DOCUMENTS 4

SERVER-SIDE SEITIGES JAVA-SCRIPTING

Programming Guide

Table of Contents

1. Introduction

The DOCUMENTS 4 server includes an integrated *scripting engine*. Thus, it enables ***server-side*** execution of *JavaScript.*

The scripting engine processes JavaScript in JS versions 1.0 to 1.9; from version 1.3, in compliance with the ECMA script 262 specification.

These scripts are defined in the DOCUMENTS Manager. This is performed in the form of global script objects, which are initially created in a scripting library of the logged-in principal (Fig. 1).

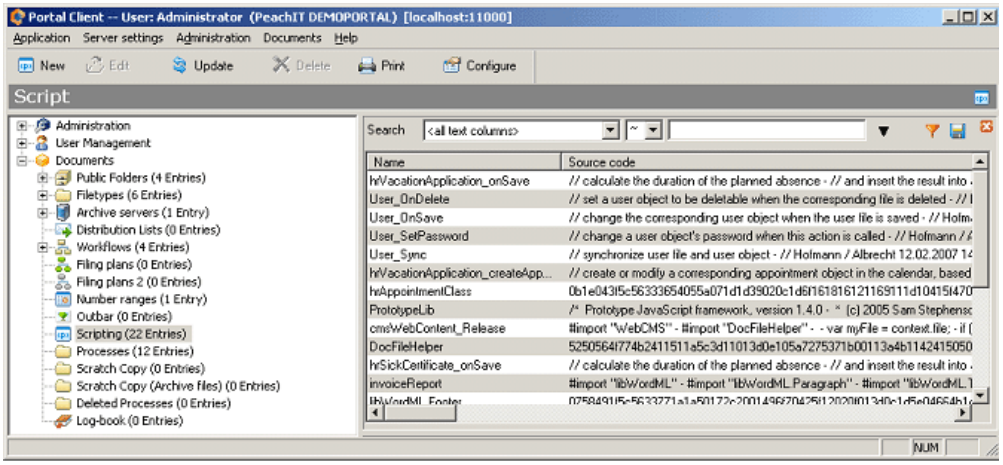


Fig. 1: Scripting library in the DOCUMENTS Manager

These are then embedded into predefined positions for use. Execution is performed according to the embedded position during runtime with specific actions. The following uses of server-side scripts are available here:

- Default actions on DOCUMENTS files. The script is integrated with a defined action on the file type:
  - On creating new files
  - On editing
  - On saving
  - After saving
  - On archiving
  - On deleting
  - As "Allowed actions" script: This controls globally for the DOCUMENTS file which actions are generally available to the user.

Fig. 2 shows for a file type in the DOCUMENTS Manager how assigning existing scripts to specific actions is implemented.

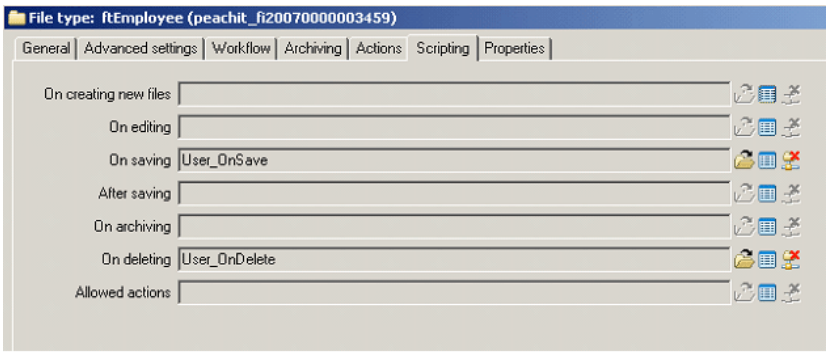


Fig. 2: Script actions of a file type

- As user-defined actions on the DOCUMENTS file, and on folders
- For describing conditions (guards) in workflows
- As send signal in workflows
- In the context of field use actions in workflows
- For defining enumeration values of a field of the enumeration type

This allows implementing specific requirements in DOCUMENTS 4 such as:

- Drop-down lists that are dependent on the values of other fields
- Access to third-party databases for filling drop-down lists or field values
- Complex guard conditions that cannot be defined through simple expressions
- Automation of processes in the form of job-driven scripts
- Starting external DLLs to navigate towards third-party systems
- For calculating data

The DOCUMENTS 4 scripting runtime environment facilitates accessing various file and field properties. Runtime constants can also be read, such as the currently involved user or the name of the current workflow step.

Successful creation of scripts requires at least basic knowledge in the programming language *JavaScript* and the various classes, objects and properties provided by the DOCUMENTS 4 scripting interface. Practical knowledge in administration and configuration of DOCUMENTS 4 are also imperative prerequisites for understanding particularly the connections between the individual classes and objects and to be aware of the impacts of specific script constructions on system performance.

2. Defining Java Scripts

The central library for Java scripts resides in the DOCUMENTS Manager's tree structure below the *Documents* node (see Fig. 1).

> *Please remember that this entry will only be available in the tree if you are logged in to a complete DOCUMENTS 4 principal. Simple archive search principals (so-called EASY Web clients) do not have script capabilities.*

Scripts are globally created, tested and administered here. A script object is essentially composed of a unique *name* and the *source code* (Fig. 3). Other elements, such a script parameters, are not necessarily required for the executions with defined actions.

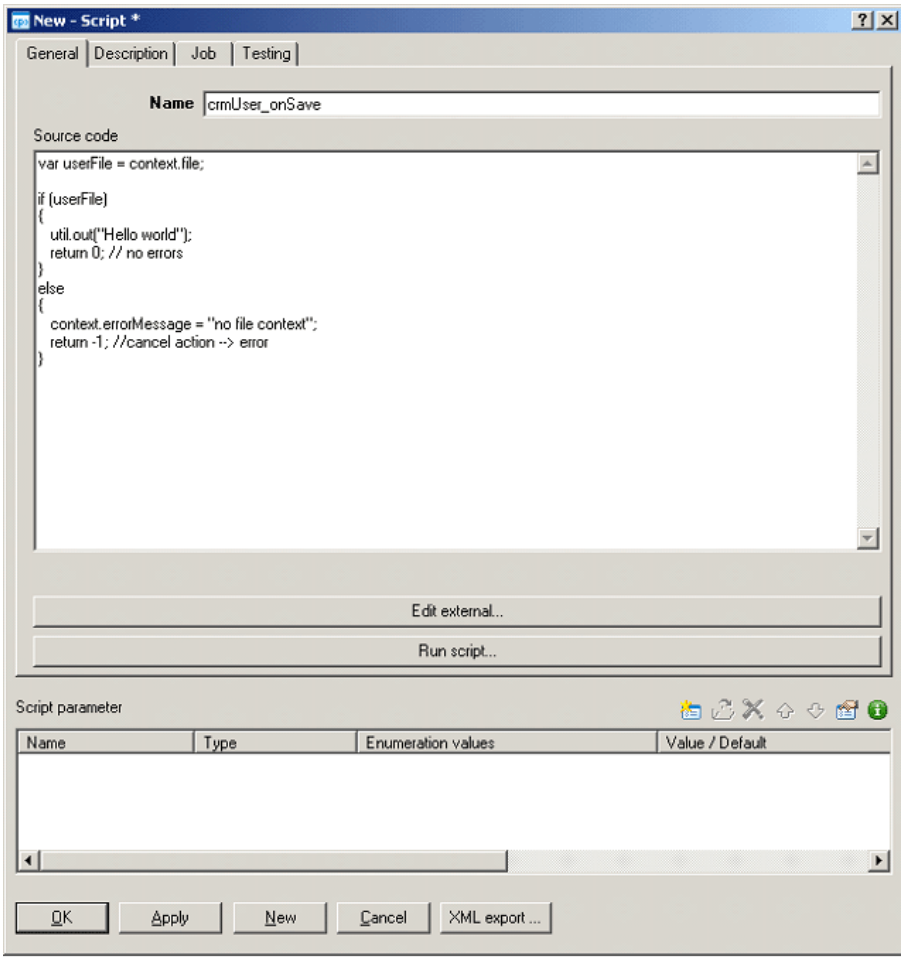


Fig. 3: Script structure

Although allocating names does not require any conventions, the use of telling names which significantly simplify later access to the scripts defined here is meaningful for use and re-evaluation.

For instance, preceding each script with an abbreviation used to uniquely identify the project part to which the script belongs may be a good idea. For instance, the

names of all scripts from the RELATIONS Solution Package start with `crm`; all CONTRACT scripts start with `lcm`; the scripts of the LDAP interface start with `Ldap`, etc.

It is also helpful to append the name of the related file type or folder as well as the calling action.

A script that should be executed on saving a DOCUMENTS file of the file type *crmUser* could be allocated the name `crmUser_onSave` (see example in Fig. 3).

This makes integration easier because the corresponding script can be quickly retrieved from the library and erroneous assignments are avoided.

Once created, a script can then be integrated into and activated on various events or anchor points.

The extension option to integrate scripts *via so-called* properties is available. An example of this can be found in CONTRACT: When sending a contract via e-mail, the sent e-mail message is automatically saved in this connection as a file of the *Note for the file* type. The script is integrated via a *property* of the sent contract file.

Code input using an external editor

The input field for the source code only represents it; however, formatting options are not provided.

When you click the *Edit external* button, the script opens, by default, in the Windows Editor.

However, the option to integrate any external editor and to edit the code using its functions is available.

In order to be able to use a specific editor as an external tool, this editor must only meet one essential condition: it must enable passing path and file names in a text file to be directly opened at program startup.

The external editor is integrated via a *property* which you enter on the Properties tab of your principal object. The name of this property is

```
scriptEditor
```

The value of the property must be the full path and file name of the editor to be executed, e.g.

```
C:\Program Files\tools\notepad++\notepad++.exe
```

Fig. 4 shows this setting with a sample principal.

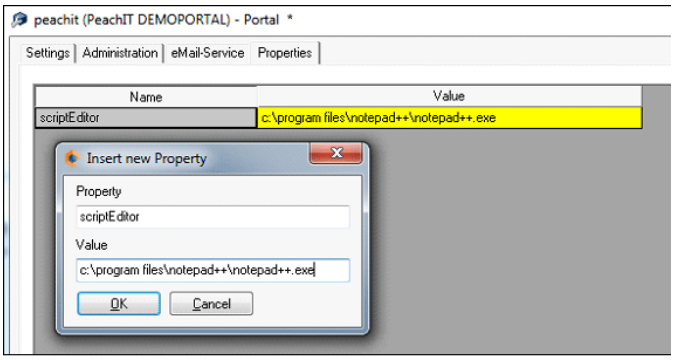


Fig. 4: Integrating an external editor

The use of an external editor includes a variety of advantages for script development:

- Syntax highlighting for JavaScript code
- Indenting the entered source text
- Retrieving and highlighting in color associated bracket pairs
- Simple creation of backup copies during code input
- Integrating versioning tools such as CVs or SVN, for example

3. Rules and Conventions

Some important general conditions arise in the script programming environment from project experience; considering them is recommended to guarantee long-term project success and convenient project maintenance.

3.1 Technical names and variable names

Thus, for instance, DOCUMENTS file structure and its replica as an object of the *DocFile* class may be a problem unless you adhere to the so-called programming language conventions when allocating the technical names of the file fields. These conventions are particularly known from programming in C/C++ and JAVA, but they are also valid for many other programming languages, and are generally considered good programming practice.

Because file fields of a DOCUMENTS file are directly accessed within scripting (as a publicly visible attribute of a DocFile object), these conventions also fully apply to scripts:

Technical names and variable names should **never be longer than 32** characters. Reason: only the first 32 characters are significant. If two different variables are identical in the first 32 characters, which of the two variables is actually used is, as it were, a coincidence. To illustrate this, here is a very bad example:

```
Field 1: psThamesSteamLinerVesselCaptainsBunksKey
Field 2: psThamesSteamLinerVesselCaptainsBunksDoorHandle
```

The part of the two labels that is bold represents the identical first 32 characters, whereas a difference is only obvious at the end of the labels.

Special characters in technical names and variable names are **absolutely forbidden**. This means in particular that umlauts (ä ö ü Ä Ö Ü ß), spaces and punctuation are forbidden, but also nearly all other special characters to be found on a standard keyboard.

Only exception to the previous rule: the **under_score is allowed**; in some respects, it replaces both spaces and dashes.

So, only use **letters** (a-z and A-Z) and **numbers** (0-9). **Never** start a technical name or variable name with a number!

Allowed are, for example: `psField123` or `_100_internal_field`.

Absolutely forbidden: `333_at_Procter_factory` or `EquivalentNumber`.

If you do not adhere to these conventions when defining your file types and their fields, DOCUMENTS 4 will not produce any direct errors; however, you will effectively obstruct field access from scripts or the other APIs of DOCUMENTS 4.

Be firm in sticking to the naming conventions described when defining your access profiles, file types, fields, tabs, number ranges, workflows and public folders for the technical names.

3.2 The working copy concept

The scripting interface provides two options to edit a DOCUMENTS file. On one hand, there is the command pair `startEdit()` and `commit()`, which emulates editing and saving a file via the Web interface, while on the other hand, changes to a DocFile object can also be made directly and synchronized to the underlying DOCUMENTS file using the `sync()` command.

To be able to understand this essential difference in processing methods, you need to be familiar with and understand the concept of working copies. This concept is based on a central significance in DOCUMENTS as a multiuser application.

It works as follows:

A user searches a DOCUMENTS file, optionally via search or by navigating through the public folders.

The user opens an individual DOCUMENTS file, and displays the index fields in the browser.

When, at that moment, other users open the same DOCUMENTS file, all users (leaving aside specific permissions here) will view exactly the same contents.

Now, if a user sets the file to edit mode, they will receive exclusive write permissions on that file right at that moment. No other user can now also edit the file at the same time. Internally, DOCUMENTS now has created a working copy of the file for the editing user. This working copy is modified by the user, but not the original DOCUMENTS file.

This allows other system users to view the current As-Is state of the file during search/retrieval, but they cannot edit that file because the system is aware that user A is already editing the file.

Only when user A saves their changes (and, with them, the file) will the changes made to the working copy be actually be written back to the genuine DOCUMENTS file. So, only from that moment will these changes be visible again to all other system users.

Whereas if the editing user discarded their changes, the system would only have to delete the working copy, so rollback on the genuine DOCUMENTS file would not be required.

Thus the concept of the working copy is not only a mechanism that guarantees transaction security, it is also an important performance feature.

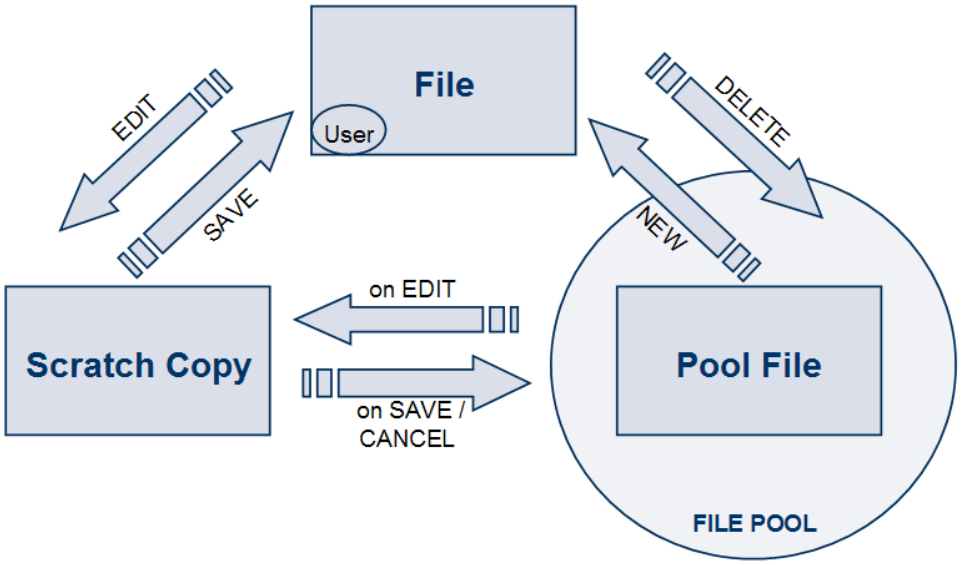


Fig. 5: Working copy concept

However, this working copy concept means for programming scripts that you need to be clear about when you may use which instructions because `startEdit()` also creates a working copy of the DOCUMENTS file, while `commit()` generates the genuine DOCUMENTS file from that working copy. The `abort()` statement must also be seen analogous to the Cancel button in the Web interface, i.e. it discards an existing working copy of the DOCUMENTS file.

The result is that these statements may never be used in specific call contexts of scripts, because otherwise data coherence and system stability can no longer be guaranteed.

A working copy of the file already exists for the following scripting events:

- On creating new files
- On editing
- On saving
- After saving

This means that when you program scripts for exactly these events, you may never edit the DocFile objects via `startEdit() / commit() / abort();` instead, you need to use `sync()`.

Moreover, setting the file object available via `context.file` to edit mode is also forbidden for the scripting events "On archiving" and "On deleting".

Additionally, experience has shown that editing files within a workflow or distribution via `startEdit()` is critical. More often than not, editing files fails in that an individual user or user group locks the file due to the current workflow step; however, other combinations are conceivable which would generally enable `startEdit()`. These specific circumstances might result in users simultaneously accessing the DOCUMENTS file via the Web stumbling on errors that seem to be inexplicable to them.

Consequence: Sacrifice `startEdit()` if the file object is in a workflow (now the current DOCUMENTS 4 version automatically prevents the use within the workflow. The resultant error message can be read via the `DocFile.getLastError()` method).

Inasmuch as you set a file object to edit mode via `startEdit()` within a script, you should take some more important basic rules to heart:

Make sure that the return value of the `startEdit()` call returns "`true`" on the DOCUMENTS file.

Make dead sure that by the script's runtime end the corresponding `commit()` or `abort()` has been performed for each `startEdit()`; otherwise, you will run the danger of generating an orphaned file in edit mode which can no longer be edited by any user. Although the scripting engine cleans up such working copies itself at the script end, there is not always the guarantee that this is performed quickly, and it is generally considered no good programming practice to improperly close your own created objects.

Each creation of a working copy via `startEdit()` by necessity causes the corresponding load on the database, because the complete file object must be initially duplicated. Thus, the use of `startEdit()` represents access to expensive resources (see below).

So, the above warnings leave the impression that `sync()` would be the generally preferable operation for making changes to files via scripting. However, this is only partially correct because synchronization of a DOCUMENTS file can also be

performed when it is being edited by a user over the Web (i.e. a working copy exists).

As a result, the possibility of loss of information exists during synchronizing, at least in theory, because the Web user can generally overwrite changes made to field values on the script. The previously listed scripting events, where on principle a working copy already exists at script start time, are exceptions – the `sync()` impacts on the working copy here, not on the DOCUMENTS file "below" it.

So, you should always ensure for each script which has to modify file contents that you do not risk any critical loss of information where users might be able to concurrently edit the same files as your scripts.

Besides a simple organizational guarantee of this security, scripting also provides the option to determine the DocFile object's edit status by reading the "Locked" and "LockedBy" attributes.

Important rules:

- Where possible, do without startEdit() and commit().
- Instead, preferably use the sync() operation.
- If startEdit() is essential, make sure that the matching commit() or (in case of failure) abort() is always available.
- Use these operations sparsely because they cause database load.

3.3 Handling so-called expensive resources

In programming parlance, some operations are referred to as access to so-called **expensive resources**. These are usually data sources which – unlike the computer's main memory – are based on significantly slower media, i.e. particularly data sources reverting to read-only memory media such as hard disks or DVD drives, but also resources that are exclusively available via the network (network shares, external databases such as ODBC data sources, etc.).

These resources are "expensive" in that accessing them takes much longer, compared to a variable in program memory, and in that the popular operating systems to which DOCUMENTS is available control file system, network resource and external (ODBC) data source access via a mostly limited number of so-called handles.

So it is obvious that programmers handle these resources very carefully, and should use them as seldom as possible.

Moreover, you should learn to carefully close each "expensive" resource that you access via scripting afterwards to make the access handles used available to the system again as soon as possible.

So what does this actually imply regarding your programming practice?

- Each `DBConnection` object that you open should be properly closed via calling the corresponding `close()` method.
- Not only is each `DBResultSet` object exclusively bound to a `DBConnection` object each, it should also always be closed properly on the object by calling the `close()` method.
- Avoid generating tons of `DBConnections` and `DBResultSets` within loop constructs.
- `File` objects (i.e. handles on data files of the server file system) must also be closed properly via `close()` after using them.

In combination with `startEdit()` / `commit()` / `abort()` and when building very extensive `FileResultset` iterators, even the server's own database is an expensive resource because these operations can cause very expensive database operations, depending on the structure of their file types, which may become noticeable through the corresponding high load on the database. Consequently, you need to handle these methods and objects very carefully.

Accesses to expensive resources nearly always impact on the script's runtime. Depending on where a script with accesses to expensive resources is used, the impact may also occur in unforeseeable places for the individual Web user, and it is even conceivable that intensive use of expensive resources also impacts on the system's general response times for all users.

A classic example of this are scripts for generating enumeration value which optionally generate the available drop-down list values from ODBC data sources or access DOCUMENTS files of another `file type` via a FileResultset (in the latter case, the question usually arises of why not use the essentially more elegant reference field? However, there are reputed to be use cases of this sophisticated construction of a drop-down list).

When this enumeration field is selected as "Show in hit list", the result is that starting a dynamic public folder (even an empty one) that filters on the named file type, or performing a search (even an unsuccessful one) results in that the script stored in the drop-down list must be executed, as a result of which the Web interface's response times can be noticeably extended.

Important rules:

- Each "`new DBConnection()`" call requires its matching `close()` command.
- Each generated `DBResultSet` must be closed properly by calling its `close()` method.
- Preferably, use neither `DBConnection` nor `DBResultSet` within loops.
- Enumeration fields containing script-generated enumeration values should not appear in hit lists.
- File handles must also be closed properly via `close()` after using them.

3.4 Script length

On the database side, the code field of a script object in DOCUMENTS (versions 5.0, 5.1 and 6.0, all patch levels) is limited to 64 KByte. So, your individual scripts may not contain more than about 64,000 characters of source text. In encrypted scripts, the maximum length is even reduced to about 32,000 characters. When using umlauts and special characters in the code (e.g. in Inline comments), the maximum length will be further reduced because the umlauts must be saved in coded format.

By comparison, more sophisticated projects quickly reach these limits, which is why it is a good idea to get used to modular programming practice.

This simply means that you store code parts in helper functions in library scripts and for possible configuration of your scripts you also use your own respective configuration scripts. In your individual main modules you then import the respective libraries using the statement

```
#import "ScriptName"
```

The solution packages CONTRACT, RELATIONS and INVOICE, as well as LDAP interface, among others, make intensive use of this.

Important rules:

- Proceed carefully with conception of your scripts
- Separate code, helper functions and configuration

3.5 Event cascading

In particular, the events configurable directly on the file type on the "Scripting" tab are cascaded within a script. This means that if, for instance, from Script "A" you create a new DOCUMENTS file of a file type "B", with a script "C" being stored on that file type for execution "on creating new files", then processing Script "A" will cause processing Script "C" exactly at the time the new file "B" is created.

In practice, this is preferably used when file types are intensely linked among each other with reference fields and link tabs (i.e. so-called 1:n relationships exist) and deleting a company file should also automatically delete all contact persons and discussion notes connected to that company; or, in reverse, deletion of parent master data from the system should be prevented for as long as accounting data in the form of transaction files still accessing this master data exists in the system.

Such constructs must inevitably be very well documented; they require a lot of discipline in project planning and implementation. Also, you need to be particularly aware of the by no means uncritical impact on the runtime of the scripts and the Web interface's response time felt by the user.

In this respect, event cascading is a powerful as much as a dangerous tool which should be used only in small doses and very carefully. In any event, testing such sophisticated execution contexts very extensively with a variety of test data is inevitable, as is considering each conceivable edge case in the tests, because otherwise this might quickly cause loss of valuable real-time values.

3.6 Potentially dangerous workflow scripts

A fixed object relationship exists between DOCUMENTS files and the workflow containing the DOCUMENTS files. For obvious reasons it is therefore extremely unwholesome for the workflow if a script that, for example, is modeled as a receive or send signal deletes the DOCUMENTS file that is still part of the workflow. In plain English: You would bite the hand that feeds you.

In this case, a valid file object that could be routed through the workflow to the defined end would no longer exist and, consequently, the file and workflow construct would no longer be in a defined state. In earlier versions, this could even lead to a complete crash of the DOCUMENTS 4 Server.

Faulty programming such as the one described above also vividly illustrate the responsibility that rests with you as a script developer; it should also clearly

explain why a script programmer needs to be very familiar with administrative practice and operations from the user's viewpoint.

3.7 Always declare variables

Particularly in Visual Basic programming, the bad habit of declaring variables improperly instead of using them the moment they are needed for the first time is very widespread among developers.

This extremely bad programming practice has particularly fatal impact in scripting because a variable declared improperly via `var variableName = initialvalue;` is automatically created through the initial assignment of the value as a global variable.

However, in this connection "Global" does not refer to individual script execution, but to the entire server runtime until next reboot.

EACH variable that you use must be declared properly**.**

4. Examples

The chapter below will introduce the different options arising from the use of scripts in a variety of examples. Besides an overview of usage options of the scripting engine you will find examples described in detail which also demonstrate the use of the classes provided by the scripting engine.

The examples are deliberately kept simple to get you started. In the further course these examples are used gradually to implement increasingly more sophisticated algorithms or requirements.

4.1 Call on creating new files

When you define a script on the file type to execute **on creating new files**, it will be executed every time you create a new file of that type. This can be particularly useful if you want to revert to data from an external database on creating new files, e.g. to automatically populate the DOCUMENTS file's fields.

However, the following example still omits database accesses; instead, two file fields are directly populated.

In the DOCUMENTS Manager, create a new script. Give this script the name `psScriptSample_onFileCreation` and enter the following source text:

```
// 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();
```

Then save the script by clicking OK.

Now define a simple file type named `ScriptSample` that consists of only two fields: a string field with the technical name `EmptyField1` and a date field named `EmptyField2`. Integrate the script you have just created to the file type

by opening, on the "Scripting" tab, to the right of "On creating new files", the selection list containing the available scripts and double-click to select the `psScriptSample_onFileCreation` script that has just been defined.

Save the file type after you have released it, and then open a Web browser to log in to DOCUMENTS. Now create a new file of the file type you have just defined; the new file is opened in edit mode but both fields have already been filled; the string field entry reads "Filled" and the current date is entered in the date field although no default values have been defined on the file type.

In these few lines, this simple example illustrates several powerful tools of the scripting engine. Initially, it gets the current file from the permanently existing `context` object. Because the script is executed on creating new files, this is the freshly created new file that still does not have any contents except for the stored default values and auto texts. This `DocFile` object is referenced in the form of the `myFile` variable.

This example then illustrates a programming practice called "**restrictive programming"**, although it is definitely clear in this exercise on which event this script is integrated, this is not always guaranteed in practice. Particularly when fewer experienced users are to integrate scripts into DOCUMENTS you, being the developer, cannot safely guarantee that your scripts are integrated correctly. Being the programmer of a script, you are required to ensure that the script has really been integrated in such a manner that it provides a `DocFile` object via the `context` object – and this is performed in the above example in that the existence of a valid object is queried in the `if()` statement.

The "`EmptyField1`" of this DOCUMENTS file is then assigned the value "Filled". So, you see that you can access a file's fields using simple statements regardless of whether the field resides on the default field tab or whether it resides on a field tab that you have additionally defined.

Write access to the second file field is a bit more complex than with the first field because this is a date field. Initially, in the code you therefore define a new variable named "`datumString`". All you do now is assign it the current date. To use this opportunity to illustrate accessing file independent auto texts, this example does not revert to the Javascript-own "`new Date()`", but the auto text `%currentDate%` is read. This date is automatically formatted in the current language format, i.e. to `mm/dd/yyyy` for English (U.S.) formatted DOCUMENTS.

However, the date field cannot respond to this string because it expects a Date object. For this reason it is necessary that you generate a Date object from this string. If you have already tried doing this using javaScript, you will appreciate the following functions: `util.convertStringToDate()`. This expects as parameter a string representing the date as well as another string that defines the format of this date string, i.e. in the example, "`mm/dd/yyyy`". The result is a genuine Date object that is now assigned to the "`EmptyField2`" file field.

The DOCUMENTS file must then only be made aware that it should import the field values that have just been set. Because it is already in edit mode, however, this may not be performed through "`startEdit()`" and "`commit()`"; instead, it is performed via a simple "`sync()`".

4.2 Call on saving files

If a DOCUMENTS file is in edit mode, you can define a script that is called once the data from the form has been sent to the server. Before synchronizing the working copy data to the actual DOCUMENTS file takes place, the script can still change the field data or respond to error conditions.

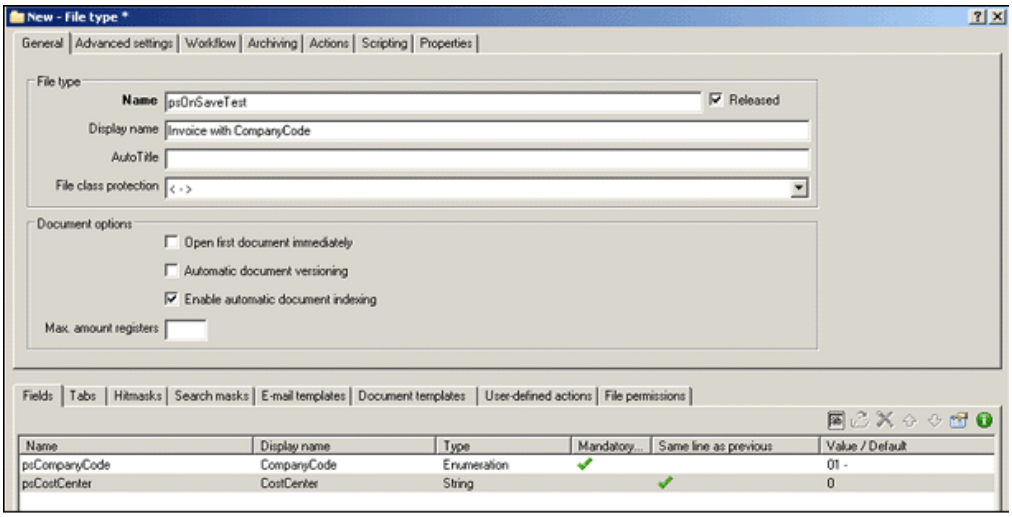
In this connection the option to cancel the save operation and to keep the user in the Web interface in edit mode in the DOCUMENTS file is an interesting feature. This, for instance, allows implementing constraint checks which can check multiple file fields for syntactically correct coherence, and the like.

If you intend to cancel file saving via script, you should send the Web users a meaningful error message including the reason for cancellation. This is done by storing this error message in the `context.errorMessage` at short notice. When finally ending the script with a negative return value (to do this, you need to end the script after assignment with the error message `return -1;`), it is exactly this error message that will be made visible in the browser as `Alert()` window.

The script described on the next page illustrates such a constraint check.

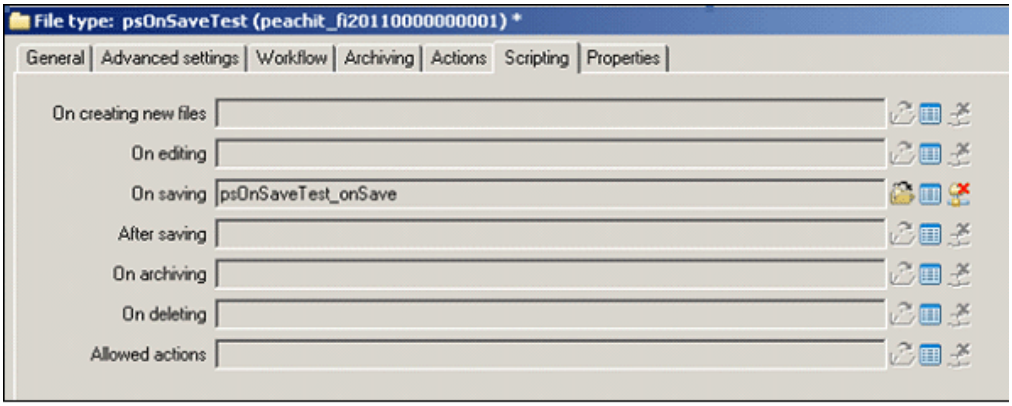
```
// Ausfuehrungskontext pruefen
var evt = context.event;
if (evt != "onSave") {
    context.errorMessage = "Falscher Scriptaufruf!";
    return -1;
}
// wir wissen nun, dass es ein "Beim Speichern"-Script ist

// 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;
```

For practical rehearsal of this script, we will require a new file type in this example as well. So, please create a new file type named `psONSaveExercise` via the client. In this file type we will then require at least the fields visible from the following screenshot:



As you can see, the Company code field should be a drop-down list containing the three enumeration values "01", "02", and "03". If in addition to non-selection of a field value you want to test a wrong company code, you can also define additional enumeration values.

On the Scripting tab you then enter the previously defined script with the previously displayed source code on the "On saving" event:



So, how does the script work?

At the start of its execution the script initially determines its execution context. We want to ensure that the scriptis started "On saving", and only in that context. If the script detects that it has been started in a different context, it will terminate with an error message. You can test this validation by alternately integrating the script once "After saving".

Following this, an attempt is made to get the current DOCUMENTS file, and the usual restrictive check on whether a valid file object has actually been found, is performed.

If so, our two file fields are read and the cost center can then be checked for the correct value via the `switch()` statement illustrated here, depending on the selected company code.

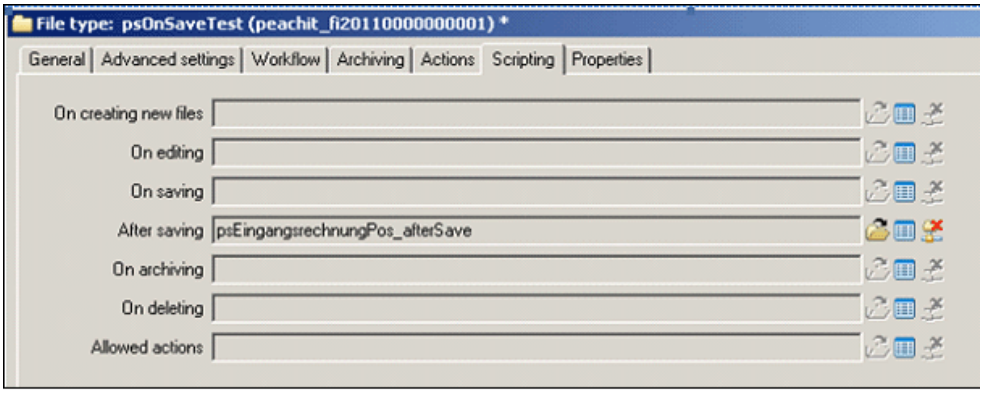
Should a cost center number be found outside the respectively valid range, the previously defined error message will be stored in `context.errorMessage` and the save operation will be canceled.

4.3 Call after saving files

An alternative to execution prior to performing the save operation is the option to execute a script directly after saving it. At the time of calling this scripting event, a working copy of the DOCUMENTS file no longer exists. The side-effect of this, however, is that you may never cancel such a script via "`return -1;`", because the result of such a cancellation would be that although saving the working copy back to the file will be executed, the due refresh of the user's browser to follow will not be performed. As a result, the user will receive the (consequent!) error message saying "This DOCUMENTS file does not exist!" next time they click "Save" or "Cancel".

Yet in practice this type of script is generally used to automatically calculate specific field values depending on other fields. The following example illustrates this using a fully automatic calculation of sales tax/VAT.

To do this, on the file type, `ftInvoice` (of the peachit *demo prinicipal), define a script that should be executed after* saving the file. In the following example, sales tax/VAT is calculated from the amount entered (total amount).



The source text of the `psftInvoice_afterSave` script linked in the previous screenshot looks as follows:

```
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
```

The script once again deliberately illustrates restrictive programming, because despite the previous statement saying returning -1 is forbidden, there is an exception to this rule: if the script has been started in the wrong execution context, an error message may and must be logically generated. So, in practice you should, of course, never omit the security prompts of restrictive programming.

4.4 Call on deleting

You can also influence the process through a script on deleting a DOCUMENTS file. The following example, taken from CONTRACT, illustrates in excerpts how, on deleting a company file, a prior check is made on whether the company, as a contracting partner, has been assigned contacts or even contracts.

```
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;
}
```

Initially, of course, the query on the execution context, now commonplace, on whether this is a valid file object is performed in the code.

Based on the content of the `crmId` field, the script then creates a `FileResultset` each on contract files or contact files. In doing so, `lcmCompany` represents a reference field each in both file types in advanced syntax whose key condition is configured on company's `crmId` field.

To ensure that the current company file may be safely deleted, both `FileResultsets` must be empty (otherwise, deleting the company file would result in violating data integrity because the references of the found DOCUMENTS files would be destroyed by deleting the company file).

Moreover, the example illustrates the use of the `delete` command from the normal Javascript language range. This is used to release the memory area used by complex objects. The DOCUMENTS 4 Server's scripting engine allows using this to influence the integrated Garbage Collector.

4.5 Accessing the DOCUMENTS 4 file system

Accessing the file system of the computer where the server is running constitutes a regular requirement that may be dangerous to system stability.

In particular, the File class, integrated into the scripting engine, is popularly used to implement self-defined log files. However, because file system access via this file class is not threadsafe, this requires considerable additional expenditure to guarantee exclusive access to a specific data file in the file system.

Normally you guarantee this, for example, by working with a unique, temporary file name allocated by the server itself per script execution (this approach is also used in the following sample code). Alternately, you could, with jobs, for example, be looking for a property at the beginning of script execution on the principal which, if not found, is immediately created and which, if existent, in reverse results in that the actual script is not executed until the deliberate property has been deleted by the other script.

```
// 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 Dynamically determining enumeration values

If the versions of a field of the "Enumeration" type are not constant, you can also define these via a script. The definition of which script is to be executed is defined as enumeration value through the required `runscript:Scriptname`.

In project experience, this is popular use to maintain an enumeration field with the constantly same enumeration values required in various file types at the same time in a central place. The advantage in this context of implementing this as a script is in particular that subsequent adjustments to the enumeration values will become automatically valid for files that already exist in DOCUMENTS, i.e. no "Change existing files" run will be required. (In reverse, the disadvantage is that such changes are automatically used for all DOCUMENTS files although this is exceptionally not desired).

Drop-down list fields dynamically generated via script should never be displayed in hit lists, because owing to the way in which DOCUMENTS internally handles

multilingual enumeration values automatically causes the enumeration script to be executed by starting an unsuccessful search or by displaying an empty dynamic folder (only this allows the server to determine which ergonomic labels the technical field values of the enumeration field correspond to).

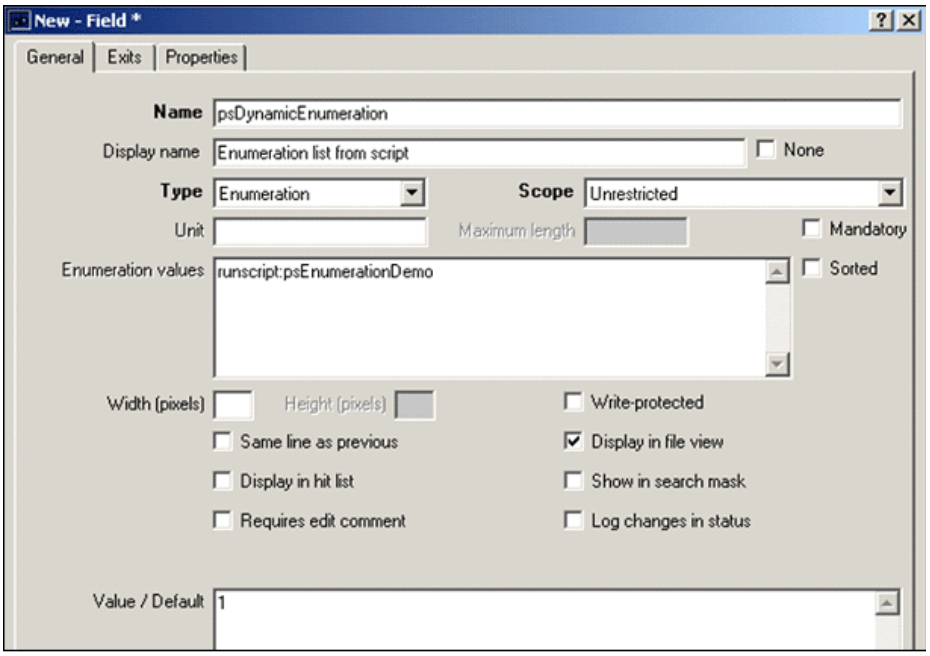
Thus, a dynamically structured enumeration field is automatically a potentially very expensive resource!

Another result from dynamic determination of the enumeration values is that particularly determining the enumeration value as the result of database queries from external data sources may become very resource-intensive. We strongly recommend not to dynamically implement enumeration fields with more than 20 to about 30 enumeration values via script from database queries. Instead, in such cases, the use of the Tabledata Userexits (see documentation on the so-called SQL Taglib), which in addition to careful handling of the resources includes considerable increase in convenience as opposed to a drop-down list, particularly for the DOCUMENTS user, would be recommendable.

Another essential restriction with enumeration scripts is that you may never return your own return value via the `return` statement in such scripts because the array, which the execution context automatically and implicitly contains, named `enumval` is also automatically used as a return value at script end. The following little sample script illustrates the use of `enumval` for enumeration values in practice. In doing so, the enumeration values are deliberately configured in multiple languages:

```
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!
```

Integrating this script into a file field (it is assumed that the above script has been saved as `psEnumerationDemo`):



4.7 Database accesses via scripting

With the `DBConnection` and `DBResultset` classes, the scripting engine provides essential support for accessing (external) relational databases.

The little example below illustrates the basic mechanisms to establish a connection to an ODBC data source to enter a new entry via INSERT command in a database table for logging purposes.

The (fictional) log table named "`ProtocolTable`" includes the `login`, `tstamp`, `role`, `wfstep`, `filekey` and `filetitle` columns (all these are string fields):

```
// 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;
}
```

The basic aspects for database handling include consistent error handling, as well as proper returning of open database handles because these usually are very expensive resources.

The SampleScript below is used as a receive signal of a DOCUMENTS file in a workflow. This sample is an invoice, the document number of which (e.g. a barcode) is used as a primary key in a database table. We assume that a(n) (fictional) ERP system should influence a column named "`BookingState`". When the invoice in the ERP system has been posted, this system should set the posting state of the invoice to the "`booked`" value. The following script then signalizes to the workflow that the invoice file may continue with the workflow, otherwise the process must continue to be in the wait state.

```
// 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;
```

The above sample code illustrates, among others, that DBResultsets should also be closed as early as possible. Moreover, the DocumentNo column from the table is also selected although its value is not considered further afterwards when processing the result. This is due to the circumstance that some RDBMS are available in the market which absolutely expect that the key columns of the WHERE clause must be part of the selection. This trick should usually be unnecessary in modern MSSQL, MySQL or Oracle versions.

4.8 Caching the data of expensive resources

A very regular feature of projects with accesses to external databases in particular is that – as enumeration script, for example – the same information is always read from the connected database tables although this only sporadically changes. Yet this regular access to the expensive "Database" resource is a not insignificant load on the server.

Consequently, you want an option to buffer this data somehow in the DOCUMENTS 4 Server's cache.

Exactly this option is available via the PropertyCache available via scripts. This is always and implicitly available under the name of `propCache` without any other instancing and is normally used for exactly this buffer.

The sample below determines some enumeration values from the Northwind database insofar as the cache named "`Contacts`" has yet to be created:

```
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]);
}
```

Note: To be able to test an enumeration script in the DOCUMENTS Manager, you need to enable the "Script for enumeration values" checkbox on the "Test" tab.

A daily executed Job script (the one-liner below) allows ensuring that the cache is re-populated every day:

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

4.9 User-defined actions on files

Enriching the functional scope of the system in the context of a single DOCUMENTS file with additional functionality has now become part of project practice in the DOCUMENTS environment. A popular option is to define a so-called user-defined action on a file type. This provides the function thus defined to each DOCUMENTS file created on the basis of this file type.

This functionality is regularly used to create reports based on the data saved in the DOCUMENTS file, or to make an export in a specific file format available to the users.

The sample below creates a Docimport-compatible XML export of the current DOCUMENTS file (but without documents that the file may contain):

```
// Check for correct execution event
var event = context.event;
if (event != "fileAction")
{
    context.errorMessage = "Not called as a file action!";
    return -1;
}

// Check for current file
var myFile = context.file;
if (!myFile)
{
    context.errorMessage = "Not in a file context!";
    return -1;
}

// get all fields and their contents
var fieldNames = new Array();
var fieldObj = new Object();
var i = 0;
while (fieldNames[i] = myFile.getFieldName(i))
{
    fieldObj[fieldNames[i]] = myFile.getFieldAttribute(fieldNames[i], "Value");
    i++;
}

// create XML structure
var xmlHeader = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n";
xmlHeader += "<!DOCTYPE files SYSTEM \"files.dtd\">\n";

var myXml = new XML("<files />"); // Container-Tag, MANDATORY!
var fileXml = new XML("<file />"); // new single file
fileXml.@type = myFile.getAutoText("fileType");
var owner = myFile.getFileOwner();
if (owner)
    fileXml.@owner = owner.login;

```

An initial check is made here on whether the script has been started in the correct context. This illustrates another form of restrictive programming because, being a

script developer, you cannot necessarily ensure that the recipient of your work actually uses the script correctly.

Things start to get interesting from line 18 onwards: The technical names of all fields that the DOCUMENTS file contains are generically (i.e. regardless of file type!) read at the same time, and their current field values are also temporarily buffered in generic form and as a string. The advantage of this procedure is that you do not have to deal with the different data types of different field types; above all, the script can be used in virtually unchanged form on any file types.

The actual building of the XML structure starts from line 28. In doing so, the script reverts to an extension of the Java script language range named E4X (EcmaScript for XML) which can be used in DOCUMENTS.

You will find an in-depth basic article in the magazine i'X, issue 11/2005, published by Heise Verlag, which very avidly treats the aspects of this script extension for the DOCUMENTS environment. Additional information can be found using the popular search engines.

After creating the header data of the DOCUMENTS file, the previously temporarily buffered field data must be processed. This is illustrated by the second part of the source text displayed below:

```
37
38  for (var field in fieldObj)
39  {
40      // new field
41      var fieldXml = <field>{fieldObj[field]}</field>;
42      fieldXml.@name = field;
43      // add field to xml structure
44      fileXml.field += fieldXml;
45      // free memory
46      delete fieldXml;
47  }
48
49  // place file xml structure into container
50  myXml.file += fileXml;
51  var outputString = xmlHeader + myXml.toString();
52  context.returnType = "download:XMLexport.xml";
53  return outputString;
54
```

Here, too, a preferably generic procedure is desirable to keep individual customization expenditure as low as possible.

Eventually, the individual fields are added one after another to the complete XML structure, the completely built file XML is loaded to the container and the result is converted to a string using the XML header required for Docimport.

This is then provided to the user for download under the suggested file name XMLexport.xml.

This script must then be integrated on the "User-defined actions" tab of a file type. You can choose whether you want the action with a name and label defined

by you as a button (layout as with workflow action) or as an entry in the Actions drop-down list.

4.10 User-defined actions an folders

A popular requirement in the DOCUMENTS environment is selecting a selection of processes (read: DOCUMENTS files) from a folder and exporting the information saved therein in some format.

The sample script below has been taken from the current ongoing development of CONTRACT; it illustrates export of deadlines in iCal format to be able to continue using the deadline data in Outlook or a desktop calendar program:

```
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");
```

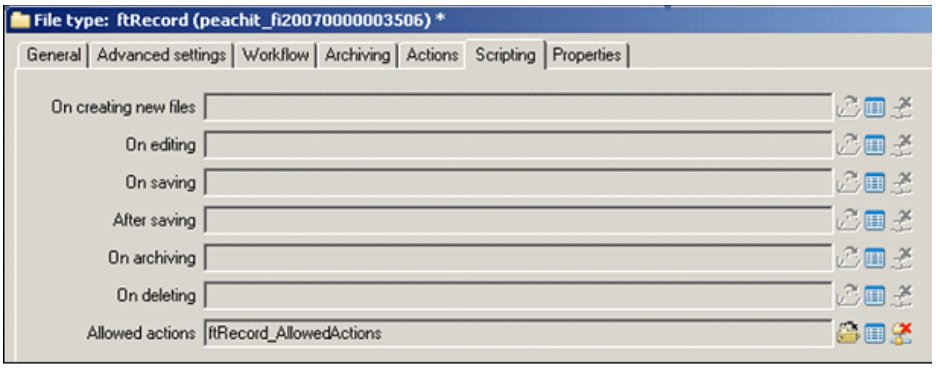
The script has been entered as a user-defined action on the deadline calendar folder in CONTRACT.

4.11 Permissioning user-defined actions

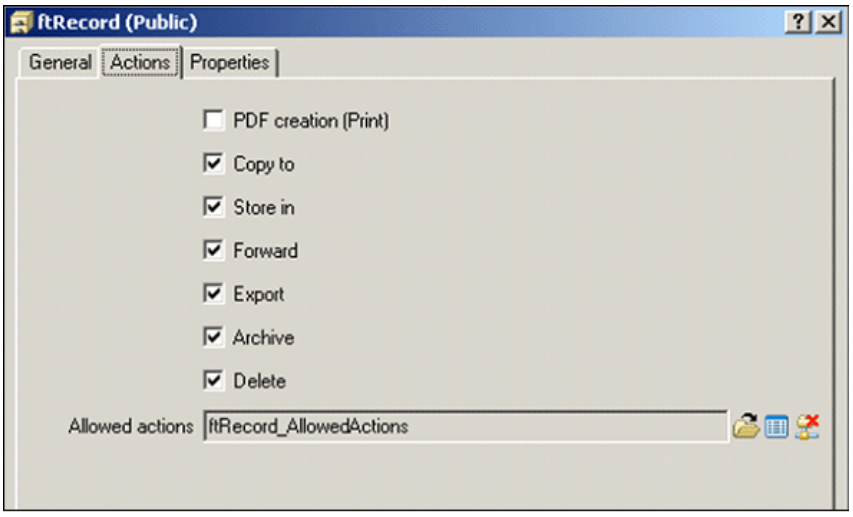
After we have become familiar in the previous two sections with how to extend the Web interface's functional scope through user-defined actions, the follow-up question immediately arises on how to ensure that specific user-defined actions are available for specific user groups, i.e. we are looking for an option to permission the buttons or action drop-down list entries that we have created ourselves.

Such permissioning can be performed both on the file type and on public folders using a specific script that must be entered as an "Allowed actions" script.

Use on the file type:



Use on public folder:



A so-called `AllowActions` script always has an identical structure that results from the fact that the scripting engine automatically provides an array with the technical names of all user-defined actions you entered on the file type or folder.

This array named `enumval` must iterate through a `for()` loop and you must compare the value of each entry with the action name you are looking for.

Once you have found the right action, you check the respective permission that you want. Inasmuch as the result of this check is that the the user should not be able to use this action, you only need to remove the current entry from `enumval`.

A simple sample script that illustrates exactly this general structure could look as follows:

```
// check all userdefined actions
for (var i = 0; i < enumval.length; i++)
{
  // show/hide approval-button
  if (enumval[i] == "btnApproval")
  {
    // hide Button, if current user is not member of privileged group
    var currentUser = context.getSystemUser();
    if (!currentUser.hasAccessProfile("SUPERADMINS")
      enumval[i] = "";
  }
}
```

Please note that it generally always iterates through the loop; the desired action is found within the loop and rights validation is only performed within the searched-for action.

The advantage of this general structure is that you actually execute potentially "expensive" rights validations only when the action to be permissioned in this way is actually present.

Another special feature is the circumstance that the script itself does not contain a `return` statement. As with enumeration value scripts, the `enumval` array implicitly represents the return value; specifying a `return` statement on your own is therefore strictly forbidden.

Action buttons and drop-down list entries do not need to be differentiated. In case both the one and the other have been defined on the file type or folder, the `AllowedAction` script will be executed for each action type once. So, please only make sure that your user-defined actions are always allocated unique and programming language fit technical names!

4.12 Run script as a job

Scripts are frequently used for automating regular recurring tasks performed without user interaction, e.g. automatic archiving of processes. This type of task can be very easily implemented in the form of so-called Job scripts. The following example illustrates how such a Job script is implemented, based on automated archiving of deprecated processes.

To do this, define a script using the following source code:

```
// 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;
```

This script illustrates again the functions for date manipulation and how to filter DOCUMENTS files based on date comparison operators. Our example filters all DOCUMENTS files containing a file field named "Expiration date" and whose value is more than 90 days prior to the current date (Caution! Empty date fields are automatically less than the current date!). All DOCUMENTS files found through this filter are now archived in sequence, and removed from the system. Should an error occur here, the script will terminate with an error message saying which DOCUMENTS file has caused terminating the job for which reason.

The script requires a user context, because otherwise there will be no access to DOCUMENTS files. Normally, you enter the matching job user in the DOCUMENTS settings (this user must, of course, have enough permissions on all DOCUMENTS files which you want to influence through job control).

Alternatively, you can overwrite this job user for a single job script by entering the desired login name on the "Test" tab of the respective script.

If you want to understand this example, please add a date field named "Expiration date" to one of your file types and create some new DOCUMENTS files, some of

whose expiration dates are more than 90 days in the past. You can significantly simplify this task when assigning the expiration date the auto-text "`%currentDate - 91%`" as the value/default setting.

In the script code, customize Line 8 in such a manner that it filters on the file type you selected (this example refers to the "Default" file type of the old toastup demo principal). You can then test the job script.

4.13 Keeping the file pool populated via JobScript

The file pool is a specific temporary buffer in DOCUMENTS where all DOCUMENTS files deleted by the file type used in the system and discarded working copies are temporarily stored. If a DOCUMENTS file of a specific file type is created, DOCUMENTS will preferably take a file from the file pool because this takes place at a significant faster rate than completely creating a new file object on the database.

The file pool, however, is also used through automated file creation such as through Docimport of the DOCUMENTS Factory; the result of this may be that the file pool will almost permanently be empty if a large number of new DOCUMENTS files are generated every day through these automatic imports. The users will feel, through considerably tougher handling of the system, that the file pool is empty because in this way new files and working copies must necessarily be generated directly on the database.

For this reason, there are two methods named `countPoolFiles()` and `createPoolFile()`, which allow a Job script to automatically re-populate the file pool every night when server load is low.

The script for this looks as follows:

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

The script will check for the file type "Default" how large the file pool is at runtime in order to fill up the difference result set up to a threshold (in this case, 3000). This threshold, of course, depends on the quantity of new DOCUMENTS files of each individual file type created on average per day; it should be individually customized.

4.14 Decisions and guards in the workflow

When designing workflows you will often encounter the problem that the comparison operators available by default for creating a condition (so-called guards) are no longer enough, e.g. when calculations are required for validation, or sophisticated, linked Boolean algebra is required. In this case, too, scripts will help.

Guard scripts can be used in different functions within the workflow engine. On one hand, they are used as a condition on a decision shape, while on the other they can (without having to customize a single line of script code) be used as so-called constraint checks on control flows emanating from an action (task for user or groups).

Guard scripts are generally executed within the context of a DOCUMENTS file (which is available via `context.file`). In case of a constraint script, the user running the script is always the user who has selected this control flow in the Web interface.

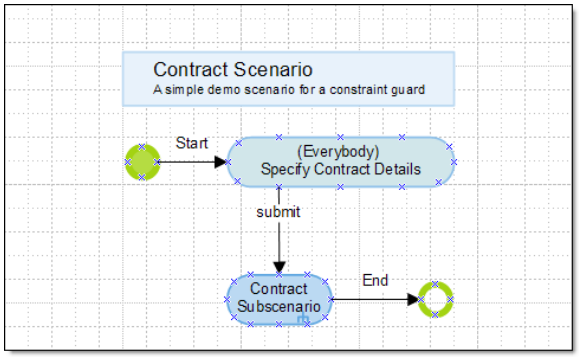
In case of a condition on a decision, script execution usually takes place within the context of the user who last clicked the DOCUMENTS file, i.e. who triggered the last manual forwarding action of the file. An exception to this, however, is the situation in which the DOCUMENTS file is delayed by a Delay shape or through a receive signal before the decision is checked. In that case the Guard is executed within the job user's context.

A very popular request for a constraint script is - for example, in contract management scenarios - a check on whether documents have already been added to the contract file. Submitting the process to the release process is to be prevented if the DOCUMENTS file does not contain any documents yet.

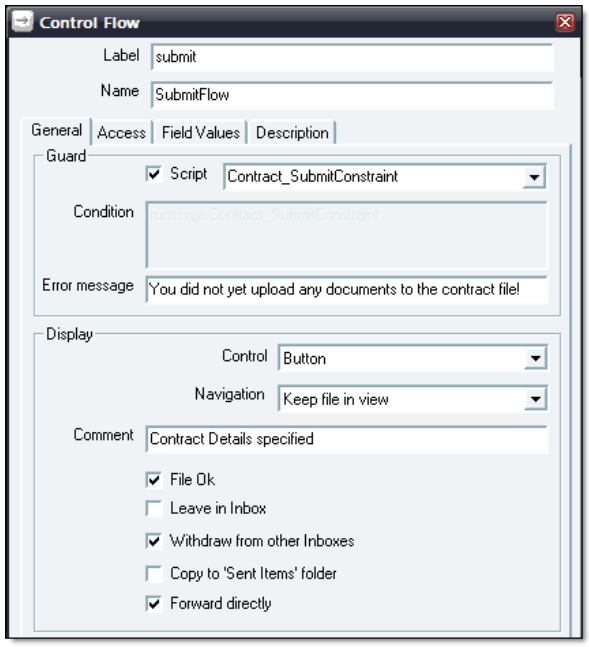
Such a script could, for example, have the following structure:

```
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;
```

This script is stored on the server under the name "`Contract_SubmitConstraint`", for example, to be able to integrate it into a workflow in Visio afterwards. Let us assume the following workflow with a simple structure is used:



For convenience we assume the actual release scenario of the contract is encapsulated within the subworkflow. Only the marked task and the control flow with the "`submit`" label emanating from it is relevant to our script.



As you can see in the above screenshot, our script has been configured as a guard. Because this is a simple check, in our example, the error message that will be output when a document is missing from the DOCUMENTS file has been hard-wired in the control flow itself. Alternately, you might store your own error message in `context.errorMessage` before ending the script via `return 0;`.

This simple practical example additionally illustrates that guards and constraints only permit two specific return values – `return 0;` means the condition is not met, while `return 1;` means it is met.

4.15 Receive signals in the workflow

In project experience, many workflow scenarios require that the DOCUMENTS files contained in the workflow be left waiting at a specific point for an external event to occur. This, for instance, occurs very frequently on purchase invoice scenarios, where waiting to complete posting an invoice in the customer's ERP system within the workflow is part of the customer's requirements.

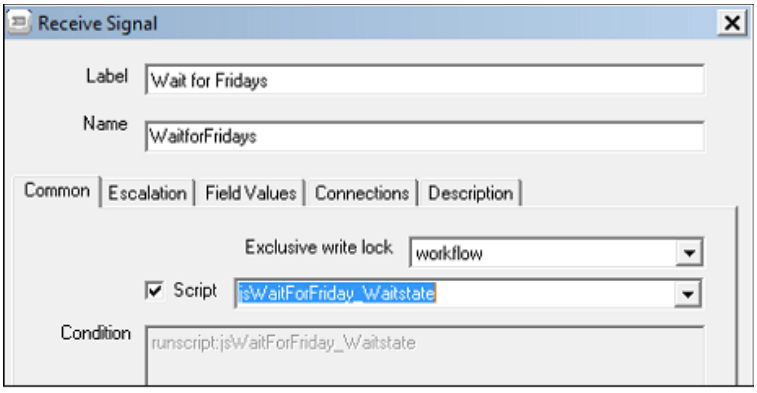
This wait state is usually modeled in receive signal format within the workflow. The syntactic restrictions of the definable conditions, however, frequently require that you no longer define the signal condition inline, but as a separate script.

Such a receive signal script can ALWAYS access the DOCUMENTS file in wait state via `context.file`. The other script code is completely arbitrary; only the allowed return values are restricted to the two Integer values 0 (this condition has not occurred so far) and 1 (the signal condition has been met and the workflow can continue iterating).

A very simple example, for instance, enforces that the workflow on the DOCUMENTS file may continue only on Fridays:

```
1  var today = new Date();
2
3  var result = 0;
4
5  // true if currentDate is Friday
6  // 0 is Sunday, 6 is Saturday
7  if (today)
8  {
9      result = (today.getDay() == 5) ? 1 : 0;
10 }
11 return result;
12
```

Integration of this small script into the workflow could then look as follows:



4.16 Send signals in the workflow

Requests of the type that the data saved in the DOCUMENTS files must be exported to third-party software in any format can often be found in workflow-aided DOCUMENTS scenarios. Depending on which interfaces the third-party software provides for this, these exports can be performed by using a temporary database. Increasingly, however, ever more software manufacturers also provide interfaces such as Web services or interfaces for an XML import.

As a rule, the only snag behind these interfaces is in the structure of the data required by the third-party provider to be exchanged via these interfaces.

Here the circumstance that the send signal in DOCUMENTS workflows can also be used to run any script is helpful. The code can then generate any export format, as it were.

We have already introduced detailed sample code of output in an SQL database or of creating any separate XML export format in the previous exercises.

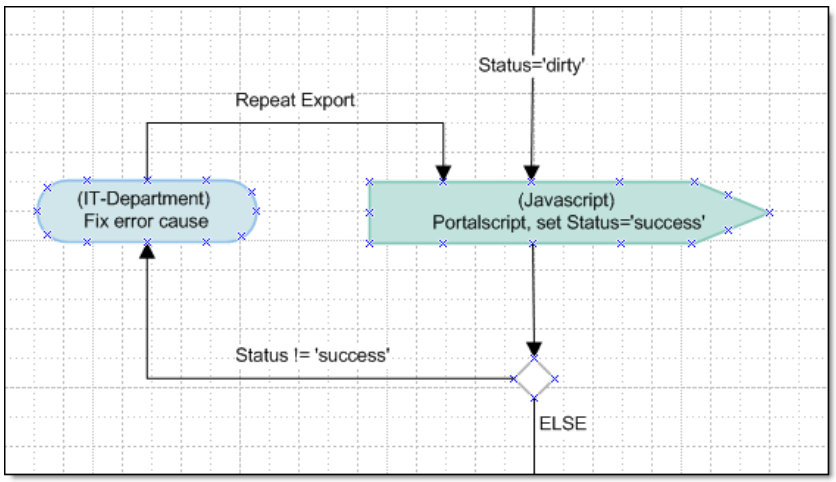
To guarantee trouble-free export to third-party system, you only need to consider more specific workflow modeling because script send signals are completely iterated even in case of a script error.

Setting a status flag (e.g. "dirty") in the DOCUMENTS file **prior to** script execution of the send signal has proven very effective.

Only with **successful** processing of the complete script of the send signal will the script then set this status flag to a different value as the last statement (e.g. "success").

Integrating a decision directly after the send signal which checks the status flag lets you respond to possible errors in script processing and send the DOCUMENTS file to a group of administrative users who can troubleshoot the error cause and re-trigger the export.

Sample implementation of this query could look as follows:



4.17 loginscript, afterLoginScript, setPasswordScript

Three principal properties allow intervention with the DOCUMENTS login mechanism. These provide the respectively separate additional environment variable implicitly at execution time:

- **loginScript** – executed **prior to** user login

login	user login
password	password entered by user, in plain text
source	"SOAPAPI", "instance", "unit", "asUser"

- **afterLoginScript** – will be executed **after** successful login

login	user login of user who has just logged in

- **setPasswordScript** – will be executed **prior to** changing password

login	user login
oldpassword	value of the input field "Previous password"
newpassword	value of the input field "New password"

The following small example of a `loginScript` only checks the current weekday. For security reasons logins are only permitted from Monday to Friday; login is denied on Saturdays and Sundays:

```
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
```

The following example of an `afterLoginScript` defines a counter as property on the user which can be used to log the number of successful logins. This, for instance, would enable statistical evaluation of regular use of the system via another script:

```
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);
}
```

The example below of a `setPasswordScript` extends the password check integrated into DOCUMENTS with a test that users may not set their own first name as the password:

```
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

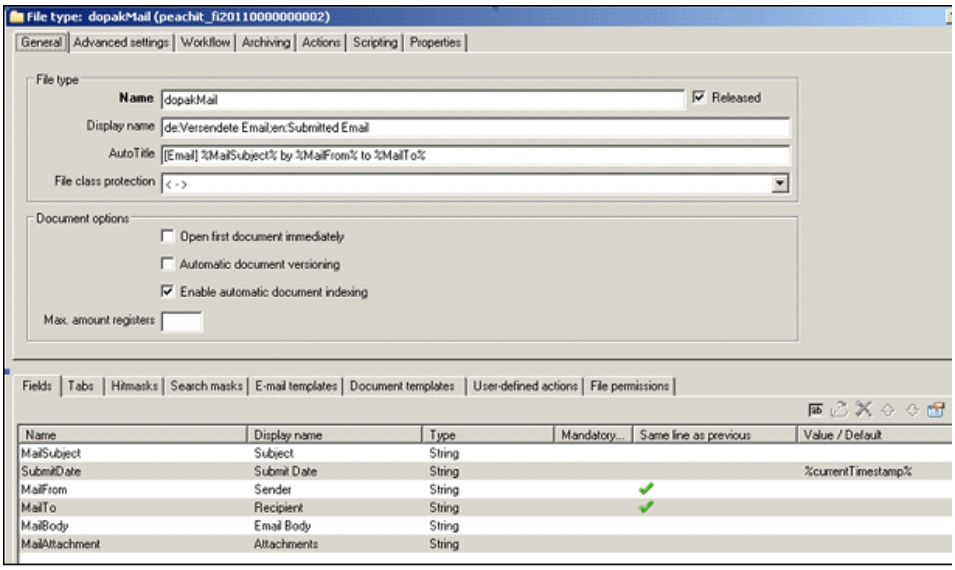
A frequent point of criticism of ad hoc e-mail distribution integrated into DOCUMENTS is that the information sent via e-mail is virtually not logged. Although you can see in file status **that** en e-mail message was sent from the DOCUMENTS file, you do not learn which contents that message had.

Whereas using a simple file type and the script described below allows implementing complete logging.

To do this, you need to create the afterMailScript property on the `file type`; the value must be the name of the script to be executed after sending the e-mail message. The following environment variable will then be implicitly available within this script:

- mailSubject Enter subject as with sender in mail dialog
- mailFrom Sender's e-mail address
- mailto Recipient(s)'s e-mail address
- mailBody E-mail content, as entered in the Send dialog
- mailAttachments Names (!) of the documents sent via e-mail

A possible structure of the file type created for logging could then look as follows:



A useful extension of this file type might be establishing a direct reference between e-mail files and the original files initially sent via mail using a reference field. This approach, for instance, has been consistently chosen in the solution package named RELATIONS to render the complete Support Ticket scenario highly transparent to all employees involved.

The source text of an AfterMailScripts matching the file type documented in the screenshot might then look as follows:

```
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 AccessScript on the file type

The DOCUMENTS rights mechanism absolutely enables very far-reaching and flexible restrictions of accessing a single DOCUMENTS file; using file class protection, ACLs or GACLs also enables permissioning at the level of file contents.

Yet some projects regularly require a restriction of write permissions on a DOCUMENTS file at content level when a larger group of users should at least have read permissions on the file.

DOCUMENTS allows implementing such requirements using a hidden property on the file type named "AccessScript", which then requires the name of the script to be executed.

Such an AccessScript is executed every time read access to a file is attempted; it can check any conceivable condition (in doing so, it is imperative that you consider careful resource procedure, because this event is one of the extremely expensive resources!).

An AccessScript provides four different rights in the form of the familiar array named `enumval`:

"DlcFile_RightRead", "DlcFile_RightWrite", "DlcFile_RightChangeWorkflow" and "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
```

The previous page shows the basic structure of an AccessScript in the form of the sample code. You need to always consider particularly Lines 25-38 in the displayed form; you must never forget them.

The actual condition, in the example, only a check on whether the currently logged-in user is the owner of the DOCUMENTS file or not, can be found in code lines 19-23. You can expand these conditions in any format.

4.20 Extending script classes

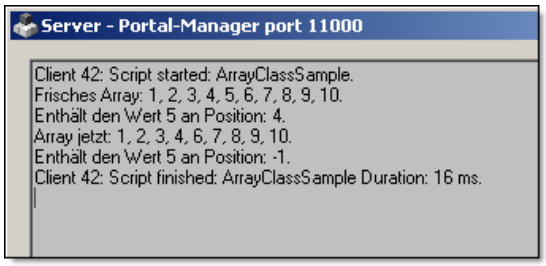
Compared to other high-level languages, developers occasionally miss this or that function in the language range which, however, is very popular in other languages and would significantly increase convenience in programming. You usually program the missing functionality yourself, though you will then be faced with the problem of unique allocation of the helper function xy() to the correct date type. JavaScript, however, provides a convenient mechanism here to enrich the existing classes with additional functions.

This is performed by using the keyword prototype in the function declaration. The following example extends the Java script array with the two regularly searched-for method `inArray()` and `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 }
```

The small code below illustrates the use based on a simple array with ten elements (you simply enter the numbers 1-10). The system initially outputs whether the element is saved with the value 5 in this array (yes, it is). Then this particular element is deleted from the array (Line 7) and the system in turn outputs whether the element named "5" is still found (no, it is no longer found now).

```
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 files

Occasionally, exactly no or exactly one DOCUMENTS file of a specific file type is to be allowed in the entire system (or user context). A classic example of this requirement, for instance, is the configuration file of the DOCUMENTS-LDAP interface, or configuration in CONTRACT v2.

To guarantee simple retrievability of these files, aka "Singleton", the option to link a script solidly with a dynamically filtered folder is available. Instead of the usual generation of a hit list on clicking a folder name in the folder tree, the stored script is executed which then ensures uniqueness of the DOCUMENTS file being searched for.

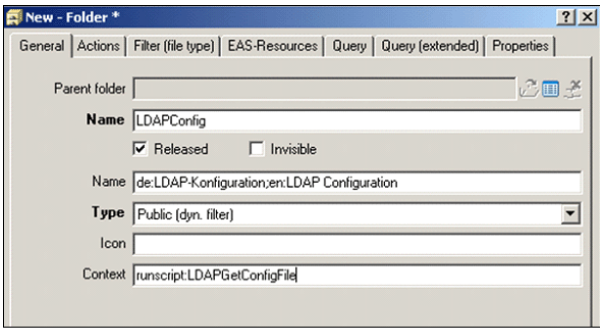
Such a script might be structured as follows:

```
1  var myFRS = new FileResultset("LdapConfiguration", "", "");
2  var myMappe = null;
3  if (!myFRS || myFRS.size() <= 0) {
4    myMappe = context.createFile("LdapConfiguration");
5  } else {
6    myMappe = myFRS.first();
7    delete myFRS; // free memory!
8  }
9  if (! myMappe) {
10   context.errorMessage = "Could not read file!";
11   return -1;
12 }
13 return myMappe.getAutoText("id");
```

Explaining how it works: An attempt is made to generate a FileResultset on LdapConfiguration files within the context of the current user. Inasmuch as this should be unsuccessful or, inasmuch as the created FRS does not contain any hits, the script will create a new DOCUMENTS file. Whereas if the script finds any hits, it will return the first file found.

The example deliberately contains two inadequacies: First it is assumed that the current user has at least read and create permissions on the file type. Secondly, the script neglects the potential existence of more than one LdapConfiguration file visible to the user.

The script will then be bound to the desired folder as follows:



It is a good idea to configure the folder's filters exactly to match the file type used in the script. This will ensure retrievability.

4.22 Downloading binary files via user-defined action

The problem has previously been that on the server side automatically generated binary files such as PDFs or automatically created Microsoft Office documents could not be used as a return value of user-defined actions. The `context.returnType "file:filename.ext"` that is usually used for this can generally only be used with text information because the output within the script must always be read into a string variable.

In the cases described above, the only option previously available was of processing in an upload as a new document in a DOCUMENTS file – and the user then had to manually download the document.

The option to actually download the document directly is available. To do this, the desired document must be stored in a directory that the server can read.

Der following code illustrates this use:

```
// first, find current file
var myMappe = context.file;
if (!myMappe) {
  context.errorMessage = "Not in a file context!";
  return -1;
}
// now get the documents tab
var reg = myMappe.getRegisterByName("Documents");
if (!reg) {
  context.errorMessage = "Register not found!";
  return -1;
}
// find the desired document
var docIter = reg.getDocuments();
if (docIter && docIter.size() > 0) {
  for (var doc = docIter.first(); doc; doc = docIter.next()) {
    if (doc.fullname == pDocument) {
      // pDocument is a Script parameter - if we find it, return the file
      context.returnType = "download:" + doc.fullname;
      return doc.downloadDocument();
    }
  }
}
```

The example requests a script parameter named `pDocument`. All documents stored in the current DOCUMENTS file are then examined on whether their file name matches exactly the user's input (when demonstrating this feature at DoPaK 2009 an enumeration script in the parameter dialog ensured that the user always had to select a correct file name).

Inasmuch as the desired document is retrieved, the script downloads it into the temporary directory of the DOCUMENTS 4 Server and returns it to the user's browser.

The trick is in combining code lines 19 and 20. The matching `returnType` is defined first. The temporary download, which automatically returns Path+Filename of the temporary file as a return value, is performed in Line 20. This is then used to return the document to the client via the `return` statement.

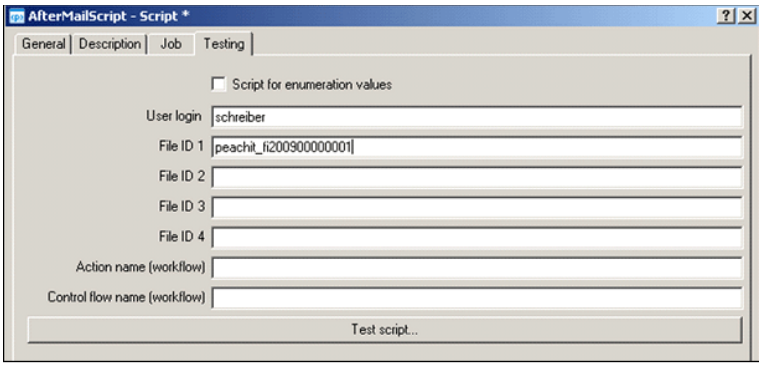
The disadvantage of this example is that the `temp` directory on the server is gradually populated with shreds. So, in practical projects you should ensure that this directory is regularly cleaned.

5. Testing, Debugging and Encrypting

5.1 Testing scripts

To test scripts regardless of DOCUMENTS files and workflows, test scenarios can be defined on the scripts. To do this, you need to define file and user and workflow parameters, and create the corresponding fields with field values.

The action button *Test script...* can then be used to run the script. After ending the script a dialog of return values and field values are displayed.



5.2 Extending log output with script executions

The configuration file named `partnernet.ini` lets you configure advanced logging of executing scripts. To do this, the button named

```
$ScriptLog 1 is used
```

After enabling this option and restarting the DOCUMENTS 4 Server, each execution of a script will be logged in the server window (or its log file).

In doing so, two additional output lines appear in the server output per script. At the start of execution the running script is referred to, and on ending the script another line is output which refers to the fact that the script named XY was ended after ABC milliseconds of processing time.

This advanced logging is particularly useful for profiling purposes and when searching for possible event cascades of scripts.

Instead of 1, you may also configure Path+File name to a CSV file writable by the DOCUMENTS 4 Server. In this CSV file, the script name, execution duration and, where necessary, occurring errors are then logged per execution.

5.3 Customizing script execution parameters

By default, the script engine's server provides memory area of 4 MB (4096 KB) per script execution. In specific cases, e.g. when a job script processes a large number of processes at the same time or when DOCUMENTS files with very extensive Gentable data are manipulated via E4X, this allocated memory area may not be enough. In such cases, the respective script would be canceled with an "Out of memory" error message. You can remedy this by customizing the `JSMaxMemory` 4096 in the server directory's

```
partnernet.ini file
```

accordingly. We point out here that the default setting of 4 MB has previously been proven in practice.

The scripting engine additionally contains some specific check routines to prevent execution of potential endless loops and, with it, inadvertent paralysis of the DOCUMENTS 4 Server. Branchings, iterator loops and recursive function calls are actually audited. These audits should normally be left untouched because they essentially contribute to the scripting engine's stability. However, in specific situations, individual protection can be optionally extended or turned off completely in the server directory's `partnernet.ini` file using the following three parameters:

```
JsMaxBranch          Set # to 0 to turn off checking
JsMaxIterator Set # to 0 to turn off checking
JsMaxRecursion       Set # to 0 to turn off checking
```

Caution: Turning the security mechanisms off completely is not recommended on production systems!

5.4 Debugging using the Script Debugger

A licensed version of the JSRemote tools provided by otris (now offered as JANUS Script Debugger) gives you the option to debug your scripts in similar fashion as in modern development environments.

In doing so, the JANUS Script Debugger provides, among others, a single step mode incl. setting breakpoints; moreover, you can define so-called watches, i.e. audits of variable values and expression.

For a detailed description of how to handle the Debugger, please read the product documentation referring to this.

5.5 Encrypting scripts

The option to automatically encrypt scripts via a maintenance operation is available. Encryption has no gauging impact on processing speed of scripts handled in this way; encryption is only used to protect your intellectual property or against inadvertent changes to potentially dangerous codes.

The maintenance operation `encryptScripts:filter` is started as usual via the client from the "Administration > Perform Maintenance Operation" menu item. You can also use wildcards as `filters`, e.g. to specifically encrypt all scripts en bloc with a common prefix (example: `encryptScripts:crm*`).

In addition, another maintenance operation named `encryptMarkedScripts` exists. This is used to automatically encrypt all scripts whose source code contains the specific comment statement

```
//#crypt
```

This specific comment does not necessarily need to be entered as the first code line of the source code; it can also appear in the later course of the script. In that case, the first part of the script remains readable in plain text, and customizable; only the code after the preprocessor statement occurs will then be encrypted.

This circumstance can be used to provide the user / customer with a configuration scope, but to encrypt the actual script functionality for security reasons.

> *Warning: The encryption cannot be undone!*

This means you are not capable of channeling scripts that have once been encrypted back to their unencrypted form! This is why it is highly recommended that you create an unencrypted backup copy of the original script prior to its encryption!

In addition, you need to consider that debugging encrypted scripts is not possible. In particular, this also means that scripts importing encrypted libraries cannot be analyzed using the ScriptDebugger.