DOCUMENTS

SERVERSEITIGES JAVA-SCRIPTING
Programmierhandbuch

DOCUMENTS 5.0

Inhaltsverzeichnis

1. Einleitung 4

2. Definition von Java-Skripten 6

3. Regeln und Konventionen 9

3.1 Technische Namen und Variablennamen 9
3.2 Das Arbeitskopienkonzept 9
3.3 Umgang mit sogenannten teuren Ressourcen 13
3.4 Scriptlänge 14
3.5 Eventkaskadierung 15
3.6 Potenziell gefährliche Workflow-Scripte 15
3.7 Variablen immer deklarieren 16

4. Fallbeispiele 17

4.1 Aufruf bei Neuanlage einer Mappe 17
4.2 Aufruf bei Mappenspeicherung 19
4.3 Aufruf nach Mappenspeicherung 22
4.4 Aufruf beim Löschen 23
4.5 Zugriff auf das Dateisystem von DOCUMENTS 24
4.6 Aufzählungswerte dynamisch ermitteln 25
4.7 Datenbankzugriffe via Scripting 27
4.8 Caching der Daten teurer Ressourcen 29
4.9 Benutzerdefinierte Aktionen an Mappen 30
4.10 Benutzerdefinierte Aktionen an Ordnern 31
4.11 Verrechtung benutzerdefinierter Aktionen 32
4.12 Skript als Job ausführen 33
4.13 Mappenpool per JobS-kript gefüllt halten 35
4.14 Entscheidungen und Wächter im Workflow 35
4.15 Signaleingänge im Workflow 37
4.16 Signalausgänge im Workflow 39
4.17 loginscript, afterLoginScript, setPasswordScript 40
4.18 afterMailScript 41
4.19 AccessSkript am Mappentypen 43
4.20 Skriptklassen erweitern 44
4.21 Singleton-Mappen 46
4.22 Download von Binärdateien per benutzerdefinierter Aktion 47

5. Testen, Debuggen und Verschlüsseln 49

5.1 Skripte testen 49
5.2 Logoutput um Scriptausführungen erweitern 49
5.3 Parameter der Scriptausführung anpassen 50
5.4 Debuggen mit dem Script-Debugger 50
5.5 Skripte verschlüsseln 51

Abbildungsverzeichnis 52

1. Einleitung

Der **DOCUMENTS 5**-Server verfügt über eine integrierte *Scripting-Engine* und ermöglicht so die *serverseitige* Ausführung von *JavaScript.*

Die Scripting-Engine verarbeitet *JavaScript* in den JS-Versionen 1.0 bis 1.9, ab Version 1.3 konform mit der ECMA-Skript-262 Spezifikation.

Diese Skripte werden im **DOCUMENTS-Manager** definiert. Dies geschieht in Form von globalen Skript-Objekten, die zunächst in einer Scripting-Bibliothek des angemeldeten Mandanten abgelegt werden (Abb. 1).

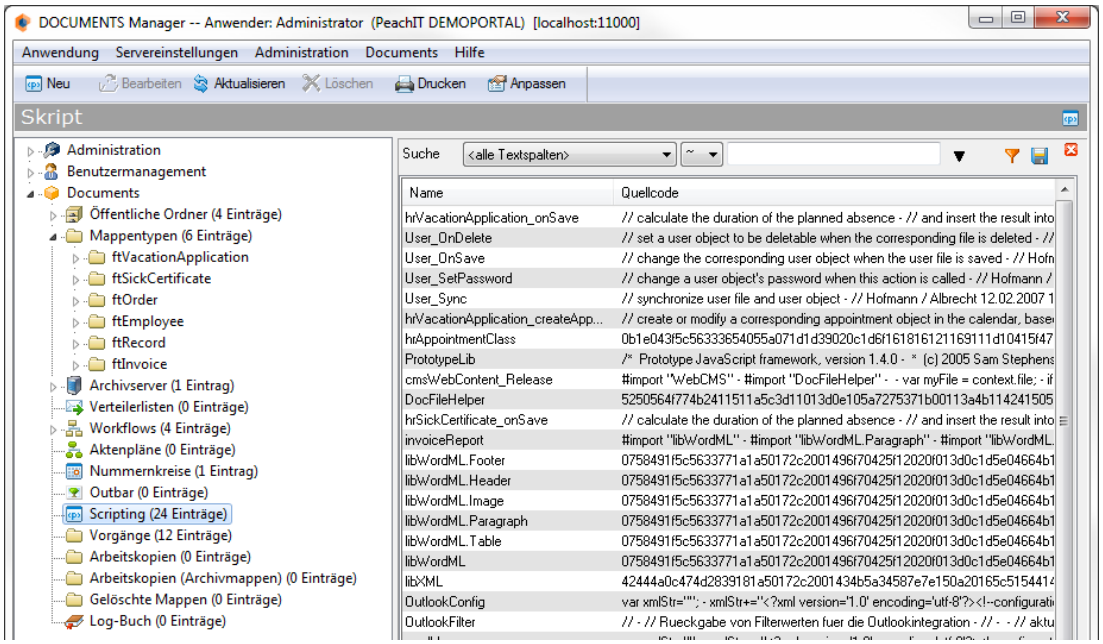


Abb. 1: Scripting-Bibliothek im DOCUMENTS-Manager

Anschließend werden diese für die Verwendung an vordefinierten Stellen eingebunden. Die Ausführung erfolgt entsprechend der eingebundenen Stelle zur Laufzeit bei bestimmten Aktionen. Folgende Einsatzgebiete für serverseitige Skripte stehen hierbei zur Verfügung:

- Standardaktionen an Mappen. Hierbei erfolgt die Einbindung des Skripts an einer definierten Aktion am Mappentyp:
    - Neuanlage einer Mappe
    - Beim Bearbeiten
    - Beim Speichern
    - Nach dem Speichern
    - Beim Archivieren
    - Beim Löschen
    - Als „Erlaubte Aktionen“-Skript: Dieses steuert global für die Mappe, welche Aktionen dem Benutzer generell zur Verfügung stehen.

Abb. 2 zeigt für einen Mappentyp im **DOCUMENTS-Manager**, wie die Zuordnung vorhandener Skripte an bestimmte Aktionen umgesetzt wird.

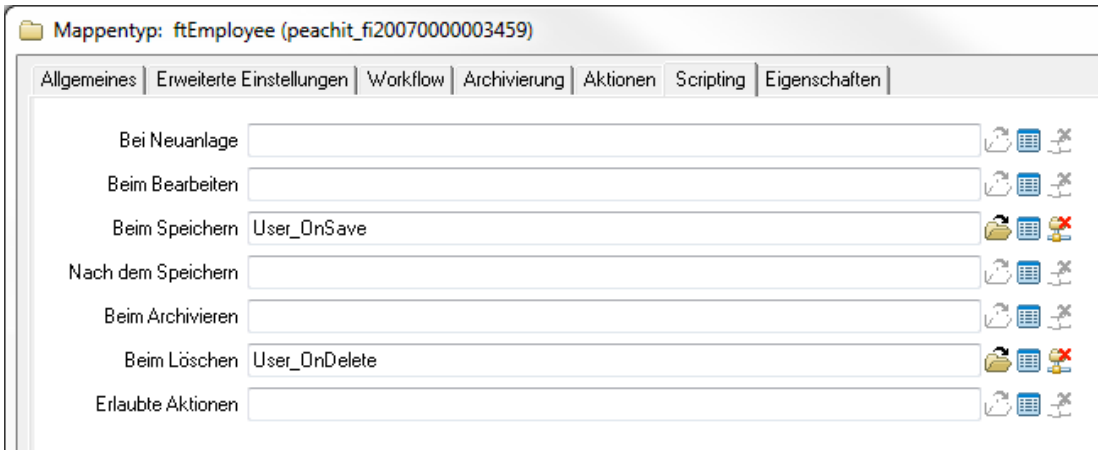


Abb. 2: Skript-Aktionen eines Mappentyps

- Als benutzerdefinierte Aktionen an der Mappe und an Ordnern
- Zur Beschreibung von Bedingungen (Guards) in Workflows
- Als Signalausgang in Workflows
- Im Rahmen von Feldbelegungsaktionen in Workflows
- Zur Definition der Aufzählungswerte eines Feldes vom Typen Aufzählungstyp

Damit lassen sich spezielle Anforderungen in **DOCUMENTS** realisieren, wie zum Beispiel:

- Klapplisten, die von den Werten anderer Felder abhängig sind
- Zugriff auf externe Datenbanken zum Füllen von Klapplisten oder Feldwerten
- Komplexe Guard-Bedingungen, die sich nicht durch einfache Ausdrücke definieren lassen
- Automatisierung von Vorgängen in Form von jobgesteuerten Skripten
- Aufruf externer DLLs , um Fremdsysteme anzusteuern
- zur Berechnung von Daten

Die **DOCUMENTS**-Skripting-Laufzeitumgebung ermöglicht hierzu den Zugriff auf verschiedene Mappen- und Feldeigenschaften. Ebenso können Laufzeit-Konstanten ausgelesen werden, wie etwa der gerade beteiligte Anwender oder der Name des aktuellen Workflow-Schrittes.

Voraussetzung für das erfolgreiche Erstellen von Skripten sind zumindest grundlegende Kenntnisse der Programmiersprache *JavaScript* und der verschiedenen von der **DOCUMENTS**-Skripting-Schnittstelle zur Verfügung gestellten Klassen, Objekte und Eigenschaften. Praktische Erfahrung in der Administration und Konfiguration von **DOCUMENTS** sin ebenfalls zwingende Voraussetzung, um insbesondere die Zusammenhänge zwischen den einzelnen Klassen und Objekten nachvollziehen zu können und sich über Auswirkungen bestimmter Skript-Konstruktionen auf die Systemleistung bewusst zu sein.

2. Definition von Java-Skripten

Die zentrale Bibliothek für Java-Skripte befindet sich in der Baumstruktur im **DOCUMENTS-Manager** unterhalb des Knotens *Documents* (vgl. Abb. 1).

Bitte beachten Sie, dass dieser Eintrag im Baum nur dann vorhanden ist, wenn Sie an einem vollständigen DOCUMENTS-Mandanten angemeldet sind. Reine Archivrecherche-Mandanten (sogenannte EASYWeb-Clients) verfügen nicht über Skript-Fähigkeiten.

Hier werden Skripte global angelegt, getestet und verwaltet. Ein Skript-Objekt besteht im Kern aus einem eindeutigen *Namen* und dem *Quellcode* (Abb. 3). Weitere Elemente, wie bspw. Skript-Parameter werden für die Ausführungen bei definierten Aktionen nicht zwangsläufig benötigt.

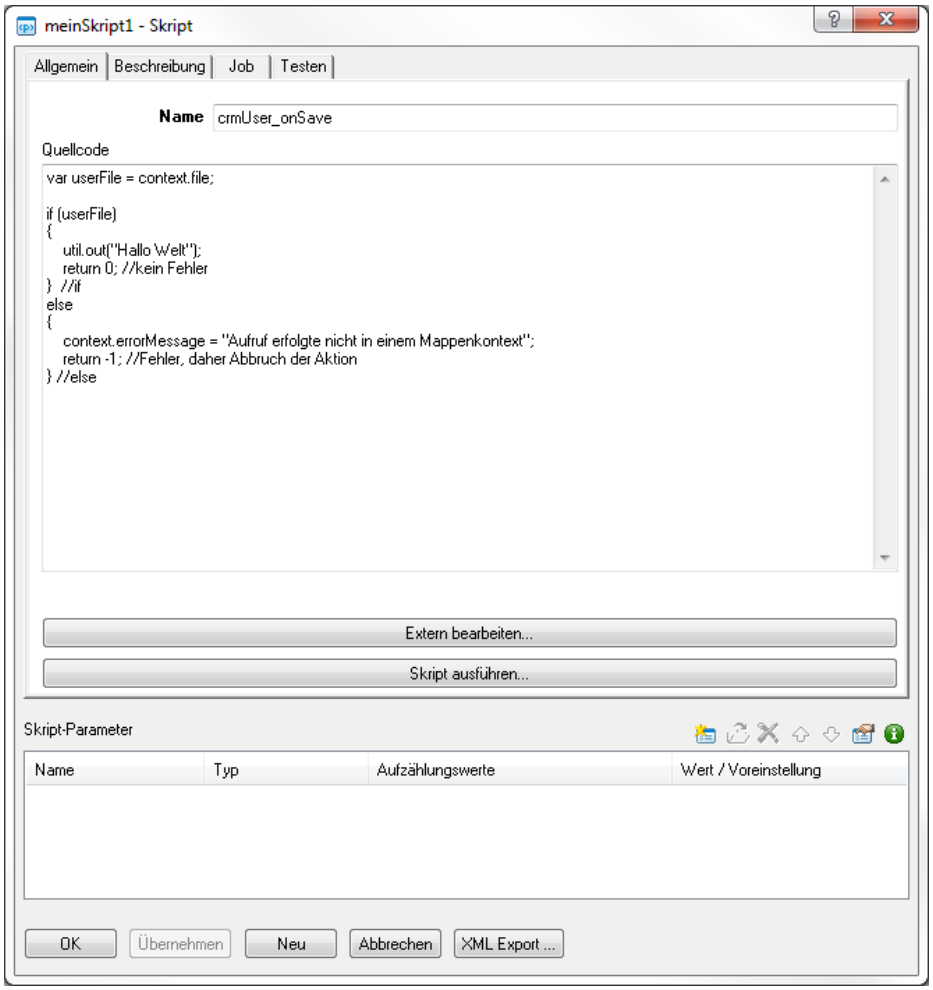


Abb. 3: Aufbau eines Skripts

Bei der Vergabe von Namen ist die Einhaltung sind zwar keine Konventionen erforderlich, allerdings ist es für die Verwendung und Wiederverwertbarkeit sinnvoll, sprechende Namen zu verwenden, die den späteren Zugriff auf die hier definierten Skripte von den verschiedenen Ankerpunkten wesentlich erleichtern.

Beispielsweise kann es sinnvoll sein, jedem Skript ein Kürzel voranzustellen, mit dem der Projektteil eindeutig identifiziert wird, zu dem das Skript gehört. Beispielsweise beginnen die Namen aller Skripte aus dem **RELATIONS Solution Package** mit `crm`, alle **CONTRACT**-Skripte mit `lcm`, die der **LDAP-Kopplung** mit `Ldap` und so weiter.

Ebenso hilfreich ist es, den Namen des zugehörigen Mappentypen oder Ordners sowie der aufrufenden Aktion anzufügen.

Ein Skript, das etwa beim Speichern einer Mappe des Mappentyps *crmUser* ausgeführt werden soll, könnte den Namen `crmUser_onSave` erhalten (vgl. Bsp in Abb. 3).

Dies erleichtert die Einbindung, da das entsprechende Skript in der Bibliothek schnell auffindbar ist und fehlerhafte Zuweisungen vermieden werden.

Ein einmal angelegtes Skript können Sie anschließend an verschiedenen Ereignissen bzw. Ankerpunkten einbinden und aktivieren.

Eine Erweiterungsmöglichkeit besteht darüber, Skripte über sogenannte *Eigenschaften* einzubinden. Ein Beispiel hierfür findet sich in **CONTRACT**: Wird ein Vertrag per Mail gesendet, wird in diesem Zusammenhang automatisch die versendete Mail als Mappe vom Typ *Aktenvermerk* gespeichert. Die Einbindung des Skripts erfolgt über eine *Eigenschaft* der versendeten Vertragsmappe.

Codeeingabe mit einem externen Editor

Das Eingabefeld für den Quellcode stellt diesen lediglich dar, allerdings werden hier keinerlei Formatierungsmöglichkeiten angeboten.

Klicken Sie auf die Schaltfläche *Extern bearbeiten*, öffnet sich standardmäßig das Skript im Windows-Editor.

Es besteht jedoch die Möglichkeit, einen beliebigen externen Editor einzubinden und den Code mit dessen Funktionen zu bearbeiten.

Damit Sie einen bestimmten Editor als externes Tool nutzen können, muss dieser lediglich eine wesentliche Bedingung erfüllen – es muss möglich sein, beim Programmstart Pfad und Dateinamen einer direkt zu öffnenden Textdatei mit zu übergeben.

Die Einbindung des externen Editors geschieht über eine *Eigenschaft*, die Sie auf dem Eigenschaften-Register Ihres Mandanten-Objektes hinterlegen. Die Eigenschaft lautet

```
scriptEditor
```

Der Wert der Eigenschaft muss der volle Pfad und Dateiname des auszuführenden Editors sein, beispielsweise

```
C:\programme\tools\notepad++\notepad++.exe
```

Abb. 4 zeigt diese Einstellung an einem Beispielmandanten.

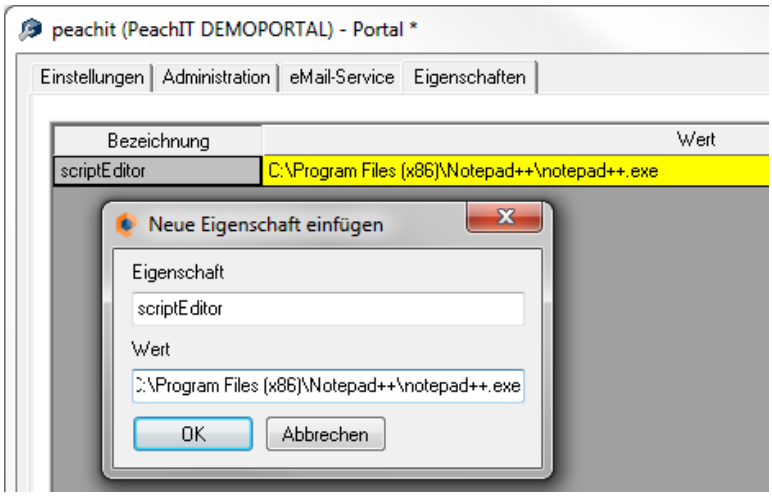


Abb. 4: Einbindung eines externen Editors

Die Verwendung eines externen Editors bringt für die Skriptentwicklung eine Reihe von Vorteilen mit sich:

- Syntax-Highlighting für JavaScript-Code
- Einrückung des eingegebenen Quelltextes
- Auffinden und farbliches Hervorheben von zueinander gehörigen Klammerpaaren
- Einfache Anlage von Sicherheitskopien während der Codeeingabe
- Einbindung etwa von Versionierungstools wie CVS oder SVN

Alternativ kann der Script-Editor auch in der Datei `documents.ini` der Documents-Installation eingetragen werden:

```
$SCRIPTEDITOR=C:\Program Files (x86)\Notepad++\notepad++.exe -multiInst
```

In diesem Fall wird der Editor in allen Mandanten der Installation verwendet.

3. Regeln und Konventionen

Im Umfeld der Programmierung von Skripten ergeben sich aus der Projektpraxis einige wichtige Rahmenbedingungen, deren Berücksichtigung sich empfiehlt, um den langfristigen Projekterfolg und eine komfortable Projektwartung zu gewährleisten.

3.1 Technische Namen und Variablennamen

So stellt beispielsweise der Aufbau einer DOCUMENTS-Mappe und deren Abbild als Objekt der *DocFile*-Klasse unter Umständen ein Problem dar, sofern Sie sich bei der Vergabe der technischen Namen der Mappenfelder nicht an die sogenannten programmiersprachenüblichen Konventionen halten. Diese Konventionen sind insbesondere aus der Programmierung in C/C++ und JAVA bekannt, haben aber auch für viele andere Programmiersprachen Gültigkeit und gelten im Allgemeinen als guter Programmierstil.

Da auf Mappenfelder einer Mappe im Scripting direkt (als öffentlich sichtbares Attribut eines DocFile-Objektes) zugegriffen wird, gelten diese Konventionen auch für Scripts in vollem Umfang:

Technische Namen und Variablennamen sollten **nie länger als 32 Zeichen** werden.

Sonderzeichen in technischen Namen und Variablennamen sind **absolut verboten**. Insbesondere bedeutet das, Umlaute (ä ö ü Ä Ö Ü ß), Leerzeichen und Satzzeichen sind tabu, aber auch nahezu alle sonstigen auf einer DIN-Tastatur befindlichen Sonderzeichen.

Einzige Ausnahme der vorherigen Regel: der **Unter_strich ist erlaubt** und ersetzt somit in gewisser Hinsicht sowohl Leerzeichen als auch Bindestrich.

Verwenden Sie also nur **Buchstaben** (a bis z und A bis Z) sowie **Ziffern** (0 bis 9). Beginnen Sie einen technischen Namen oder einen Variablennamen **niemals** mit einer Ziffer!

Erlaubt sind also etwa `psFeld123` oder `_100_internes_Feld`.

Absolut verboten wären `333_bei_Issos_Keilerei` oder `Äquivalenzziffer`.

Sollten Sie sich bei der Definition Ihrer Mappentypen und deren Feldern nicht an diese Konventionen halten, wird zwar **DOCUMENTS** mit seinen Bordmitteln keine direkten Fehler zu Tage fördern, allerdings verbauen Sie sich somit wirkungsvoll den Feldzugriff aus Skripten oder den anderen APIs von **DOCUMENTS**.

Halten Sie sich schon bei der Definition Ihrer Zugriffsprofile, Mappentypen, Felder, Register, Nummernkreise, Workflows und öffentlichen Ordner bei den technischen Namen konsequent an die beschriebenen Namenskonventionen.

3.2 Das Arbeitskopienkonzept

Die Scripting-Schnittstelle kennt zwei Möglichkeiten, eine DOCUMENTS-Mappe zu bearbeiten. Zum einen existiert das Befehlspaar `startEdit()` und `commit()`, welches

exakt dem Bearbeiten und Speichern einer Mappe über die Weboberfläche nachempfunden ist, zum anderen können Änderungen an einem DocFile-Objekt auch direkt vorgenommen und mit dem Befehl `sync()` auf die zugrundeliegende DOCUMENTS-Mappe synchronisiert werden.

Um den gravierenden Unterschied in der Verarbeitungsweise nachvollziehen zu können, muss man das Konzept der Arbeitskopien kennen und verstehen. Diesem Konzept kommt in DOCUMENTS als multiuser-tauglicher Applikation eine zentrale Bedeutung zu.

Es funktioniert folgendermaßen:

Ein Benutzer recherchiert eine DOCUMENTS-Mappe, wahlweise über eine Suche oder durch Navigieren durch die öffentlichen Ordner.

Er öffnet eine einzelne Mappe und zeigt sich die Indexfelder im Browser an.

Wenn in diesem Moment andere Anwender dieselbe Mappe öffnen, bekommen alle Anwender (spezielle Verrechtungen einmal außen vor gelassen) exakt die gleichen Inhalte zu Gesicht.

Versetzt nun ein Anwender die Mappe in den Bearbeitungsmodus, erhält er in dem Moment exklusive Schreibrechte auf die Mappe. Kein anderer Anwender kann die Mappe nun zeitgleich ebenfalls bearbeiten. Intern hat DOCUMENTS für den bearbeitenden User nun eine Arbeitskopie der Mappe erzeugt. Diese Arbeitskopie wird vom Anwender modifiziert, nicht aber die ursprüngliche DOCUMENTS-Mappe.

Das bewirkt, dass andere Anwender des Systems nach wie vor bei der Recherche den bisherigen Ist-Zustand der Mappe zu Gesicht bekommen – und die Mappe nicht bearbeiten können, weil das System erkennt, dass Anwender A die Mappe schon bearbeitet.

Erst wenn Anwender A seine Änderungen (und damit die Mappe) speichert, werden die an der Arbeitskopie erfolgten Änderungen tatsächlich in die echte DOCUMENTS-Mappe zurückgeschrieben. Erst ab diesem Moment werden diese Änderungen also auch für alle anderen User des Systems wieder sichtbar.

Würde der bearbeitende User hingegen seine Änderungen verwerfen, müsste das System lediglich die Arbeitskopie löschen, ein Rollback an der echten DOCUMENTS-Mappe wäre somit also nicht erforderlich.

Damit ist das Konzept der Arbeitskopie also nicht nur ein Mechanismus zur Gewährleistung der Transaktionssicherheit, sondern auch ein wichtiges Performance-Merkmal.

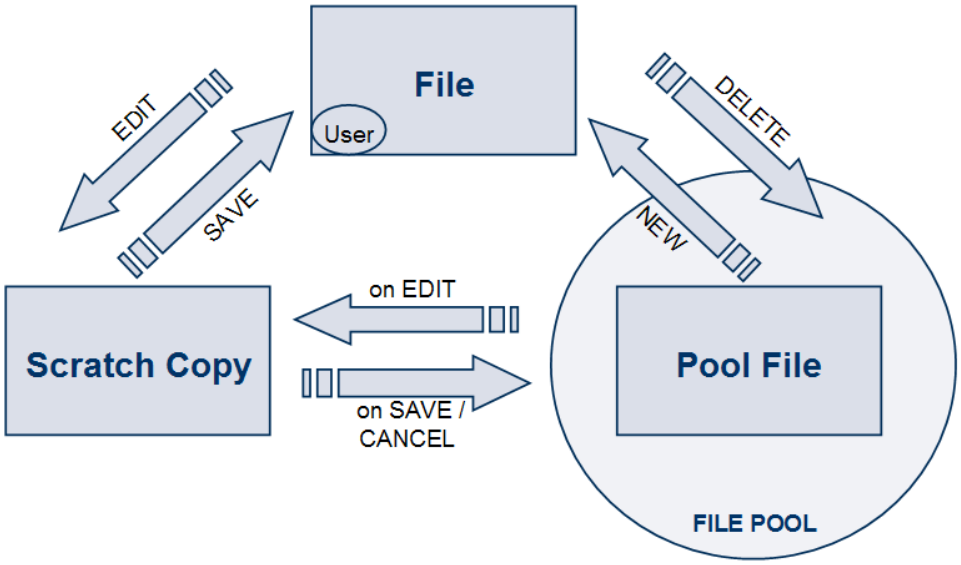


Abb. 5: Arbeitskopienkonzept

Dieses Arbeitskopienkonzept bedeutet aber in der Konsequenz für die Programmierung von Skripten, dass man sich darüber im Klaren sein muss, wann welche Instruktionen angewendet werden dürfen, da `startEdit()` ja ebenfalls eine Arbeitskopie der Mappe erzeugt, während `commit()` aus dieser Arbeitskopie die echte Mappe generiert. Die Anweisung `abort()` ist ebenfalls analog zum Abbrechen-Button in der Weboberfläche zu sehen, verwirft also eine vorhandene Arbeitskopie der Mappe.

Daraus ergibt sich, dass diese Anweisungen in bestimmten Aufrufkontexten von Skripten auf keinen Fall eingesetzt werden dürfen, da die Datenkohärenz und die Systemstabilität sonst nicht mehr gewährleistet werden können.

Bei folgenden Scripting-Events existiert schon eine Arbeitskopie der Mappe:

- Bei Neuanlage
- Beim Bearbeiten
- Beim Speichern
- Nach dem Speichern

Das heißt, wenn Sie Skripte für genau diese Events programmieren, dürfen Sie die DocFile-Objekte auf keinen Fall über `startEdit()` / `commit()` / `abort()` bearbeiten, sondern müssen mit `sync()` arbeiten.

Darüber hinaus ist es bei den Scripting-Events „Beim Archivieren“ und „Beim Löschen“ ebenfalls verboten, das über `context.file` verfügbare Mappenobjekt in den Bearbeitungsmodus zu versetzen.

In der Praxis hat es sich außerdem als kritisch erwiesen, Mappen über `startEdit()` zu bearbeiten, die sich in einem Workflow oder einer Versendung befinden. Meist scheitert das Bearbeiten der Mappe dann daran, dass ein einzelner Benutzer oder eine Benutzergruppe die Mappe aufgrund des aktuellen Workflowschrittes sperrt, jedoch sind auch andere Konstellationen denkbar, die prinzipiell ein `startEdit()` ermöglichen würden. Diese

speziellen Umstände könnten dann aber dazu führen, dass gleichzeitig auf die Mappe zugreifende Anwender übers Web in für sie unerklärliche Fehler stolpern.

Folgerung: Verzichten Sie auf `startEdit()`, wenn sich das Mappenobjekt in einem Workflow befindet (inzwischen verhindert die aktuelle **DOCUMENTS**-Version die Verwendung im Workflow automatisch. Über die Methode `DocFile.getLastError()` kann man die resultierende Fehlermeldung auslesen).

Sofern Sie in einem Skript ein Mappen-Objekt per `startEdit()` in den Bearbeitungsmodus versetzen, sollten Sie einige weitere wichtige Grundregeln beherzigen:

Vergewissern Sie sich, dass der Rückgabewert des Aufrufs von `startEdit()` an der Mappe ein „`true`“ zurückgibt.

Stellen Sie hundertprozentig sicher, dass spätestens zum Laufzeitende des Scripts zu jedem `startEdit()` ein korrespondierendes `commit()` oder `abort()` erfolgt ist – Sie laufen sonst Gefahr, dass Sie eine im Bearbeitungsmodus befindliche Mappenleiche generieren, die kein Anwender mehr bearbeiten kann. Zwar räumt die Scripting-Engine solche Arbeitskopien zum Skriptende selbst auf, jedoch ist nicht immer gewährleistet, dass dies zeitnah genug erfolgt, und es gilt allgemein nicht als guter Programmierstil, selbst erzeugte Objekte nicht sauber zu schließen.

Jede Erzeugung einer Arbeitskopie über `startEdit()` bewirkt zwangsläufig entsprechende Last auf der Datenbank, da ja das Mappen-Objekt zunächst komplett dupliziert werden muss. Damit stellt die Verwendung von `startEdit()` einen Zugriff auf teure Ressourcen dar (siehe unten).

Die obigen Warnhinweise erwecken also den Anschein, als wäre `sync()` die generell zu bevorzugende Operation, wenn Änderungen an Mappen per Scripting durchzuführen sind. Dies ist allerdings in einem Punkt auch nur bedingt korrekt, denn die Synchronisation einer Mappe kann auch dann durchgeführt werden, wenn diese von einem Anwender übers Web gerade bearbeitet wird (mithin also eine Arbeitskopie existiert).

Infolgedessen besteht beim Synchronisieren also zumindest die theoretische Möglichkeit eines Informationsverlustes, da der Webanwender beim Speichern seiner Arbeitskopie prinzipiell vom Skript vorgenommene Änderungen an Feldwerten wieder überschreiben kann. Ausnahmen bilden die zuvor aufgelisteten Scripting-Events, bei denen prinzipbedingt zum Aufrufzeitpunkt des Scripts schon eine Arbeitskopie existiert – hier wirkt sich das `sync()` auf die Arbeitskopie aus, nicht auf die „darunter“ liegende DOCUMENTS-Mappe.

Sie sollten also für jedes Skript, welches Mappeninhalte verändern muss, stets gewährleisten, dass Sie keine kritischen Informationsverluste riskieren, indem zeitgleich Benutzer die gleichen Mappen bearbeiten könnten wie Ihre Skripte.

Neben einer rein organisatorischen Gewährleistung dieser Sicherheit besteht auch aus dem Scripting heraus die Möglichkeit, durch Auslesen der Attribute „Locked“ und „LockedBy“ den Bearbeitungsstatus des DocFile-Objektes zu eruieren.

Wichtige Regeln:

- Verzichten Sie nach Möglichkeit auf startEdit() und commit().
- Verwenden Sie stattdessen bevorzugt die sync()-Operation.
- Sollte startEdit() unverzichtbar sein, stellen Sie sicher, dass immer ein passendes commit() bzw. (im Fehlerfall) ein abort() vorhanden ist.
- Gehen Sie sparsam mit diesen Operationen um, da sie Datenbanklast verursachen.

3.3 Umgang mit sogenannten teuren Ressourcen

In der Programmierung spricht man bei einigen Operationen von einem Zugriff auf sogenannte **teure Ressourcen**. Dies sind in der Regel solche Datenquellen, die auf gegenüber dem Hauptspeicher des Rechners wesentlich langsameren Medien basieren – also insbesondere Datenquellen, die auf Festspeichermedien wie Festplatten oder DVD-Laufwerke zurückgreifen, aber auch Ressourcen, die ausschließlich über das Netzwerk verfügbar sind (Netzwerkfreigaben, externe Datenbanken wie ODBC-Datenquellen etc.).

„Teuer“ sind diese Ressourcen in der Hinsicht, dass der Zugriff auf sie verglichen mit einer Variable im Programmspeicher wesentlich mehr Zeit erfordert, und unter dem Aspekt, dass die gängigen Betriebssysteme, für die DOCUMENTS verfügbar ist, den Zugriff auf das Dateisystem, Netzwerkressourcen und externe (ODBC-) Datenquellen über eine meist limitierte Anzahl sogenannter Handles reglementieren.

Es liegt also auf der Hand, dass man als Programmierer sehr sorgfältig mit diesen Ressourcen umgehen und sie so selten wie möglich einsetzen sollte.

Außerdem sollten Sie sich angewöhnen, jede „teure“ Ressource, auf die Sie über Scripting zugreifen, sorgsam wieder zu schließen, um die eingesetzten Zugriffshandles so schnell wie möglich dem System wieder zur Verfügung zu stellen.

Was bedeutet dies für die Programmierpraxis konkret?

- Jedes von Ihnen geöffnete `DBConnection`-Objekt sollte sauber über den Aufruf der korrespondierenden `close()`–Methode wieder geschlossen werden.
- Jedes `DBResultSet`-Objekt ist nicht nur exklusiv an je ein `DBConnection`-Objekt gebunden, sondern sollte ebenfalls stets sauber durch Aufruf der `close()`–Methode am Objekt wieder geschlossen werden.
- Vermeiden Sie es, innerhalb von Schleifenkonstrukten massenweise `DBConnections` und `DBResultSets` zu erzeugen.
- `File`-Objekte (also Handles auf Dateien des Serverdateisystems) sind ebenfalls stets per `close()` nach ihrer Verwendung sauber wieder zu schließen.

In Verbindung mit `startEdit()` / `commit()` / `abort()` und beim Aufbau sehr umfangreicher `FileResultset`-Iteratoren stellt sogar die server-eigene Datenbank eine teure Ressource dar, da diese Operationen je nach Struktur Ihrer Mappentypen sehr

aufwendige Datenbankoperationen bedingen können, was sich durch entsprechend hohe Last auf der Datenbank bemerkbar machen kann. Folglich ist mit diesen Methoden und Objekten sehr sorgsam umzugehen.

Zugriffe auf teure Ressourcen wirken sich fast immer auf die Laufzeit des Skripts aus. Je nachdem, wo ein Skript mit Zugriffen auf teure Ressourcen eingesetzt wird, können die Auswirkungen für einen einzelnen Webanwender auch an unvermuteten Stellen auftreten, und es ist sogar denkbar, dass sich intensiver Gebrauch teurer Ressourcen auch auf die allgemeinen Antwortzeiten des Systems für alle Benutzer auswirkt.

Ein klassisches Beispiel dafür sind Skripte zur Generierung von Aufzählungswerten, die wahlweise die möglichen Klapplistenwerte aus ODBC-Datenquellen generieren oder über ein `FileResultset` auf Mappen eines anderen Mappentyps zugreifen (im letzteren Fall stellt sich zwar in der Regel die Frage, warum nicht einfach das wesentlich elegantere Referenzfeld verwendet wird, jedoch soll es auch Anwendungsfälle für diese komplexe Konstruktion einer Klappliste geben).

Wenn dieses Aufzählungsfeld als „in Trefferliste anzeigen“ markiert ist, hat dies zur Folge, dass schon der Aufruf eines (durchaus auch leeren) dynamischen öffentlichen Ordners, der auf den genannten Mappentypen filtert, oder die Durchführung einer (durchaus auch erfolglosen) Suche bewirkt, dass das in der Klappliste hinterlegte Skript ausgeführt werden muss, infolgedessen die Antwortzeiten der Weboberfläche spürbar verlängert werden können.

Wichtige Regeln:

- Zu jedem „`new DBConnection()`“-Aufruf gehört ein passender `close()`–Befehl.
- Jedes generierte `DBResultSet` muss durch Aufruf seiner `close()`–Methode sauber geschlossen werden.
- Setzen Sie nach Möglichkeit weder `DBConnection` noch `DBResultset` innerhalb von Schleifen ein.
- Aufzählungsfelder mit skriptgenerierten Aufzählungswerten sollten nicht in Trefferlisten dargestellt werden.
- File-Handles gehören nach ihrer Verwendung ebenfalls sauber über `close()` geschlossen.

3.4 Scriptlänge

Datenbankseitig ist das Codefeld eines Script-Objektes in DOCUMENTS (versionen 5.0, 5.1 und 6.0, alle Patchlevel) auf 64 KByte begrenzt. Ihre einzelnen Scripts dürfen also nicht mehr als etwa 64.000 Zeichen Quelltext enthalten. Bei verschlüsselten Scripts reduziert sich die Maximallänge gar auf ca. 32.000 Zeichen. Werden Umlaute und Sonderzeichen im Code verwendet (z.B. in Inline-Kommentaren), reduziert sich die Maximallänge weiter, da die Umlaute in einer codierten Form gespeichert werden müssen.

Komplexere Projekte stoßen vergleichsweise schnell an diese Grenzen, und daher ist es sinnvoll, sich eine modulare Programmierweise anzugewöhnen.

Das bedeutet nichts anderes als dass Sie Codeteile in Hilfsfunktionen in Bibliotheksscripten ablegen, und für eine eventuelle Parametrisierung Ihrer Skripte verwenden Sie ebenfalls jeweils eigene Konfigurations-Skripte. In Ihren einzelnen Hauptmodulen importieren Sie die jeweiligen Bibliotheken dann mit Hilfe der Anweisung

```
#import "ScriptName"
```

Unter anderem machen die Solution Packages CONTRACT, RELATIONS und INVOICE sowie die LDAP-Kopplung intensiven Gebrauch davon.

Wichtige Regeln:

- Gehen Sie sorgfältig bei der Konzeption Ihrer Skripte vor
- Trennen Sie Code, Hilfsfunktionen und Konfiguration voneinander

3.5 Eventkaskadierung

Speziell die direkt am Mappentypen auf dem Reiter „Scripting“ konfigurierbaren Events werden innerhalb eines Skripts kaskadiert. Das heißt, wenn Sie zum Beispiel aus Skript „A“ heraus eine neue Mappe eines Mappentypen „B“ erzeugen, und an diesem Mappentypen ist ein Skript „C“ zur Ausführung „bei Neuanlage“ hinterlegt, dann bewirkt die Verarbeitung des Scripts „A“ genau zum Zeitpunkt der Neuanlage der Mappe „B“ die Verarbeitung des Scripts „C“.

In der Praxis wird dies gerne verwendet, wenn Mappentypen intensiv mit Referenzfeldern und Verknüpfungsregistern untereinander verknüpft sind (also sogenannte 1:n-Beziehungen bestehen) und etwa das Löschen einer Firmenmappe auch automatisch alle an diese Firma gekoppelten Ansprechpartner und Gesprächsnotizen löschen soll – oder im Umkehrschluss verhindert werden soll, dass übergeordnete Stammdaten aus dem System gelöscht werden können, solange noch Bewegungsdaten in Form von Vorgangsmappen im System vorliegen, die auf diese Stammdaten noch zurückgreifen.

Solche Konstrukte müssen zwangsläufig sehr gut dokumentiert werden und erfordern sehr viel Disziplin bei der Projektplanung und –realisierung, und insbesondere müssen Sie sich den durchaus nicht unkritischen Auswirkungen auf die Laufzeit der Skripte und die vom Anwender gefühlte Antwortzeit der Weboberfläche bewusst sein.

Die Eventkaskadierung ist so gesehen ein ebenso mächtiges wie gefährliches Werkzeug und sollte nur sehr dosiert und sehr vorsichtig eingesetzt werden. In jedem Fall ist es unabdingbar, solche komplexen Ausführungskontexte sehr umfassend mit einer Vielzahl von Testdaten zu testen und jeden denkbaren Grenzfall in den Tests zu berücksichtigen, da es sonst schnell zum Verlust wertvoller Echtdaten kommen könnte.

3.6 Potenziell gefährliche Workflow-Scripte

Zwischen DOCUMENTS-Mappen und dem Workflow, in dem sich die Mappen befinden, besteht zur Laufzeit eine feste Objektbeziehung. Aus eigentlich naheliegenden Gründen ist es für den Workflow daher sehr ungesund, wenn ein Skript, welches beispielsweise als

Signalausgang oder Signaleingang modelliert ist, die noch im Workflow befindliche Mappe löscht. – Zu gut deutsch: Sie würden den Ast absägen, auf dem Sie sitzen.

In diesem Fall gäbe es dann ja kein gültiges Mappen-Objekt mehr, welches bis zum definierten Ende durch den Workflow weitergeleitet werden könnte, und infolgedessen befände sich das Konstrukt aus Mappe und Workflow in keinem definierten Zustand mehr. In früheren Versionen konnte dies sogar zu einem vollständigen Absturz des **DOCUMENTS**-Servers führen.

Fehlprogrammierungen wie die eben beschriebene demonstrieren also anschaulich, in welcher Verantwortung Sie als Skript-Entwickler stehen, und ebenso deutlich sollte es erklären, warum sich ein Skript-Programmierer auch mit der administrativen Praxis und den Abläufen aus Anwendersicht sehr gut auskennen muss.

3.7 Variablen immer deklarieren

Insbesondere aus der Visual Basic Programmierung ist unter Entwicklern die Unsitte sehr weit verbreitet, Variablen nicht sauber zu deklarieren, sondern einfach in dem Moment einzusetzen, da sie das erste Mal benötigt werden.

Dieser sehr schlechte Programmierstil führt im Scripting zu besonders verheerenden Auswirkungen, da eine nicht sauber per `var variablenName = initialwert;` deklarierte Variable durch die erste Wertzuweisung automatisch als globale Variable erzeugt wird.

„Global“ ist in diesem Zusammenhang aber nicht auf die einzelne Skriptausführung bezogen, sondern auf die gesamte Laufzeit des Servers bis zum nächsten Neustart.

JEDE von Ihnen verwendete Variable ist sauber zu deklarieren.

4. Fallbeispiele

Im folgenden Kapitel werden Ihnen in einer Reihe von Fallbeispielen die verschiedenen Möglichkeiten vorgestellt, die sich durch den Einsatz von Scripting ergeben. Neben einem Überblick über die Einsatzmöglichkeiten der Scripting-Engine finden Sie hier ausführlich beschriebene Beispiele, die auch die Anwendung der von der Scripting-Engine zur Verfügung gestellten Klassen vorführen.

Die Beispiele sind anfangs bewusst einfach gehalten, um den Einstieg zu erleichtern. Im weiteren Verlauf diene die Beispiele nach und nach der Umsetzung stetig komplexer werdender Algorithmen oder Anforderungen.

4.1 Aufruf bei Neuanlage einer Mappe

Wenn Sie ein Skript am Mappentypen für die Ausführung **bei Neuanlage** definieren, so wird es jedes Mal dann ausgeführt, wenn Sie eine neue Mappe dieses Typs erzeugen. Dies kann insbesondere dann nützlich sein, wenn Sie direkt beim Erstellen einer Mappe auf Daten aus einer externen Datenbank zurückgreifen möchten, um etwa Felder der Mappe automatisch zu befüllen.

Das folgende Beispiel verzichtet allerdings noch auf Datenbankzugriffe; stattdessen werden zwei Mappenfelder direkt befüllt.

Erstellen Sie im DOCUMENTS-Manager ein neues Skript. Nennen Sie dieses Skript `psScriptSample_onFileCreation` und geben Sie folgenden Quelltext ein:

```
// aktuelle Mappe holen
var myFile = context.file;

if (!myFile)
{
  context.errorMessage = "Ausfuehrung erfolgt nicht im Mappenkontext!";
  return -1;
}
// Textfeld fuellen
myFile.Leerfeld1 = "Filled";

// Datumsfeld fuellen - demonstriert Autotexte am Context
// sowie die Handhabung eines Datumsfelds auf PortalScript-
// Ebene - dazu muss das %currentDate% in ein Date-Objekt
// umgewandelt werden.
var datumString = context.getAutoText("%currentDate%");
var dateObj = util.convertStringToDate(datumString, "dd.mm.yyyy");
myFile.Leerfeld2 = dateObj;

// Mappe im Bearbeitenmodus, also fuehren wir statt startEdit()
// und commit() einen sync() aus
myFile.sync();
```

Anschließend speichern Sie das Skript durch einen Klick auf OK.

Definieren Sie sich nun einen einfachen Mappentypen namens `ScriptSample`, der aus lediglich zwei Feldern besteht – einem Stringfeld mit dem technischen Namen `Leerfeld1` und einem Datumsfeld namens `Leerfeld2`. Binden Sie das soeben erstellte Skript an den Mappentypen, indem Sie auf dem Register „Scripting“ rechts neben „Bei Neuanlage“ die Auswahlliste mit den verfügbaren Scripts öffnen und das soeben definierte Skript `psScriptSample_onFileCreation` mit einem Doppelklick auswählen.

Speichern Sie den Mappentypen, nachdem Sie ihn freigegeben haben, und öffnen Sie einen Webbrowser, mit dem Sie sich in DOCUMENTS anmelden. Legen Sie nun eine neue Mappe Ihres gerade definierten Mappentyps an – die neue Mappe wird im Bearbeitungsmodus geöffnet, aber beide Felder sind schon vorgefüllt – im Stringfeld steht „Filled“ und im Datumsfeld ist das aktuelle Tagesdatum eingetragen – und das, obwohl keine Defaultwerte am Mappentypen definiert wurden.

Dieses einfache Beispiel demonstriert in diesen wenigen Zeilen gleich mehrere mächtige Werkzeuge der Scripting-Engine. Zunächst holt es sich aus dem permanent vorhandenen `context`-Objekt die aktuelle Mappe. Diese ist, da das Skript bei Neuanlage ausgeführt wird, die frisch erzeugte neue Mappe, die außer eventuell hinterlegten Defaultwerten und Autotexten noch keine Inhalte hat. Dieses `DocFile`-Objekt wird in Form der Variablen `myFile` referenziert.

Das Beispiel demonstriert anschließend einen Programmierstil, den man „**restriktives Programmieren**“ nennt – obwohl in dieser Übung ja definitiv feststeht, an welchen Event dieses Skript eingebunden wird, ist dies in der Praxis nicht immer gewährleistet. Insbesondere wenn weniger erfahrene Anwender Skripte in DOCUMENTS einbinden sollen, können Sie als Entwickler nicht sicher gewährleisten, dass Ihre Skripts auch korrekt eingebunden werden. Als Programmierer eines Skripts sind Sie also gefordert sich zu vergewissern, dass das Skript auch tatsächlich in einer Art und Weise eingebunden wurde, die über das `context`-Objekt ein `DocFile`-Objekt zur Verfügung stellt – und das geschieht im obigen Beispiel dadurch, dass die Existenz eines gültigen Objektes in der `if()`–Anweisung erfragt wird.

Anschließend wird dem Feld „`Leerfeld1`“ dieser Mappe der Wert „Filled“ zugewiesen. Sie sehen also, dass Sie mit einfachen Zuweisungen schreibend auf die Felder einer Mappe zugreifen können – unabhängig davon, ob sich das Feld auf dem Standard-Feldregister befindet oder ob es sich auf einem zusätzlichen von Ihnen definierten Feldregister befindet.

Der schreibende Zugriff auf das zweite Mappenfeld gestaltet sich etwas komplexer als beim ersten Feld, da es sich hier um ein Datumsfeld handelt. Zunächst definieren Sie daher im Code eine neue Variable namens „`datumString`“. Dieser weisen Sie nun das aktuelle Tagesdatum zu. Um bei dieser Gelegenheit den Zugriff auf die mappenunabhängigen Autotexte zu demonstrieren, wird im Beispiel nicht auf das Javascript-eigene „`new Date()`“ zurückgegriffen, sondern der Autotext `%currentDate%` ausgelesen. Dieses Tagesdatum wird automatisch im Format der aktuellen Sprache formatiert – bei einem deutschsprachigen DOCUMENTS also im Format `tt.mm.jjjj`.

Das Datumsfeld kann allerdings mit dieser Zeichenkette nichts anfangen, da es ein Date-Objekt erwartet. Aus diesem Grund ist es erforderlich, aus der Zeichenkette ein Date-Objekt

zu erzeugen. Wenn Sie dies schon einmal mit den Bordmitteln von JavaScript versucht haben, werden Sie die folgende Funktion zu schätzen wissen: `util.convertStringToDate()`. Diese erwartet als Parameter eine Zeichenkette, die das Datum repräsentiert, sowie einen zweiten String, der das Format dieses Datumsstrings definiert – im Beispiel also „`dd.mm.yyyy`“. Das Ergebnis ist ein echtes Date-Objekt, welches nun dem Mappenfeld „`Leerfeld2`“ zugewiesen wird.

Abschließend muss der Mappe noch mitgeteilt werden, dass sie die gerade gesetzten Feldwerte übernehmen soll. Weil sie sich aber schon im Bearbeitungsmodus befindet, darf dies nicht durch „`startEdit()`“ und „`commit()`“ geschehen, sondern wird über ein einfaches „`sync()`“ vorgenommen.

4.2 Aufruf bei Mappenspeicherung

Wenn eine Mappe sich im Bearbeitungsmodus befindet, kann ein Skript definiert werden, welches aufgerufen wird, sobald die Daten aus dem Formular zum Server geschickt worden sind. Bevor die Synchronisation der Daten der Arbeitskopie in die tatsächliche Mappe stattfindet, kann das Skript dann die Felddaten noch verändern bzw. auf Fehlerbedingungen reagieren.

Ein interessantes Feature stellt in diesem Zusammenhang die Möglichkeit dar, den Speichervorgang abzubrechen und den Benutzer in der Weboberfläche in der Mappe im Bearbeitungsmodus festzuhalten. Auf diese Weise lassen sich beispielsweise Constraint-Prüfungen umsetzen, mit denen etwa mehrere Mappenfelder auf syntaktisch korrekten Zusammenhang überprüft werden können oder dergleichen.

Wenn man die Mappenspeicherung per Skript gewollt abbricht, sollte man den Webanwendern eine aussagekräftige Fehlermeldung mit einer Begründung für den Abbruch liefern. Dies geschieht, indem diese Fehlermeldung kurzerhand in das Attribut `context.errorMessage` abgelegt wird. Wenn man das Skript abschließend mit einem negativen Rückgabewert beendet (dazu muss das Skript nach der Zuweisung der Fehlermeldung mit `return -1;` beendet werden), wird genau diese Fehlermeldung als `Alert()`–Fenster im Browser sichtbar gemacht.

Eine solche Constraintprüfung demonstriert das auf der nachfolgenden Seite beschriebene Skript.

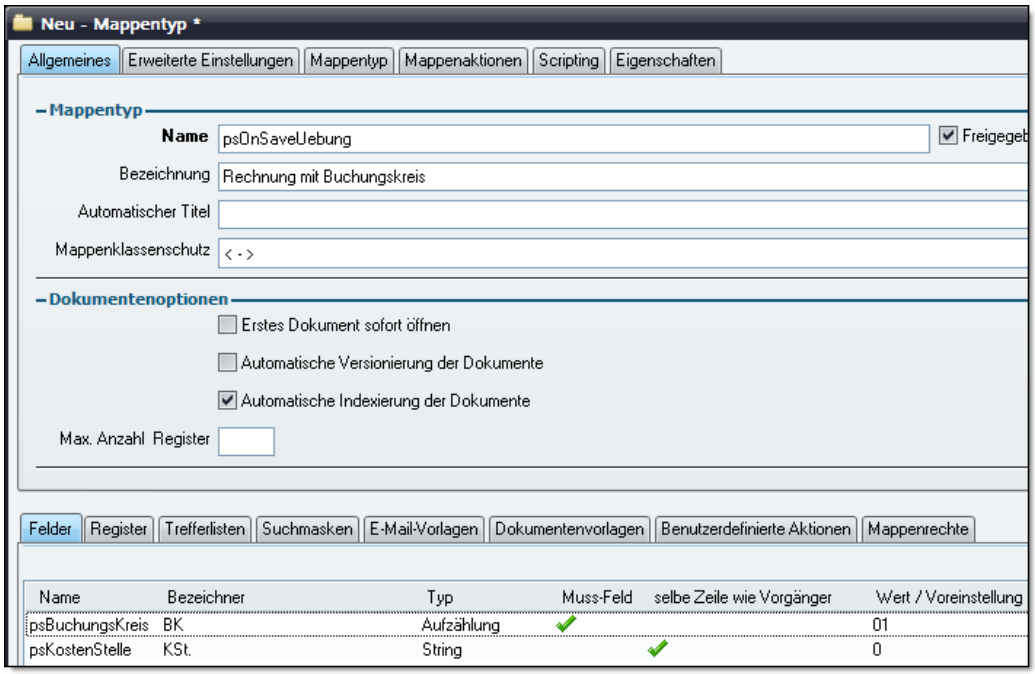
```
// Ausfuehrungskontext pruefen
var evt = context.event;
if (evt != "onSave") {
    context.errorMessage = "Falscher Scriptaufruf!";
    return -1;
}
// wir wissen nun, dass es ein "Beim Speichern"-Script ist

// aktuelle Mappe holen
var myFile = context.file;
if (!myFile) {
    context.errorMessage = "Keine Mappe!";
    return -1;
}

var buchungsKreis = myFile.psBuchungsKreis; // Klappliste mit Buchungskreis
var kostenStelle = myFile.psKostenStelle; // numerisches Feld fuer KSt.
var message = "Kostenstelle passt nicht zu Buchungskreis!";

// die Kostenstellen muessen zum selektierten Buchungskreis passen
switch (buchungsKreis) {
    case "01":
        if ((kostenStelle < 1000) || (kostenStelle > 1999)) {
            context.errorMessage = message;
            return -1;
        }
        break;
    case "02":
        if ((kostenStelle < 2000) || (kostenStelle > 2999)) {
            context.errorMessage = message;
            return -1;
        }
        break;
    case "03":
        if ((kostenStelle < 3000) || (kostenStelle > 3999)) {
            context.errorMessage = message;
            return -1;
        }
        break;
    default:
        context.errorMessage = "Noch kein Buchungskreis ausgewaehlt!";
        return -1;
        break;
}
// wenn das Script bis hier noch nicht abgebrochen wurde, ist alles in Ordnung
return 0;
```

Für eine praktische Erprobung dieses Scripts benötigen wir auch bei diesem Beispiel einen neuen Mappentypen. Legen Sie also bitte über den Client einen neuen Mappentypen namens `psOnSaveUebung` an. In diesem Mappentypen benötigen wir anschließend mindestens die aus dem folgenden Screenshot ersichtlichen Felder:



Wie Sie sehen, sollte das Buchungskreis-Feld eine Klappliste mit den drei Aufzählungswerten „01“, „02“ und „03“ sein. Wenn Sie neben der Nichtauswahl eines Feldwertes auch noch einen falschen Buchungskreis testen möchten, können Sie auch noch zusätzliche Aufzählungswerte definieren.

Auf dem Reiter Scripting hinterlegen Sie anschließend das zuvor definierte Skript mit dem vorher dargestellten Quellcode auf den „Beim Speichern“-Event:

Wie funktioniert das Skript nun?

Zum Beginn der Ausführung ermittelt das Skript zunächst seinen Ausführungskontext. Wir wollen sicherstellen, dass das Skript „Beim Speichern“ aufgerufen wird, nirgends sonst. Falls das Skript feststellt, dass es in einem anderen Kontext gestartet wurde, bricht es mit einer Fehlermeldung ab. Sie können diese Prüfung testen, indem Sie das Skript alternativ einmal „Nach dem Speichern“ einbinden.

Anschließend wird versucht die aktuelle Mappe zu holen, und es erfolgt die übliche restriktive Prüfung, ob tatsächlich ein gültiges Mappenobjekt gefunden wurde.

Wenn ja, werden nun unsere zwei Mappenfelder ausgelesen, und über die hier demonstrierte `switch()`-Anweisung kann dann, je nach selektiertem Buchungskreis, die Kostenstelle auf korrekten Wert hin überprüft werden.

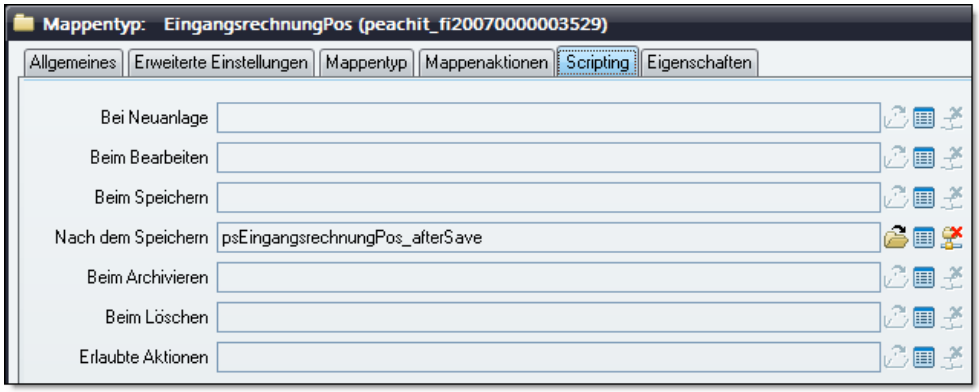
Sollte hierbei eine Kostenstellennummer außerhalb des jeweils gültigen Bereichs gefunden werden, wird die zuvor vordefinierte Fehlermeldung in `context.errorMessage` abgelegt und der Speichervorgang abgebrochen.

4.3 Aufruf nach Mappenspeicherung

Alternativ zur Ausführung vor Durchführung des Speicherungsvorgangs gibt es auch die Möglichkeit, ein Skript direkt nach dem Speichern durchzuführen. Zum Zeitpunkt des Aufrufs dieses Scripting-Events existiert keine Arbeitskopie der Mappe mehr. Das bringt es allerdings als Seiteneffekt auch mit sich, dass Sie ein solches Skript keinesfalls mit „`return -1;`“ abbrechen dürfen. Dieser Abbruch hätte nämlich zur Folge, dass zwar die Speicherung der Arbeitskopie zurück in die Mappe ausgeführt wird, der anschließend fällige Refresh des Browsers des Anwenders aber unterbleibt. In der Folge bekommt der Anwender beim nächsten Klick auf „Speichern“ oder „Abbrechen“ die (folgerichtige!) Fehlermeldung „Diese Mappe existiert nicht“!

Dennoch wird diese Sorte Skript in der Praxis gerne verwendet, um automatisiert abhängig von anderen Feldern bestimmte Feldwerte zu berechnen. Das folgende Beispiel demonstriert dies anhand einer vollautomatischen Berechnung der Mehrwertsteuer.

Definieren Sie dazu am Mappentypen `ftInvoice` (des peachit-Demomandanten) ein Skript, welches *nach Mappenspeicherung* ausgeführt werden soll. Im folgenden Beispiel wird aus dem eingegebenen Betrag (Gesamtbetrag) die MwSt berechnet.



Der Quelltext des im vorherigen Screenshot verlinkten Scripts `psftInvoice_afterSave` sieht wie folgt aus:

```
var vatRate = 0.19; // konfigurierbarer Mwst-Satz

var myFile = context.file; // aktuelle Mappe holen
if (!myFile)
{
  // einzig erlaubte Ausnahme im afterSave-Script, eine -1 zurueckzugeben!
  context.errorMessage = "Not in a file context!";
  return -1;
}

myFile.MwSt = myFile.Betrag * vatRate; // MwSt berechnen
myFile.sync(); // Mappe synchronisieren
```

Das Skript demonstriert bewusst nochmals die restriktive Programmierung. Denn trotz der vorherigen Aussage, die Rückgabe von -1 sei verboten, gibt es eine Ausnahme von dieser Regel: falls das Skript im falschen Ausführungskontext gestartet wurde, darf und muss folgerichtig eine Fehlermeldung generiert werden. In der Praxis sollten Sie auf die Sicherheitsabfragen des restriktiven Programmierens also selbstverständlich nie verzichten.

4.4 Aufruf beim Löschen

Sie können auch beim Löschen einer Mappe noch durch ein Skript Einfluss auf den Vorgang nehmen. Das folgende aus CONTRACT entlehnte Beispiel demonstriert auszugsweise, wie beim Löschen einer Firmenmappe zuerst noch eine Überprüfung erfolgt, ob der Firma nicht noch Kontakte oder gar Verträge als Vertragspartner zugewiesen sind.

```
var myFile = context.file;
if (!myFile)
{
  context.errorMessage = "No File!";
  return -1;
}

var filter = "lcmCompany|~'" + myFile.crmId + "'";
var contractFRS = new FileResultset("lcmContract", filter, "");
if (contractFRS && contractFRS.size() > 0)
{
  delete contractFRS; // free Memory!!!
  context.errorMessage = "Found contracts for this company!";
  return -1;
}

var contactFRS = new FileResultset("lcmContact", filter, "");
if (contactFRS && contactFRS.size() > 0)
{
  delete contactFRS; // free Memory!!!
  context.errorMessage = "Found contacts for this company!";
  return -1;
}
```

Zunächst erfolgt im Code natürlich die inzwischen geläufige Abfrage auf den Ausführungskontext, ob überhaupt ein gültiges Mappenobjekt vorliegt.

Anschließend erstellt das Skript auf Basis des Inhalts des Feldes `crmId` je ein `FileResultset` auf Vertragsmappen bzw. auf Kontaktmappen. Dabei stellt `lcmCompany` in beiden Mappentypen jeweils ein Referenzfeld in erweiterter Syntax dar, dessen Schlüsselbedingung auf das Feld `crmId` der Firma konfiguriert ist.

Um sicherzugehen, dass die aktuelle Firmenmappe gefahrlos gelöscht werden darf, müssen beide `FileResultsets` leer sein (anderenfalls würde das Löschen der Firmenmappe zu einer Verletzung der Datenintegrität führen, da ja die Referenzen der gefundenen Mappen durch das Löschen der Firmenmappe zerstört würden).

Das Beispiel veranschaulicht darüber hinausgehend noch die Verwendung des Befehls `delete` aus dem normalen Javascript-Sprachumfang. Dieser dient dazu, den von komplexen Objekten belegten Speicherbereich wieder freizugeben. In der Scripting-Engine des DOCUMENTS Servers kann man dies nutzen, um den integrierten Garbage Collector zu beeinflussen.

4.5 Zugriff auf das Dateisystem von DOCUMENTS

Der Zugriff auf das Dateisystem des Rechners, auf dem der Server läuft, stellt eine gleichermaßen regelmäßige wie unter Umständen für die Systemstabilität gefährliche Anforderung dar.

Insbesondere wird die in der Scripting-Engine integrierte Klasse File gerne zur Umsetzung selbstdefinierter Logfiles genutzt. Da aber der Dateisystemzugriff über diese File-Klasse nicht threadsafe ist, erfordert dies erheblichen Zusatzaufwand zur Gewährleistung eines exklusiven Zugriffs auf eine bestimmte Datei im Dateisystem.

Im Normalfall gewährleistet man dies etwa dadurch, je Skriptausführung mit einem eindeutigen, vom Server selbst vergebenen temporären Dateinamen zu arbeiten (dieser Ansatz wird auch im folgenden Beispielcode verwendet). Alternativ könnte man, etwa bei Jobs, zu Beginn der Skriptausführung am Mandanten eine Eigenschaft suchen, die, falls nicht gefunden, sofort angelegt wird, und bei Existenz umgekehrt dazu führt, dass das eigentliche Skript nicht ausgeführt wird, bis die bewusste Eigenschaft vom anderen Skript wieder gelöscht wurde.

```
// Tempfile erzeugen
var tempFileName = context.getTmpFilePath();
var fileHandle = new File(tempFileName, "a+t");
if (!fileHandle || !fileHandle.ok())
{
  context.errorMessage = "Could not open temporary file!";
  return -1;
}

// sicherstellen, dass geoeffnete Datei leer ist
var buffer = fileHandle.read(1);
if (fileHandle.eof())
{
  util.out("Created a new temporary file");
} else {
  util.out("Opened an existing temporary file");
}

// Schreibzugriff versuchen
var success = fileHandle.write("Hello World!");
if (!success)
{
  context.errorMessage = "Error writing data to fileHandle";
  fileHandle.close();
  return -1;
}
fileHandle.close(); // Datei schliessen

// zuletzt einige Hilfsfunktionen aus der util-Klasse
util.out(util.fileSize(tempFileName)); // sollte 12 Byte sein
var tempPath = util.getTmpPath(); // temp. Pfad holen
util.out(tempPath);
tempPath += "\\hello\\world";
util.out(util.makeFullDir(tempPath));
var newTempName = tempPath + "\\HelloWorld.txt";
util.out(util.fileMove(tempFileName, newTempName));
util.unlinkFile(newTempName);
```

4.6 Aufzählungswerte dynamisch ermitteln

Wenn die Ausprägungen eines Feldes vom Typ „Aufzählung“ nicht konstant sind, können Sie diese auch durch ein Skript definieren. Die Festlegung, welches Skript ausgeführt werden soll, wird dazu durch die Vorschrift `runscript:Scriptname` als Aufzählungswert festgelegt.

In der Projektpraxis nutzt man dies gerne, um ein in gleichzeitig in verschiedenen Mappentypen erforderliches Aufzählungsfeld mit den immer gleichen Aufzählungswerten zentral an einer Stelle zu pflegen. Die Umsetzung als Skript hat in diesem Zusammenhang insbesondere den Vorteil, dass nachträgliche Anpassungen an den Aufzählungswerten automatisch auch für schon bestehende Mappen in DOCUMENTS gültig werden, also kein „Bestehende Mappen ändern“-Lauf erforderlich wird. (Umgekehrt bringt dies den Nachteil mit sich, solche Änderungen automatisch für alle Mappen zu verwenden, obwohl dies ausnahmsweise gar nicht gewünscht ist).

Dynamisch per Skript generierte Klapplistenfelder sollten auf keinen Fall in Trefferlisten angezeigt werden, denn aufgrund der Art und Weise, wie DOCUMENTS intern mit

mehrsprachigen Aufzählungswerten umgeht, führt schon der Aufruf einer erfolglosen Suche oder die Anzeige eines leeren dynamischen Ordners automatisch zur Ausführung des Aufzählungsskripts (nur so kann der server ermitteln, welchen ergonomischen Bezeichnern die technischen Feldwerte des Aufzählungsfeldes entsprechen).

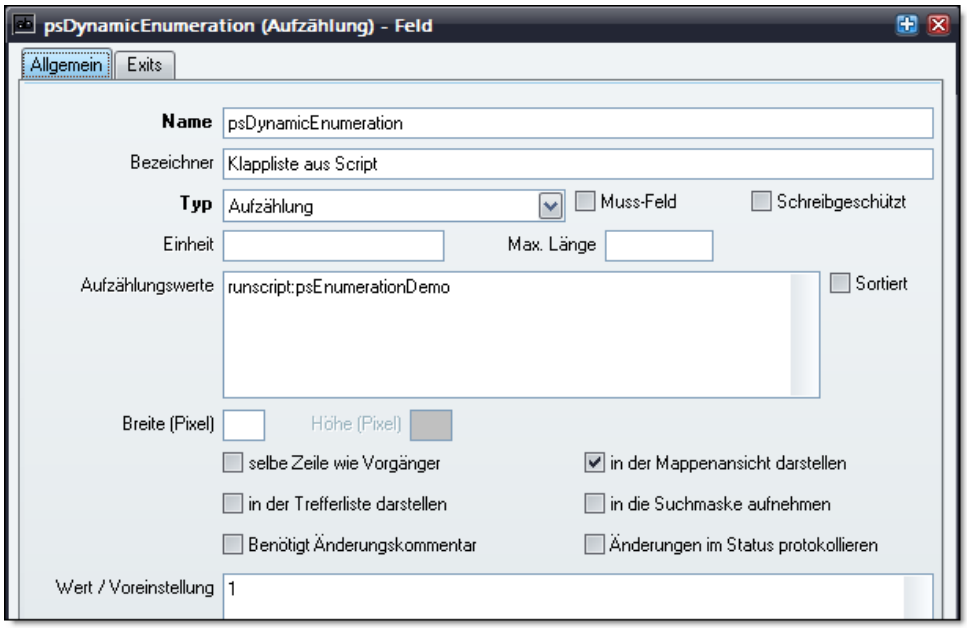
Damit handelt es sich bei einem dynamisch aufgebauten Aufzählungsfeld also automatisch um eine potenziell sehr teure Ressource!

Als Folge der dynamischen Ermittlung der Aufzählungswerte ergibt sich weiterhin, dass insbesondere eine Ermittlung der Aufzählungswerte als Resultat von Datenbankabfragen externer Datenquellen sehr ressourcenintensiv werden kann. Wir raten dringend davon ab, Aufzählungsfelder mit mehr als 20 bis ca. 30 Aufzählungswerten dynamisch per Skript aus Datenbankabfragen umzusetzen. Stattdessen wäre in solchen Fällen der Einsatz des Tabledata-Userexits (siehe hierzu die Dokumentation zur sogenannten SQL-Taglib) ratsam, der neben dem schonenden Umgang mit den Ressourcen erheblichen Komfortzuwachs gegenüber einer Klappliste mit sich bringt, insbesondere für den DOCUMENTS-Anwender.

Eine weitere wesentliche Einschränkung bei Aufzählungsskripten besteht darin, dass Sie in solchen Scripts auf keinen Fall einen eigenen Rückgabewert per `return`-Anweisung zurückgeben dürfen, da das im Ausführungskontext automatisch und implizit vorhandene Array namens `enumval` bei Skriptende ebenso automatisch als Rückgabewert genutzt wird. Das folgende kleine Beispielskript demonstriert die Nutzung von `enumval` für Aufzählungswerte in der Praxis. Dabei werden die Aufzählungswerte bewusst mehrsprachig konfiguriert:

```
1  // Dieses Beispiel demonstriert die zentrale Befuellung der Aufzaehlungswerte
2  // eines Klapplistenfelds per Script. Das Beispiel demonstriert dabei auch die Mehrsprachigkeit
3
4  enumval.push("1;de:Eins;en:One;");
5  enumval.push("2;de:Zwei;en:Two;");
6  enumval.push("3;de:Drei;en:Three;");
7  enumval.push("4;de:Vier;en:Four;");
8  enumval.push("5;de:Fuenf;en:Five;");
9
10 // In diesen enumval-Scripts ist es strikt verboten, einen return-Wert zurueckzugeben
11 // das enumval-Array stellt selbst und implizit den return-Wert dar!
```

Einbindung dieses Skripts in ein Mappenfeld (es wird davon ausgegangen, dass das obige Skript als `psEnumerationDemo` gespeichert wurde):



4.7 Datenbankzugriffe via Scripting

Die Scripting-Engine stellt mit den Klassen `DBConnection` und `DBResultset` grundlegende Unterstützung zum Zugriff auf (externe) relationale Datenbanken zur Verfügung.

Das nachfolgende kleine Beispiel demonstriert die wesentlichen Mechanismen, eine Verbindung zu einer ODBC-Datenquelle aufzubauen, um in einer Datenbanktabelle zu Protokollierungszwecken einen neuen Eintrag per INSERT-Befehl einzutragen.

Die (fiktive) Protokolltabelle namens „`ProtocolTable`“ habe dabei die Spalten `login`, `tstamp`, `role`, `wfstep`, `filekey` und `filetitle` (alles Stringfelder):

```
// the usual stuff - accessing the current DocFile object
var myFile = context.file;
if (!myFile)
{
  context.errorMessage = "Not in a file context!";
  return -1;
}

// try to connect
var myDB = new DBConnection("odbc", "ProtocolDB", "aUser", "aPwd");
if (myDB)
{
  var lastError = myDB.getLastError();
  if (lastError == null) // connection is OK
  {
    // collect data to export to protocol table
    var loginCol     = context.currentUser;          // login of current user
    var fmt          = "dd.mm.yyyy HH:MM";
    var tStampCol    = util.convertDateToString(new Date(), fmt);
    var roleCol      = "Directors";                  // Access Profile
    var wfStepCol    = "Directors signature";        // WorkFlowAction label
    var fileKeyCol   = myFile.getAutoText("id");     // current file's unique ID
    var fileTitleCol = myFile.getAutoText("title"); // current file's title

    var queryString = "INSERT INTO ProtocolTable ";
    queryString    += "(login,tstamp,role,wfstep,filekey,filetitle) ";
    queryString    += "VALUES ('" + loginCol +
                      "','" + tStampCol +
                      "','" + roleCol +
                      "','" + wfStepCol +
                      "','" + fileKeyCol +
                      "','" + fileTitleCol + "')";
    util.out("INSERT-QUERY: " + queryString); // DEBUG OUTPUT
    var success = myDB.executeStatement(queryString);
    if (!success)
    {
      context.errorMessage = "Error in INSERT: " + myDB.getLastError();
      return -1;
    }
    myDB.close(); // IMPORTANT! Close DB handle!
  }
} else {
  context.errorMessage = "Could not connect to database!";
  return -1;
}
```

Zu den wesentlichen Aspekten beim Datenbank-Handling gehören ein konsequentes Errorhandling sowie die saubere Rückgabe geöffneter Datenbank-Handle, da es sich hierbei im Regelfall um sehr teure Ressourcen handelt.

Das nachfolgende BeispielSkript dient als Signaleingang einer Mappe in einem Workflow. Im Beispiel handelt es sich um eine Eingangsrechnung, deren Belegnummer (z.B. ein Barcode) in einer Datenbanktabelle als Primärschlüssel dient. Ein (fiktives) ERP-System soll dort unserer Annahme zufolge eine Spalte „`BookingState`“ beeinflussen. Wenn die Rechnung im ERP-System verbucht worden ist, soll dieses System den Buchungsstatus der Rechnung in der Tabelle auf den Wert „`booked`“ setzen. Das nachfolgende Skript signalisiert dem Workflow dann, dass die Eingangsrechnungs-Mappe den Workflow weiter durchlaufen darf, anderenfalls muss der Vorgang weiter im Wartezustand verharren.

```
// the usual stuff - accessing the current DocFile object
var myFile = context.file;
if (!myFile)
{
  context.errorMessage = "Not in a file context!";
  return -1;
}

var returnCode = 0; // assume to stay in the waitstate shape

// try to connect
var myDB = new DBConnection("odbc", "ErpDB", "aUser", "aPwd");
if (myDB)
{
  var lastError = myDB.getLastError();
  if (lastError == null) // connection is OK
  {
    var fileKeyCol   = myFile.DocumentID;     // unique Document ID of invoice
    var queryString = "SELECT DocumentNo,BookingState FROM BookingTable ";
    queryString    += "WHERE DocumentNo='" + fileKeyCol + "'";

    util.out("SELECT-QUERY: " + queryString); // DEBUG OUTPUT

    var myRS = myDB.executeQuery(queryString);
    if (myRS && myRS.next())
    {
      // 1st column contains the DocumentNo, 2nd contains BookingState
      var bookingState = myRS.getString(1);
      if (bookingState == "booked")
      {
        returnCode = 1;
      }
      myRS.close(); // IMPORTANT! Close RS handle!
    }
    myDB.close(); // IMPORTANT! Close DB handle!
  }
}

return returnCode;
```

Der obige Beispielcode demonstriert unter anderem, dass auch DBResultsets so früh wie möglich wieder geschlossen werden sollten. Außerdem wird hier die Spalte DocumentNo aus der Tabelle mit selektiert, obwohl deren Wert anschließend bei der Verarbeitung des Ergebnisses nicht weiter berücksichtigt wird. Das trägt dem Umstand Rechnung, dass es einige RDBMS am Markt gibt, die zwingend erwarten, dass die Schlüsselspalten der WHERE-Klausel Bestandteil der Selektion sein müssen. Bei modernen Versionen von MSSQL, MySQL oder Oracle sollte dieser Trick üblicherweise nicht notwendig sein.

4.8 Caching der Daten teurer Ressourcen

Es kommt insbesondere bei Projekten mit Zugriffen auf externe Datenbanken sehr regelmäßig vor, dass – beispielsweise als Aufzählungsskript – stets dieselben Informationen immer wieder aus den angebundenen Datenbanktabellen gelesen werden, obwohl sich diese nur sporadisch ändern. Dennoch bedeutet dieser regelmäßige Zugriff auf die teure Ressource „Datenbank“ eine nicht unerhebliche Belastung des Servers.

Folglich wünscht man sich eine Möglichkeit, diese Daten irgendwie im Speicher des DOCUMENTS-Servers zu puffern.

Genau diese Möglichkeit besteht über den in Scripts verfügbaren PropertyCache. Dieser ist unter dem Namen `propCache` stets und implizit ohne weitere Instanzierungen verfügbar und wird üblicherweise für genau diese Pufferung verwendet.

Das nachfolgende Beispiel ermittelt einige Aufzählungswerte aus der Nordwind-Datenbank, sofern der Cache namens „`Contacts`“ noch nicht angelegt sein sollte:

```
var enums = propCache.Contacts;

if (!enums)
{
  util.out("Requiring data from DB, no cache found!");
  enums = new Array();
  var myDB = new DBConnection("odbc", "Nordwind", "", "");
  if (!myDB || myDB.getLastError() != null)
  {
    enums.push("DBConnect-Error");
  } else {
    var myRS = myDB.executeQuery("SELECT Nachname, Vorname FROM Personal");
    if (myRS)
    {
      while (myRS.next())
      {
        enums.push(myRS.getString(0) + ", " + myRS.getString(1));
      }
      myRS.close();
      delete myRS; // free memory
    } else {
      enums.push("DBResult-Error");
    }
    myDB.close();
    delete myDB; // free memory
  }
  propCache.Contacts = enums;
} else {
  util.out("Found cached data");
}
for (var i = 0; i < enums.length; i++)
{
  enumval.push(enums[i]);
}
```

Übrigens: Um ein Aufzählungsskript im **DOCUMENTS-Manager** testen zu können, ist es erforderlich, auf dem „Testen“-Reiter die Checkbox „Skript für Aufzählungswerte“ zu aktivieren.

Über ein täglich ausgeführtes Jobskript (nachfolgender Einzeiler) kann sichergestellt werden, dass der Cache jeden Tag neu befüllt wird:

```
propCache.removeProperty("Contacts");
```

4.9 Benutzerdefinierte Aktionen an Mappen

Mittlerweile zählt es zum Projektalltag im Umfeld von DOCUMENTS, den Funktionsumfang des Systems im Kontext einer einzelnen Mappe um zusätzliche Funktionalität zu ergänzen. Eine gängige Möglichkeit besteht in der Definition einer sogenannten benutzerdefinierten Aktion an einem Mappentypen. Dies stellt die so definierte Funktion dann jeder Mappe zur Verfügung, die auf Basis dieses Mappentyps erstellt worden ist.

Die Einbindung eines Skripts muss dann auf dem Reiter „Benutzerdefinierte Aktionen“ eines Mappentyps erfolgen. Sie haben die Wahl, ob Sie die Aktion mit einem von Ihnen zu definierenden Namen und Label als eigenen Button (Layout wie Workflow-Aktion) oder als Eintrag in der Aktionen-Klappliste wünschen.

4.10 Benutzerdefinierte Aktionen an Ordnern

Eine gängige Anforderung im DOCUMENTS-Umfeld liegt darin, aus einem Ordner heraus eine Auswahl an Vorgängen (lies: Mappen) zu selektieren und die darin gespeicherten Informationen in irgendeiner Form zu exportieren.

Das nachfolgende Beispielskript ist der aktuellen Weiterentwicklung von CONTRACT entnommen und demonstriert den Export von Fristen im iCal-Format, um die Fristendaten beispielsweise in Outlook oder einem Desktop-Kalender-Programm weiterverwenden zu können:

```
var terms = context.selectedFiles;
if (!terms || terms.size() <= 0)
{
    context.errorMessage = "No terms selected";
    return -1;
}

var completeArr = new Array(
    "BEGIN:VCALENDAR\r\n"
  + "VERSION:2.0\r\n"
  + "PRODID:CONTRACT\r\n"
  + "METHOD:PUBLISH\r\n"
);

for (var myFile = terms.first(); myFile; myFile = terms.next())
{
  var sId     = myFile.getAutoText("id") + "@contract";
  var sName   = "CONTRACT Verwaltung";
  var sEmail  = context.getPrincipalAttribute(
                  "eMailDefaultSender"
                  );
  var sDate   = util.convertDateToString(
                  myFile.lcmTerm, "yyyymmdd"
                  );
  var sNow    = util.convertDateToString(
                  new Date(), "yyyymmddTHHMM"
                  ) + "00Z";
  var sDesc   = myFile.lcmDescription.replace(/\r\n/g, "\\n ");
  sDesc       = myFile.lcmTermDescription + "\\n " + sDesc;
  var sBody = "BEGIN:VEVENT\r\n"
      + "UID:" +  sId + "\r\n"
      + "ORGANIZER;CN=\"CONTRACT\":MAILTO:" + sEmail + "\r\n"
      + "SUMMARY:" + myFile.getAutoText("title") + "\r\n"
      + "DESCRIPTION:" + sDesc + "\r\n"
      + "CLASS:PUBLIC\r\n"
      + "DTSTART;VALUE=DATE:" + sDate + "\r\n"
      + "DTSTAMP:" + sNow + "\r\n"
      + "END:VEVENT\r\n";
  completeArr.push(sBody);
}
context.returnType = "file:terms.ics";
return util.transcode("windows-1252", completeArr.join(""), "UTF-8");
```

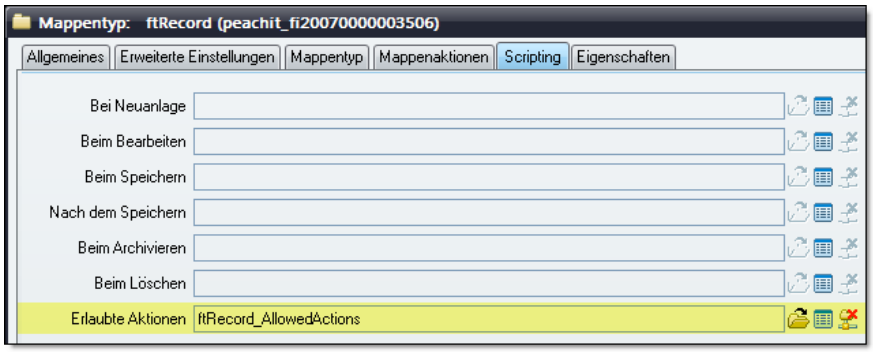
Das Skript ist als benutzerdefinierte Aktion am Fristenkalender-Ordner in CONTRACT hinterlegt worden.

4.11 Verrechtung benutzerdefinierter Aktionen

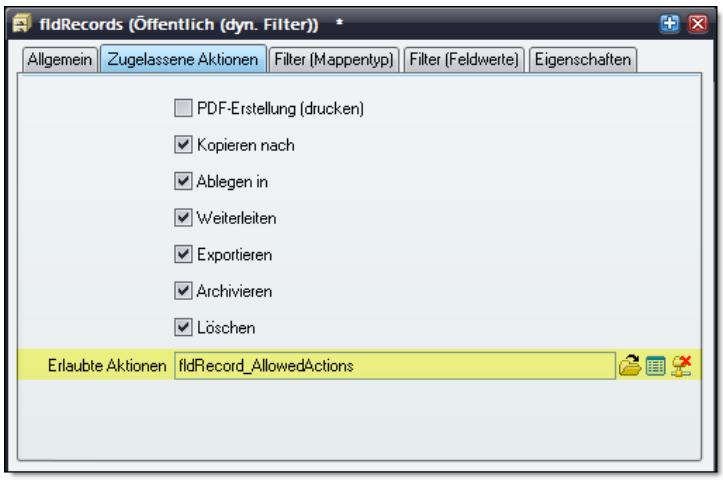
Nachdem wir nun in den vorherigen beiden Abschnitten kennengelernt haben, wie man den Funktionsumfang der Weboberfläche durch benutzerdefinierte Aktionen erweitern kann, stellt sich unmittelbar die Folgefrage, wie man dafür Sorge trägt, dass bestimmte benutzerdefinierte Aktionen auch nur für bestimmte Benutzerkreise zur Verfügung stehen – es wird also faktisch eine Möglichkeit zur Verrechtung der von uns selbst angelegten Buttons bzw. Aktionsklapplisten-Einträge gesucht.

Eine solche Verrechtung ist sowohl am Mappentypen als auch an öffentlichen Ordnern unter Zuhilfenahme eines speziellen Skripts möglich, das als „Erlaubte Aktionen“-Skript hinterlegt werden muss.

Einsatz am Mappentypen:



Einsatz am öffentlichen Ordner:



Ein sogenanntes `AllowedActions`-Skript hat dabei stets einen identischen Aufbau, der daraus resultiert, dass Ihnen die Scripting-Engine automatisch ein Array mit den technischen Namen sämtlicher von Ihnen am Mappentypen bzw. Ordner hinterlegten benutzerdefinierten Aktionen zur Verfügung stellt.

Dieses Array namens `enumval` müssen Sie in einer `for()`–Schleife durchlaufen und den Wert jedes Eintrags mit dem von Ihnen gesuchten Aktionsnamen vergleichen.

Sobald Sie die richtige Aktion gefunden haben, überprüfen Sie dann die jeweilige von Ihnen gewünschte Berechtigung. Sofern diese Prüfung ergibt, dass die Aktion vom Anwender nicht genutzt werden können soll, muss lediglich der aktuelle Eintrag aus `enumval` entfernt werden.

Ein einfaches Beispielskript, welches genau diesen generellen Aufbau demonstriert, könnte etwa wie folgt aussehen:

```
1  // check all userdefined actions
2  for (var i = 0; i < enumval.length; i++)
3  {
4    // show/hide approval-button
5    if (enumval[i] == "btnApproval")
6    {
7      // hide Button, if current user is not member of privileged group
8      var currentUser = context.getSystemUser();
9      if (!currentUser.hasAccessProfile("SUPERADMINS")
10       enumval[i] = "";
11   }
12 }
```

Bitte beachten Sie, dass hier prinzipiell immer zuerst durch die Schleife iteriert wird, innerhalb der Schleife dann die gewünschte Aktion gesucht wird und erst innerhalb der gesuchten Aktion dann die Rechteprüfung erfolgt.

Dieser generelle Aufbau hat unter anderem den Vorteil, dass man potentiell „teure“ Rechteprüfungen tatsächlich nur dann ausführt, wenn die auf diese Weise zu verrechtende Aktion auch wirklich vorhanden ist.

Eine weitere Besonderheit stellt der Umstand dar, dass es im Skript selbst keine `return`-Anweisung gibt. Das `enumval`-Array stellt nämlich wie bei Aufzählungswerte-Scripts implizit den Rückgabewert dar, die eigenhändige Angabe einer `return`-Anweisung ist daher strikt verboten.

Es muss nicht zwischen Aktionsbuttons und –klapplisteneinträgen unterschieden werden. Für den Fall, dass am Mappentypen bzw. Ordner sowohl das eine als auch das andere definiert worden ist, wird das `AllowedAction`-Skript für jede Aktionsart je einmal ausgeführt. Sorgen Sie also bitte lediglich dafür, dass Ihre benutzerdefinierten Aktionen stets eindeutige und programmier-sprachentaugliche technische Namen erhalten!

4.12 Skript als Job ausführen

Ein häufiges Einsatzgebiet von Skripten dürfte die Automatisierung von regelmäßig wiederkehrenden Aufgaben sein, die ohne Benutzerinteraktion erfolgen, beispielsweise eine automatische Archivierung von Vorgängen. Diese Art Aufgabe lässt sich sehr einfach in Form sogenannter Job-Skript realisieren. Das folgende Beispiel demonstriert anhand der automatisierten Archivierung veralteter Vorgänge, wie ein solches Job-Skript realisiert wird.

Definieren Sie hierzu ein Skript mit dem folgenden Quellcode:

```
// vom Tagesdatum 90 Tage abziehen
var toDate = new Date();     // aktuelles Datum auslesen
toDate = context.addTimeIntervall(toDate, -90, 'days');
var strDate = util.convertDateToString(toDate, 'dd.mm.yyyy');

// Filtere nun die Mappen gemaess der definierten Kriterien
var filter = "Verfallsdatum<'" + strDate + "'";   // Filter setzen
var myFRS = new FileResultset("Standard", filter, "");
if (myFRS && myFRS.size() > 0)
{
  // Iteriere durch die Ergebnismenge und archiviere und
  for (var myFile = myFRS.first(); myFile; myFile = myFRS.next())
  {
    var success = myFile.archiveAndDelete();
    if (!success)    // Fehlermeldung und Abbruch im Fehlerfall
    {
      var msg = "Fehler beim Archivieren und Loeschen";
      msg += " der Mappe " + myFile.getAutoText("id");
      msg += ": " + myFile.getLastError();
      context.errorMessage = msg;
      return -1;
    }
  }
}
return 0;
```

Dieses Skript demonstriert nochmals die Funktionen zur Datumsmanipulation, und wie man Mappen anhand von Datums-Vergleichsoperationen filtern kann. Unser Beispiel filtert alle Mappen heraus, die ein Mappenfeld namens „Verfallsdatum“ beinhalten und dessen Wert mehr als 90 Tage älter ist als das aktuelle Tagesdatum. Alle durch diesen Filter gefundenen Mappen werden nun der Reihe nach archiviert und aus dem System entfernt. Sollte hierbei ein Fehler auftreten, so bricht das Skript mit einer Fehlermeldung ab, die wiedergibt, welche Mappe aus welchem Grund zum Abbruch des Jobs geführt hat.

Das Skript erfordert zwingend einen Benutzerkontext, da anderenfalls keinerlei Zugriff auf DOCUMENTS-Mappen besteht. Im Normalfall hinterlegen Sie dazu in den DOCUMENTS-Einstellungen einen passenden Jobuser (dieser muss selbstverständlich über ausreichende Berechtigungen an allen Mappen verfügen, auf die Sie jobgesteuert Einfluss nehmen möchten).

Alternativ können Sie diesen Jobuser auch für ein einzelnes Jobskript überschreiben, indem Sie den gewünschten Loginnamen auf dem Reiter „Testen“ des jeweiligen Scripts hinterlegen.

Wenn Sie das Beispiel nachvollziehen möchten, fügen Sie bitte einem Ihrer Mappentypen ein Datumsfeld namens „Verfallsdatum“ hinzu und erstellen einige neue Mappen, deren Verfallsdatum teilweise länger als 90 Tage zurückliegt. Wenn Sie dem Verfallsdatum als Wert/Voreinstellung den Autotext „`%currentDate - 91%`“ zuweisen, können Sie sich diese Aufgabe wesentlich erleichtern.

Passen Sie im Skriptcode bitte Zeile 8 dahingehend an, auf dem von Ihnen gewählten Mappentypen zu filtern (das Beispiel bezieht sich auf den Mappentypen „Standard“ des alten toastup-Demomandanten). Anschließend können Sie das Job-Skript testen.

4.13 Mappenpool per Job-Skript gefüllt halten

Der Mappenpool ist ein spezieller Zwischenpuffer in DOCUMENTS, in dem von allen im System verwendeten Mappentypen gelöschte Mappen und verworfene Arbeitskopien zwischengelagert werden. Wird nun eine neue Mappe eines bestimmten Mappentyps erzeugt, wird seitens DOCUMENTS bevorzugt eine Mappe aus dem Mappenpool entnommen, da dies deutlich schneller vonstattengeht als die komplette Neuanlage eines Mappenobjekts auf der Datenbank.

Der Mappenpool wird allerdings auch durch automatisierte Mappenerstellung wie etwa durch Docimport oder die DOCUMENTS Factory beansprucht, und dies kann dazu führen, dass der Mappenpool nahezu dauerhaft leer ist, wenn durch diese automatisierten Importe täglich sehr viele neue Mappen generiert werden. Die Benutzer spüren dies in einer deutlich zäheren Handhabung des Systems, dass der Mappenpool leer ist, da neue Mappen und Arbeitskopien so zwangsweise direkt auf der Datenbank erzeugt werden müssen.

Aus diesem Grund gibt es zwei Methoden namens `countPoolFiles()` und `createPoolFile()`, mit denen es einem Job-Skript ermöglicht wird, nächtlich, wenn die Serverlast gering ist, den Mappenpool automatisch neu zu füllen.

Das Skript hierzu sieht wie folgt aus:

```
1  var fileType = "Standard";                          // Mappentyp
2  var poolSize = context.countPoolFiles(fileType); // Mappenpoolgroesse
3  for (var i = poolSize; i < 3000; i++)
4    context.createPoolFile(fileType);
```

Das Skript schaut für den Mappentypen „Standard“ nach, wie groß der Mappenpool zur Laufzeit ist, um anschließend die Differenzmenge bis zu einem Schwellwert (hier 3000) aufzufüllen. Dieser Schwellwert ist natürlich abhängig von der durchschnittlich pro Tag erzeugten Menge neuer Mappen jedes einzelnen Mappentyps und sollte individuell angepasst werden.

4.14 Entscheidungen und Wächter im Workflow

Oft wird es Ihnen beim Design von Workflows passieren, dass die standardmäßig verfügbaren Vergleichsoperatoren für die Erstellung einer Bedingung (sogenannte Guards) nicht mehr ausreichen, zum Beispiel, wenn zur Prüfung Berechnungen notwendig sind oder komplex verknüpfte boolsche Algebra notwendig ist. Auch in diesem Fall helfen Scripts weiter.

Guard-Scripts können innerhalb der Workflow-Engine in unterschiedlicher Funktion eingesetzt werden. Zum einen dienen sie als Bedingung an einem Entscheidungsshape, zum anderen können sie (ohne dafür auch nur eine einzige Zeile Skriptcode anpassen zu müssen)

als sogenannte Constraintprüfungen auf Kontrollflüssen eingesetzt werden, die aus einer Aktion (Aufgabe für Benutzer bzw. Gruppen) hinauslaufen.

Guard-Scripts werden prinzipiell im Kontext einer Mappe ausgeführt (die per `context.file` zur Verfügung steht). Der das Skript ausführende Benutzer ist im Fall eines Constraint-Skripts stets der Anwender, der diesen Kontrollfluss in der Weboberfläche selektiert hat.

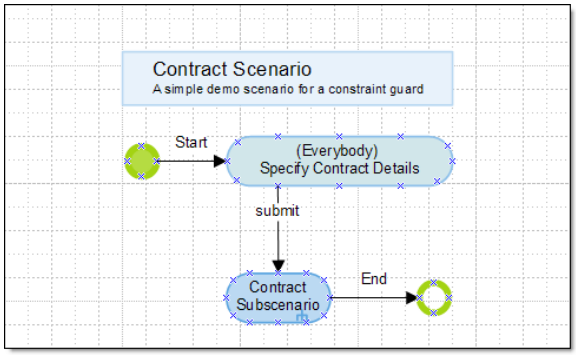
Im Fall einer Bedingung an einer Entscheidung findet die Skriptausführung üblicherweise im Kontext des Benutzers statt, der den letzten Mausklick mit der Mappe getätigt hat, also die letzte manuelle Weiterleitung der Mappe angestoßen hat. Eine Ausnahme davon bildet allerdings die Situation, dass die Mappe von einem Delay-Shape oder durch einen Signaleingang aufgehalten wurde, bevor es zur Überprüfung der Entscheidung kommt. In diesem Fall wird der Guard im Kontext des Jobusers ausgeführt.

Eine sehr gängige Anforderung für ein Constraint-Skript stellt, etwa in Vertragsmanagement-Szenarien, eine Prüfung dar, ob der Vertragsmappe schon Dokumente hinzugefügt wurden. Eine Einreichung des Vorgangs in den Freigabeprozess soll unterbunden werden, sofern noch keine Dokumente in der Mappe vorhanden sind.

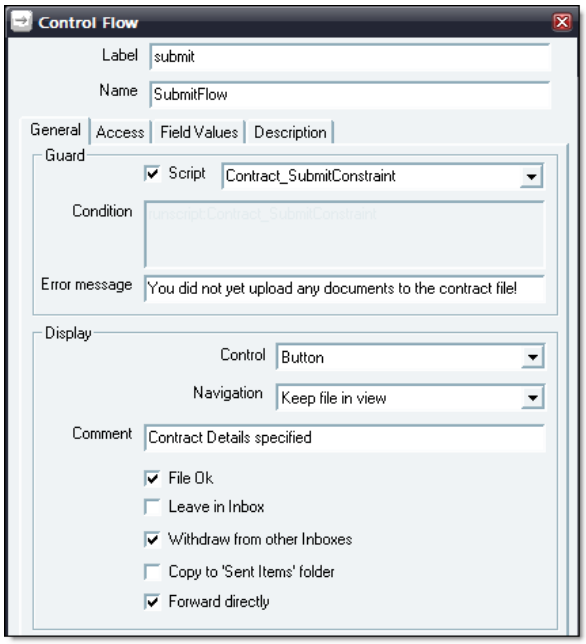
Ein solches Skript könnte beispielsweise folgende Struktur haben:

```
var myFile = context.file;
if (!myFile)
{
  context.errorMessage = "No File!";
  return -1;
}
// get all DocumentTabs
var docRegs = myFile.getRegisters("documents");
if (docRegs && docRegs.size() > 0)
{
  for (var reg = docRegs.first(); reg; reg = docRegs.next())
  {
    var docs = reg.getDocuments(); // get DocumentIterator
    if (docs && docs.size() > 0)
    {
      return 1;
    }
  }
}
return 0;
```

Dieses Skript wird im Server beispielsweise unter dem Namen „`Contract_SubmitConstraint`“ abgelegt, um es anschließend in Visio in einen Workflow einbinden zu können. Nehmen wir an, es kommt der folgende einfach strukturierte Workflow zum Einsatz:



Der Einfachheit halber nehmen wir an, das eigentliche Freigabeszenario des Vertrags ist in dem Subworkflow gekapselt. Für unser Skript ist nur die eingezeichnete Aufgabe sowie der von ihr ausgehende Kontrollfluss mit dem Label „`submit`“ relevant.



Wie Sie im obigen Screenshot sehen, ist unser Skript als Guard konfiguriert worden. Da es sich um eine einfache Prüfung handelt, wurde im Beispiel die Fehlermeldung, die bei Fehlen eines Dokuments in der Mappe ausgegeben wird, im Kontrollfluss selbst hartverdrahtet. Alternativ könnten Sie auch eine eigene Fehlermeldung in `context.errorMessage` ablegen, bevor Sie das Skript mit `return 0;` beenden.

Dieses einfache Praxisbeispiel veranschaulicht außerdem, dass Guards und Constraints nur zwei spezielle Rückgabewerte gestatten – mit `return 0;` gilt die Bedingung als nicht eingetroffen, mit `return 1;` gilt sie als erfüllt.

4.15 Signaleingänge im Workflow

In der Projektpraxis ergibt sich bei vielen Workflow-Szenarien die Notwendigkeit, die im Workflow befindlichen Mappen an einem bestimmten Punkt auf den Eintritt eines externen Ereignisses warten zu lassen. Sehr häufig trifft dies etwa auf Eingangsrechnungs-Szenarien zu, in denen es zu den Kundenanforderungen gehört, im Workflow auf den Abschluss der Verbuchung einer Rechnung im ERP-System des Kunden zu warten.

Dieser Wartezustand wird im Workflow üblicherweise in Form von Signaleingängen modelliert. Die syntaktischen Beschränkungen der definierbaren Bedingungen machen es allerdings oft erforderlich, die Signalbedingung nicht mehr inline zu definieren, sondern als eigenständiges Skript.

Ein solches Signaleingangs-Skript hat IMMER über `context.file` Zugriff auf die im Wartezustand befindliche Mappe. Der weitere Skriptcode ist vollkommen wahlfrei, lediglich die erlaubten Rückgabewerte beschränken sich auf die zwei Integer-Werte 0 (Bedingung ist

bisher nicht eingetroffen) und 1 (Die Signalbedingung wurde erfüllt, der Workflow darf weiter durchlaufen werden).

Ein sehr einfaches Beispiel erzwingt etwa, dass der Workflow an der Mappe ausschließlich an Freitagen weiterlaufen darf:

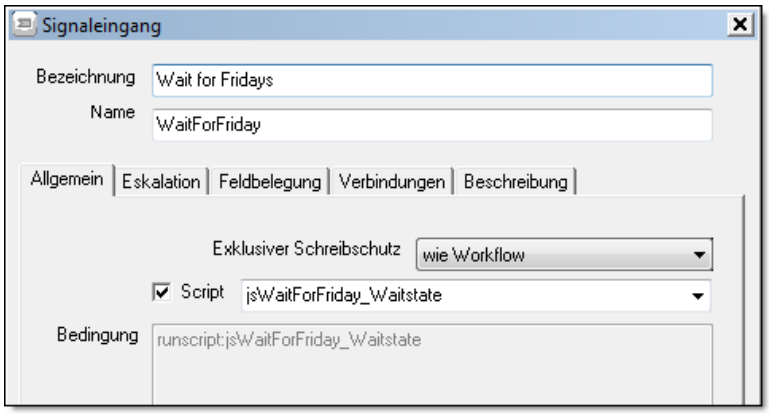
```
var today = new Date();

var result = 0;

// true if currentDate is Friday
// 0 is Sunday, 6 is Saturday
if (today)
{
    result = (today.getDay() == 5) ? 1 : 0;
}
return result;

```

Die Einbindung dieses kleinen Skripts im Workflow könnte dann etwa wie folgt aussehen:



4.16 Signalausgänge im Workflow

Oft finden sich in Workflow-gestützten DOCUMENTS-Szenarien Anforderungen in der Art, dass die in den Mappen gespeicherten Daten in irgendeiner Form an Software von Drittanbietern exportiert werden muss. Je nachdem, welche Schnittstellen die Fremdsoftware hierzu bietet, können diese Exporte etwa unter Zuhilfenahme einer Zwischendatenbank erfolgen. In wachsendem Maß stellen immer mehr Softwarehersteller allerdings auch Schnittstellen wie Webservices oder für einen XML-Import zur Verfügung.

Der einzige Haken bei diesen Schnittstellen liegt im Regelfall in der vom Drittanbieter geforderten Struktur der Daten, die über diese Schnittstellen ausgetauscht werden sollen.

Hier hilft der Umstand, dass der Signalausgang in den Workflows von DOCUMENTS ebenfalls dazu genutzt werden kann, ein beliebiges Skript auszuführen. Der Code kann von Ihnen dann quasi jedes beliebige Exportformat erzeugen.

Ausführlicher Beispielcode für eine Ausgabe in eine SQL-Datenbank oder zur Erzeugung eines beliebigen eigenen XML-Export-Formates wurde in den vorherigen Übungen schon vorgestellt.

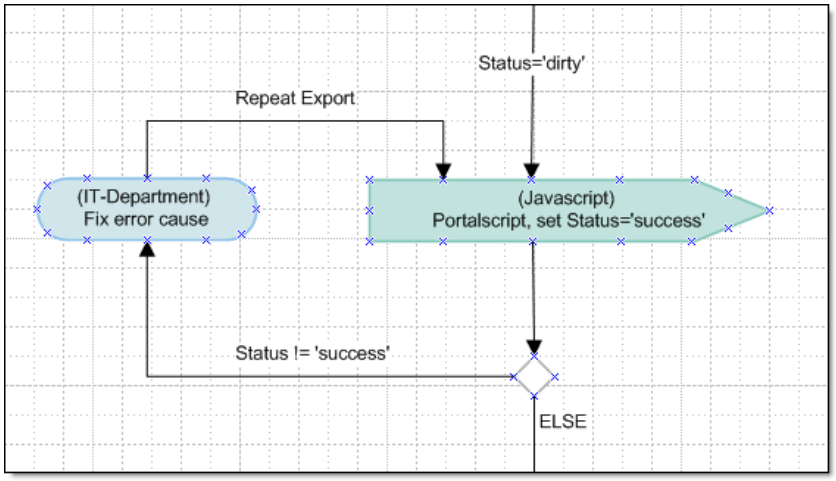
Zur Gewährleistung eines einwandfreien Exports ans Drittsystem ist lediglich auf eine speziellere Modellierung des Workflows zu achten, da Script-Signalausgänge auch im Fall eines Skriptfehlers komplett durchlaufen werden.

Es hat sich in der Praxis bewährt, **vor** der Skriptausführung des Signalausgangs ein Statusflag (z.B. „dirty“) in der Mappe zu setzen.

Ausschließlich bei **erfolgreicher** Verarbeitung des kompletten Scripts des Signalausgangs setzt das Skript dann als letzte Anweisung dieses Statusflag auf einen abweichenden Wert (z.B. „success“).

Indem man nun direkt nach dem Signalausgang eine Entscheidung einbindet, die das Statusflag überprüft, kann man auf eventuelle Fehler in der Skriptverarbeitung reagieren und die Mappe einer Gruppe administrativer User zusenden, die sich um eine Behebung der Fehlerursache kümmern und den Export erneut anstoßen können.

Eine beispielhafte Implementierung dieser Abfrage könnte wie folgt aussehen:



4.17 loginscript, afterLoginScript, setPasswordScript

In den Login-Mechanismus von DOCUMENTS kann mit Hilfe dreier Mandanten-Eigenschaften eingegriffen werden. Diese stellen jeweils eigene zusätzliche Umgebungsvariablen implizit zur Ausführungszeit zur Verfügung:

- **loginScript** – wird **vor** dem Login des Benutzers ausgeführt

login	Benutzerlogin
password	vom Benutzer eingegebenes Passwort im Klartext
source	„SOAPAPI“, „instance“, „unit“, „asUser“

- **afterLoginScript** – wird **nach** erfolgreichem Login ausgeführt

login	Benutzerlogin des gerade eingeloggten Benutzers

- **setPasswordScript** – wird **vor** Passwortänderung ausgeführt

login	Benutzerlogin
oldpassword	Wert des Eingabefelds “Bisheriges Kennwort”
newpassword	Wert des Eingabefelds “Neues Kennwort”

Das folgende kleine Beispiel für ein `loginSkript` prüft lediglich den aktuellen Wochentag. Aus Sicherheitsgründen werden Logins nur von Montag bis Freitag gestattet, an Samstagen und Sonntagen wird der Login verweigert:

```
var today = new Date();
var weekDay = today.getDay();

if ((weekDay == 0) || (weekDay == 6))
{
  context.errorMessage = "Am Wochenende wird nicht gearbeitet";
  return -1; // LOGIN VERWEIGERN
}
return 0; // LOGIN ZULASSEN
```

Das nachfolgende Beispiel für ein `afterLoginScript` definiert einen Zähler als Eigenschaft auf dem Benutzer, mit dem die Anzahl der erfolgreichen Logins protokolliert werden kann. Dies würde beispielsweise eine statistische Auswertung über die regelmäßige Nutzung des Systems über ein zweites Skript ermöglichen:

```
var su = context.findSystemUser(login);
if (su) {
  var loginCount = 1; // Default
  var strLoginCount = su.getAttribute("$SuccessfulLogins");
  if (strLoginCount != "") {
    loginCount = parseInt(strLoginCount) + 1;
  }
  su.setAttribute("$SuccessfulLogins", loginCount);
}
```

Das nachfolgende Beispiel für ein `setPasswordScript` erweitert die in DOCUMENTS integrierte Passwortprüfung um einen Test, dass der Anwender nicht seinen eigenen Vornamen als Passwort setzen darf:

```
var su = context.findSystemUser(login);
if (!su)
{
  context.errorMessage = "Could not find user!";
  return -1;
}
if (!su.checkPassword(oldpassword))
{
  context.errorMessage = "Typo in your old password!";
  return -1;
}
if (newpassword.toLowerCase() == su.firstName.toLowerCase())
{
  context.errorMessage = "First name as password not allowed!";
  return -1;
}
return 0; // all fine
```

4.18 afterMailScript

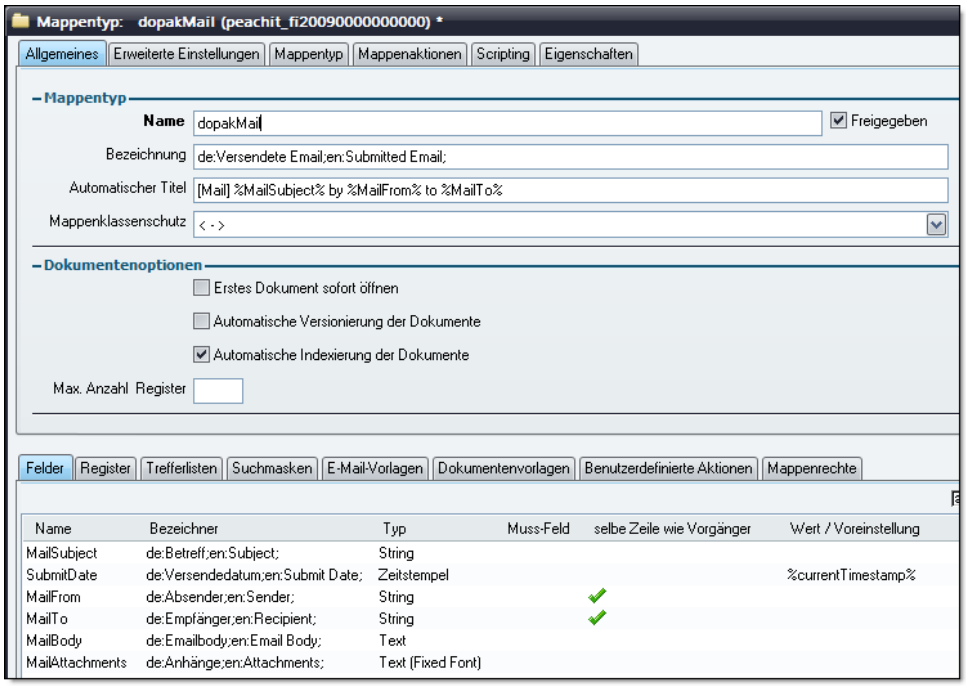
Ein häufiger Kritikpunkt der in DOCUMENTS integrierten Adhoc-Emailversendung liegt darin, dass praktisch keine Protokollierung der per Email versendeten Informationen erfolgt. Im Status einer Mappe sieht man zwar, **dass** eine Email aus der Mappe heraus versendet wurde, man erfährt jedoch nicht, welche Inhalte diese hatte.

Unter Zuhilfenahme eines einfachen Mappentypen und des nachfolgend beschriebenen Scripts kann hingegen eine vollständige Protokollierung umgesetzt werden.

Dazu ist am gewünschten Mappentypen die Eigenschaft `afterMailScript` anzulegen, der Wert muss der Name des nach erfolgtem Emailversand auszuführenden Scripts sein. Innerhalb dieses Scripts stehen dann implizit die folgenden Umgebungsvariablen zur Verfügung:

- mailSubject Betreff wie vom Absender im Maildialog eingegeben
- mailFrom Emailadresse des Absenders
- mailto Emailadresse des bzw. der Empfänger(s)
- mailBody Inhalt der Email, wie im Versendedialog eingegeben
- mailAttachments Namen (!) der mit der Email versendeten Dokumente

Eine mögliche Struktur des zur Protokollierung geschaffenen Mappentypen könnte dann wie folgt aussehen:



Eine sinnvolle Erweiterung dieses Mappentypen könnte darin bestehen, über ein Referenzfeld einen direkten Bezug zwischen den Emailmappen und den ursprünglich per Mail versendeten Originalmappen herzustellen. Dieser Ansatz ist beispielsweise konsequent im Solution Package namens RELATIONS gewählt worden, um beispielsweise das komplette Support-Ticket-Szenario auf diese Weise hochgradig transparent für alle beteiligten Mitarbeiter zu gestalten.

Der Quelltext eines auf den im Screenshot dokumentierten Mappentypen passenden AfterMailScripts könnte dann wie folgt aufgebaut sein:

```
var note = context.createFile("dopakMail");
if (!note)
{
  context.errorMessage = "Could not create new note file!";
  return -1;
}

note.MailSubject = mailSubject;
note.MailFrom = mailFrom;
note.MailTo = mailTo;
note.MailBody = mailBody;

if (mailAttachments != "")
{
  note.MailAttachments = mailAttachments;
}

// State is not new
note.setUserStatus(context.currentUser, "Standard");
note.setUserRead(context.currentUser,true);
note.sync();
```

4.19 AccessSkript am Mappentypen

Der Rechte-Mechanismus von DOCUMENTS ermöglicht durchaus sehr weitreichende und flexible Einschränkungen des Zugriffs auf eine einzelne Mappe, und durch den Einsatz von Mappenklassenschutz, ACLs oder GACLs ist auch eine Verrechtung auf der Ebene der Mappeninhalte möglich.

Dennoch erfordern manche Projekte regelmäßig auch eine Einschränkung etwa der Schreibrechte auf eine Mappe auf Ebene des Inhalts, wenngleich eine größere Gruppe von Benutzern wenigstens Leserechte auf die Mappe haben sollen.

DOCUMENTS ermöglicht die Umsetzung solcher Anforderungen mit Hilfe einer versteckten Eigenschaft am Mappentypen namens „AccessScript“, die als Wert dann den Namen des auszuführenden Scripts erfordert.

Ein solches AccessSkript wird bei jedem Versuch eines lesenden Zugriffs auf eine Mappe ausgeführt und kann jede erdenkliche Bedingung überprüfen (dabei ist jedoch zwingend auf eine ressourcenschonende Vorgehensweise zu achten, da dieser Event zu den extrem teuren Ressourcen zählt!).

In einem AccessSkript stehen in Form des bekannten Arrays namens `enumval` derzeit vier verschiedene Rechte zur Verfügung:

"DlcFile_RightRead", "DlcFile_RightWrite", "DlcFile_RightChangeWorkflow" sowie "DlcFile_RightReactivate".

```
1  // The script has to specified as filetype property:
2  // AccessScript=jsAccessScript
3  // It modifies the user's access privileges to the file
4  // whenever a user tries to access the file for viewing
5  // sample: only the file owner (creator) has the write-right at the file
6
7  var myFile = context.file;
8  if (!myFile)
9  {
10     context.errorMessage = "Not in a file context!":
11     return -1;
12 }
13
14 var read = "1";
15 var write = "0";
16 var changeWorkflow = "0";
17 var reactivate = "0";
18
19 var loginFileOwner = myFile.getAutoText("fileOwner.login");
20 var loginCurrentUser = context.currentUser;
21
22 if (loginFileOwner == loginCurrentUser)
23     write = "1";
24
25 for (var i = 0; i < enumval.length; i++)
26 {
27     if (enumval[i] == "DlcFile_RightRead")
28         enumval[i] = read;
29
30     if (enumval[i] == "DlcFile_RightWrite")
31         enumval[i] = write;
32
33     if (enumval[i] == "DlcFile_RightChangeWorkflow")
34         enumval[i] = changeWorkflow;
35
36     if (enumval[i] == "DlcFile_RightReactivate")
37         enumval[i] = reactivate;
38 }
39
```

Der grundlegende Aufbau eines AccessSkripts ist in Form des Beispielcodes auf der vorhergehenden Seite dargestellt. Insbesondere die Zeilen 25 bis 38 sind stets in der dargestellten Form zu berücksichtigen und dürfen niemals vergessen werden.

Die eigentliche Bedingung, im Beispiel lediglich eine Überprüfung, ob der aktuell angemeldete Benutzer der Besitzer der Mappe ist oder nicht, finden Sie in den Codezeilen 19 bis 23. Diese Bedingungen können Sie in beliebiger Form ausbauen.

4.20 Skriptklassen erweitern

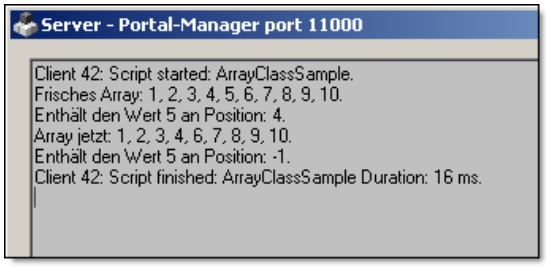
Verglichen mit anderen Hochsprachen vermisst man als Entwickler gelegentlich die eine oder andere Funktion im Sprachumfang, die in anderen Sprachen hingegen sehr gängig ist und den Komfort beim Programmieren deutlich erhöhen würde. Üblicherweise programmiert man sich die fehlende Funktionalität dann selbst hinzu, steht dann aber vor dem Problem der eindeutigen Zuordnung der Hilfsfunktion xy() zum korrekten Datentypen. JavaScript bietet hierfür allerdings einen bequemen Mechanismus, die vorhandenen Klassen um zusätzliche Funktionen zu ergänzen.

Dies geschieht unter Zuhilfenahme des Schlüsselworts prototype in der Funktionsdeklaration. Das folgende Beispiel erweitert das JavaScript-Array um die zwei regelmäßig gesuchten Methoden `inArray()` und `removeElement()`:

```
1  // Dieses Script wird als Lib in andere Scripts importiert
2  // Es erweitert das Javascript-Array um die Methoden
3  // inArray(element) und removeElement(element)
4
5  Array.prototype.inArray = function (value) {
6    for (var i = 0; i < this.length; i++) {
7      if (this[i] === value)
8        return i;
9    }
10   return -1;
11 }
12
13 Array.prototype.removeElement = function (value) {
14   for (var i = 0; i < this.length; i++) {
15     if (this[i] === value) {
16       this.splice(i, 1);
17       return i;
18     }
19   }
20   return -1;
21 }
```

Der nachfolgende kleine Code demonstriert die Nutzung anhand eines einfachen Arrays mit zehn Elementen (es werden einfach die Zahlen 1 bis 10 eingetragen). Zunächst wird ausgegeben, ob das Element mit dem Wert 5 in diesem Array gespeichert ist (ja, ist es). Anschließend wird genau dieses Element aus dem Array gelöscht (Zeile 7) und wiederum ausgegeben, ob das Element namens „5“ noch gefunden wird (nein, nun nicht mehr).

```
1  // #import "ArrayClass"
2
3  var liste = new Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
4
5  util.out("Frisches Array: " + liste.join(", "));
6  util.out("Enthält den Wert 5 an Position: " + liste.inArray(5));
7  liste.removeElement(5);
8  util.out("Array jetzt: " + liste.join(", "));
9  util.out("Enthält den Wert 5 an Position: " + liste.inArray(5));
```



4.21 Singleton-Mappen

Gelegentlich kommt es vor, dass von einem bestimmten Mappentypen exakt keine oder genau eine Mappe im gesamten System (bzw. Benutzerkontext) erlaubt sein soll. Ein klassisches Beispiel für diese Anforderung stellt etwa die Konfigurationsmappe der DOCUMENTS-LDAP-Kopplung dar, oder die Konfiguration in CONTRACT v2.

Zur Gewährleistung einer einfachen Auffindbarkeit dieser auch „Singleton“ genannten Mappe besteht die Möglichkeit, ein Skript fest mit einem dynamisch gefilterten Ordner zu verknüpfen. Statt wie sonst üblich beim Klick auf einen Ordnernamen im Ordnerbaum dann eine Trefferliste zu generieren, wird stattdessen das hinterlegte Skript ausgeführt, welches dann seinerseits die Einzigartigkeit der gesuchten Mappe gewährleistet.

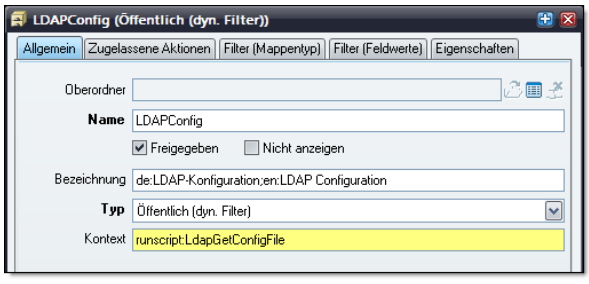
Ein solches Skript könnte etwa wie folgt gestaltet sein:

```
var myFRS = new FileResultset("LdapConfiguration", "", "");
var myMappe = null;
if (!myFRS || myFRS.size() <= 0) {
  myMappe = context.createFile("LdapConfiguration");
} else {
  myMappe = myFRS.first();
  delete myFRS; // free memory!
}
if (! myMappe) {
  context.errorMessage = "Could not read file!";
  return -1;
}
return myMappe.getAutoText("id");
```

Erläuterung der Funktionsweise: es wird versucht, im Kontext des aktuellen Benutzers ein FileResultset auf LdapConfiguration-Mappen zu erzeugen. Sofern dies nicht gelingen sollte bzw. sofern das erstellte FRS keine Treffer enthält, legt das Skript eine neue Mappe an. Wird das Skript hingegen fündig, liefert es die erste gefundene Mappe zurück.

Das Beispiel enthält dabei bewusst zwei Unzulänglichkeiten: Erstens wird vorausgesetzt, dass der aktuelle Benutzer mindestens Lese- und Anlagerechte am Mappentypen hat. Zweitens vernachlässigt das Skript die potenzielle Existenz von mehr als einer für den Anwender sichtbaren LdapConfiguration-Mappe.

Das Skript wird anschließend wie folgt an den gewünschten Ordner gebunden:



Es ist eine gute Idee, die Filter des Ordners genau auf den im Skript verwendeten Mappentypen zu konfigurieren. So wird eine Auffindbarkeit sichergestellt.

4.22 Download von Binärdateien per benutzerdefinierter Aktion

Bislang bestand in der Scripting-Engine die Problematik, dass serverseitig automatisch generierte Binärdateien wie beispielsweise PDFs oder automatisch erzeugte Office-Dokumente nicht als Rückgabewert benutzerdefinierter Aktionen genutzt werden konnten. Der dafür üblicherweise genutzte `context.returnType „file:dateiname.ext"` kann prinzipiell nur mit Textinformationen genutzt werden, da der Output innerhalb des Scripts stets in eine String-Variable eingelesen werden muss.

In den beschriebenen Fällen bestand daher bisher die einzige Möglichkeit einer Verarbeitung in einem Upload als neues Dokument in eine Mappe – und der Anwender musste den Download des Dokuments dann manuell ausführen.

Es besteht die Möglichkeit, tatsächlich das Dokument direkt herunterzuladen. Dafür muss das gewünschte Dokument in einem für den Server lesbaren Verzeichnis abgelegt sein.

Der folgende Code veranschaulicht die Anwendung:

```
1  // first, find current file
2  var myMappe = context.file;
3  if (!myMappe) {
4    context.errorMessage = "Not in a file context!";
5    return -1;
6  }
7  // now get the documents tab
8  var reg = myMappe.getRegisterByName("Documents");
9  if (!reg) {
10   context.errorMessage = "Register not found!";
11   return -1;
12 }
13 // find the desired document
14 var docIter = reg.getDocuments();
15 if (docIter && docIter.size() > 0) {
16   for (var doc = docIter.first(); doc; doc = docIter.next()) {
17     if (doc.fullname == pDocument) {
18       // pDocument is a Script parameter - if we find it, return the file
19       context.returnType = "download:" + doc.fullname;
20       return doc.downloadDocument();
21     }
22   }
23 }
```

Das Beispiel erfragt einen Scriptparameter namens `pDocument`. Anschließend werden alle in der aktuellen Mappe hinterlegten Dokumente darauf untersucht, ob ihr Dateiname exakt der Eingabe des Benutzers entspricht (bei der Vorführung dieses Features auf der DoPaK 2009 wurde durch ein Aufzählungs-Skript im Parameterdialog sichergestellt, dass der Anwender stets einen korrekten Dateinamen auswählen musste).

Sofern das gewünschte Dokument gefunden wird, lädt das Skript es ins temporäre Verzeichnis des DOCUMENTS-Servers herunter und gibt es an den Browser des Anwenders zurück.

Der Trick liegt in der Kombination der Codezeilen 19 und 20. Zunächst wird der passende `returnType` definiert. In Zeile 20 erfolgt dann der temporäre Download, der als Rückgabewert automatisch Pfad+Dateiname der temporären Datei zurückgibt. Dies wird dann genutzt, über die `return`-Anweisung das Dokument zum Client zurückzuschicken.

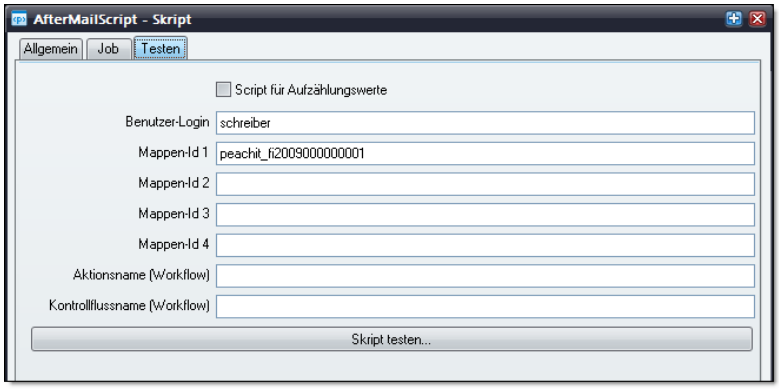
Das Beispiel hat den Nachteil, nach und nach das `temp`-Verzeichnis auf dem Server mit Dateileichen zu füllen. In der Projektpraxis sollte also gewährleistet werden, dass dieses Verzeichnis regelmäßig aufgeräumt wird.

5. Testen, Debuggen und Verschlüsseln

5.1 Skripte testen

Um Skripte unabhängig von Mappen und Workflows zu testen, können an den Scripts Test-Szenarien definiert werden. Dazu sind die Mappen- und Benutzer- und Workflow-Parameter zu definieren und entsprechende Felder mit Feldwerten anzulegen.

Mit dem Aktionsknopf *Skript testen...* kann das Skript dann ausgeführt werden. Nach Beendigung des Scripts werden in einem Dialog der Rückgabewerte und die Feldwerte angezeigt.



5.2 Logoutput um Skriptausführungen erweitern

Über die Konfigurationsdatei namens `partnernet.ini` kann man ein erweitertes Logging der Ausführung von Scripts konfigurieren. Dazu dient der Schalter namens

```
$ScriptLog 1
```

Nach Aktivierung dieser Option und Neustart des DOCUMENTS-Servers wird jedwede Ausführung eines Scripts im Serverfenster (bzw. dessen Logfile) protokolliert.

Dabei erscheinen je Skript zwei zusätzliche Ausgabezeilen im Serveroutput. Zu Beginn einer Ausführung wird auf das gestartete Skript hingewiesen, und bei Beendigung des Scripts wird eine weitere Zeile ausgegeben, die darauf hinweist, dass das Skript namens XY nach ABC Millisekunden Verarbeitungszeit beendet wurde.

Dieses erweiterte Logging ist insbesondere zu Profilingzwecken und bei der Recherche auf mögliche Event-Kaskaden von Skripten sinnvoll.

Anstelle der 1 darf auch ein Pfad+Dateiname zu einer vom DOCUMENTS-Server beschreibbaren CSV-Datei konfiguriert werden. In dieser CSV-Datei werden dann je Ausführung der Skriptname, die Ausführungsdauer und ggf. auftretende Fehler protokolliert.

5.3 Parameter der Skriptausführung anpassen

Per Default stellt der Server der Scripting-Engine je Skriptausführung einen Speicherbereich von 4 MByte (4096 KByte) zur Verfügung. In speziellen Fällen, etwa wenn ein Job-Skript sehr viele Vorgänge gleichzeitig bearbeitet oder Mappen mit sehr umfangreichen Gentable-Daten per E4X manipuliert werden, kann es vorkommen, dass dieser zugeteilte Speicherbereich nicht ausreicht. In solchen Fällen käme es zu einem Abbruch des jeweiligen Scripts mit einer „Out of memory“-Fehlermeldung. Dem können Sie entgegen wirken, indem Sie in der Datei `partnernet.ini` des Serververzeichnisses den Parameter

```
JSMaxMemory 4096
```

entsprechend anpassen. Wir weisen darauf hin, dass sich die Standardeinstellung von 4 MByte bisher in der Praxis bewährt hat.

Die Scripting-Engine beinhaltet darüber hinaus einige spezielle Prüfroutinen, um die Ausführung potenzieller Endlosschleifen und damit ein versehentliches Lahmlegen des DOCUMENTS-Servers zu verhindern. Konkret überwacht werden Verzweigungen, Iteratorschleifen und rekursive Funktionsaufrufe. Im Normalfall sollten diese Überwachungen unangetastet bleiben, da sie wesentlich zur Stabilität der Scripting-Engine beitragen. Jedoch ist es für spezielle Situationen möglich, die einzelnen Schutzmaßnahmen in der Datei `partnernet.ini` des Serververzeichnisses mit den folgenden drei Parametern wahlweise zu erweitern oder ganz abzuschalten:

```
JsMaxBranch     # auf 0 setzen, um die Pruefung abzuschalten
JsMaxIterator   # auf 0 setzen, um die Pruefung abzuschalten
JsMaxRecursion  # auf 0 setzen, um die Pruefung abzuschalten
```

Achtung: Eine komplette Abschaltung der Sicherheitsmechanismen empfiehlt sich nicht auf Produktivsystemen!

5.4 Debuggen mit dem Script-Debugger

Mit Hilfe einer lizensierten Version des von otris angebotenen Tools JSRemote (nun als JANUS Script Debugger geführt) haben Sie die Möglichkeit, Ihre Scripts ähnlich wie in modernen Entwicklungsumgebungen zu debuggen.

Der JANUS Script Debugger stellt Ihnen dabei unter anderem einen Einzelschrittmodus inkl. Setzen von Breakpoints zur Verfügung, außerdem können Sie sogenannte Watches, also Überwachungen von Variablenwerten und Ausdrücken, definieren.

Für eine detaillierte Beschreibung der Handhabung des Debuggers lesen Sie bitte die diesbezügliche Produktdokumentation.

5.5 Skripte verschlüsseln

Es besteht die Möglichkeit, über eine Wartungsoperation Scripts automatisch zu verschlüsseln. Die Verschlüsselung hat dabei keine messbare Auswirkung auf die Verarbeitungsgeschwindigkeit der so behandelten Scripts, die Verschlüsselung dient lediglich zum Schutz Ihres geistigen Eigentums bzw. vor unbedachter Änderung potenziell gefährlichen Codes.

Der Aufruf der Wartungsoperation `encryptScripts:filter` erfolgt wie üblich über den Client unter dem Menüpunkt „Administration > Wartungsoperation durchführen“. Als `filter` können Sie auch Wildcards verwenden, etwa um gezielt alle Scripts mit einem gemeinsamen Prefix en bloc zu verschlüsseln (Beispiel: `encryptScripts:crm*`).

Darüber hinaus existiert eine weitere Wartungsoperation namens `encryptMarkedScripts`. Diese dient dazu, automatisch alle Scripts zu verschlüsseln, deren Quellcode die spezielle Kommentaranweisung

```
//#crypt
```

enthält. Der spezielle Kommentar muss nicht zwingend als erste Codezeile des Quellcodes eingetragen werden, sondern kann auch im späteren Verlauf des Skripts auftreten. In diesem Fall verbleibt dann der erste Teil des Skripts im Klartext lesbar und anpassbar, erst der Code nach dem Auftreten der Präprozessoranweisung wird dann verschlüsselt.

Dieser Umstand kann dazu genutzt werden, dem Anwender / Kunden einen Konfigurationsbereich zur Verfügung zu stellen, die eigentliche Skriptfunktionalität aber aus Sicherheitsgründen zu verschlüsseln.

Achtung: Die Verschlüsselung kann nicht mehr rückgängig gemacht werden!

Das bedeutet, Sie sind nicht in der Lage, einmal verschlüsselte Skripts wieder in unverschlüsselte Form zu überführen! Daher ist es dringend ratsam, von jedem Skript vor seiner Verschlüsselung eine unverschlüsselte Sicherheitskopie des Originals anzufertigen!

Darüber hinaus ist zu berücksichtigen, dass es nicht möglich ist verschlüsselte Scripts zu debuggen. Insbesondere bedeutet das auch, dass Skripte, die verschlüsselte Bibliotheken importieren, mit dem ScriptDebugger nicht untersucht werden können.

Abbildungsverzeichnis

Abb. 1: Scripting-Bibliothek im DOCUMENTS-Manager 4
Abb. 2: Skript-Aktionen eines Mappentyps 5
Abb. 3: Aufbau eines Skripts 6
Abb. 4: Einbindung eines externen Editors 8
Abb. 5: Arbeitskopienkonzept 11