Coding Styles in ADITO

AID 001 DE

Autor: FA, MW, JK, KN. Version 10.8. Zuletzt geändert 27.09.2017

Version	Änderungen
10.8	Kapitel zu JSDoc eingefügt
10.7	Anpassung der Formatierungen
10.6	Allgemeine Beispiele zur Variablendeklarationen ergänzt
10.5	Hinweis auf die Others-Dokumente hinzugefügt Hinweis für die Tabreihenfolge eingefügt
10.4	Beschriftung von Ribbon-Buttons eingefügt
10.3	Logging, Coding und Debugger eingefügt
10.2.2	Coding Guidelines für Schlüsselwörter eingefügt
10.2.1	Hinweis auf Sprache von 4.13 nach 4.6 verschoben “4.4 SQL Querys” erweitert Hinweis bei “4.5 Anweisungsblöcke” ergänzt Hinweis auf onSave im Designer (4.15)
10.2	Standardwerte für Checkboxen geändert
10.1	Coding Guidelines für Strings eingefügt
10.0	Anpassung an ADITO4.6
5.2	Erweiterungen nach ADITO-Friday vom 01.07.2016
5.1	Version zur ersten Freigabe als Basis für ADITO-Customizing

Inhaltsverzeichnis

1. Allgemein

1.1. Zweck dieses Dokuments

Warum eine Konvention für die Gestaltung von JDito-Code und die Benennung von Komponenten? Konventionen dieser Art haben sich im Bereich der Softwareentwicklung seit Jahren bewährt und sind aus einer Reihe von Gründen vorteilhaft:

- Code wird sehr viel häufiger gelesen als geschrieben. Daher sollte das Lesen durch gemeinsame Regeln für die Gestaltung des Codes so einfach wie möglich sein.
- Konventionen verbessern die Lesbarkeit von Code und reduzieren den Aufwand für die Einarbeitung in fremden Code.
- Konventionen befreien von unnötigem Nachdenken über formale Details und sind gerade für typschwache Sprachen von großem Vorteil.
- Konventionen verdeutlichen das Bekenntnis eines Unternehmens hin zu standardisierten Abläufen und damit einer Erhöhung der Produktqualität.
- Namenskonventionen für Komponenten erleichtern die Arbeit im Team, wenn sich JDito-Code auf Komponenten bezieht.
- Gleiche Variablen- und Komponentennamen erlauben das Hinzufügen von Frames zu einer bestehenden Installation mit minimalem Aufwand.

Diese Coding Guidelines gelten auch als Grundlage für die ADITO internen Code Reviews.

1.2. Objektnamen

Die Namen von ADITO-Objekten (Frames, Reports, Prozesse, etc.) sollten möglichst in Englisch gehalten werden. Grundsätzlich gilt:

- Name eines Frames = Name der verknüpften Datenbanktabelle (CAMPAIGN)
- Funktionsbibliothek (Prozess, der nur Funktionen enthält und keinen sonstigen Code) = Name beginnt mit lib_
- Serverprozess (Prozess, der nächtlich in bestimmten Abständen am Server ausgeführt wird): Name beginnt mit sp_
- ADITO Mobile (Prozess für die Arbeit mit ADITO4 mobile): Prozessname beginnt mit weptun_

2. Komponenten

Für alle Komponenten, die in JDito-Code oder Formeln benutzt werden, sollte eine einheitliche Art und Weise der Namensgebung üblich sein. Dies erlaubt die leichtere Einarbeitung bzw. Übernahme von fremdem Code und macht Formeln und JDito-Code wartbarer.

Vor allem beim Arbeiten mit der Versionsverwaltung GIT, welche bei ADITO verwendet wird, sollte auf die Bezeichnung der Komponenten geachtet werden.

Hier kann es zu Problemen beim Merge kommen, die oft schwer zu lösen sind und eine Absprache mit den Entwicklern bedeuten, welche in diesem Fall unnötig ist.

Beim Hinzufügen von Komponenten schreibt der Designer zunächst die Bezeichnung der jeweiligen Komponenten (“Editfeld”) und ergänzt diese um die Anzahl der bereits verwendeten Komponenten mit diesem Namen (Editfeld1, Editfeld2, …) da der Name einer Komponente in ADITO eindeutig ist.

Ein Beispiel

Anna fügt ein Edit-Feld ein und vergisst die Benennung anzupassen. Sie schreibt einen entsprechenden `valueProcess` und hinterlegt eine Hintergrundfarbe.

Maximilian fügt in seiner lokalen Revision ein Edit-Feld hinzu und benennt dieses auch nicht um. Er hinterlegt einen `valueProcess` und ein Suchkriterium.

Beim Merge werden nun zwei völlig verschiedene Felder betrachtet, und wenn Anna sich z.B. entscheidet ihre Revision zu nutzen, würde trotzdem die Kennzeichnung des Suchkriteriums im AUTO-Merge hinzugefügt werden - das Feld von Maximilian würde ganz verschwinden.

Und wenn jetzt auch noch Prozesse nachgelagert in der Revision von Maximilian schon auf die Komponente zugreifen, müssen diese ebenfalls angepasst werden. Es zieht sich somit schlimmstenfalls durch einen ganzen Commit!

2.1. Präfix

Komponenten, die nicht mit einem Datenbankfeld verbunden sind, tragen zu Beginn des Namens ein Präfix. Danach folgt der restliche Name. Ein Unterstrich zwischen Präfix und Name muss zur Trennung verwendet werden.

Präfix	Komponente
btn	Button
cmb	Combobox
chk	Checkbox
clr	Farbkomponente (color)
doc	Dokumentenmappe

Präfix	Komponente
docv	Dokumentenansicht
dpic	Datenbankbild
edt	editfeld
lst	Listbox / Listenfeld
lbl	Labelkomponente
lup	Lookup-Komponente
memo	Memo-Komponente
rb	Radiobutton
reg	Register
regtab	Registerreiter
spic	statisches Bild
tbl	Tabelle
tt	Treetable
gnt	Gantt
dv	Datenvisualisierung
bro	Browser

2.2. Komponentennamen

Falls eine Komponente mit einem Datenbankfeld verknüpft ist, stimmt der Name der Komponente mit dem Datenbankfeld überein (Rechtsklick auf die Komponente, „Name aus Datenbankfeld übernehmen“). Ansonsten sollte aus dem Namen der Komponente der Zweck hervorgehen. Der Name sollte so eindeutig und lang wie nötig, aber so kurz wie möglich gestaltet werden.

Falls mehrere Komponenten logisch gesehen zusammengehören, sollte nach dem Namen eine laufende Nummer (ohne einer führenden Null) folgen, wie beispielsweise in `edt_Name1`, `edt_Name2`. Bei einer alphabetischen Sortierung der Komponenten sind die Gruppierungen dann leicht zu erkennen.

2.3. Beschriftungen von Ribbon-Buttons

Die Beschriftung eines Ribbon-Buttons beginnt

- Mit einem Kleinbuchstaben, wenn die Beschriftung mit einem Verb, Adjektiv oder Artikel beginnt (z.B. `neue Aufgabe anlegen`, `kopieren`)
- Mit einem Großbuchstaben, wenn die Beschriftung mit einem Nomen oder Eigennamen beginnnt (z.B. `Artikel anlegen`, `Homepage öffnen`)

2.4. Tabreihenfolge in Frames

Die Tabreihenfolge ist immer so anzupassen, dass eine sinnvolle Abarbeitungsreihenfolge der Komponenten entsteht. Zusammengehörige Felder, wie zum Beispiel Adressfelder sollten nacheinander angesprungen werden und keine anderen Felder dazwischen liegen.

3. JDito-Bezeichner

3.1. Variablendeklarationen

Jede Deklaration einer Variablen in JDito sollte in einer eigenen Zeile erfolgen. Dies ermöglicht einen Kommentar für jede Variable mit minimalem Aufwand. Daher ist die Darstellung

```
var query;  // String mit der SQL-Abfrage
var resultCount;  // Anzahl der Datensätze in der Tabelle
```

dieser vorzuziehen:

```
var qry, count;
```

Die Deklarationen müssen sinnvoll gruppiert werden.

3.1.1. Schreibweise

Die Schreibweise sollte sich an der Java-Konvention orientieren. Dies bedeutet, dass bei einem Variablen- oder Funktionsnamen das erste Wort des Bezeichners kleingeschrieben wird (lowerCamelCase). Weitere Worte werden ohne Unterstrich und mit dem ersten Buchstaben in Großschreibung ergänzt. Die folgenden Beispiele sollen dies verdeutlichen:

```
passwordExpiration  lineCount
getUserFirstname()  setUserFirstname()
```

Es ist darauf zu achten, bei Funktionen eine entsprechende Vorsilbe für die Datenrichtung anzugeben. Alle Funktionen, die Werte lesen, erhalten ein `get` als Präfix, alle Funktionen, die Daten ändern, ein `set` (Beispiel: `getMailCount` und `setMailCount`). Handelt es sich weder um get- oder set-Methoden, so sollte ein sinnvoller Bezeichner gewählt werden (`importSourceCSV`). Hier gilt: Lieber lang und verständlich als möglichst kurz und kryptisch.

3.1.2. Initialisierung

Variablen sollten sofort bei der Deklaration initialisiert werden. Nur wenn der erste Inhalt einer Variablen durch eine Berechnung oder eine Datenbankabfrage gebildet wird, die nicht direkt bei der Initialisierung stehen kann, ist eine direkte Initialisierung nicht nötig.

3.1.3. Konstanten

Die Namen aller Konstanten (Variablen mit festen Werten, die sich nicht ändern) werden großgeschrieben. So ist aus dem Namen sofort ersichtlich, dass es sich um einen konstanten Wert handelt. Beispiel: `MAX_LOGINS`.

3.1.4. Beispiele zur Deklaration

```
var tableCount = 13;
const MAX_LOOP = 5000;
var dataObj = {};
function getEmployeeNames(){
…
```

```
}
function FrameData(){
this.version = 1.1;
}
var fd = new FrameData();
fd.version = 1.2;
```

3.2. Variablen

Variablennamen aus nur einem Buchstaben sollten auf temporäre „Wegwerfvariablen“ wie Schleifenzähler, Zwischenwerte oder anderes beschränkt werden. Klassisches Beispiel ist die Verwendung der Variablen „i“ als Zähler.

Namen von Variablen sind in **Englisch** zu schreiben, werden mit **kleinem Buchstaben** begonnen und dann in **Camel Case** Schreibweise weiter geschrieben.

Variablen müssen verständlich und selbsterklärend sein, im Zweifel erhalten Sie Name und Datentyp, z.B.

- Objekt (`memberObj["name"] = "Klaus Reinbach"`)
- Objekte mit Key / Value-Aufbau (`countryMap["DE"] = "Deutschland"`)
- Array (`memberData[0] = "O1001"`)
- String (`memberNameStr = "Klaus"`)
- Numerische Werte (`memberCountInt = 100`)
- Multistring (`namesMs = "; Reinbach; Schtolteheim; "`)

Variablen können über die JavaScript-Befehle var oder let initialisiert werden. Bitte darauf achten, dass bei der Definition der Variablen im Debugger let nicht verwendet werden kann. Details zum Definieren von Variablen finden sich in der gängigen JavaScript-Dokumentation.

Kurz zusammengefasst: Variablen, die mit var definiert werden befinden sich im Kontext des nächsten function Blocks, wohingegen let nur im Kontext des einschließenden Blocks (for, if etc.) existiert.

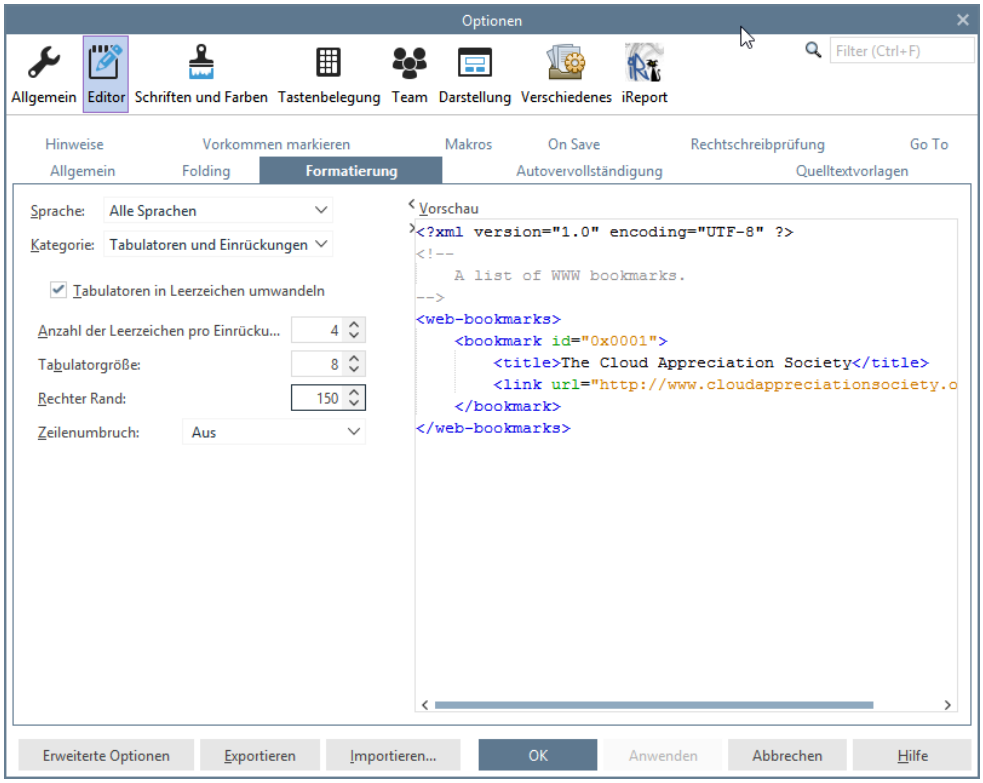
3.3. Schlüsselwörter

Schlüsselbegriffe werden nach dem Prinzip „Tabelle.Spalte“ benannt, damit diese den jeweiligen Modulen besser zugeordnet werden können. Beispielsweise sollte der Schlüsselbegriff Sprache, welcher sich auf die Sprache der Firma oder Person bezieht, besser als RELATION.LANG betitelt werden.

4. Prozesse & JDito-Code

4.1. Zeilenlänge

Die Zeilenlänge einer Codezeile sollte 150 Zeichen nicht überschreiten. Unter den allgemeinen Designereinstellungen kann der "rechte Rand" konfiguriert werden, damit dieser als Hilfslinie dient.



4.2. Funktionen

Der Name einer Funktion ist in **Englisch** zu schreiben, beginnt immer mit einem Buchstaben in **Kleinschreibung**, danach wird das jeweils erste Zeichen eines Wortes groß, den Rest kleingeschrieben (**Camel Case** eben). Auf Unterstriche ist nach Möglichkeit zu verzichten, da diese auf deutschen Tastaturen umständlicher einzugeben sind als auf US-Tastaturen. Ausnahmen siehe im Kapitel "Interne Funktionen".

Aus dem Namen der Funktion sollte wie bei Variablen auch die Datenrichtung hervorgehen. So sollten Funktionen, die Daten von Objekten abfragen, ein „`get`“ vorangestellt bekommen. Funktionen, die Daten schreiben oder in Tabellen eintragen, sollten ein „`set`“ als Vorsatz im Namen tragen. Beispiele: `getCustomerCount`, `setMailSubject`.

Die Funktionen sollten nicht **länger** als eine Bildschirmseite sein. Ganze Codeblöcke sollen und dürfen nicht gedankenlos von einer Funktion in die andere kopiert werden.

Beispiele für Methodennamen: `checkEmailAddress()`, `updateActivities()`, etc.

In Funktionen dürfen keine $comp-Variablen verwendet werden. Werden Werte aus Komponenten benötigt, so sind diese über Parameter in der Funktion zu übergeben.

4.2.1. Parameter

Parameternamen werden mit einem p-Präfix versehen, lauten also beispielsweise `pTableData`, `pHeaders`.

4.2.2. Interne Funktionen

Werden in Komponentenprozessen interne Funktionen benötigt, so werden diese mit vorangestelltem Unterstrich geschrieben:

```
function _sendMessage()
{
 // fancy code
}
```

4.3. Strings

Strings werden mit Anführungszeichen (") gekennzeichnet. Muss man in einem String " verwenden und möchte diese nicht escapen, kann auch auf einfache Hochkommas (') zurückgegriffen werden. Diese Praktik sollte aber nur in Ausnahmefällen verwendet werden.

4.4. SQL Querys

SQL-Abfragen werden in Strings angegeben. Im SQL-Text wird nach Keywords umgebrochen. Anweisungen werden **klein** geschrieben, Tabellen- und Spaltennamen **GROß**.

Es sollte immer ein Tabellen-Alias angegeben sein (Tabelle.Spalte).

Ein eigener Alias sollte nur definiert werden, wenn:

- Eine Tabelle mehrfach gejoint wird (eindeutige Aliase z.B.: ORGREL PERSREL vergeben)
- Es der Übersichtlichkeit eines SQLs dient

Zusätzlich kann bei komplexeren SQLs per Kommentar angegeben werden, an welcher Stelle in einem möglichen Array die Spalten angezeigt werden.

```
var sql = "select PERS.FIRSTNAME, PERS.LASTNAME, ORG.ORGNAME "
          // 0 - 2
        + "from RELATION"
        + "join ORG on RELATION.ORG_ID = ORG.ORGID "
        + "join PERS on RELATION.PERS_ID = PERS.PERSID";
```

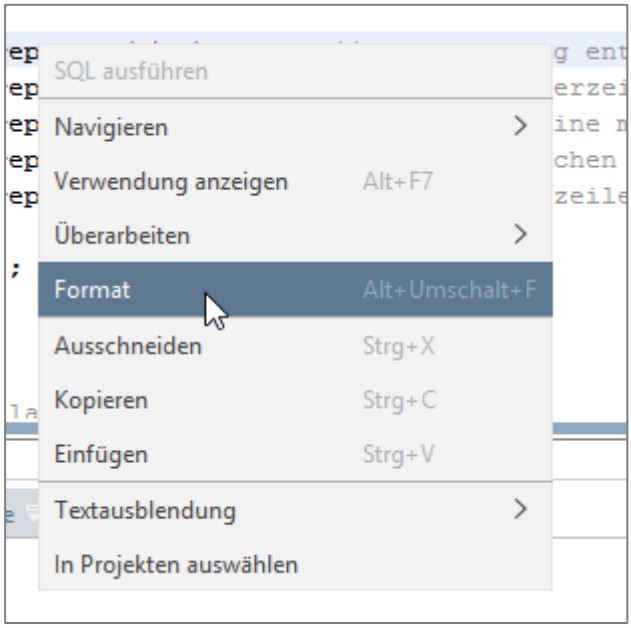
Diese Regel kann unter Umständen bei bestimmten Datenbanken nicht angewendet werden. Manche Datenbanken sind so konfiguriert, dass Tabellen- und Spaltennamen Case-Sensitiv sind. In diesem Fall gelten dann natürlich die Vorgaben des Datenbanksystems.

4.5. Anweisungsblöcke

Codeblöcke werden in JavaScript und auch in JDito in geschweifte Klammern eingeschlossen. Diese sollten jeweils in einer eigenen Zeile stehen und auf der Einrückungsebene der Anweisung oder Funktion, die den Block enthält, stehen. Ein Muster findet sich im nächsten Beispiel unten.

Enthält eine Anweisung nur ein Kommando, so können auch die geschweiften Klammern entfallen. Damit ist auch beim Lesen des Codes klar, dass nur eine Anweisung folgt, wenn z.B. nach einem `if` die geschweiften Klammern fehlen.

Über die Eigenschaft `Format` im Designer kann man sich die Formatierung vereinfachen.



Beispiel

```
var document = chooseTemplate( [7], pLanguage );
if (document)
{
     exportData( pCondition, document );
}
```

```
if(vars.getString("$global.isAdmin") == "true")
   result.string("Admin-Zugang"); // Eingerückt
```

In bestimmten Fällen ist dies nicht sinnvoll, z.B. wenn ein Import mithilfe des IMPORTERS gemacht wird.

4.6. Kommentare und Leerraum

Kommentare können nach dem Code bis zum Ende der Zeile gestaltet werden (sogenannte Zeilenkommentare oder *inline comments*) oder Sie können als eigener Block über mehr als eine Zeile (Zeilenkommentare oder *block comments*) geschrieben werden.

Kommentare sind in **Englisch** zu verfassen, da dies das Verständnis von Quellcodes international arbeitender Unternehmen deutlich erleichtert.

Zeilenkommentare (mit einem // eingeleitet) eignen sich zur Kommentierung von Details innerhalb des Codes. Sie sollten für das allgemeine Verständnis des Ablaufs bzw. des Algorithmus nicht notwendig sein.

Die Blockkommentare bzw. wichtige einzeilige Kommentare sollten immer in der derzeitigen Einrückungsspalte beginnen. Diese Kommentare beschreiben aus relativ abstrahierter Sichtweise die Funktion des Codes, indem das „**Warum**“ geschildert wird.

Auf jeden Fall **NICHT** so:

```
var count = 0;//Zählervariable count mit Number von 0 deklarieren
und definieren
```

Leerzeilen sollten zur Strukturierung des Codes eingesetzt werden. Sie verbessern die Lesbarkeit und die Gruppierung logisch zusammenhängender Codestrecken deutlich.

Beispiel

```
as = as.replace(/^\n/, "");       // remove CR at start of line
as = as.replace(/  /g, " ");      // remove double spaces
as = as.replace(/\\n/ig, "\n");   // replace new line marker
as = as.replace(/ *\n */g, "\n");// replace space at start and end
as = as.replace(/\s(?=\s)/g, ""); // remove blank lines
```

4.7. JSDoc

JSDoc wird in einem Kommentarblock über einer Funktion angegeben. Dieser Kommentar ist folgendermaßen aufgebaut:

Die erste Zeile beschreibt die Funktion selbst, danach werden die Parameter, ein Bespiel und der Rückgabewert beschrieben.

```
/*
* @description move import data to target
*
* @param {Object} pObject req the mapping line
*
* @example [iMove, { Source: 3, Target: "RELATION.ADDRESS" } ]
```

```
*
* @return {Boolean} false, if the import of the row is not possible.
otherwise true
*/
```

Folgende Tags sollten mindestens verwendet werden:

Tag	Beschreibung
description	Beschreibung der Funktion, also eine Erklärung, was die Funktion macht.
param	Beschreibt einen Parameter. Der Aufbau entspricht @param {Datentyp} Parametername Beschreibung. Als Typen stehen “String”, “Number”, “Object” und “Boolean” zur Verfügung. Werden mehrere Typen erwartet, können diese über | (Pipe) getrennt angegeben werden, z.B. {String|Number}. Durch Angabe von [] oder [[]] (eckige Klammern) nach dem Parameternamen kann angegeben werden, dass diese als Array erwartet werden. Werden die Parameternamen in eckige Klammern geschrieben, dann wird dieser Parameter als optional definiert, z.B. `@param {String} [name] Der Name der Person, die den Datensatz angelegt hat.`
return	Gibt den Rückgabewert der Funktion an. Diese Werte werden über @return {Datentyp} Beschreibung angegeben. Als Datentypen stehen alle Werte, die auch für die Parameter verwendet werden können, zur Verfügung.
example	Gibt ein Beispiel für die Verwendung an. Unterhalb dieses Tags können auch mehrzeilige Beispiele angegeben werden.

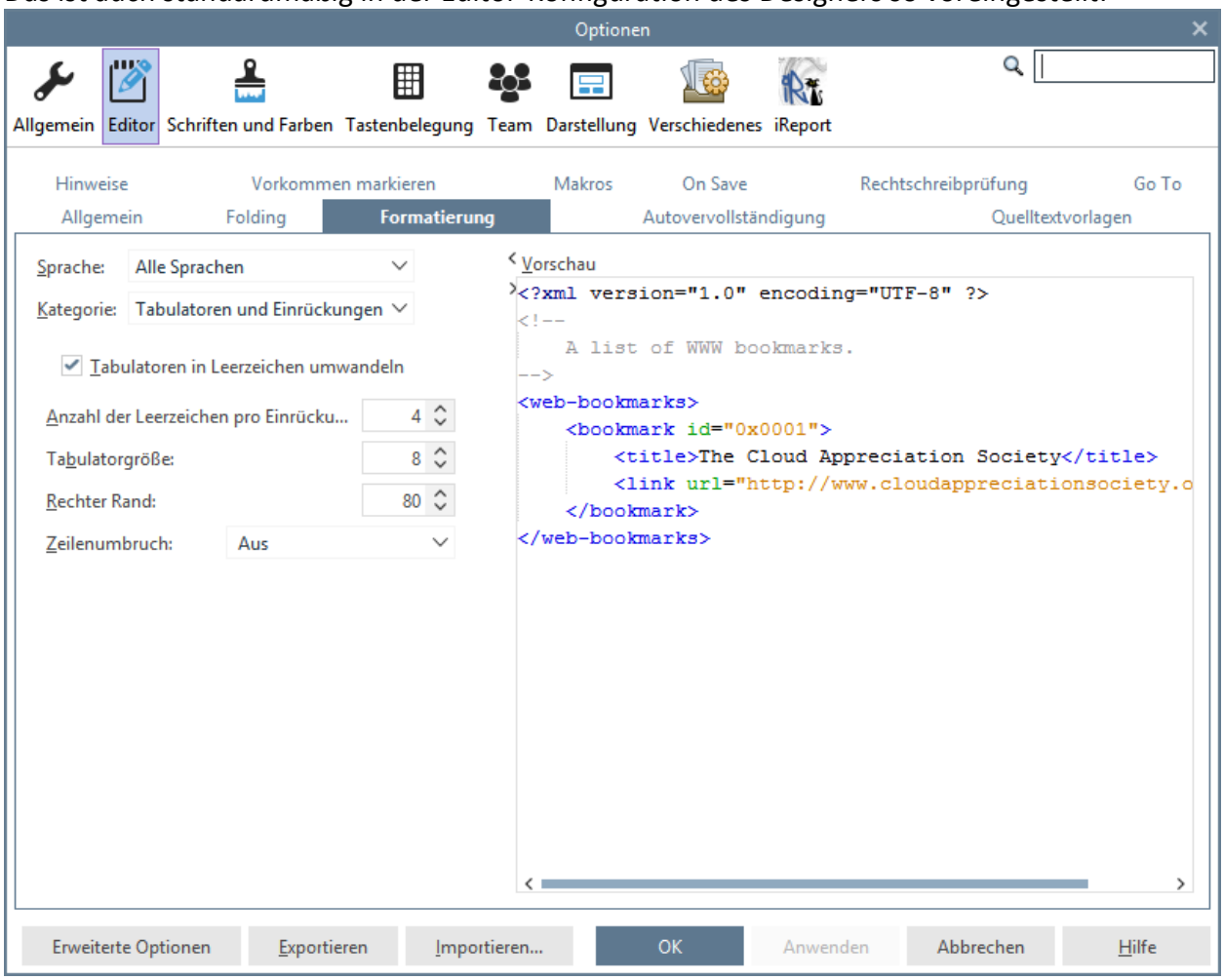
Weiter Informationen zu JSDoc, sowie weitere Parameter finden sich im JavaDito-Javascript-Handbuch.

Sowie unter http://code.google.com/p/jsdoc-toolkit/wiki/TagReference und http://usejsdoc.org

4.8. Einrückungen

Einrückungen sollten pro Blockebene (geschweifte Klammern) mit einem festen Wert von Leerzeichen oder Tabulatoren (4 Leerzeichen) erfolgen. Der einfachste Weg ist auch hier die Verwendung von `Format` im Designer.

Das ist auch standardmäßig in der Editor-Konfiguration des Designers so voreingestellt:



4.9. Autorenkennzeichnungen

Auf Autorenkennzeichnungen im Kommentar ist zu **verzichten**.

Autoren und verantwortliche Entwickler werden über die Kennzeichnungen im Git-Commit ermittelt.

4.10. Ticket-, CR- und Sprint-Referenzen im Code

Auf Verweise zu Tickets, Change Requests oder Sprints ist im Code zu verzichten.

4.11. Ausdrücke und Operatoren

Bei Ausdrücken werden Operatoren bzw. Variablen jeweils durch ein Leerzeichen getrennt. Damit erhöht sich die Lesbarkeit des Codes besonders bei komplexen Ausdrücken deutlich.

```
so:    ...  if(year < 1583 || year > 9999) ...
so nicht: ... if(year<1583||year>9999) ...
```

4.12. Bibliotheksprozesse

4.12.1. Präfix für den Funktionsnamen

Immer wieder benötigte Funktionen werden in Bibliotheken zusammengefasst, die über die Anweisung `import` in einem Prozess eingebunden werden können. Diese Einbindung durch Import erfolgt intern durch ein Hinzufügen des Prozesscodes in den aktuellen Prozess (analog dem "`include`" anderer Sprachen). Funktionen müssen daher systemweit eindeutig benannt werden. Aus diesem Grund erhalten alle Funktionen einer Funktionsbibliothek einen Präfix, der diese logisch zusammenfasst.

Beispiele:

- lib_knowledgemanagement
- lib_sql
- lib_util

Ausnahmen sind Prozesse, welche zwar als Funktionsbibliotheken genutzt werden, allerdings nur bestimmte Komponenten beim Aufbau ihrer Daten unterstützen (aditoTreeTable, aditoTreeTableAdditional) oder Bibliotheken, die eigene Module darstellen (z.B. IMPORTER).

4.12.2. Modularisierung der Bibliotheken

Zur besseren Kennzeichnung von Code und der Analyse von Bibliotheken sollte auf JSDoc zugegriffen werden. Dabei handelt es sich um eine Beschreibung der Funktion vor der Funktionsdefinition. Die dort angebotenen Funktionen können auch zur automatischen Code-Vervollständigung verwendet werden.

Auf Angabe einer Versionsnummer wird verzichtet (ausgenommen bei Bibliotheken, die ganze Module darstellen wie beispielsweise dem IMPORTER).

Weiter Informationen zu JSDoc finden sich im JavaDito-Javascript-Handbuch.

4.13. Logging

4.13.1. Arten von Logging

Grundsätzlich gibt es zwei Arten von Logging: Debug und Log.

Debug gibt dem Entwickler Informationen, die ihm helfen ein Problem zu lösen. Beim Debuggen können viele Informationen geloggt werden, da dies nur erfolgt, wenn -Dadito.debug=PROJECT aktiv ist.

Beim Log kommt es auf die Priorität der Ausgaben an. Interessante Informationen, die z.B. dem Fachadministrator oder dem Support helfen können, sollten mit geringer Priorität geloggt werden. Fehler und Warnungen sollten hingegen mit hoher Priorität geloggt werden.

4.13.2. Anwendungsfälle

Geloggt wird zum Beispiel in folgenden Fällen:

- Serverprozesse: Datenimport, Schnittstellen (z.B. CTI), …
- Plugins
- Webserviceaufrufe

Zu viel zu loggen kann genauso schlecht sein wie zu wenig Logausgaben. Durch zu viel Logging kann die Performance leiden.

4.13.3. Logging-Methoden

Zum Loggen gibt es Methoden aus der `lib_log`:

- `logMsg(ex)`: Loggt die Exception direkt ins Log.
- `logMsg(ex, …)`: Loggt eine benutzerdefinierte Meldung, die den Exception-Text enthält. Anstatt ex kann auch ein String mit einer benutzerdefinierten Meldung verwendet werden.
- Loggen einer benutzerdefinierten Meldung mit einer Objektinstanz, z.B. für lange Serverprozesse.

Zusätzlich dazu gibt es jede Methode auch zum Debuggen:

- `debugMsg(ex)`: Loggt die Exception direkt ins Log, wenn der Debug-Serverparameter für das Projekt gesetzt wurde.
- `debugMsg(ex, …)`: Loggt eine benutzerdefinierte Meldung, die den Exception-Text enthält. Anstatt ex kann auch ein String mit einer benutzerdefinierten Meldung verwendet werden. Auch hier wird nur geloggt, wenn der Debug-Serverparameter für das Projekt gesetzt wurde.
- Loggen einer benutzerdefinierten Meldung mit einer Objektinstanz, z.B. für lange Serverprozesse.

Des Weiteren gibt es auch Kernprozesse zum Loggen:

- `logging.log`
- `logging.show`

4.13.4. JavaExceptions vs rhinoExceptions

Die Kernmethoden können nur bei JavaExceptions auf den StackTrace zugreifen, dies ist bei rhinoExceptions nicht möglich. Daher kann es schwierig sein, auf den genauen Grund des Fehlers zuzugreifen. Die Methoden der `lib_log` `logMsg()` und `debugMsg()` fangen dies bereits ab.

4.14. try / catch

try-catch-Anweisungen sollen in Prozessen nicht verwendet werden, es sei denn

- es handelt sich um Serverprozesse oder
- es können erwartbare Fehler auftreten (Datei nicht vorhanden etc.) die nicht anderweitig abgefangen werden können.

Wenn unbedingt try-catch verwendet werden muss, dann muss die Exception in das Serverlog geschrieben werden. Es gibt sehr selten Fälle, in denen es sinnvoll ist, nicht zu loggen. In diesen Fällen sollte im Catch-Block ein Kommentar stehen, warum nicht geloggt wird.

Beispiel

```
try
{
    importDataFromCSV();
}
catch(ex)
{
    logging.log(ex, logging.ERROR); //Important: log level ERROR
}
```

4.15. Auskommentierter Code

Im Rahmen der Softwareentwicklung kann es passieren, dass bestimmte Code-Bestandteile nicht mehr benötigt werden, beispielsweise wenn man auf der Suche nach einem bestimmten Verhalten oder Fehler ist.

Beispiel

```
for (i = 0; i <  pUsersName.length; i++)
{
 // logging.show(vars.getString("$global.firstLastName"));
 if (vars.exists("$global.firstLastName") &&
vars.get("$global.firstLastName"))
 {
    //question.showMessage("in Schleife");
  userp = userMap[ text.decodeMS(pUsersName[i])[1].split(":")[1]
][tools.PARAMS];
  result.push(new Array(pUsersName[i], userp[tools.LASTNAME] + " " +
userp[tools.FIRSTNAME]));
  // logging.log(i);
 }
 else result.push(new Array(pUsersName[i], pUserTitle[i]));
}
```

Diese Codebestandteile müssen unbedingt vor einem Commit entfernt werden.

4.16. Übersetzungen

Damit Meldungen an den Benutzer in jedem Fall bei Verwendung von Mehrsprachigkeit übersetzt werden können, muss jede Ausgabe mit `question.showMessage` oder jede Rückgabe in einer Komponente mit `result.string` in `translate.text` gekapselt werden.

Beispiel

```
question.showMessage(translate.text("Beispiel"));
```

4.17. Dokumente im Others-Ordner des Designers

In diesem Ordner werden Hinweise zu Schlüsselwörtern (keywords.txt), Attributen (attributes.txt) und SQL-Statements (db changes.sql) abgelegt. Die Inhalte dieser Dateien werden auf Englisch verfasst.

4.17.1. Schlüsselwörter

In dieser Datei werden anfangs folgende Informationen angegeben. Die folgenden Schlüsselwörter werden eingetragen, wie in diesen Informationen beschrieben.

```
/*
add new keywords or keyword-changes here if necessary.
Append your changes at the end. (newest at the bottom)

here is an example with some values:
--
-- <<date ISO 8601: YYYY-MM-DD>> <<user>> <<commit-message from your
ticket>>
Keyname1: ProductAjstKeys
Keyname2: Ajst-Produktkennzeichen
+------+----------+
| KV   | Keyname1 |
+------+----------+
| 1    | AX930    |
+------+----------+
| 2    | PTLS40x  |
+------+----------+
*/
```

4.17.2. Attribute

Folgende Informationen werden am Anfang dieser Datei angegeben.

```
/*
add new attributes or attribute-changes here if necessary.
```

```
Append your changes at the end. (newest at the bottom)


here is an example with some values:
--
-- <<date ISO 8601: YYYY-MM-DD>> <<user>> <<commit-message from your
ticket>>
<<attribute-structure>>
*/
```

4.17.3. Datenbankänderungen

In dieser Datei werden anfangs folgende Informationen angegeben:

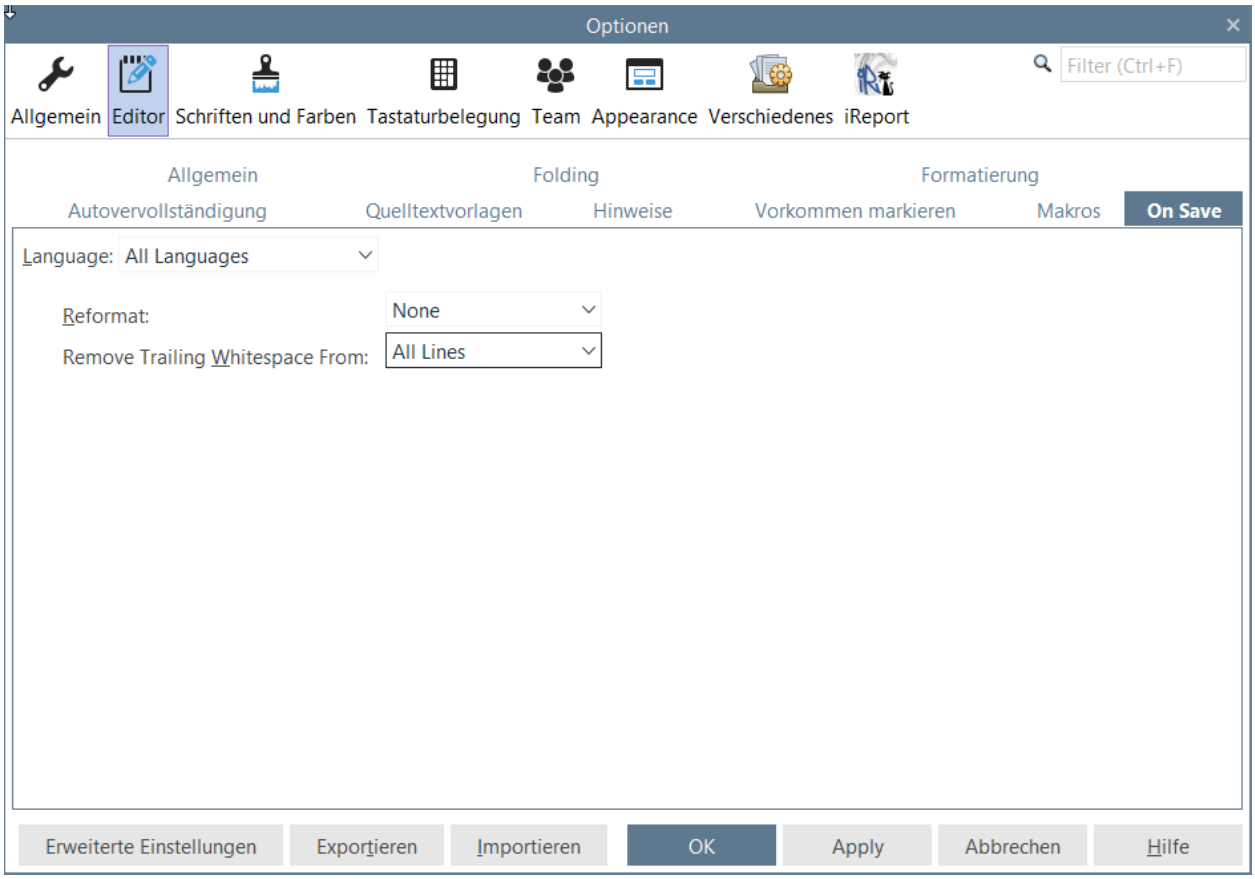
```
/*
place SQL-
Scripts for changes (DLL, DML, etc.) here. Append your changes at th
e end. (newest at the bottom)
Remember to refresh your repository-
structure (aliasDefinition) after a DML statements
Format:
-- <<date ISO 8601: YYYY-MM-DD>> <<user>> <<commit-
message from your ticket>>
<<sql-statement1>>;
<<sql-statementN>>;
*/
==> Remember to refresh your repository-
structure (aliasDefinition) after a DML statement
```

Ein SQL-Statement kann folgendermaßen aussehen:

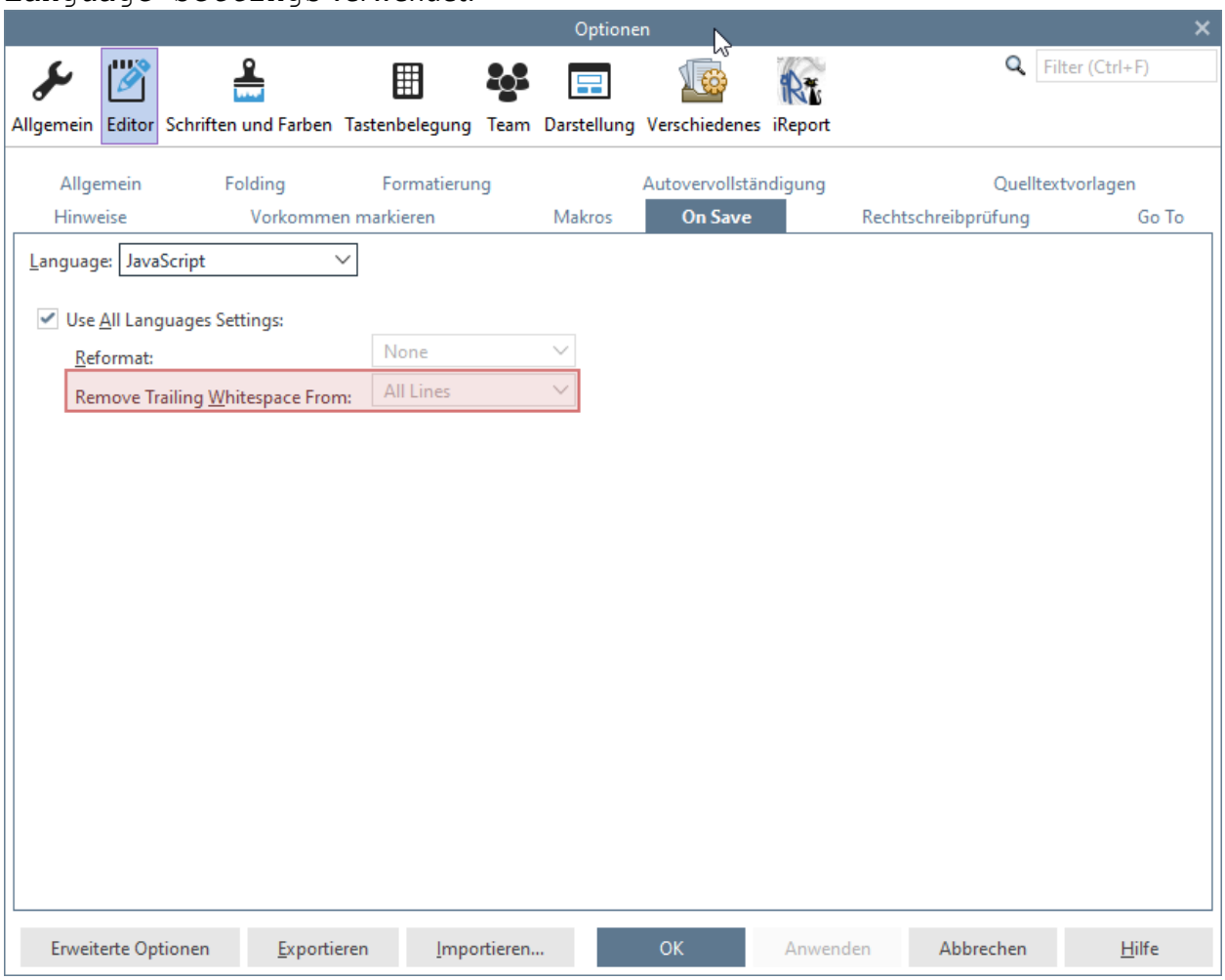
```
--12.10.2016 XY [Projekt: ADITO SolutionManagement - xRM
Modelle][TicketNr.: 2068][Impl.][Vorgangsverwaltung - Status]
alter table ACTIVITY add ACTIVITY_STATUS int;
update ACTIVITY set ACTIVITY_STATUS = 1 where ACTIVITY_STATUS is
null;
alter table ACTIVITY add EXPECTED_WORKING_PERIOD int;
drop table AOSYS_COLUMNREPOSITORY;
drop table AOSYS_INDEXREPOSITORY;
drop table AOSYS_TABLEREPOSITORY;
```

4.18. Hinweis auf onSave im Designer

Es gibt im Designer die Einstellungsmöglichkeit, dass beim Speichern automatisch Leerzeichen am Ende von JavaScript-Code entfernt werden. Um dies einzustellen, muss man zuerst in der Auswahlliste `Language` „All Languages“ auswählen. Dort kann man in der Auswahlliste `Remove Trailing Whitespaces From` „All Lines” auswählen.



Um diese Einstellung bei bestimmten Sprachen zu verwenden, wird ein Haken bei `Use All Language Settings` verwendet.



4.19. Standardwerte

Folgende Werte werden von Komponenten standardmäßig zurückgegeben:

Komponente	Standardwert
Checkbox	valueTrue: t valueFalse: f Jeweils CHAR(1)
Combobox (editable false)	Keyvalue / Keyname1 aus Schlüsselwörtern
Radiobutton	Keyvalue / Keyname1 aus Schlüsselwörtern

4.20. Coding und Debugger

Um Code möglichst einfach zu Debuggen, sollten folgende Formatierungen eingehalten werden.

4.20.1.If-Blöcke

Besitzt ein If-Block nur eine Codezeile, die ausgeführt werden soll, sollte diese folgendermaßen formatiert werden:

```
if (insData.length > 0)
  db.inserts(insData);
```

Dadurch kann im Debugger leichter in das if gesprungen werden, als bei diesem Code:

```
if (insData.length > 0) db.inserts(insData);
```

4.21. Demo-Code

Beispielcode zur Verdeutlichung der obigen Punkte. Schwer zu lesen? Einfach zoomen.

```
/*
 * adds an address to a specific tour
 *
 * @param {String} pTourid req ID tour where a specific address should be added
 * @param {String} pHomeRelID req relationid of the requested address
 * @param {String} pAttendanceType opt ATTENDANCETYPE, default intermediate target
 *
 *
 * @return {String} the ATTENDANCEID of the new record with that added address
 */
function insertAttendaceHomeAddress(pTourid, pHomeRelID, pAttendanceType)
{
    var attendanceType = pAttendanceType || "2";

    var cols = ["ATTENDANCEID", "TOUR_ID", "ROW_ID"
                , "DATE_NEW", "USER_NEW", "DURATION", "ATTENDANCETYPE", "FIXED"
                , "COUNTRY", "ZIP", "CITY", "ADDRESS","BUILDINGNO"];

    var table = "ATTENDANCE";
    var types = db.getColumnTypes(table, cols);

    var addr = db.array(db.ROW, "select RELATION.RELATIONID, ADDRESS.ADDRESS, ADDRESS.CITY "
        + ",ADDRESS.ZIP, ADDRESS.BUILDINGNO, ADDRESS.COUNTRY "
        + "from RELATION "
        + "join ADDRESS on (ADDRESS.ADDRESSID = RELATION.ADDRESS_ID) "
        + "where RELATION.RELATIONID = '" + pHomeRelID + "'");



    var attendenceId = util.getNewUUID();
    var vals = [attendenceId, pTourid, addr[0]
                , date.date(), vars.getString("$sys.user"), "0", attendanceType, "2"
                , addr[5], addr[3], addr[2], addr[1],addr[4]];

    db.insertData(table, cols, types, vals);

    return attendenceId;
}
```