsbefehl in eine Warteschlange gestellt und bei der nächsten Möglichkeit ausgeführt.

Das hat zwei Konsequenzen: Das Client-Skript kann sich nicht darauf verlassen, dass mit der Abarbeitung des Befehls tatsächlich auch die Operation durchgeführt wurde. Falls das für den weiteren Skriptverlauf wichtig ist, muss das im Skript selbst geprüft werden und in eine Warteschleife integriert werden. Bedenken Sie dabei, dass der ELOas möglicherweise gerade sehr umfangreiche andere Aktionen ausführt. Im Allgemeinen sollte ein Skript also nicht auf den Abschluss einer ELOas Aktion warten.

Weiterhin kann es sein, dass ein ungeduldiger Benutzer das Triggern mehrfach veranlasst. In diesem Fall wird das Ruleset auch mehrfach ausgeführt. Er sollte also so angelegt sein, dass dieses mehrfache Triggern nicht zu Fehlern führt, z. B. indem das Objekt vorab geprüft wird und die wiederholte Ausführung dann abgebrochen wird.

Noch eine Folge entsteht aus der asynchronen Ausführung: Fehler, die beim Abarbeiten des Rulesets aufgetreten sind, können nicht über den Aufruf zurückgemeldet werden.

2.3.2 Synchrones Triggern des Rulesets

Beim synchronen Triggern wird das Ruleset direkt ausgeführt und kann auch ein Ergebnis zurückliefern. Das synchrone Triggern wird vor allem vom ELO Workflow (Formulareditor) verwendet. Bei dem Aufruf wird statt "cmd=run" ein "cmd=get" benötigt. Zudem müssen die Rulesets für den synchronen Aufruf nicht im Ordner "Rules" sondern im Ordner *Direct* stehen. Die synchronen Rulesets werden unabhängig von den asynchronen Rulesets in einem eigenen Thread ausgeführt.

2.3.3 Berechtigungsprüfung

Beim Aufruf aus dem Windows Client kann optional das Client Ticket der Anmeldung mit übertragen werden. In diesem Fall kann der ELOas die Anmeldung prüfen und den aktuellen Benutzer ermitteln. Bei kritischen Aktionen sollte im Ruleset auf jeden Fall eine Prüfung auf einen berechtigten Benutzer erfolgen und bei fehlender oder unzureichender Anmeldung eine Ausführung abgebrochen werden.

Ein anonymes Triggern kann in bestimmten Fällen durchaus akzeptabel sein. Zum Beispiel, weil ein bestimmtes festgelegtes Objekt bearbeitet wird. In diesem Fall muss man darauf achten, dass die Objekt-ID nicht durch den Aufruf verändert werden kann. Das kann am einfachsten im onstart Event erfolgen indem der Wert von EM_SEARCHVALUE im Skript gesetzt wird. In diesem Fall wird für die Suche nicht der Parameter, sondern der voreinstellte Wert aus dem Ruleset-Skript verwendet.

2.3.4 Ausführungsreihenfolge

Ein manuell getriggertes Ruleset fügt sich ganz normal in die Ausführungsreihenfolge des Rulesets ein. Wenn zu einem Ruleset mehrere Trigger vorliegen, werden erst alle Trigger abgearbeitet bevor das nächste Ruleset bearbeitet wird.

3 Aufbau der Rule-Struktur

Dieses Dokument beschreibt den Aufbau der XML Rule Struktur des ELOas. Diese Struktur wird im Normalfall über ein grafisches User Interface gepflegt werden. Falls trotzdem manuell eingegriffen werden muss, kann diese Beschreibung als Referenz verwendet werden. Ebenso dient diese Beschreibung als Referenz für die Implementierung des GUI.

3.1 Allgemeiner Aufbau

Die komplette Struktur ist in ein Tag <ruleset> eingebettet. Dieses besteht aus zwei Teilen, einem <base> Eintrag am Anfang, gefolgt von beliebig vielen <rule> Einträgen.

Der <base> Eintrag beinhaltet die Informationen zur Suche der zu bearbeitenden Einträge. Diese umfassen die Suchzeile, den Suchbegriff, Masken und Datumseinschränkungen.

Die <rule> Einträge beinhalten jeweils eine Verarbeitungsvorschrift. Jede Regel kann mit einer Bedingung belegt werden, sie kann das Ablageziel verändern oder Inhalte der Indexzeilen anpassen. Zudem kann eine Rule auch einen JavaScript-Inhalt haben. Wenn dieser definiert ist, dann werden die anderen Einträge ignoriert, sie dürfen aber mit Werten gefüllt bleiben.

Wenn die Bedingung einer Regel „OnError“ lautet, dann handelt es sich bei dieser Regel um eine Fehlerbehandlungs-Regel. Nach jeder Regel darf eine Fehlerbehandlung erfolgen, ganz am Ende muss eine Fehlerbehandlungs-Regel eingetragen werden. Diese abschließende Fehlerregel wird auch dann aufgerufen, wenn beim Verschieben oder Speichern ein Fehler auftritt. Wenn bei der Bearbeitung innerhalb einer normalen Regel ein Fehler auftritt, wird die nächstmögliche Fehlerbehandlungs-Regel aufgerufen und danach die Bearbeitung abgebrochen.

Beispiel für ein einfaches Ruleset:

```
<ruleset>
  <base>
    <name>Name des Rulesets</name>
    <search>
      <name>Name der Indexzeile im JavaScript Code</name>
      <value>Suchbegriff im JavaScript Code</value>
      <mask>Nummer der Dokumentenmaske für die Suche</mask>
    </search>
    <interval>5M</interval>
  </base>
  <rule>
    <name>Name der Regel</name>
    <destination mask=“Ordnermaske“> Neues Ziel im JavaScript
Code</destination>
    <index>
      <name>Name der Indexzeile im JavaScript Code</name>
      <value>Neuer Inhalt der Indexzeile im JavaScript Code</value>
    </index>
```

```
  </rule>
  <rule>
    <name>Name der Fehlerbehandlungs-Regel</name>
    <condition>OnError</condition>
  </rule>
</ruleset>
```

3.1.1 Alle Einträge im <base> Abschnitt

Tag	Funktion	Beispiel
name	Name des Rulesets. Dieser Name wird auf der Statusseite angezeigt, er wird aber nicht weiterverarbeitet.	SAP Verarbeitung
search	Parameter für die Suche der zu verarbeitenden Dokumente. Beschreibung siehe nachfolgenden Abschnitt *3.1.2 Alle Einträge im Abschnitt ‚<search>'*	
masks	Wenn im Rahmen der Verarbeitung auf einen anderen Ablagemaskentyp umgeschaltet werden soll, dann muss hier eine Liste aller möglichen Zielmaskennummern aufgeführt werden. Jedes Maskennummer wird durch ein <mask> Tag umrahmt.	<mask>3</mask> <mask>4</mask>
interval	Wiederholungsintervall für die Abarbeitung der Suche. Dieses Intervall kann in Minuten (5M) oder Stunden (1H) angegeben werden. Weiterhin kann es einmal pro Tag zu einer bestimmten Uhrzeit (15:30) ausgeführt werden, einmal pro Wochentag (17:20/SA) oder einmal pro Monat (22:00/31). Falls bei der monatlichen Ausführung ein Tag angegeben wird, der im aktuellen Monat nicht vorkommt (z. B. der 31. In einem Februar), dann wird jeweils der letzte Tag des Monats verwendet.	5M 1H 15:30 17:20/SA 22:00/31

3.1.2 Alle Einträge im Abschnitt ‚<search>'

Die Einträge im Abschnitt *<search>* bestimmen welche Dokumente bearbeitet werden. Zum Beginn jedes Durchlaufs wird eine Suche mit diesen Parametern ausgeführt. Die Trefferliste wird dann anhand der Regeln bearbeitet.

```
<search>
  <name>Name der Indexzeile im JavaScript Code</name>
  <value>Suchbegriff im JavaScript Code</value>
  <mask>Nummer der Dokumentenmaske für die Suche</mask>
  <max>Maximale Anzahl der Dokumente pro Durchlauf</max>
</search>
```

Tag	Funktion	Beispiel
name	Name der Indexzeile im JavaScript-Code. Wenn der Name feststehend ist, kann direkt ein Text in Anführungszeichen eingegeben werden. Es kann aber auch ein beliebiger JavaScript-Ausdruck verwendet werden.	"ELOOUTL2"
value	Suchbegriff im JavaScript-Code. Wenn der Wert feststehend ist, kann direkt ein Text in Ausführungszeichen eingegeben werden. Es kann aber auch ein beliebiger JavaScript-Ausdruck verwendet werden.	"ELO*"
mask	Nummer der Dokumentenmaske für die Suche. Es kann hier nur eine Dokumentenmaske, keine reine Suchmaske verwendet werden, da beim Einlesen davon ausgegangen wird, dass alle Treffer die gleiche Maskendefinition besitzen.	2
max	Maximale Anzahl der Dokumente pro Durchlauf bei der Suchabfrage zum Indexserver. Falls mehr Treffer vorliegen, werden diese in einem weiteren Durchlauf bearbeitet nachdem alle anderen Rulesets durchlaufen wurden. Damit soll verhindert werden, dass ein umfangreiches Ruleset die Abarbeitung aller anderen Rulesets unterdrückt. Es sind maximal 1000 Dokumente pro Durchlauf zulässig.	200
idate xdate	Die Trefferliste kann durch ein Datumsbereich im Ablage- (idate) oder Dokumentendatum (xdate) eingeschränkt werden. Dieses Datum kann entweder in absoluten Werten im ISO Datumformat (YYYYMMTT) oder in relativen Werten zum aktuellen	`<idate>   <from>-5</from>   <to>+0</to> </idate>`

	Tag (-5) eingetragen werden. Der Bereich besteht jeweils aus einem Startdatum in einem <from> Tag und einem Endedatum in einem <to> Tag.	

3.1.3 Alle Einträge im Abschnitt ‚<rule>'

Nach dem Abschnitt *<base>* kommen beliebige viele <rule>-Abschnitte. Diese werden bei der Abarbeitung in der Reihenfolge der Definition durchlaufen.

Eine Rule kann in zwei unterschiedlichen Ausprägungen vorliegen: als normale Regel und als Fehlerregel. So eine Fehlerregel wird in der normalen Ausführung einfach übersprungen. Nur im Fehlerfall wird die nächste verfügbare Fehlerregel in der Reihenfolge aufgerufen, danach wird die Abarbeitung für dieses Dokument abgebrochen, d.h. – nach einer Fehlerregel werden keine weiteren Regeln abgearbeitet.

Die letzte Regel in der <rule>-Kette muss immer eine Fehlerregel sein. Damit ist sichergestellt, dass auf jeden Fall eine Fehlerbehandlung zur Verfügung steht. Zudem wird diese Regel auf aufgerufen, wenn beim Verschieben oder Speichern ein Fehler auftritt.

Tag	Funktion	Beispiel
name	Name der Regel, wird nur zur besseren Lesbarkeit und zur Dokumentation benötigt.	Indexergänzung
condition	Ausführungsbedingung für diese Regel. Falls es sich um eine Fehlerregel handelt, wird hier der feste Text „OnError" eingetragen. Hier ist auf eine korrekte Schreibweise zu achten, andernfalls wird die Regel nicht als Fehlerregel erkannt. Die Ausführungsbedingung liegt als JavaScript-Code vor. Nur, wenn die Bedingung „true" ist, wird die Regel ausgeführt.	KDNR == "123"
destination	Neues Ablageziel des Dokuments als Archivpfad. Der Eintrag ist optional und kann leer bleiben, in diesem Fall bleibt das Dokument an seiner ursprünglichen Position. Falls es mehrere Regeln mit destination gibt,	<destination mask="1">¶ELO¶Mails¶" + ELOOUTL1</destination>

	wird das erste Ziel als neuer Ablageort verwendet, alle weiteren Ziele werden zusätzlich als Referenzen eingetragen. Wenn ein Ablageziel noch nicht vorhanden ist, wird es automatisch angelegt. Das destination Tag kann ein zusätzliches Attribut „mask“ enthalten, welches die Nummer der Ordnermaske für neu erzeugte Ordner enthält. Wenn dieses Attribut nicht vorhanden ist, wird als Default die „1“ verwendet, die Nummer der Ordnermaske in einem Standardarchiv.	
mask	Neue Verschlagwortungsmaske des Dokuments. Wenn dieser Eintrag vorhanden ist oder die Maskennummer -1 ist, bleibt die ursprüngliche Verschlagwortungsmaske erhalten. Wenn die Maske gewechselt wird, werden automatisch alle Einträge mit gleichem Gruppennamen übernommen. Das wird auch dann korrekt ausgeführt, wenn die Indexzeilenaufteilung unterschiedlich ist. Falls die ursprüngliche Ablagemaske Indexzeilen enthielt, die die neue Maske nicht besitzt, werden diese Daten automatisch und ohne Fehlermeldung verworfen. Der ELOas kann keine Dokumente bearbeiten, deren Ablagemaske den gleichen Gruppennamen für mehrere Einträge verwendet, da die interne Verarbeitung und die Struktur der Regeln von einer eindeutigen Zuordnung ausgehen.	<mask>20</mask>
index	Innerhalb einer Regel kann es beliebig viele Indexeinträge geben. Jeder Indexeintrag enthält den Namen der betroffenen Indexzeile und einen JavaScript-Ausdruck mit dem neuen Wert. Bei Indexzeilen mit ISO-Datum und den	<index> <name>DOCDATE</name> <value>"20070930"</value> </index>

	Feldern für das Ablage- und Dokumentendatum muss die Eingabe auch im ISO Datumsformat erfolgen. Neben den Indexfeldern mit den Gruppennamen der Suchmaske stehen alle Gruppennamen der alternativen Ablagemasken und eine Reihe von Pseudo-Indexzeilen mit einigen Standardwerten der Verschlagwortung zur Verfügung: NAME: Kurzbezeichnung DOCDATE: Dokumentendatum ABLDATE: Ablagedatum ARCHIVINGMODE: Revisionsstatus 0, 1 oder 2 für Freie Bearbeitung, Versionskontrolliert oder Revisionssicher. ACL: mit „PARENT“ die ACL des neuen Ablageziels übernehmen. Mit <Rechte>:<Name> beliebige Gruppenberechtigungen definieren. OBJCOLOR: Farbnummer des Eintrags OBJDESC: Zusatztext OBJTYPE: Dokumenten- oder Ordnertyp des Eintrags. Achtung: durch eine falsche Zuordnung kann es zu Störungen bei der weiteren Verarbeitung kommen. Dokumente dürfen nur einen OBJTYPE zwischen 254 und 286 besitzen.	
script	Eine Regel kann auch direkt den auszuführenden JavaScript-Code enthalten. In diesem Fall werden allen anderen Parameter dieser Regel ignoriert, sie dürfen aber (z. B. zu Dokumentationszwecken) weiter enthalten sein.	

3.1.4 Berechtigungsänderungen

In der Pseudo-Indexzeile ACL kann eine geänderte Berechtigung hinterlegt werden. Im einfachsten Fall trägt man hier „PARENT“ ein, dann wird beim Speichern die Berechtigung des Zielordners für diesen Eintrag übernommen. Es kann aber auch eine komplette Berechtigungsliste hinterlegt werden. Diese Liste besteht aus einer Folge von Einzelberechtigungen, die durch ein Pilcrow-Symbol getrennt werden. Jede Einzelberechtigung besteht aus der Berechtigungsmaske (RWDEL – Read, Write, Delete, Edit, List) gefolgt von einem Doppelpunkt und dem Gruppennamen. Bei UND Gruppen wird statt des einfachen Gruppennamen eine Folge von Namen, getrennt jeweils auch durch einen Doppelpunkt angegeben.

```
R:Jeder¶RW:Controlling¶RWDEL:Verwaltung:Stuttgart:Leitungsteam
```

Im Beispiel erhält die Gruppe *Jeder* einen Lesezugriff, die Gruppe *Controlling* Lese- und Schreibzugriff und die UND-Gruppe *Verwaltung und Stuttgart und Leitungsteam* Vollzugriff auf das Dokument.

Falls eine Berechtigung auf einen Benutzer statt auf eine Gruppe gesetzt werden soll, muss in die Berechtigungsliste zusätzlich noch ein „U“ aufgenommen werden.

```
UR:Administrator
```

3.2 Anmerkungen

Bei der Generierung des JavaScript-Codes werden alle Gruppennamen der Suchmaske und der alternativen Ablagemasken als Variable in vollständiger Großschreibweise angelegt. Dieses Verfahren minimiert das Risiko, dass sich Gruppennamen mit Standardbezeichnern aus JavaScript oder der ELO Laufzeitumgebung überschneiden. Prinzipiell kann es aber trotzdem zu Problemen kommen, falls einer der Gruppennamen identisch zu einem Standardbezeichner oder einer der Übersetzungslisten ist.

```
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: Diese Liste kann sich im Laufe des Projektfortschritts erweitern, insbesondere kann sie auch durch lokales Customizing um zusätzliche Einträge ergänzt werden.

Die Nummer der Ablagemaske des aktuellen Dokuments kann über eine Regel geändert werden. Falls es sich dabei um eine ungültige Maskennummer oder um eine Nummer handelt, die nicht in der Liste der alternativen Zielmasken aufgeführt wurde, erzeugt das erst beim Speichern des Dokuments einen Laufzeitfehler und nicht direkt bei der Maskenzuweisung.

Wenn aufgrund eines Laufzeitfehlers eine Fehlerregel aufgerufen wird, dann löscht diese alle bereits vorbelegte Ablageziele der zuvor abgearbeiteten Regeln. Wenn die Fehlerregel keine eigene <destination> besitzt, bleibt das Dokument an seiner ursprünglichen Position, andernfalls wird das Ziel der Fehlerregel verwendet.

Das Verschieben und die Speicherung der geänderten Verschlagwortung passieren erst am Ende nach der Abarbeitung der letzten Regel. Wenn es dabei zu einem Fehler kommt, wird also die letzte Fehlerregel aufgerufen und nicht die Fehlerregel, die zu der Regel gehört, die das Ziel festgelegt hat (was aber identisch ist, wenn es nur eine einzige Fehlerregel gibt).

3.3 Beispielhafte Struktur

Im Folgenden wird eine beispielhafte Definition aufgeführt, dazu wird der generierte Code aufgelistet. Diese Information dient nur der Orientierung.

```
<ruleset>
 <base>
  <name>Mailmaske Thiele</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>Regel 1</name>
  <destination mask="5">"¶Thiele¶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-Kopie</name>
  <destination mask="1">"¶Thiele¶Journale¶" + ELOOUTL1</destination>
 </rule>
<rule>
  <name>Script rule</name>
  <script>
    moveTo(Sord, "¶Ablage¶Ziele1¶" + ELOOUTL1);
    moveTo(Sord, "¶Ablage¶Ziele2¶" + ELOOUTL2);
    moveTo(Sord, "¶Ablage¶Ziele3¶" + 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>
```

4 Beispiel - Mailpostfach überwachen

Das JavaScript-Library der ELO Automation Services enthält ein Modul zum Senden und Empfangen von E-Mails. Diese Anleitung soll zeigen, wie man die ELOas zur Überwachung eines Postfachs verwenden kann.

> **Information:** Dieses Beispiel soll keine Mail Archivierung simulieren. Dafür haben wir andere Module in unserer Produktliste, die dafür geeignet sind. Es soll vielmehr als Basis für „Autoresponder" dienen, also Programme, die auf eine Mail automatisch eine Aktion auslösen (z. B. ein Benutzer sendet eine Registrierungsmail und daraufhin wird sein Konto freigeschaltet).

4.1 Allgemeine Vorgehensweise

Bevor ein Ruleset zur Bearbeitung von Postfächern erstellt werden kann, muss im Modul *mail* eine Postfachverbindung erstellt werden. Da es hier viele Unterschiede und Möglichkeiten gibt, kann man hier nicht mit einer einfachen Konfigurationsliste arbeiten. Stattdessen ist es notwendig, dass für jede Postfachverbindung eine „connect" Methode erstellt wird. Diese muss die Verbindung zum Mailserver aufbauen, das richtige Postfach auswählen und die Nachrichtenliste einlesen.

Jede Postfachverbindung erhält einen einfachen, kurzen Namen – z. B. GMAIL. Dieser Name wird an verschiedenen Stellen benötigt und muss „Identifier-Tauglich" sein, d.h. er muss mit einem Buchstaben beginnen und darf danach weitere Buchstaben oder Ziffern enthalten (aber keine Umlaute, das sind aus amerikanischer Sicht keine Buchstaben). Dieser Name wird dann an verschiedenen Stellen im Ruleset und in der JavaScript Implementierung benötigt.

4.1.1 Verbindungsaufbau

Die JavaScript Library bringt bereits in der Standardinstallation eine Definition für eine Verbindung mit dem Namen GMAIL mit. Diese werden wir für das Beispiel verwenden. Da der Verbindungsname in die speziellen Funktionen einfließt, können Sie auch mehrere Verbindungen parallel definieren und in unterschiedlichen Rulesets verwenden.

Die Standardfunktion für den GMAIL Verbindungsaufbau sieht so aus:

```
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", "<ANWENDER>@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_DELETE_ARCHIVED = false;
},
```

Das Beispiel verbindet sich mit dem GOOGLEMAIL-Server „imap.gmail.com“ auf dem Port „993“ über eine verschlüsselte Verbindung („mail.store.protocol“ – „imaps“). Diese Informationen werden in ein Property Objekt eingetragen. Ihr eigener Mail Server benötigt evtl. andere Werte – diese müssen Sie der Mailserver-Anleitung entnehmen.



Information: Wenn Sie ein Google-Mail-Konto einrichten, müssen Sie es zuerst für einen IMAP-Zugriff freischalten. Das ist möglich unter *Einstellungen > Weiterleitung und POP/IMAP> IMAP aktivieren*.

Die Anmeldung wird dann über den Befehl *MAIL_STORE.connect* ausgeführt. Hier muss nochmals der Servername angegeben werden und der Postfach-Benutzer mit Passwort.

Nach der Anmeldung wird als nächstes der Ordner *Posteingang* aufgesucht. Man könnte durchaus auch andere Ordner überwachen, z. B. *Gesendet*:

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

Über den Befehl *MAIL_INBOX.getMessages()* werden schließlich alle Mails des Ordners eingelesen und in die interne Message Liste aufgenommen. Diese Liste wird dann später abgearbeitet indem das Ruleset für jeden Eintrag dieser Liste einmal aufgerufen wird.

Die Variable *MAIL_DELETE_ARCHIVED* bestimmt, ob das Ruleset nach erfolgreicher Bearbeitung die Nachricht löschen oder als bearbeitet markieren darf. Wenn er, wie in der Voreinstellung, auf „false“ steht, wird der Nachrichtenstatus nicht verändert. Das ist besonders in der Testphase sehr praktisch, damit man sich nicht ständig neue Mails erzeugen muss. Im Betrieb wird dieser Eintrag im Normalfall auf „true“ stehen.

4.1.2 Ruleset erstellen

Ein einfaches Ruleset zur Abarbeitung des Postfachinhalts besteht aus zwei wesentlichen Teilen: der Definition der Suche und dem Skript zur Abarbeitung der Mail.

Die Suche wird folgendermaßen definiert:

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

```
<max>200</max>
</search>
```

Der Suchname „MAILBOX_GMAIL“ signalisiert, dass es sich nicht um eine normale Archivsuche, sondern um ein Postfach mit dem Verbindungsnamen GMAIL handelt. Die erzeugten ELO Dokumente werden im Archivschrank „IMAP“ abgelegt (über „ARCPATH:¶IMAP“) und werden mit der Maske 2 (Email in einem Standard ELO Archiv) erzeugt. Die Zahl der Treffer wird im Normalfall nicht weiter beachtet, sie sollte trotzdem eingetragen werden damit es im Designer nicht zu einer Fehlermeldung kommt.

Das Skript zur Ausführung wird im Wesentlichen von der benötigten Funktion bestimmt. Ein ganz einfacher Rahmen könnte so aussehen:

```
<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>
```

Bei der Skriptausführung ist die Nachricht in der Variablen MAIL_MESSAGE verfügbar. Hieraus können dann Standardwerte wie Mailtext, Absender und Empfänger ausgelesen werden. Damit das etwas einfacher geht, stellt das Modul mail hierfür die Hilfsroutinen getBodyText, getSender und getRecipients zur Verfügung.

Der Betreff wird automatisch in die Kurzbezeichnung (NAME) übernommen. Der Mailkörper wird in den Zusatztext geschrieben und Absender und Empfänger in die jeweiligen Indexzeilen übertragen. Zuletzt wird die Nachricht über MAIL_ALLOW_DELETE als Bearbeitet markiert oder gelöscht.

Das komplette Beispiel sieht dann so aus:

```
<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>
```

4.2 Überwachte Bearbeitung

Das einfache Beispiel hat einen wesentlichen Nachteil: wenn eine Mail bereits als „Bearbeitet“ markiert oder gelöscht wurde und der Prozess abgebrochen wird, bevor die Daten im Archiv gespeichert werden konnten, bleibt ein Datensatz unbearbeitet. Dieses Problem kann man komplett vermeiden, indem man mit einem zweistufigen Ansatz arbeitet: eine neue Mail wird zuerst nur im ELO gespeichert aber noch nicht gelöscht. Erst wenn bei einem späteren Lauf eine Mail gefunden wird, die bereits im ELO vorhanden ist, wird sie gelöscht.

Für diese Vorgehensweise sind zwei Voraussetzungen nötig: die Mail muss eindeutig erkennbar sein und bei der Abarbeitung muss geprüft werden, ob sie bereits im Archiv vorhanden ist. Der erste Punkt ist leicht erfüllbar: jede Mail hat eine interne Mail-ID. Diese kann im ELO in einer Indexzeile gespeichert werden (z. B. in der Standard Mail Maske in der Indexzeile ELOOUTL3, diese ist für die Mail-ID vorgesehen). Der zweite Punkt kann durch eine Hilfsroutine aus dem Modul ix leicht erfüllt werden: ix.lookupIndexByLine.

Das geänderte Skript sieht dann so aus:

```
<script>
  log.debug("Process Mailbox: " + NAME);
  // wenn die Nachricht bereits im Archiv ist: dann löschen.
  var msgId = MAIL_MESSAGE.messageID;
  if (ix.lookupIndexByLine(EM_SEARCHMASK, "ELOOUTL3", msgId) != 0) {
    log.debug("Mail bereits im Archiv 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;
  }
</script>
```

4.3 Markieren statt löschen

In der Standardimplementierung wird eine bearbeitete Mail aus dem Postfach gelöscht. Das ist in manchen Fällen nicht gewünscht. Stattdessen kann man auch eine Markierung vornehmen. Ein möglicher Kandidat ist das Flag „Gelesen". Eine bearbeitete Mail wird vom ELOas auf „gelesen" gesetzt und unterscheidet sich damit von einer neuen Mail. In diesem speziellen Fall müssen neben der connectImap Methode noch weitere Methoden in der mail JavaScript Library definiert werden:

nextImap_GMAIL(): Diese Funktion schaltet auf die nächste Nachricht weiter. Sie muss in diesem Beispiel prüfen, ob eine Mail bereits als gelesen markiert wurde und diese dann bei Bedarf überspringen.

finalizeImap_GMAIL(): Diese Funktion markiert die bearbeitete Nachricht. In der Standardimplementation wird die Nachricht gelöscht. In unserem Beispiel hingegen soll sie nur als Gelesen markiert werden.

4.3.1 nextImap_GMAIL

Diese Funktion schaltet auf die nächste Nachricht weiter. Dafür läuft sie sequenziell über die Nachrichtenliste, die aktuelle Position wird in der Variablen MAIL_POINTER gespeichert. Wenn eine Nachricht bereits als gelesen markiert wurde, wird sie übersprungen. Bei der ersten ungelesenen Nachricht wird diese aktiviert (sprich – in die Variable MAIL_MESSAGE kopiert) und der Wert true zurück geliefert. Wenn es keine weiteren Nachrichten gibt, wird ein false zurückgeliefert. Der ELOas beendet dann die Abarbeitung dieses Rulesets und wechselt zum Nächsten.

```
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;
},
```

Neben der Weiterschaltung wird noch eine Initialisierung vorgenommen: die Variable MAIL_ALLOW_DELETE wird auf false gesetzt. Nur dann, wenn im Ruleset eine Bearbeitung vorgenommen wurde, sollte dieser die Variable auf true setzen. In diesem Fall wird die Mail dann in der finalizeImap Methode als bearbeitet markiert.

4.3.2 finalizeImap_GMAIL

Die Funktion finalizeImap_GMAIL muss eine Mail als Bearbeitet markieren, das passiert durch Setzen des Flags „SEEN“. Es darf aber nur dann gesetzt werden, wenn die Connect Methode es prinzipiell erlaubt (MAIL_DELETE_ARCHIVED) und das Ruleset die aktuelle Mail als archiviert markiert hat (MAIL_ALLOW_DELETE).

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

5 Beispiel - Migration einer Dokumenten-Datenbank

Für unser internes „Verbesserungs-Vorschlags-Wesen“ musste eine Datenbank mit rund 1400 Einträgen ins ELO migriert werden. In dieser Datenbank waren die Verschlagwortungsinformationen und Dokumente. Auf der ELO Seite sollte aus jeder Verschlagwortung ein Ordner erzeugt werden welches das Dokument dann als Untereintrag enthalten soll. Für die Migration wurde als führendes Tool der ELOas gewählt.

Da der ELOas im Augenblick noch keine Dokumente erzeugen kann, musste in einem Vorverarbeitungsschritt erst mal zu jedem Ordner ein Dummy Eintrag angelegt werden. Zum Glück haben die Einträge in der Datenbank eine mit 1 beginnende fortlaufende Nummer bis 1440. Man kann die Dummy Ordner deshalb relativ leicht durch ein VBS Skript erzeugen. Alle Ordner werden in einem Schrank mit der Objekt-Id 274312 angelegt.

```
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
```

Als nächstes kommt dann der ELOas zum Einsatz. Die Daten werden aus einer SQL Datenbank geholt:

```
"select Bearbeiter, Email, DerBetreff, LTrim(BunFeld1) BunFeld1,
KlassenName, Auftrag from [InetHelpDesk].[dbo].tblAuftraege a,
[InetHelpDesk].[dbo].tblBuendel b, [InetHelpDesk].[dbo].tblKlasse c,
[InetHelpDesk].[dbo].tblUser d where a.BunID = b.BunId and a.KlaID =
c.KlaID and a.UsrID = d.UsrID and AufID = " + ETS_COUNT
```

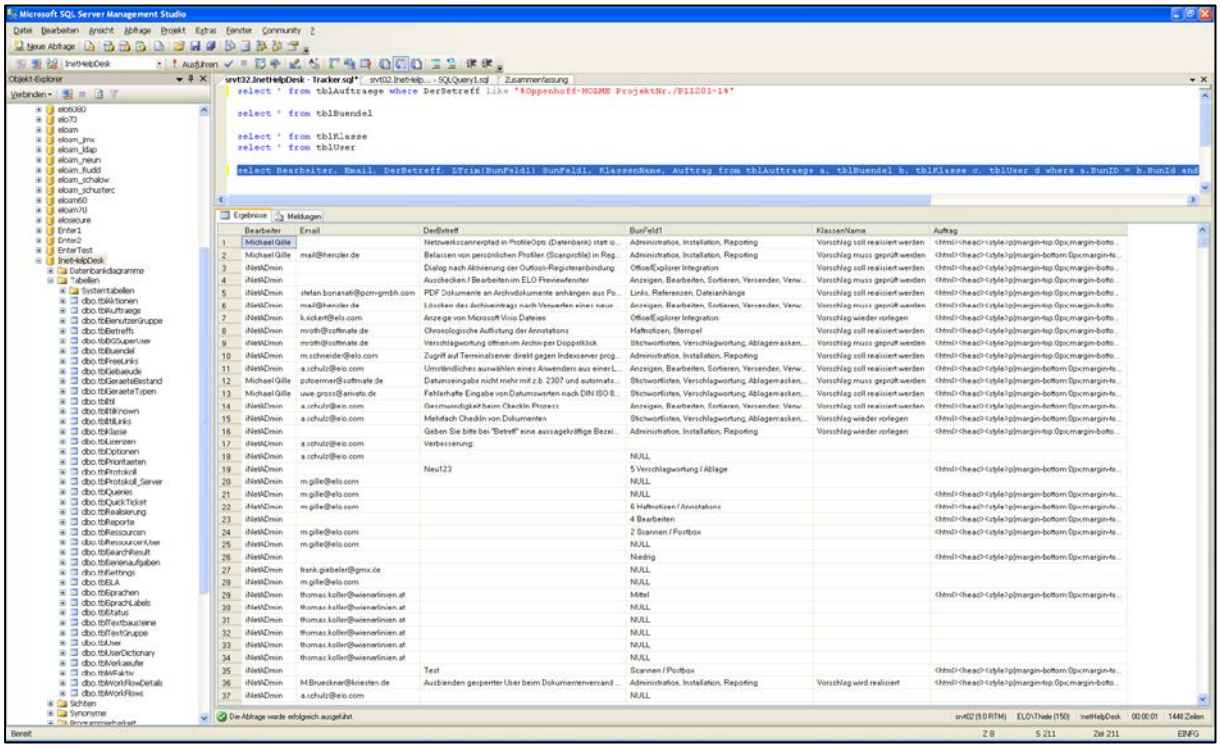


Abb. 10: SQL-Datenbank

Es handelt sich um ein etwas umfangreicheres SELECT-Statement, welches aber ansonsten keine Besonderheiten bietet. Lediglich ein Punkt ist erwähnenswert: in der Select-Liste gibt es eine Spalte *LTrim(BunFeld1) BunFeld1*. In dem Datenbankfeld *BunFeld1* liegen die Daten zum Teil mit führenden Leerzeichen, die nicht gewünscht sind. Diese werden mit LTrim entfernt. Dann hätte die Spalte aber keinen Namen mehr, deshalb wird anschließend noch der Spaltenname wieder mit *BunFeld1* angegeben. Diese Technik sollte man immer anwenden, wenn man in der Select Liste mit berechneten Werten arbeiten will.

Das komplette Ruleset sieht so aus:

```
<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>
```

```
    // Hier werden die Daten zum aktuellen Ordner
    // aus der Datenbank geladen

    var item = db.getLine(1, "select Bearbeiter, Email, DerBetreff,
LTrim(BunFeld1) BunFeld1, KlassenName, Auftrag from
[InetHelpDesk].[dbo].tblAuftraege a, [InetHelpDesk].[dbo].tblBuendel b,
[InetHelpDesk].[dbo].tblKlasse c, [InetHelpDesk].[dbo].tblUser d where
a.BunID = b.BunId and a.KlaID = c.KlaID and a.UsrID = d.UsrID and AufID = "
+ ETS_COUNT);

    // ETS_COUNT enthält die Record-Nummer, sie wird nach erfolgreicher
Bearbeitung geleert.
    ETS_COUNT = "";

    // Das Kurzbezeichnungs-Feld wird aus der Datenbank gefüllt, maximale
Feldlänge beachten!
    NAME = item.DerBetreff;
    if (NAME == "") { NAME = "unknown"; }
    if (NAME.length() > 127) { NAME = NAME.substring(0, 126); }

    // Der Initiator wird aus der Datenbank gefüllt.
    ETS_MAIL = item.Email;

    // Das Thema-Feld war in der Datenbank mit anderen Stichwörtern belegt
als im ELO Archiv
    // Deshalb gibt es hier eine Übersetzungstabelle. Im ELO wird mit
Spaltenindex gearbeitet.
    var thema = item.BunFeld1;
    if (thema == "Administration, Installation, Reporting") { thema =
"Administration¶Installation¶Reporting"; }
    if (thema == "Anzeigen, Sortieren, Bearbeiten, Versenden, Verwalten,
Suchen") { thema =
"Dokumentenbearbeitung¶Viewer¶Strukturbearbeitung¶Suche"; }
    if (thema == "Anzeigen, Bearbeiten, Sortieren, Versenden, Verwalten,
Suchen") { thema =
"Dokumentenbearbeitung¶Viewer¶Strukturbearbeitung¶Suche"; }
    if (thema == "Benutzeroberfläche, Design, Menüs, Navigation") { thema =
"Usability¶Oberfläche"; }
    if (thema == "Haftnotizen, Stempel") { thema = "Annotationen"; }
    if (thema == "Office / Explorer Integration") { thema = "Office
Integration¶OS Integration"; }
    if (thema == "Offline Verfügbarkeit") { thema = "Offline"; }
    if (thema == "Links, Referenzen, Dateianhänge") { thema =
"Links¶Referenzen"; }
    if (thema == "Scannen, Postbox, Konvertieren, Drucken") { thema =
"Scannen¶Postbox¶Konvertieren¶Drucken"; }
    if (thema == "Sicherheit, Anmeldung, Verschlüsselung, Benutzerrechte")
{ thema = "Benutzerrechte"; }
    if (thema == "Stichwortlisten, Verschlagwortung, Ablagemasken,
Versionierung") { thema = "Verschlagwortung¶Dokumentenablage"; }
    if (thema == "Workflow, Aufgaben") { thema = "Workflow¶Aufgaben"; }
```

```
    if (thema == "Schnittstellen, Scripte") { thema =
"Scripting¶Schnittstellen"; }
    ETS_THEMA = thema;

    ETS_USER = "Produktmanagement";
    ETS_STATUS_INT = item.KlassenName;

    EM_WRITE_CHANGED = true;

    // Die Datenbankinformation ist nun eingetragen. Es fehlt noch das
Dokument.
    // Dieses wird als HTML Datei mit einer XML Steuerungsdatei für den
    // ELO XML Importer erstellt. Zuerst wird die HTML Datei geschrieben:
    var id = Sord.getId();
    var dataFile = new File("d:\\temp\\trk\\" + id + ".htm");
    Utils.stringToFile(item.Auftrag, dataFile, "ISO-8859-15");

    // als nächstes wird der XML Datenstrom erzeugt. Da die eigentliche
Verschlagwortung
    // am Ordner hängt, ist hier nur eine Minimalverschlagwortung
vorhanden.
    var xmlDesc = NAME.replace("\"", "'").replace("&amp;",
"&amp;amp;").replace("&lt;", "&amp;lt;").replace("&gt;", "&amp;gt;");
    var xmlFile = new File("d:\\temp\\trk\\" + id + ".xml");
    var xmlText = "&lt;?xml version=\"1.0\" ?&gt;&lt;eloobjlist
ver=\"1.0\"&gt;&lt;obj&gt;&lt;desc value=\"";
    xmlText = xmlText + xmlDesc;
    xmlText = xmlText + "\"/&gt;&lt;type
value=\"0\"/&gt;&lt;destlist&gt;&lt;destination type=\"1\" value=\"#";
    xmlText = xmlText + id;
    xmlText = xmlText + "\"/&gt;&lt;/destlist&gt;&lt;docfile name=\"";
    xmlText = xmlText + id;
    xmlText = xmlText + ".htm\"/&gt;&lt;/obj&gt;&lt;/eloobjlist&gt;";

    // Zuletzt wird die XML Datei geschrieben.
    Utils.stringToFile(xmlText, xmlFile, "UTF-8");

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

Nachdem der ELOas die Verschlagwortung aus der Datenbank ergänzt hat und die HTML und XML Dokumentendateien erstellt hat, kommt der ELO XML Importer ins Spiel. Er importiert nun die HTML Dateien in den jeweiligen Ordner. Damit ist der Migrationsvorgang abgeschlossen. Aufwand für das komplette Projekt: ca. 4 Stunden.

6 Beispiel - Treewalk für ELOas

Für die Abarbeitung der Dokumente in den ELO Automation Services steht eine Treewalk-Funktion zur Verfügung. Es können also nicht nur Suchbereiche abgearbeitet, sondern auch komplette Baumstrukturen durchlaufen werden.

6.1 Einleitung

Normalerweise führt der ELOas eine Suche nach einem Indexfeld aus, um die Liste der zu bearbeitenden Dokumente zu ermitteln. Alternativ kann nun aber auch ein Treewalk ausgeführt werden. Über diesen Treewalk können einzelne Archivzweige oder aber auch das komplette Archiv durchlaufen werden. Dabei wird jeder Eintrag zweimal durchlaufen: einmal beim Betreten, danach werden erst mal alle Untereinträge durchlaufen und dann noch mal beim Verlassen.

Beispiel: es gibt einen Schrank mit den Ordnern 1 und 2. Ordner 1 enthält das Register 1.1. Dann ergibt sich folgender Ablauf:

```
Schrank (betreten)
Ordner 1 (betreten)
Register 1.1 (betreten)
Register 1.1 (verlassen)
Ordner 1 (verlassen)
Ordner 2 (betreten)
Ordner 2 (verlassen)
Schrank (verlassen)
```

Ob das Ruleset im aufsteigenden Zweig (betreten) oder im absteigenden Zweig (verlassen) aufgerufen wird, kann ein Skript anhand der Variablen EM_TREE_STATE abprüfen. Diese enthält 0 beim Betreten und 1 beim Verlassen. Gespeichert wird nur beim Verlassen. Änderungen, die beim Betreten ausgeführt werden, bleiben aber bis zum Verlassen erhalten, auch dann, wenn in der Zwischenzeit eine Anzahl anderer Objekte bearbeitet wurde.

Initiiert wird ein Treewalk indem als Suchindex Gruppenname der Wert „TREEWALK“ eingegeben wird und als Suchbegriff die Nummer des Startknotens. Auf dem Startknoten werden keine Regeln aufgerufen. Dies wird nur für die Untereinträge durchgeführt.

6.2 Anwendungsbeispiel

Das folgende Anwendungsbeispiel durchläuft einen Archivzweig und setzt in allen Objekten des Maskentyps 6 (Track Item) eine interne Id (Track-ID). Der Startordner hat die Id 3352.

In diesem einfachen Beispiel ist keine Fehlerbehandlung vorgesehen, die Fehlerregel ist deshalb leer.

```
<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;&amp;amp; (EM_ACT_SORD.getMask()  == 6)) {
      // nur TrackItems bearbeiten
      //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>
```

Der interessante Teil des Rulesets liegt in dem Skript-Bereich, dieser wird deshalb im Folgenden noch mal einzeln aufgeführt:

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

Das Skript soll nur beim Verlassen ausgeführt werden (EM_TREE_STATE == 1) und nur auf Objekte vom Typ TrackItem (EM_ACT_SORD.getMask() == 6).

```
      // nur TrackItems bearbeiten
      //cnt.createCounter("ETSTrackId", 10000);
```

Das Beispiel verwendet einen Counter, der muss vorher angelegt werden, z. B. durch das oben aufgeführte Kommando. Er darf aber nur einmal erzeugt werden, sonst wird die Track-ID immer wieder zurückgesetzt.

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

Nur wenn noch keine Track-ID vorhanden ist (Indexzeile ETS_TICK ist leer), dann soll eine erzeugt werden.

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

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

Zur Erzeugung von Track Ids gibt es eine praktische Methode im Counter Modul cnt: getTrackId(<CounterName>, <Prefix>). Diese Methode holt einen neuen Counter Wert und ergänzt ihn mit dem Präfix und einer Checksumme. Im Beispiel wird also aus dem Counter Wert 10001 die TrackId V10001C2.

```
      EM_WRITE_CHANGED = true;
```

Nur wenn eine neue TrackId erzeugt wurde, soll das Objekt gespeichert werden.

```
    }
  }
```

Das Ruleset wird alle 10 Minuten ausgeführt und durchläuft den kompletten Track Item Ordner. Alle Einträge ohne TrackId werden automatisch ergänzt, egal mit welchem Client sie erzeugt wurden.

6.3 Variablen der Laufzeitumgebung

Wenn das Ruleset ausgeführt wird, gibt es neben dem EM_TREE_STATUS Wert eine Reihe weiterer Variablen, die zur Bearbeitung herangezogen werden können.

Name	Inhalt
EM_TREE_STATUS	Gibt an, ob das Ruleset im aufsteigenden Ast (0) oder absteigenden Ast (1) ausgeführt wird.
EM_ACT_SORD	Enthält das Sord Objekt mit den aktuellen Objektdaten.

EM_PARENT_SORD	Enthält das Sord Objekt mit den Daten des Parent Knotens. Diese Daten dürfen prinzipiell auch verändert werden. Es muss aber darauf geachtet werden, dass diese Änderungen dann auch gespeichert werden. Dazu muss im absteigenden Ast die Veränderung erkannt werden und das EM_WRITE_CHANGED Flag auf true gesetzt werden.
EM_ROOT_SORD	Enthält das Sord Objekt mit dem Startknoten. Da auf diesen Eintrag das Ruleset nicht angewendet wird, muss bei Veränderungen ein manuelles Speichern erfolgen. Das kann durch Setzen der Variablen EM_SAVE_TREE_ROOT passieren.
EM_INDEX_LOADED	Im Gegensatz zu einer Abarbeitung nach einer Suche kann man beim Treewalk nicht sicher davon ausgehen, dass ein geladenes Sord Objekt eine bestimmte Maske besitzt. Prinzipiell kann jede beliebige Maske auftreten. Die vorbelegten Indexvariablen können aus den Indexzeilen aber nur für Masken generiert und gefüllt werden, die in der Definition unter <mask> und unter <masks> angemeldet werden. In diesem Fall wird die Variable EM_INDEX_LOADED auf true gesetzt. Falls eine unbekannte Maske vorliegt, ist ein Zugriff auf die Indexzeilen nur direkt über das EM_ACT_SORD Objekt möglich, EM_INDEX_LOADED steht dann auf false. Hinweis: Wenn die Indexvariablen gefüllt wurden, sollte man die Indexzeilen in EM_ACT_SORD nicht direkt bearbeiten. Diese Änderungen gehen vor dem Speichern wieder verloren, wenn die Indexvariablen zurückgeschrieben werden.
EM_TREE_LEVEL	Über diese Variable kann man feststellen, auf welcher Ebene innerhalb des Treewalks man sich befindet. Die Untereinträge der Startknotens befinden sich im Level 0 (für den Startknoten werden ja keine Regeln aufgerufen).
EM_TREE_MAX_LEVEL	Über diese Regel kann man die maximale Tiefe festlegen. Noch tiefer geschachtelte Untereinträge werden ignoriert. In Normalfall liegt dieser Wert bei 32. Falls er geändert werden soll, kann er vor der Abarbeitung in der onstart Routine auf den gewünschten Wert gesetzt werden.

EM_SAVE_TREE_ROOT	Für den Startknoten des Treewalk werden keine Regeln aufgerufen. Falls dieser durch den Zugriff über EM_TREE_ROOT oder EM_PARENT_SORD geändert wurde, muss durch das Setzen der Variablen EM_SAVE_TREE_ROOT eine Speicherung dieser Änderungen angemeldet werden.
	
	`<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	Wenn beim Durchlauf festgestellt wird, dass ein Unterbereich von der Verarbeitung ausgeschlossen werden soll, kann die Variable EM_TREE_CHILDREN auf false gesetzt werden. Dieser Wert wird nur beim aufsteigenden Zweig ausgewertet (beim absteigenden Ast wäre es ohnehin zu spät, dann ist der Unterbereich ja bereits durchlaufen worden) und er wird bei jedem Objekt mit true initialisiert (Standardverhalten: durchlaufe den vollständigen Unterbereich).
EM_TREE_ABORT_WALK	Wenn ein Durchlauf komplett abgebrochen werden soll, dann kann man zu einem beliebigen Zeitpunkt das Flag EM_TREE_ABORT_WALK gesetzt werden. In diesem Fall werden keine weiteren Untereinträge durchlaufen. Auch weitere Einträge der gleichen Ebene, die noch nicht abgearbeitet wurden, bleiben unbearbeitet. Dieses Flag kann man setzen, um die Arbeit nach einem schweren Fehler abzubrechen. Hinweis: In der onstart Routine können notwendige Laufzeitkontrollen durchgeführt werden, um zu prüfen, ob der Treewalk durchgeführt werden darf. Falls nicht, kann durch dieses Flag der Lauf abgebrochen werden.

7 Beispiel - Workflowbearbeitung

Es gibt eine Erweiterung zur Abarbeitung von Workflowterminen. Man kann im Workflow einzelne Personenknoten für das ELOas Konto anlegen. Wenn ein Workflow diesen Knoten aktiviert, kann über eine ELOas Suche „WORKFLOW“ eine Liste der aktiven Workflowtermine ermitteln und abarbeiten. Dabei kann die Verschlagwortung ergänzt und der Workflow weitergeleitet werden.

Die notwendigen Rulesets müssen auf XML Ebene erstellt werden.

Das Sammeln der Workflowterminliste sieht im Wesentlichen wie eine normale Suche aus. Als Indexzeilennamen wird „WORKFLOW“ eingegeben, der Suchbegriff selbst wird ignoriert und sollte leer bleiben.

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

Auch wenn für das Sammeln der Liste keine Suchmaske benötigt wird, muss trotzdem eine Suchmaske angegeben werden. Es werden aus der Terminliste nur die Workflows abgearbeitet, die diese Maske besitzen. Das ist notwendig, damit die Indexdaten in die lokalen JavaScript-Variablen geladen werden können. Falls Workflows zu mehreren Masken möglich sind, muss das Ruleset mehrfach angelegt werden.



Information: In der Terminliste wird kein „FindFirst – FindNext“ durchgeführt. Wenn es viele Termine gibt, die nicht bearbeitet werden, kann das dazu führen, dass keine neuen Termine mehr zur tatsächlichen Bearbeitung gefunden werden.

Bei der Abarbeitung von Workflows gibt es neben der Verschlagwortungsänderung zwei Aktivitäten: gezieltes Weiterleiten und Verändern des Workflows. Das folgende Beispiel soll aufzeigen, wie man in Abhängigkeit der aktuellen Verschlagwortung Einfluss auf den Workflow nehmen kann. Dazu soll ein einfacher Freigabeworkflow betrachtet werden, bei dem der Sachbearbeiter zu Beginn noch nicht feststeht. Er wird im Laufe des Workflows von der Poststelle in einem Indexfeld *SACHBEARBEITER* eingetragen. In der Vorlage wird der Sachbearbeiter Knoten erst mal mit *Eigentümer* initialisiert, der richtige Wert wird vom ELOas zur Laufzeit aus dem Indexfeld *SACHBEARBEITER* ausgelesen und in den Knoten eingetragen. Der ELOas läuft hierzu unter dem ELO Namen *elowf* und hat einen Personenknoten zwischen der Poststelle und dem Sachbearbeiter.

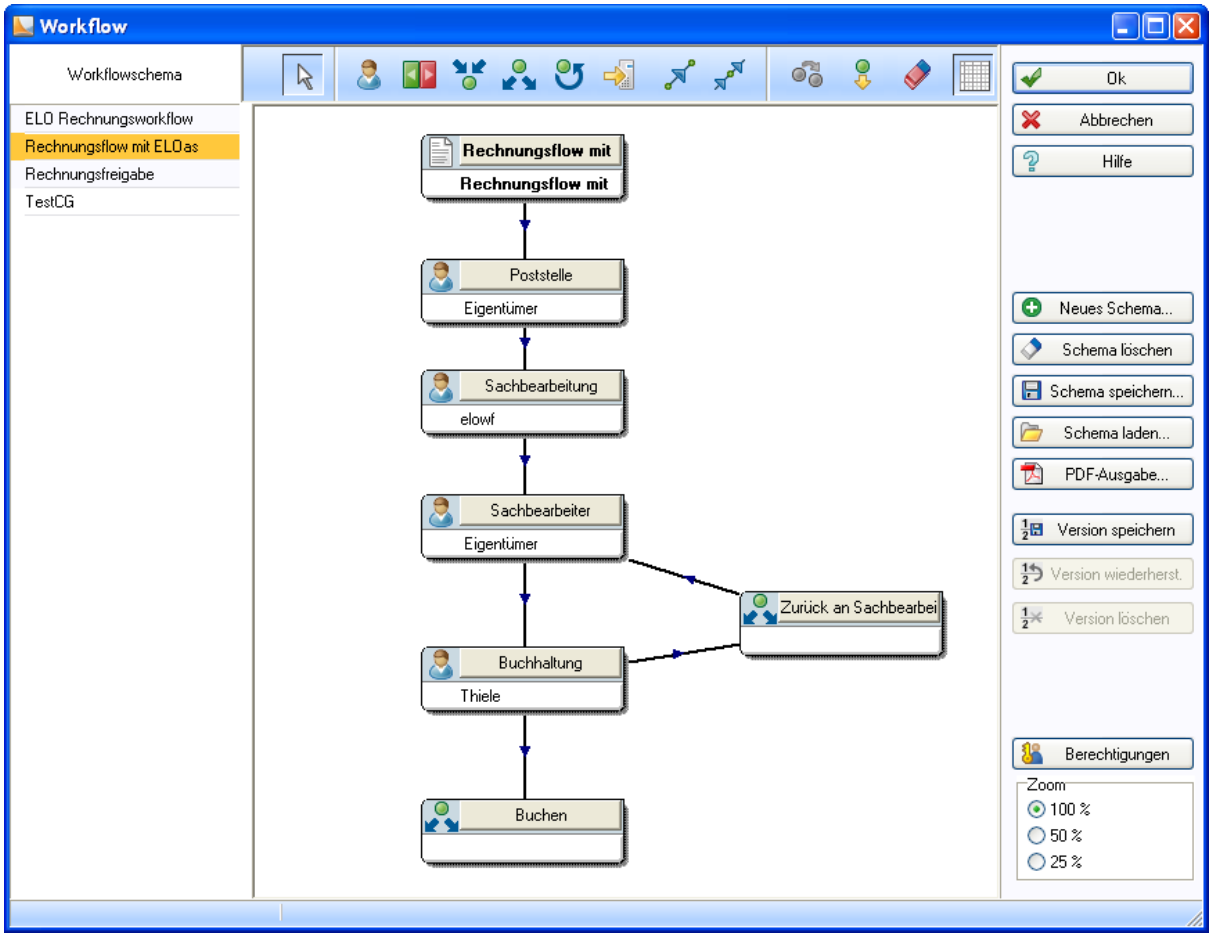


Abb. 11: Workflow-Vorlage mit ELOas

Wenn der Workflow beim ELOas ankommt, hat die Poststelle den Sachbearbeiter festgelegt. Der ELOas liest diese Indexzeile SACHBEARBEITER aus und trägt den Wert in den Folgeknoten *Sachbearbeiter* ein. Das passiert durch folgende einfache Rule:

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

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

Der Wechsel des ELO Benutzernamens wird durch den Befehl *wf.changeNodeUser* durchgeführt. Als erster Parameter wird der Workflow Knotenname angegeben und als zweiter Parameter der ELO Benutzername, der eingetragen werden soll. Um den Rest (Workflow sperren, lesen, Knoten suchen, Benutzer aktualisieren, Workflow speichern, Sperre freigeben) kümmert sich die Library *wf*.

Nachdem der Benutzername gesetzt wurde, muss der Workflow weitergeleitet werden. Das passiert durch Setzen der Variablen *EM_WF_NEXT*. Wenn diese leer bleibt, wird nichts weitergeleitet. Der Termin bleibt bestehen (was natürlich nicht für alle Zeiten so bleiben sollte, da dann irgendwann die Terminliste überläuft). Wenn alle Voraussetzungen für eine Weiterleitung erfolgt sind, dann kann entweder die Verbindungsnummer oder der Name des Nachfolgerknotens angegeben werden. Wenn es nur einen Nachfolger gibt, dann ist die Verbindungsnummer einfach einzutragen: „EM_WF_NEXT = "0"; ".

Falls es mehrere Nachfolger gibt, sollte man besser den Namen des Nachfolgeknotens angeben. Dafür wollen wir annehmen, dass der Vorgang nach der Sachbearbeitung automatisch gebucht wird, d.h. den Knoten Buchhaltung übergeben wir auch an den ELOas. Dieser führt ein Skript aus, welches die Buchungsdaten prüft. Wenn alles in Ordnung ist, dann liefert die Funktion ERPverify() true zurück und der Workflow soll an den Knoten *Buchen* weitergeleitet werden. Wenn ein Fehler vorliegt, dann soll der Workflow an den Sachbearbeiter zurückgehen. Das Skript dafür könnte so aussehen:

```
If (ERPverify()) {
  EM_WF_NEXT = "Buchen";
} else {
  EM_WF_NEXT = "Sachbearbeiter";
}
```

8 ELOas Debugger

Die Fehlersuche in einem umfangreichen Ruleset kann sehr aufwendig werden. Bei jedem Durchlauf muss die JavaScript angepasst werden. Dafür müssen Sie das Dokument auschecken, bearbeiten, einchecken und schließlich auf *Reload* klicken. Weiterhin ist der Betrieb des Rhino Debuggers unter Apache Tomcat in einer Windows-7-Umgebung problematisch. Zumindest diesen Punkt und den Checkout-/Check-in-Vorgang kann man sich durch den Einsatz des ELOas Debuggers ersparen.



Information: Für mehr Informationen zum ELOas Debugger lesen Sie die separate Dokumentation „ELO Automation Services Debugger (Java FX)“.

8.1 Programm öffnen

Das ZIP-Archiv enthält eine EXE-Datei und ein Unterverzeichnis mit verschiedenen Libraries. Dieses wird einfach in den Zielordner entpackt. Bei Bedarf kann man sich eine Desktop-Verknüpfung auf die EXE-Datei anlegen.

Der ELOas Debugger wird über die Datei EloAsDebug.exe gestartet und setzt voraus, dass mindestens JRE 1.7 auf dem Rechner installiert ist.

Der ELOas Debugger kann optional auch über die Kommandozeile mit dem folgenden Befehl gestartet werden:

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

8.2 Konfiguration

Klicken Sie auf die Schaltfläche *Config*. Hierüber erreichen Sie den Konfigurationsdialog.

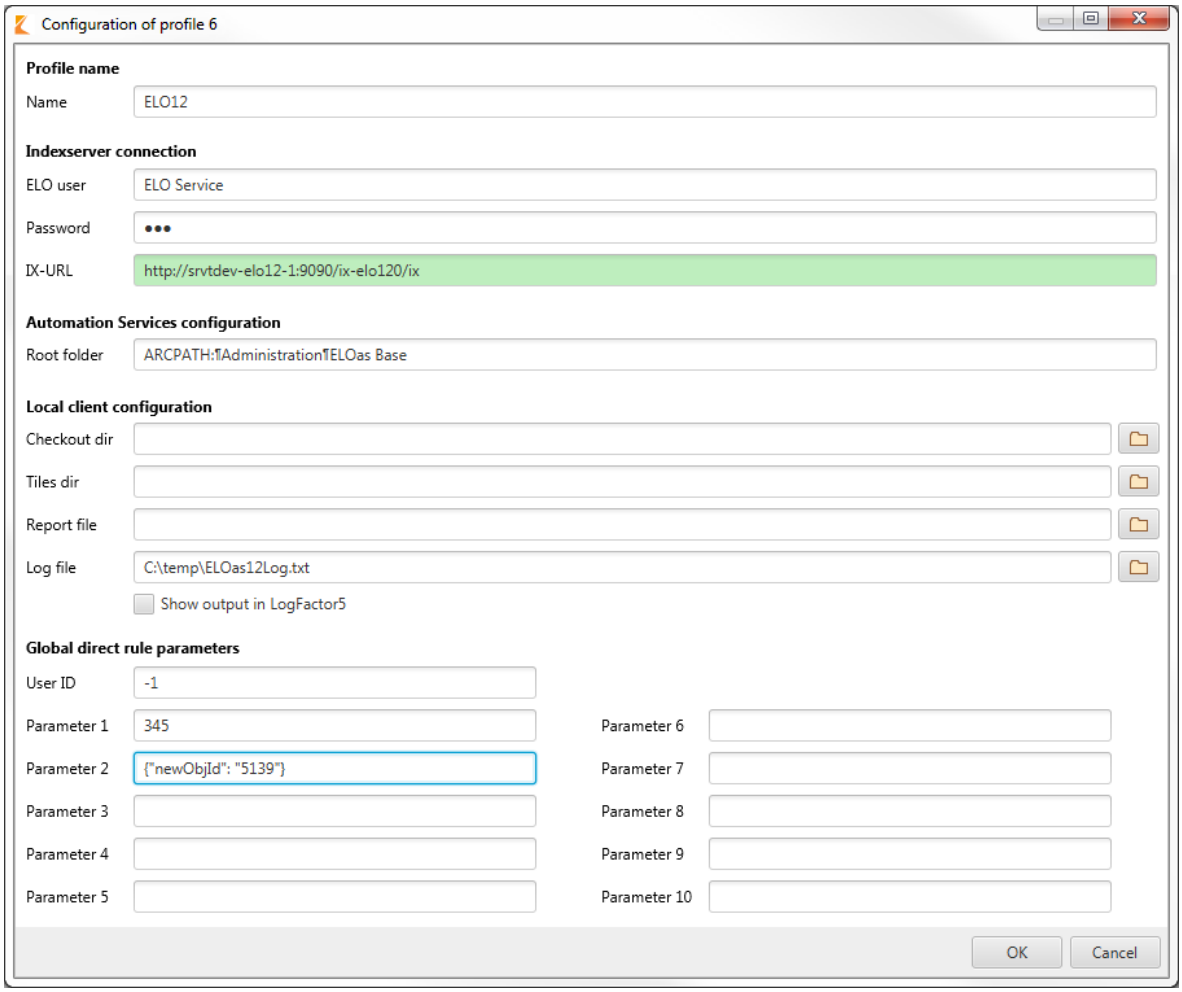


Abb. 12: Konfiguration des Debuggers

Im Dialogtitel wird die ID des zu bearbeitenden ELOas-Debugger-Profils angezeigt. Im Textfeld für das Benutzerpasswort wird das Benutzerpasswort nun maskiert.

Name: Der Profilname darf maximal 15 Zeichen lang sein.

ELO user: Der Name des ELO Benutzers.

Password: Das Passwort der Indexserver-Verbindung.

IX-URL: Die URL des Indexservers. Das Textfeld erhält eine grüne Hintergrundfarbe, wenn eine Verbindung möglich ist.

Root folder: Der Speicherpfad der ELOas Konfiguration.

Checkout dir: Die Schaltfläche hinter dem Feld *Checkout dir* ermöglicht die Auswahl des Checkout-Verzeichnisses des ELO Java Clients.

Tiles dir: Die Schaltfläche hinter dem Feld *Tiles dir* ermöglicht die Auswahl des überwachten Verzeichnisses für die referenzierten ELO Dropzone Kacheln.

Report file: Über die Schaltfläche hinter dem Feld *Report file* lässt sich eine ELOas Debugger-Reportdatei auswählen.

Log file: Die Schaltfläche hinter dem Feld *Log file* ermöglicht die Auswahl der Log-Datei.

Global direct rule parameters: Legen Sie hier die globalen Parameter für die direkten ELOas Regeln fest. Sie können die Benutzer-ID und zehn Parameter bearbeiten.

Der Profilkonfigurationsdialog hat einen Scroll-Balken, der beim Verkleinern des Dialogs ab einer bestimmten Dialoggröße angezeigt wird.

Nach dem Betätigen der OK-Schaltfläche werden die durchgeführten Profil-Einstellungen in der System-Registry gespeichert. Die Einstellungen für das aktuelle ELOas-Debugger-Profil (ID: 1) werden an der folgenden Position in der System-Registry gespeichert:

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

Über die Schaltfläche *Abbrechen* werden die aktuellen Änderungen verworfen und der Dialog wird geschlossen. Der Dialog zum Bearbeiten eines ELOas-Debugger-Profils kann auch über die ESCAPE-Taste geschlossen werden. Dieser Dialog hat auch eine passende minimale Dialoggröße. Beim Vergrößern des Dialogs werden die einzelnen Dialogkomponenten ebenfalls passend vergrößert. Somit kann der Benutzer lange Profileingaben sehen.

8.3 Ruleset bearbeiten

Nach dem Start werden automatisch alle Rulesets der aktuellen Konfiguration geladen. Sie werden aber noch nicht ausgeführt, damit Sie die Möglichkeit erhalten, im JavaScript-Code Haltepunkte zu setzen.

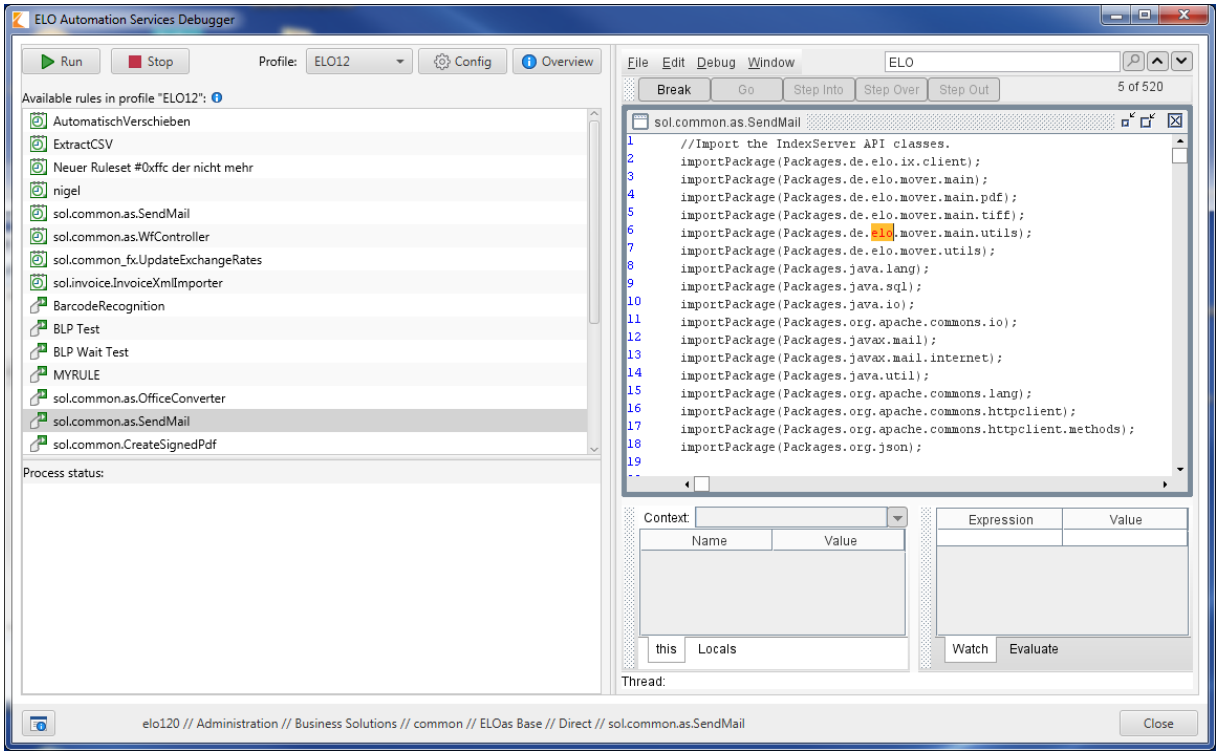


Abb. 13: Ruleset bearbeiten

Falls Sie mehrere Rulesets haben, achten Sie darauf, dass Sie im Rhino Debugger Fenster unter *Window* den richtigen ausgewählt haben. Nun können Sie an beliebigen Stellen Haltepunkte setzen und das Ruleset starten.

Der Start eines Rulesets wird durch anklicken des jeweiligen Ruleset-Eintrags in der Liste und anschließendem Klick auf *Run* ausgeführt. Das Ruleset wird zwar jetzt aktiviert wird, unterliegt jedoch weiterhin der Intervallsteuerung. Wenn Sie einen Ruleset-Start für Mitternacht eingestellt haben, dann wird er auch im Debugger erst zu diesem Zeitpunkt aktiv. Für Debugging Zwecke ist die Einstellung „1M" – also einmal pro Minute – für wiederkehrende Ausführungen sinnvoll und „10H" – also alle 10 Stunden – für einmalig auszuführende Rulesets.

Wenn Sie ein Ruleset oder eine JavaScript-Datei bearbeiten wollen, dann können Sie diese auschecken oder die bereits ausgecheckte Datei direkt mit einem geeigneten Editor aufrufen. Führen Sie die Änderungen durch und speichern Sie die Daten. Solange der Editor die Datei nicht exklusiv öffnet (ist bei Texteditoren eher unüblich), müssen Sie den Editor noch nicht einmal schließen. Klicken Sie im Debugger einfach erneut auf *Run*. Das Ruleset wird nun automatisch aus dem Archiv und dem Checkout-Verzeichnis neu geladen und neu gestartet. Auch die Log-Datei wird neu erstellt, so dass Sie sich nicht mit alten Log-Ausgaben befassen müssen.

9 Manuelle Installation der ELOas

Dieses Dokument beschreibt die manuelle Installation der ELO Automation Services (ELOas). Unter ELOprofessional wird das Modul durch die Serverinstallation automatisch mit erzeugt. Bei einer Nachinstallation oder in einer verteilten Umgebung muss man es aber manuell installieren.

> **Information:** In den Abbildungen werden die ELO Automation Services gegebenenfalls noch mit der vorherigen Bezeichnung ELO Mover benannt. Gemeint sind in diesen Fällen die ELO Automation Services.

Der ELOas ist, wie fast alle Module aus der ELOenterprise Server Linie, als Servlet programmiert und benötigt zum Betrieb eine Java-Laufzeitumgebung und einen Application-Server, z. B. den Tomcat 6.0. Es wird mindestens die Java Version 5 (bzw. 1.5 nach alter Schreibweise) benötigt.

Die Konfiguration wird in der XML-Datei *config.xml* im Standard-Konfigurationsverzeichnis von ELO hinterlegt. Somit können Updates problemlos durchgeführt werden und die Konfiguration bleibt dabei erhalten. Die Standardsprache für ELOas wird beim ELO Server Setup je nach ausgewählter Installationssprache automatisch abgeleitet und in der Datei *config.xml* im Parameter „language“ eingetragen – im folgenden Beispiel „en“ für Englisch:

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

Die Ausführungsanweisungen des ELOas mit den Regelsätzen, Übersetzungslisten und Basisskripten liegen in einem Ordner im Archiv. In der Konfiguration muss also nur der Zugang zum ELO Indexserver und dieser Basisordner eingestellt werden.

9.1 Benötigte Dateien

Im ZIP-Archiv für die manuelle Installation finden Sie folgende Dateien:

ELOas.war	Java-Archiv mit dem Servlet-Code und benötigten Libraries
ELOas.xml	Konfigurationsdatei für den Application-Server mit Verweis auf das ELO Konfigurationsverzeichnis
log4j.properties	Default-Konfiguration für die Logger Ausgabe
config.xml	ELO Konfiguration für die ELO Indexserververbindung
ELO Automation Services Konfiguration.zip	Importdatensatz mit JavaScript-Templates
GetGuid.vbs	Skript zur Ermittlung der GUID eines ELO Eintrags

Installation.pdf	Dieses Dokument, Leitfaden zur Installation
JavaScriptCode.pdf	Beschreibung des generierten JavaScript-Codes
Regeldefinition.pdf	Aufbau der Regeldefinition

9.2 Installationsvorbereitung

Das ZIP-Archiv *ELO Automation Services Konfiguration.zip* enthält eine ELOas Struktur mit den Basis-Skripten. Dabei enthält das Archiv die Vorlage mit XML-Daten im Zusatztext (vorzugsweise bei Microsoft SQL). Die Vorlage mit Textdateien (vorzugsweise für Oracle SQL, da es hier nur einen relativ kleinen Zusatztext gibt) erhalten Sie auf Anfrage von ELO. Die benötigte Version dieser Daten importieren Sie als erstes an eine geeignete Stelle in Ihr Archiv, z. B. unterhalb des Ordners *Administration*. Stellen Sie sicher, dass der ELO Benutzer, unter dem der Dienst dann später laufen wird, Zugriff auf diesen Bereich besitzt.

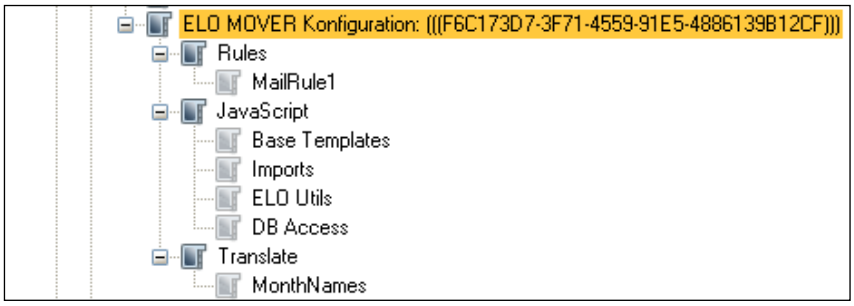


Abb. 14: Ordner für Rules im ELO Archiv

Der Unterordner *Rules* enthält die Benutzerdefinierten Regelsätze, hier liegt ein Beispiel, welches als Vorlage für eigene Lösungen verwendet werden kann.

Die Dateien *ELOas.war* und *ELOas.xml* sollten passend zum Archivnamen und der ELO Standardkonvention für Servicenamen umbenannt werden in *as-<Archivname>.war* bzw. *as-<Archivname>.xml*. Für das Archiv „elo70" also in „as-elo70.war" bzw. „as-elo70.xml". Dabei sollte die Groß/Kleinschreibweise unbedingt beachtet werden, da diese für den späteren Zugriff wichtig ist. Diese beiden Dateien werden dann in ein temporäres Verzeichnis auf den Rechner kopiert, auf dem der Application Server läuft (z. B. C:\TEMP).

In der Datei *ELOas.xml* muss der Pfad für das Konfigurationsverzeichnis Ihrer ELO Umgebung eingetragen werden:

```
<?xml version='1.0' encoding='UTF-8'?>
<Context path="/as-elo70">
  <Environment name="webappconfigdir" value="G:\ELOprofessional\config\as-
elo70" type="java.lang.String" override="false"/>
</Context>
```

Für die Dateien *log4j.properties* und *config.xml* wird im ELO Konfigurationsverzeichnis ein Unterverzeichnis für diese ELOas Konfiguration erstellt und diese beiden Dateien werden dort hin kopiert.

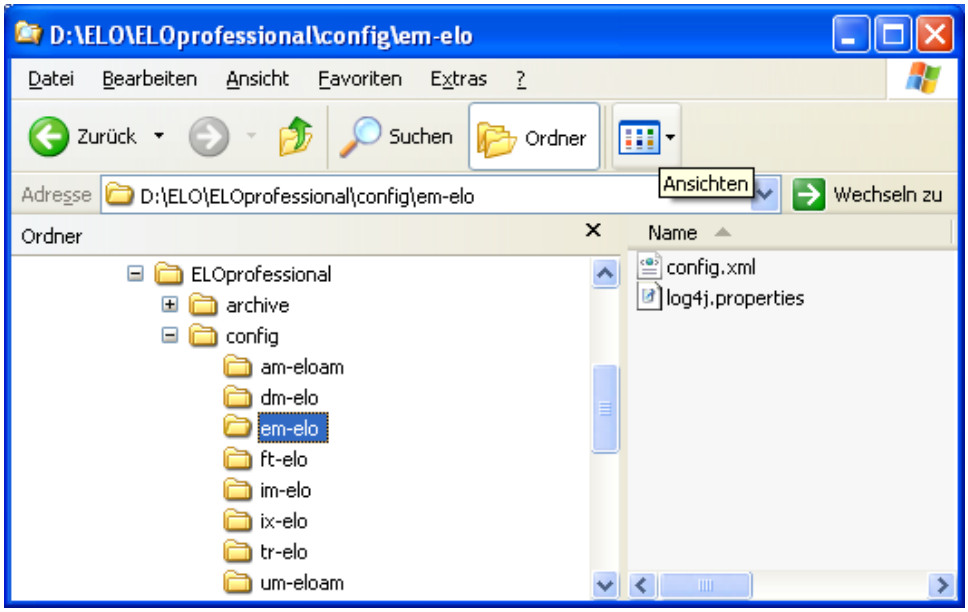


Abb. 15: Konfigurationsverzeichnis für ELOas

Der Name des Konfigurationsverzeichnisses sollte mit „as-„ beginnen und anschließend den Archivnamen enthalten. Für das Archiv „elo70“ sollte es also den Namen „as-elo70“ tragen. In der Datei *log4j.properties* muss der Pfad für das Ausgabeverzeichnis an die lokale Installation angepasst werden.

log4j.appender.FI.File=C:/Programme/Tomcat 6.0/logs/as-elo70.log

In der Datei *config.xml* müssen die Parameter für den Indexserverzugriff angepasst werden:

```
<?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-elo70/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>
```

Der Schlüssel „url“ enthält den Zugriffspfad zum ELO Indexserver. Achten Sie auch hier unbedingt auf die Groß/Kleinschreibweise, im Fehlerfall wird der Indexserver nicht gefunden werden.

Der Schlüssel „user“ enthält den ELO Anmeldenamen des ELOas am ELO Indexserver. Im Normalfall sollte man für die Zusatzdienste ein eigenes Konto anlegen, welches nicht von interaktiven Benutzern verwendet wird.

Der Schlüssel „password“ enthält das ELO Passwort. Sie können diesen Eintrag zum Testen im Klartext vornehmen. Der Report enthält dann nach dem Start des Dienstes einen Hinweis darauf, wie die zugehörende Verschlüsselung aussieht. Diesen Text können Sie dann per Cut&Paste aus dem Log-Report in die Konfiguration übernehmen.

Der Schlüssel „rootguid“ enthält die GUID des Basisordners des ELOas, voreingestellt ist die GUID des Beispielordners aus dem Importdatensatz. Falls Sie ein eigenes Register für diese Daten angelegt haben, können Sie diese GUID leicht durch folgendes Skript über den ELO Windows Client ermitteln (GetGuid.vsb Datei im ZIP-Archiv):

```
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
```

Dieses Skript ermittelt die GUID des aktuell ausgewählten Eintrags und kopiert sie in das Windows-Klemmbrett. Von dort aus können Sie es im Editor per STRG-V im Editor in die Konfiguration übernehmen.

Der Schlüssel „rootguid“ wird auch für die Konfiguration mehrerer ELOas Instanzen verwendet. Sie können bis zu zehn ELOas Instanzen betreiben. Pro Instanz legen Sie einen „ELOas Base“-Ordner im ELO Archiv unter *<Archivname> // Administration* an. Legen Sie für jeden einzelnen „ELOas Base“-Ordner in der Datei *config.xml* separate „rootguid“-Parameter wie im folgenden Beispiel an:

```
<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>
```

Die ELOas Regeln in den einzelnen „ELOas Base“-Ordnern werden dann unabhängig voneinander ausgeführt. Die Logs werden auch bei mehreren Instanzen in eine Log-Datei geschrieben. Der Pfad der Log-Datei wird in der Konfigurationsdatei *log4j.properties* definiert.

Der Schlüssel „tempdir“ enthält optional ein Verzeichnis für den temporären Download der Textdateien, wenn die XML- und JavaScript-Daten in Textdateien statt im Zusatztext liegen sollen. Falls tempdir leer ist oder nicht vorhanden ist, wird automatisch die Zusatztext-Version verwendet, andernfalls die Textdatei-Version.

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

Beachten Sie: Beim Anlegen eines neuen Benutzers für diesen Dienst reagiert der Indexserver auf diese Änderung zeitverzögert Zur Sicherheit können Sie auf der Statusseite des ELO Indexserver den Benutzer-Cache löschen, um eine sofortige Aktualisierung zu erzwingen.

9.3 Deployment der Dateien

Im Application Server tragen Sie nun die Parameter für das Deployment ein. Der Context Path (ist nicht optional, auch wenn es in der Tomcat Admin Konsole so steht) enthält den Namen der Web Applikation, die beiden Dateipfade zeigen auf Konfiguration und Programmdatei. Mit einem Klick auf *Deploy* wird die Applikation installiert.

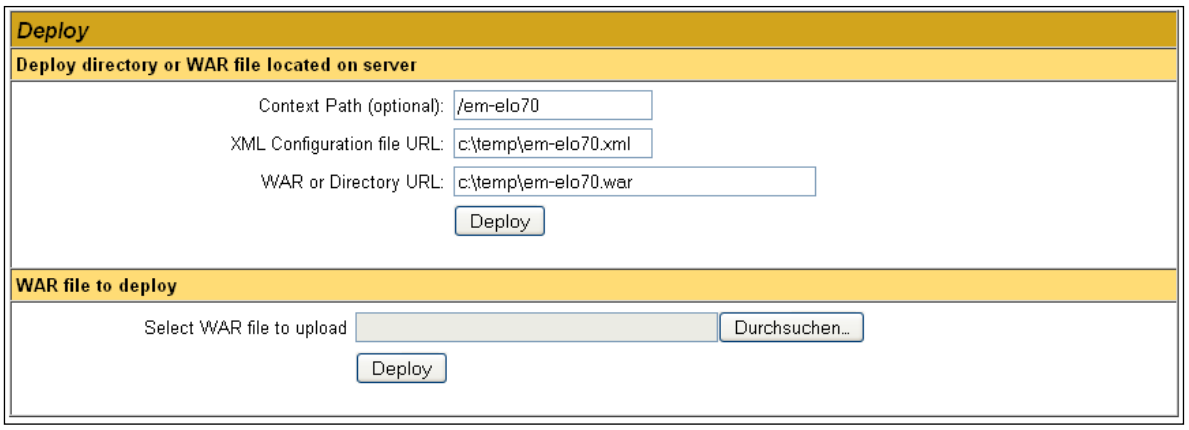


Abb. 16: Parameter für das Deployment eintragen

Die Datei „ELOas.war“ in der ELOas Version 12.00.000 Build 017 enthält eine Textdatei namens „version.txt“. Diese Datei enthält die Erweiterungen der einzelnen ELOas Versionen.

9.4 Statusseite anzeigen

Der ELOas bringt eine eigene Statusseite mit, diese kann über folgende URL erreicht werden:

http://<SERVERNAME>:9070/as-<ARCHIVNAME>/as?cmd=status



ELO MOVER status report

No active ruleset, pausing

Excecuted	Name	Next run	Status
2	Regelsatz4	2009-08-03 15:39:44.689	Idle...
2	Mailmaske Thiele	2009-08-03 15:39:44.689	Idle...
2	Mailmaske Support	2009-08-03 15:39:44.689	Idle...

Reload

Abb. 17: ELOas Statusseite mit aktiven Rules

Auf der Statusseite werden alle aktiven Rulesets aufgelistet, zusammen mit der Information, wie oft sie bereits ausgeführt wurden und wann die nächste geplante Ausführung stattfindet.

Falls ein JavaScript-Fehler aufgetreten ist, wird dieser auch auf der Statusseite angezeigt, zusammen mit der Zeilennummer des Fehlers und den Programmcode in diesem Bereich.

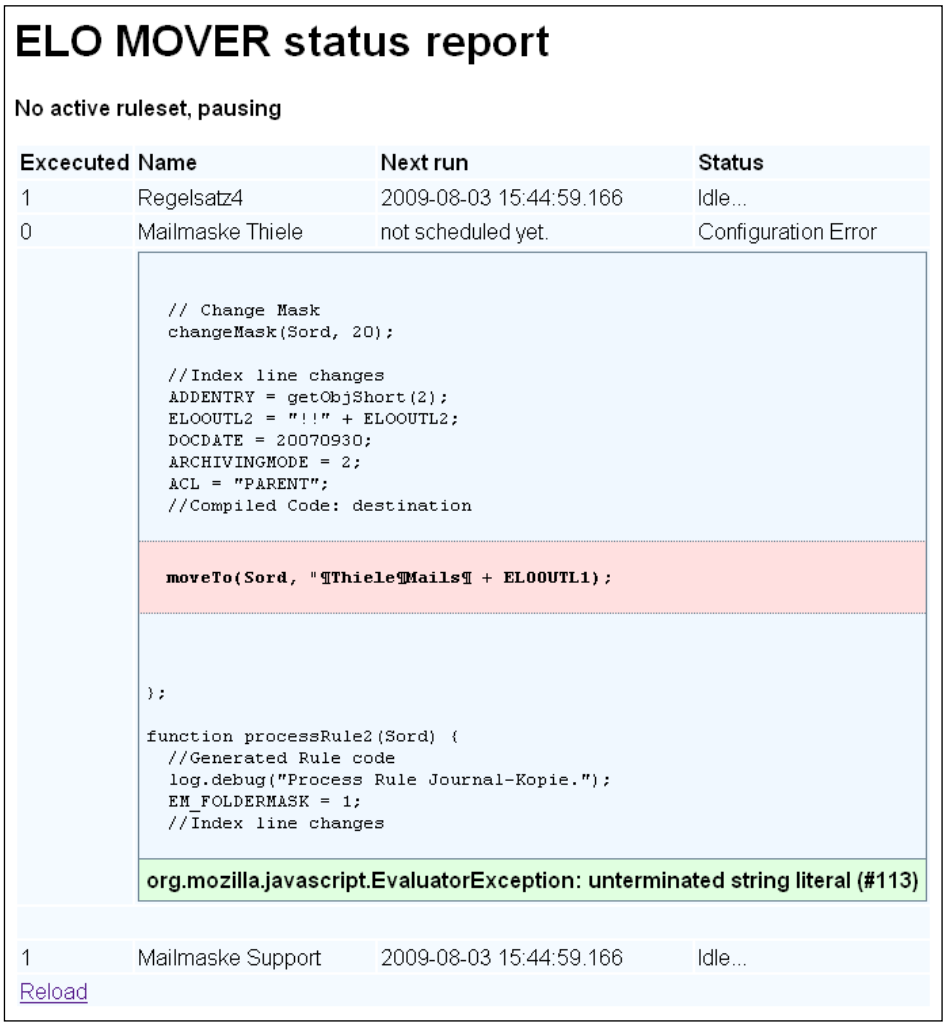


Abb. 18: Änderungen durch 'Reload' übernehmen

Änderungen im Archiv an den Regeln oder den Rahmen-Skripten können durch einen Klick auf *Reload* ohne Neustart des Servers übernommen werden.



Abb. 19: ELOas reload report

Mit *Back to Status Page* kommt man anschließend wieder in die normale Statusanzeige zurück.

Auf der Registerkarte *Einfügen* enthalten die Kataloge Elemente, die mit dem generellen Layout des Dokuments koordiniert werden sollten. Mithilfe dieser Kataloge können Sie Tabellen, Kopfzeilen, Fußzeilen, Listen, Deckblätter und sonstige Dokumentbausteine einfügen.