# ELOas Programmierhandbuch

ELOas Version 7.00.040

JavaScript Modul Version : 7.00.032

Das ELO Automation Services (ELOas) Programmierhandbuch 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.

## Inhalt

1 Funktionsweise des ELOas ..... 3
1.1 Überblick ..... 3
1.2 Suchmethoden (Indexsuche, Treewalk, Aufgabenliste, Mailbox) ..... 3
1.3 Skriptausführung ..... 4
1.4 Anlegen eigener Module ..... 6
1.5 Lazy Initialization ..... 7
2 Fehlersuche ..... 8
2.1 Syntaxfehler im Skript ..... 9
2.2 Logische- oder Laufzeitfehler ..... 11
3 Standardmodule ..... 11
3.1 cnt: ELO Counter Access ..... 11
Verfügbare Funktionen ..... 12
3.2 db:DB Access ..... 14
Verfügbare Funktionen ..... 14
Imports ..... 16
Verbindungsparameter ..... 16
JavaScript Code ..... 17
3.3 dex: Document Export ..... 20

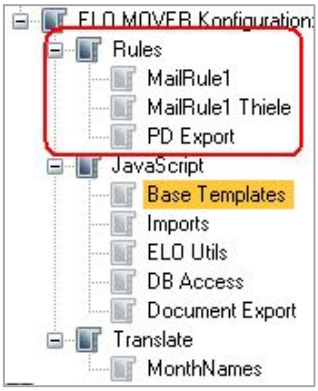
Verfügbare Funktionen ..... 21
JavaScript Code ..... 21
3.4 ix: IndexServer Functions ..... 23
Verfügbare Funktionen ..... 24
3.5 wf: Workflow Utils ..... 25
Verfügbare Funktionen ..... 26
3.6 mail: Mail Utils ..... 28
Verfügbare Funktionen zum Lesen eines Postfachs ..... 28
Verfügbare Funktionen zum Versenden von Mails ..... 33
3.7 fu: File Utils ..... 34
Verfügbare Funktionen ..... 34

# 1 Funktionsweise des ELOas

## 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 AdminConsole 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“). Der Ruleset enthält weiterhin eine Suchanfrage und eine Abfolge von Regeln zur Bearbeitung der Daten.




Der ELOas aktiviert im Betrieb in einer rotierenden Abfolge jeden Ruleset aus seiner Liste unterhalb von „ELOas\Rules“. Für jeden Ruleset wird zuerst geprüft, ob die Intervallbedingung erfüllt ist (sind seit dem letzten Lauf 10 Minuten vergangen?). Wenn diese noch nicht erfüllt ist, wird der 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.

## 1.2 Suchmethoden (Indexsuche, Treewalk, Aufgabenliste, Mailbox)

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:

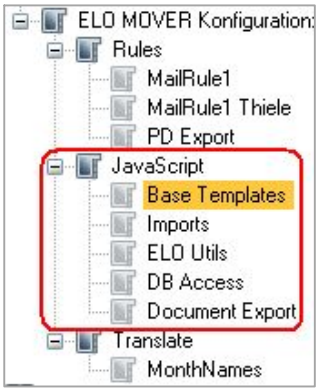
- TREEWALK: Im SEARCHVALUE wird die ObjektId oder ARCPATH zum Startobjekt hinterlegt. Es wird dann der komplette Teilbaum durchlaufen und der Ruleset zu jedem Eintrag mit der passenden Verschlagwortungsmaske aufgerufen.
- WORKFLOW: es werden alle Workflow Termine des ELOas Anwenders eingelesen und der 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. Der Ruleset wird zu jeder Mail in dem Postfach mit einem leeren Dokument aufgerufen.

## 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:

1. Die Zuweisungen in der XML Konfiguration dürfen vollwertige JavaScript Ausdrücke mit beliebigen Funktionsaufrufen enthalten.
2. In die XML Konfiguration können beliebige JavaScript Code Abschnitte mit komplexen Routinen eingebettet werden.
3. 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.
4. 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).

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 selber verändert werden muss. Man kann also mit einen Standardprogramm arbeiten, welches aber extrem 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 ELOasModule 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.

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

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

(Beachten Sie bitte, dass die einzelnen Funktionen mit einem Komma und keinem Semikolon getrennt warden, da es sich um eine Aufzählung handelt.)

Diese Funktionen können dann vom JavaScript Code mit dex.command1(x,y) oder dex.command2() angesprochen warden. 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.

## 1.4 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.


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 der 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). Wenn hier dann nicht ein sehr hoher Aufwand für die Erkennung eines ungültigen Zustands und ein automatisches Reconnect getrieben wird, führt das dazu, dass der komplette Service neu gestartet werden muss. 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 Kapitel „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.

## 1.5 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. Der Ruleset liest aus dieser Ressource per „readUser“ einen Anwendernamen 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.

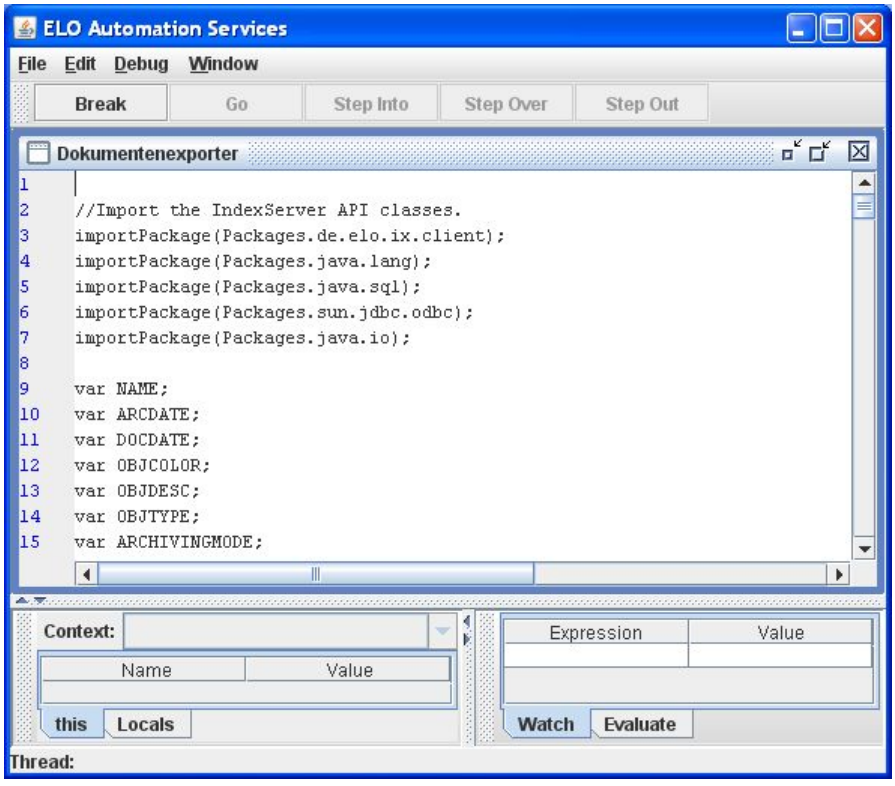
Wenn eine Regel aus dem Ruleset dann den Anwendernamen ermitteln möchte, wird die Funktion readUser() aufgerufen. Dort wird zuerst geprüft, ob bereits eine Verbindung besteht. Wenn nicht, wird sie mit Open() geöffnet und das readerInitialized auf true gesetzt. Bei nachfolgenden Aufrufen wird also kein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

Wenn der 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.

## 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>

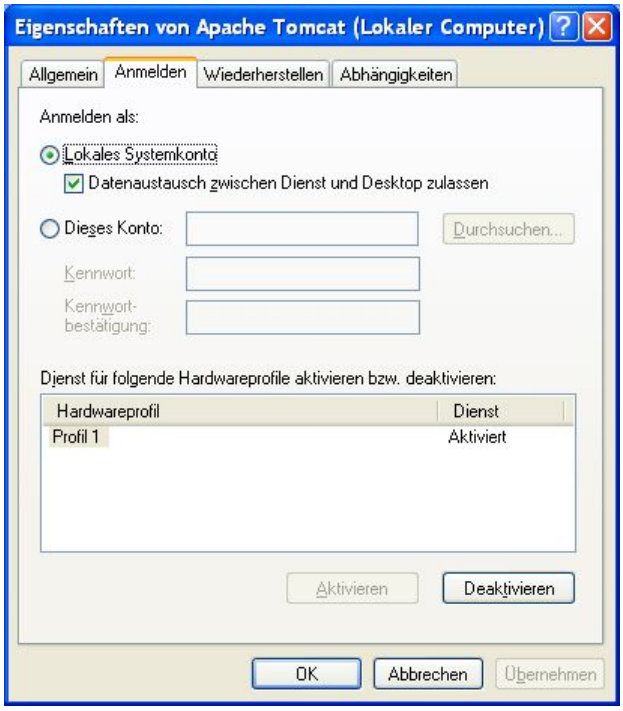



Zum Betrieb des Debuggers sollte der ELOas lokal auf der Entwicklermaschine ausgeführt werden. Falls der Tomcat unter dem Systemkonto läuft, muss explizit die Option

„Datenaustausch zwischen Dienst und Desktop zulassen“ aktiviert werden, andernfalls kann der Debugger das Ausgabefenster nicht anzeigen.

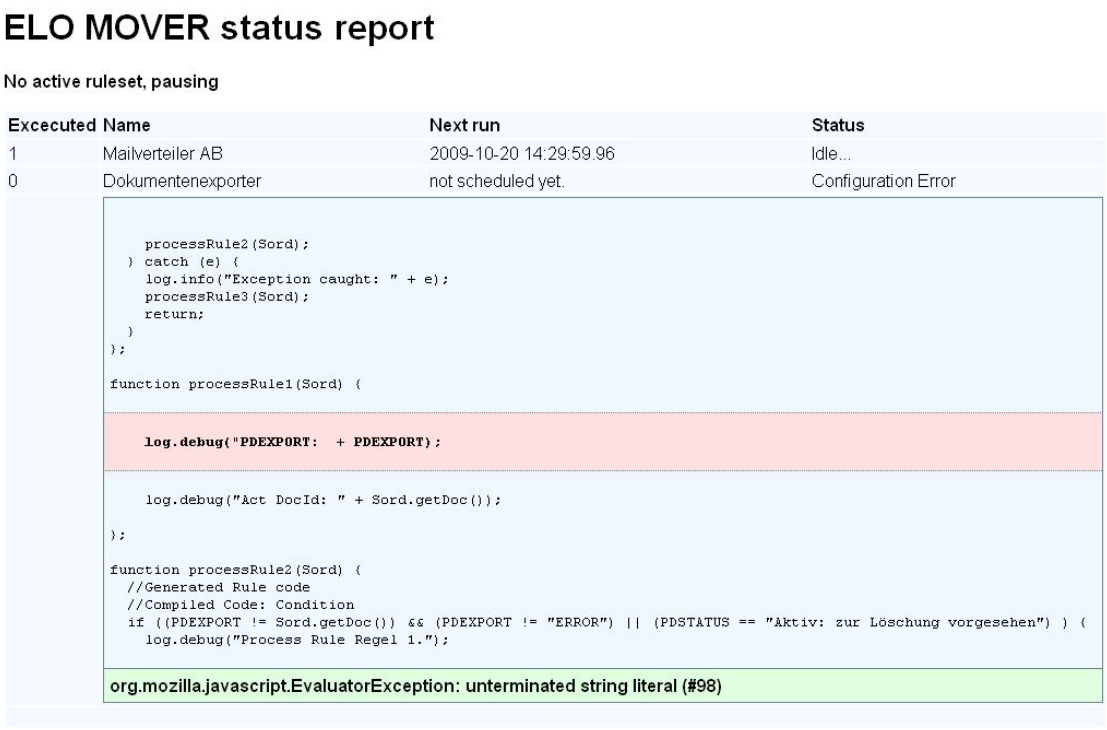
Falls Sie mehrere aktive Rulesets haben, gibt es für jeden ein eigenes Debuggerfenster. Sie können über den Menüeintrag „Window“ zwischen den einzelnen Ansichten umschalten.

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




## 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)

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 bitte 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.

## 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
```

# 3 Standardmodule

## 3.1 cnt: ELO Counter Access

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

### Verfügbare Funktionen

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

```
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 Prefix und die Länge der eigentlichen Zahl kann 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 TrackId) 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;
}
```

## 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.

### 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.

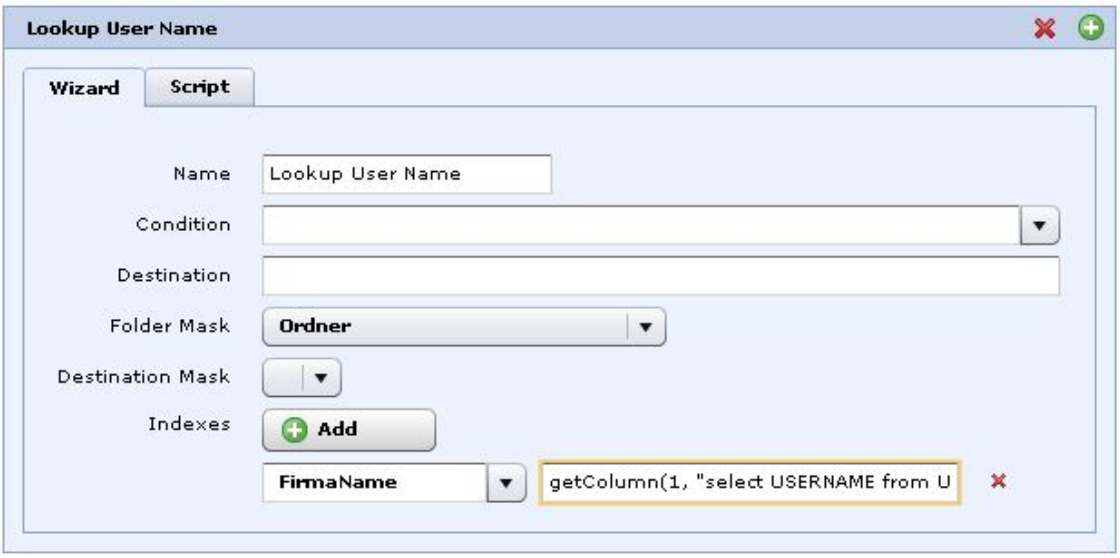
Beispiel: „select USERNAME from CUSTOMERS where USERID = 12345“

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:




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 bitte darauf, dass JavaScript Bezeichner case sensitive arbeiten. Die von der Datenbank zurückgemeldeten Spaltennamen müssen also in korrekter Groß/Kleinschreibweise im Skript verwendet werden.

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.

### Imports

Art und Umfang der benötigten Imports sind Datenbankabhängig 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);
```

### 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 | Loginname 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. |

### 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.

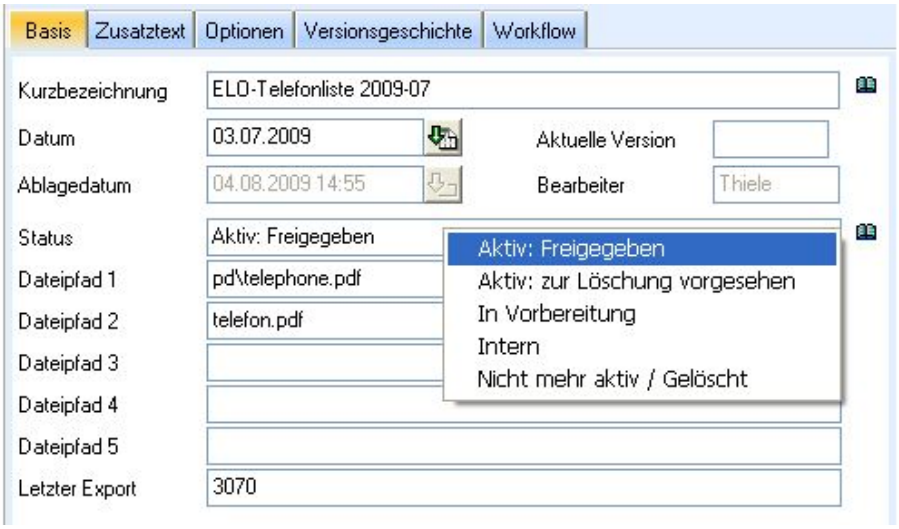
```
function getLine(connection, qry) {
  // Unterfunktion: erzeugt ein JavaScript Objekt mit
  // dem eingelesenen Datenbankinhalt
  function dbResult(connection, qry) {
    // Zuerst die Verbindung aufbauen
    dbInit(connection);

    // 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 Meta Daten 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;
}
```

## 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.




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 Anwender 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'
```

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

### JavaScript Code

Als erstes wird der Basispfad „docRoot“ für die Dokumentenablage bestimmt. Der Zielpfad wird immer aus dieser Einstellung und der Anwendereingabe in der Ablagemaske ermittelt. Prinzipiell wäre es möglich, diesen Basispfad leer zu lassen, so dass der Anwender beliebige Pfade angeben kann. Diese Vorgehensweise würde aber ein extrem großes Sicherheitsrisiko darstellen, da dann jeder Anwender 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);
}
```

## 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.

### Verfügbare Funktionen

Löschen eines Sord Eintrags: deleteSord(). Dieser Funktion werden als Parameter die ObjektIds 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 ObjektId 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 ObjektId eines Eintrags anhand einer Indexzeilensuche. Wenn der Parameter MaskId 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 zurück gegeben. Beachten Sie bitte, dass man nicht erkennen kann, 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;

}
```

## 3.5 wf: Workflow Utils

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

1. 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 extrem leicht zu verwenden, führen aber nur eine einfache Funktion aus.

2. 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 sicher stellen, dass der Workflow nur einmal gelesen und geschrieben wird und nicht x-mal für jede Operation.

### Verfügbare Funktionen

Anwendernamen eines Personenknoten ändern: changeNodeUser(). Diese Funktion tauscht im aktuellen Workflow im Workflowknoten mit dem Namen „nodeName“ den Anwender gegen einen neuen Anwender „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);
  }
}
```

Aktuellen Workflow einlesen: readActiveWorkflow(). Diese Funktion liest den aktuell aktiven Workflow in eine lokale Variable zur Bearbeitung ein. Er kann am Ende mit writeWorkflow zurück geschrieben 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;
},
```

## 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.

### 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 der 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 sicher gestellt 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:

1. Verbindung aufnehmen, Postfachfolder öffnen: connectImap_<Verbindungsname>
2. Nächste Nachricht der Liste zur Bearbeitung: nextImap_<Verbindungsname>
3. Nachricht als bearbeitet markieren oder löschen: finalizeImap_<Verbindungsname>
4. 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 warden...

  // 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 den 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 sicher stellen, dass das Postfach nicht über alle 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 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;
}
```

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

## 3.7 fu: File Utils

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

### Verfügbare Funktionen

Dateiname bereinigen: clearSpecialChars(): Wenn ein Dateiname aus der Kurzbezeichnung erzeugt werden soll, dann kann diese kritische 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 zurück gegeben.

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