Coding Guidelines in ADITO - Long Version

AID001

ADITO Academy

Version 1.1 | 2024-11-27

Version	**Changes**
1.1	• New chapter Module-specific terms
1.0	Initial release.

Character Formatting

The following signs will point you to specific sections:

ℹ	Hints and notes.
💡	Tips and tricks.
❗	This is important!
⚠	Warning! These actions are dangerous and can result in data loss!

The following font formatting applies:

Font type	Meaning
Mask	The mask, table or button to which the section refers
"Mask"	Terms that originate from the system and that need to be emphasized in the reading flow
`code();`	Code and program parts

Index

1. Purpose

The benefits of the ADITO Coding Guidelines are:

- Code is easily readable and understandable.
- Conventions help to keep formal requirements, which is especially helpful with weakly typed languages, such as JavaScript (and JDito, which is based on it).
- Conventions support standardised processes, which raises the product quality.
- Spelling conventions for ADITO models and elements enable a quick orientation in all parts of the ADITO project code, for programmers as well as for managers and customers.

These coding guidelines are also the basis of internal ADITO code reviews.

2. General rules

- All parts of ADITO code must be written in English language to achieve ever consistent code
- Generally, the naming of all code elements should be self-explanatory for easier reading
  - Examples:
    - A variable that holds the DOCUMENTTEMPLATELINKID should be named documentLinkId, not docLId.
    - A function that deletes all CONTACT data sets should be named deleteAllContacts, not delCon
- LOGGING-imports, aswell as LOGS themselves, have to usually be deleted (see Code Review). The rule does not apply to error-logs.
- If nothing is defined in this document, the usual JavaScript guidelines are to be followed
  - In programming history, general coding conventions have been evolved. In most cases, these can also be applied to ADITO projects. E.g., a function that returns the CONTACTID should be named getContactId, not retrieveContactId.
  - mozilla js guidelines
  - w3schools js conventions
- Always use semicolons at the end of a line if you can
- Do not add personal info in codes - the author can be determined through the git history
- Do not refer to tickets, CRs or sprint numbers (does not apply to file names in Liquibase)
- Always try to leave the code with a higher quality than you found it - you can always clean code

up when you stumble over it. BUT nobody has to reformat some "old" code - if the changed are quite recent ask the author to reformat.

- **Do not change code in default libraries in your project - this can cause problems with updates**
    - If you need changes for your specific purpose, copy the library & functions you need to change and add your project specific prefix (see project specific)
    - If you found a bug in a default library create a ticket for the responsible xRM Team

3. Coding (technical)

3.1. General

- Be lazy: Never write something twice, rather create a custom function - this makes it easier to maintain your code later on
- In your custom library always start out by creating a class
- Wherever possible, use the methods .forEach(), .map() or for(in) instead of equivalents that are harder to read
- A good reference of dos & don’ts in JavaScript is w3schools js best practices
- Don’t use prototype functions when nothing object specific is needed
- Do not use hardcoded values use constants

Do NOT use "for each … in" loops, they are deprecated. Use the function as mentioned above.
Never use the JavaScript function eval! It is a serious security risk and bad for performance.

3.2. Liquibase

Liquibase is an open source tool for database schema change management. It has not been developed by ADITO, but it is integrated into the ADITO Designer via a plugin (see option "Plugins" in the "Tools" menu). You can find a detailed documentation of Liquibase on the developer’s web site.
Nevertheless, ADITO has developed conventions for structure and spelling of the XML files used by Liquibase.

3.2.1. Structure

See project specific (5.1).

3.2.2. Location

In the "Projects" window of the ADITO Designer, the Liquibase xml files reside in the folder alias > Data_alias.
If your project is based on the xRM project, this folder already contains sub-folders with xml files related to xRM. Therefore, create a new folder on top level, using a name related to your project’s name. See also project specific (5.1).

3.2.3. Spelling

See overview for all spellings (5.0).

3.2.4. Error Handling / Prevention

To prevent errors and make sure that liquibase is properly running, there are some rules to follow:

- Never edit commited liquibase files - create a new one
- Use preconditions for structural change to prevent errors
- Write deletes before data inserts to prevent duplicates or errors (for example for Keywords, Attributes, etc.)

3.3. SQL

In ADITO, class SqlBuilder (in processes > libraries > Sql_lib) provides functionality for building prepared statements. Find more information in property "documentation" of Sql_lib and in the ADITO Customizing Manual.

- **Qualified column name**
  Every column name used in an SQL-statement should be preceded by the table name (Tablename.Columnname, e.g. ORGANISATION.ORGANISATIONID), as there are identical column names within different tables.

> Do NOT use any reserved keywords in your statements!

- **Prepared Statements**
  In general, prepared statements should be used in the code due to security reasons (especially, in order to prevent misusage by "SQL injection"). Its usage is supported via the code completion inside the designer.
  Example:

```
newSelect("CONTACT.CONTACTID, CONTACT.PERSON_ID, CONTACT.ORGANISATION_ID")
    .from("CONTACT")
```

```
        //The following replaces "...where CONTACT.CONTACTID = '" + pRelationId + "'..."
        .where("CONTACT.CONTACTID", pRelationId)
        .arrayRow();
```

- **Default columns**
  Every table needs the default columns USER_NEW, USER_EDIT, DATE_NEW, DATE_EDIT to see what happened when to the dataset. Add them to all new tables and use them in your entity.
- **ColumnAlias for Expressions**
  If you use an expression for a field within a record container, you should set a columnAlias that you later can easier analyse your SQL.
- **Functions that encapsulate only one SQL**
  return always the SqlBuilder object, because you can reuse the function as a subselect in another SqlBuilder, for example. All functions such as .cell() can be used after the function call.

4. Coding style (format)

For a semi-automatic formatting of code, use the shortcut [SHIFT] + [ALT] + [F]. Always check your code afterwards, because sometimes the code is misformatted!

- Each curly bracket deserves its own line in your code, this rule applies to functions, if-Statements, loops etc.
  - Exceptions: Registries (i.e. KeywordRegistry) and class declarations (i.e. function PermissionUtil () {})
- If-Statements should not be written in one line and always need curly brackets
  - High consistency makes code not only easier to read but also prevents mistakes (often occuring when not using curly brackets at all)
  - This rule does not refer to "condition ? exprIfTrue : exprIfFalse"
  - (Open to change when automatic reformatting is available)
- Make sure your indentation makes code more readable (i.e. you see immediately in which function your code belongs) - NOTE: the automatic formatting can also make mistakes
- Operators, variables, numbers, etc. are separated by blank spaces so you are able to identify them easily in your code

Example:

```
Contact.getEmployeeNames = function(pRoleId, pLength)
{
    if(pRoleId && pLength > 0)
    {
        //code
    }

    return employeeNames;
}
```

5. Spelling & Wording

5.1. Project specific

- For all new contexts, entities, database tables and libraries use a project specific prefix (i.e. FM_Congress_entity)
- Use the prefix also for all new fields in existing entities and database tables to differ from the default

5.2. JDito identifiers

- Each variable must be declared in a seperate row
  - This way a type can be immediately assigned aswell (i.e. let myObj = {};)
  - Defining variables inside an array in one row is still possible (i.e. var [contactId, date, user] = coolArray;)
- Variables must be written in camelCase
- Variable names that only contain one letter should only contain temporarily needed data (i.e. in for(let i; i < 100; i++))
- Constants are written in capital letters (i.e. const MYCONST)
- Functions and methods are written in DromedaryCase, their parameters are preceded by the letter "p" (i.e. CustomLib.myFunction(pPossibleParameter))

```
As mentioned, all of the above make your code more
readable and make it easier to identify different
components
```

Examples:

- Object (`memberObject["name"] = "Vernon Roberts";`)
- Objects with a key / value-construction (`countryMap["EN"] = "England";`)
- Array (`memberData[0] = "01001";`)

- String (`memberName = "Klaus";`)
- Numeric values (`memberCount = 100;`)

```
var tableCount = 13;
const MAX_LOOP = 5000;
var dataArray = [];

...

dataArray.forEach(function(item)
{
    CustomLib.myFunction(item, tableCount);
});
```

5.3. ADITO models

Real(!) English names should be given to ADITO models - they should provide a first insight into what exactly your model will do. The name of your context should be represented in its entity, views and providers (i.e. Contact, Contact_entity, ContactPreview_view, Contact_provider). +

- Typical structure: (ProjectPrefix_)ModelCamelCase_suffix
- Exceptions:
  - Contexts do not have suffixes
  - Entity fields: CAPITAL_LETTERS for database fields and CamelCase for all others
  - dromedaryCase is also used for: Executables and record containers
  - Database tables and colums are always written in CAPITAL_LETTERS

Model	Name pattern, description	Examples
Entity	CamelCase_entity	Activity_entity AttributeRelation_entity

Model	Name pattern, description	Examples
Entity Field	Fields holding the content of one specific database field are spelled equally: FIELDNAME, without underscores, except for 3 cases:	LASTNAME
	When a primary key of another table is referenced, the suffix "ID" is preceded by an underscore.	CONTACT_ID
	Certain columns automatically generated by the system.	DATE_EDIT
	When multiple primary keys of multiple tables are referenced, it **ALWAYS** have to be named like this. The OBJECT_TYPE is the entity that the OBJECT_ROWID is refering to.	OBJECT_ROWID, OBJECT_TYPE
	All other fields are written in CamelCase.	ReturnDate
Field Group	<SUMMARY>_fieldGroup	FULL_NAME_fieldGroup
Aggregate Field	<name of corresponding EntityField>_aggregate	Organisation_entity: count_aggregate
Context	CamelCase	Activity AttributeRelation
View	CamelCase_view CamelCaseFunction_view The naming of standardized views is : Main, Filter, Preview, Edit, PreviewMultiple, or Lookup	PersonMain_view PersonFilter_view PersonEdit_view PersonPreview_view PersonLookup_view
	Other views are named according to their function.	PersonDetail_view PersonEditDefaults_view PersonSimpleList_view

Model	Name pattern, description	Examples
View Template	CamelCase + s (if plural) The naming of a view template should be determined by its content. The plural is used, if multiple datasets can be shown at once (e.g. in a table).	Person Persons
	CamelCase + s (if plural) + View-Templatetype If a view template is related to different views (in particular, inside the CamelCaseFilter_view) then the template type should be used as suffix. The plural is used, if multiple datasets can be shown at once (e.g. in a table).	PersonsTable PersonsTreetable PersonsTimeline
	Prototypical construction of a CamelCasePreview_view: The view templates should be named as shown on the right. Additionally required view templates are given in between.	Header + [Details] + Info + [AdditionalInfo] with 2: Header, Info with 3: Header, Details, Info with 4: Header, Details, Info, AdditionalInfo
RecordContainer	dromedaryCase	Standard RecordContainers: simply always "db", "jdito", "index", or "dataless". Additional RecordContainers: arbitrary name, reflecting the purpose
Parameter	CamelCase_param	RowId_param OrderAddress_param

Model	Name pattern, description	Examples
Provider	CamelCase_provider	Activity_provider AttributeRelation_provider
Consumer	CamelCase_consumer	Activity_consumer AttributeRelation_consumer
Dashboard	CamelCase_dashboard	Sales_dashboard
Dashlet config	CamelCase	AllContacts
Dashlet	<DashletConfigName>_dashlet	AllContacts_dashlet
Action	CamelCase_action	NewActivity_action OpenEditDefaultsView_action
FilterExtension	CamelCase_filter	Favorites_filter
FilterExtensionSet	CamelCase_filter	Attribute_filter
Library	CamelCase_lib	Keyword_lib ActivityTask_lib
Executable	Project_dromedaryCase_serverProcess dromedaryCase_serverProcess dromedaryCase_rest dromedaryCase_workflowService usw.	DreSo_updateClassification_serverProcess buildSerialLetter_serverProcess workflowRoles_rest createSalesproject_workflowService usw.

Model	Name pattern, description	Examples
Database table	TABLENAME, without underscores, except for 2 cases:	OFFER OFFERITEM
	Essential tables have 'AB_' (for **A**DITO **b**asic) as a prefix.	AB_OBJECTRELATION
	System tables have 'ASYS_' (stands for **A**DITO **sys**tem) as a prefix.	ASYS_USERS
Database column	COLUMNAME, without underscores, except for 2 cases:	LASTNAME
	When a primary key of another table is being referenced, the suffix "ID" is preceded by an underscore.	CONTACT_ID
	Specific columns automatically generated by the system.	DATE_EDIT

5.4. Module-specific terms

When developing your modules, please keep strictly to the respective spelling conventions. Some of these conventions exist only for orientation purposes, while others are bindingly required, as modularization mechanisms rely on them (Example: language files).

Here is a list of the most important spelling conventions:

- Modules in Git:
  - only lowercase letters (e.g. "salutation")
  - on demand, use hyphen (e.g. "csv-importer"), but no other special characters, like underscore etc.
  - ideally, add an icon and a suitable description, as, e.g., done in the Git subgroup "xRM-Platform"
- ExtensionPoints: "<model name>ExtensionPoint", e.g., "FieldExtensionPoint", "ActionFieldExtensionPoint". [One-sided ExtensionPoints] can be named genericly, e.g., OpportunityExtensionPoint

- implementations of ExtensionPoints: Naming according to the spelling guidelines in AID001 Coding Styles, e.g., MYFIELD (upper case) for database-related EntityFields, or MyField (CamelCase) for calculated fields
- language files: The naming follows the syntax "<root file name>_<module name>" (e.g., "_____LANGUAGE_de_activity") and is automatically applied, when you create a new language file via option "New" from the context menu of folder "language". Do not change the autogenerated names, as they are technically required, in order to merge the several module-specific language files into one single file, when transpiling.
- services:
    - global services: "<module name><ServiceName>_service", e.g., "attributeUsageContexts_service"
    - Entity services: "<module name><ProcessName>_service", e.g. "activityOnDBDelete_service"
- service implementations: Although in the xRM modules, the pattern "<moduleName>_impl" (e.g., "task_impl") is frequently used, a more expressive naming is recommended, e.g., addNewActivity_impl. Temporary implementations are named "tmp_impl".
- modifications: Do not change the autogenerated names.
- keyword registries: <ModuleName>Keywords_registry, e.g., ActivityKeywords_registry or ContactManagementKeywords_registry.

5.5. Liquibase

Each liquibase file name should contain an action prefix, the tablename and what is added, changed, deleted etc.. See examples in the table below.
For your project create your own liquibase folder and sub-folders for each version, i.e., "sprint_1". In each sub-folder you should create a changelog.xml and two sub-folders to separate database structure ("struct") and database content ("data"). Now you can store all xml files in these folders.

- Always use DATETIME instead of TIMESTAMP due to the 2038-Problem
- Boolean values have to be stored in TINYINT

Operating content	Prefix	Example
creating a table	create_	create_address.xml

Operating content	Prefix	Example
inserting new content (keywords, demo data, etc.)	insert_	insert_keyword_offerstatus.xml
adding columns	add_	add_checklist_position.xml
change structure (table, column, constraint, etc.)	alter_	alter_contact_status.xml
updating content (data sets)	update_	update_keyword_medium.xml
deleting content	delete_	delete_attribute_newsletter.xml
removing structure (including constraints etc.)	remove_	remove_address_contact_nullConstraint.xml

6. Documentation and Comments

6.1. Comments

Unnecessary parts of the comments have to be removed before committing!

BUT not every comment has to be deleted! You do not have to delete comment that describe why is was done that way. It does **NOT** mean that you have to write what is happening, just why you did this. This helps to debug the code later on, especially if a coworker has to deal with it!

DON'T:

```
var resultArray = [];

userProfilesFull.forEach(function(pItem)
{
    // logging.show(vars.getString("$global.firstLastName"));

    if (vars.get("$global.firstLastName"))
    {
        // logging.show("in loop");

        userProfile = userMap[text.decodeMS(pItem)[1].split(":")[1]][tools.PARAMS];
        resultArray.push([pItem, userProfile[tools.LASTNAME] + " " + userProfile[tools.FIRSTNAME]]);

        // logging.log(i);

    }
    else
    {
        resultArray.push([pUsersName[i], pUserTitle[i]]);
    }
});
```

DO (practical comment):

```
//SqlBuilder not implemented as the statement needs to be finished in the switch case
switch (pComparison)
   {
       case "EQUAL":
       case "NOT_EQUAL":
           resultSqlCond = resultSqlCond + " '" + pRawvalue + "')";
       ...
   }
```

It is useful to think: "Will I know why I did that in a month?" and "Will somebody else understand what I was doing here?".

6.2. Documentation of Functions/Methods

- We refer to JSDoc to comment our Functions
- All functions must be documented
- A function documentation has to have at least:
    - A short description
    - All parameters described
    - The return value described (if existend)

Example:

```
/**
 * makes a SqlBuilder that checks if there's (not) a commrestriction for a contact
 *
 * @param {String} pMedium medium to check if undefined, do not check it
 * @param {boolean} pNoRestriction if true, the condition gets every contact that has no commrestriction, otherwise every contact that
has a commrestriction
 * @param {String|Number} [pStartDate=currentDate] date to check against the start date of the commrestriction
 * @param {object} [pEntityInfo] an object with two properties:
 *       <ul>
 *           <li>entityFields: array of entity fields</li>
 *           <li>entityIdField: the id field name as string</li>
 *       </ul>
 *
 * @return {SqlBuilder} the condition
 */
```

Possible parameter types: (see also JSDoc param tag & JSDoc types)

Type	**Example**	**Description**
String	{String}	A simple string has to be entered.
Boolean	{boolean}	True or False have to be entered.
Number	{Number}	Simple number
Object	{Object}	If you use an object as a parameter it is necessary that you have listed every possible value that is needed to fulfill the logic of your code. An example is shown above.
Array	{Array}	A one dimensional array like ["1", "2"]. If it has a specific type other than string (string is standard), you can include a type in [] like {Array[boolean]}.
Multi-diminesional Array	{Array[][]}	As with a one dimensional array you can enter specific types to each array you use e.g. {Array[boolean][Number]}
Special Types	{SqlBuilder}	If your parameter includes a special object that is build beforehand by a seperate function you can use their object name.
Optional parameter	{String} [pParam]	This parameter is optional for the function.
Optional parameter with default	{String} [pParam=default]	This parameter is optional and has a default value when not declared.

6.3. Documentation Property

Not only is it necessary to comment your code but to also have a documentation about a field, param, action, library or even an entire entity to describe what it does. You can use the documenation property in order to manifest your logic. The content of the documenation is written in AsciiDoc (.adoc) files. You may have to install the AsciiDoctorJ-plugin in your ADITO designer.

> You have to write a documentation if the logic behind it is not 100% self explanatory!

7. Reserved (Key-)Words

On http://www.reservedwordsearch.com/ you can enter names to be checked automatically (no guarantee for completeness!).

Other reserved words lists:

- JavaScript
- Java
- General SQL
- MariaDB
- MSSQL
- MySQL
- Oracle